Towards a verified Lustre compiler with modular reset

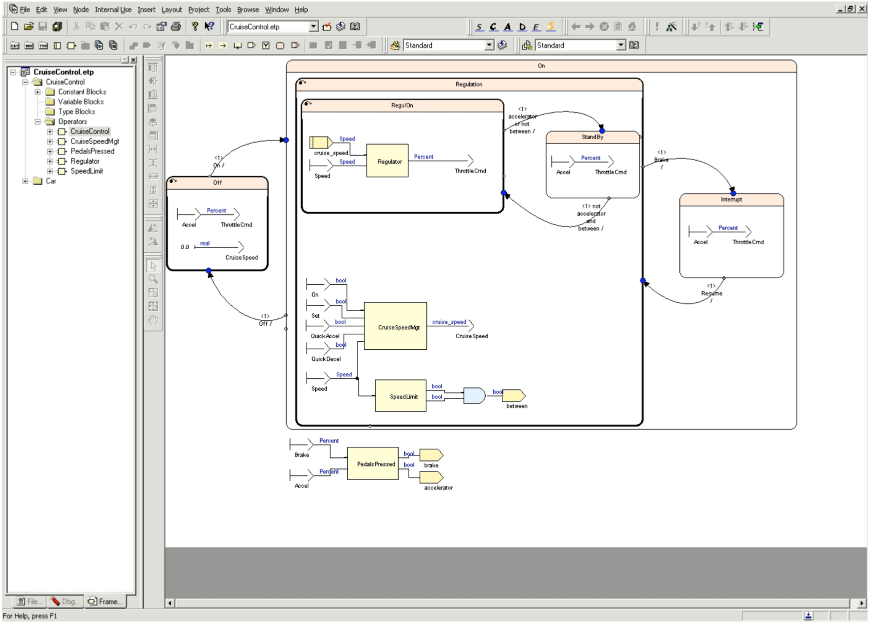
Timothy Bourke[1,2] Lélio Brun[1,2] Marc Pouzet [3,2,1]

[1]Inria Paris

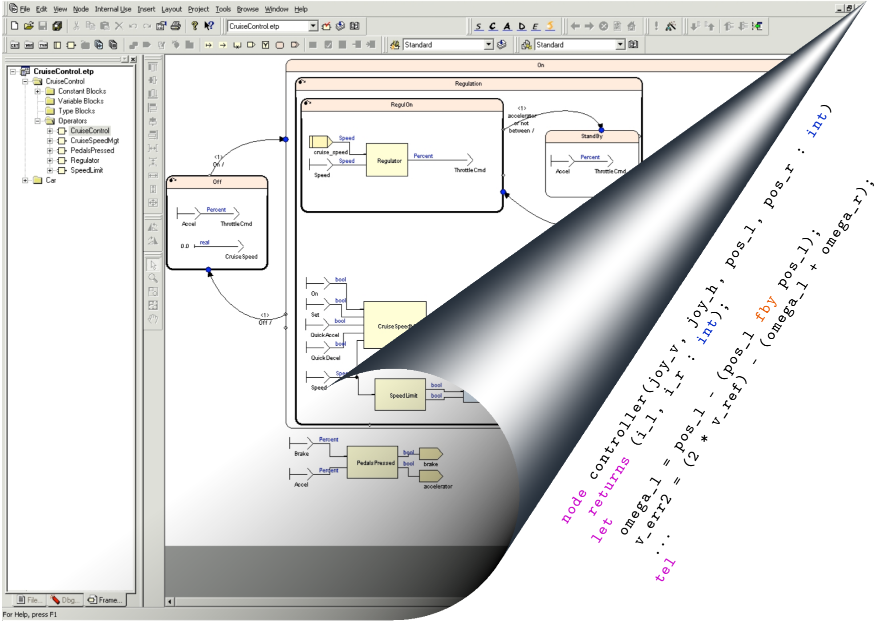
[2]DI ENS

[3]UPMC

SCOPES 2018 — May 30, 2018



Screenshot from ANSYS/Esterel Techologies SCADE Suite



Screenshot from ANSYS/Esterel Techologies SCADE Suite

Context

State of the art: Scade

- Specification norms (DO-178C), industrial certification

Context

State of the art: Scade

- Specification norms (DO-178C), industrial certification
- Onerous and expensive development process

Context

State of the art: Scade

- Specification norms (DO-178C), industrial certification
- Onerous and expensive development process
- No *formal* proof of correctness

Context

State of the art: Scade

- Specification norms (DO-178C), industrial certification
- Onerous and expensive development process
- No *formal* proof of correctness

Goal

Develop a formally verified code generator

Context

State of the art: Scade

- Specification norms (DO-178C), industrial certification
- Onerous and expensive development process
- No *formal* proof of correctness

Goal

Develop a formally verified code generator

- formal verification, mechanized proofs, proof assistant (eg. Coq[1])

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

Context

State of the art: Scade

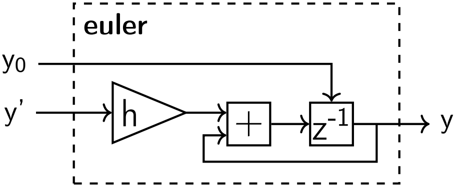
- Specification norms (DO-178C), industrial certification
- Onerous and expensive development process
- No *formal* proof of correctness

Goal

Develop a formally verified code generator

- formal verification, mechanized proofs, proof assistant (eg. Coq)
- Scade
  - Lighten the qualification to norms
  - Provide a complete semantics

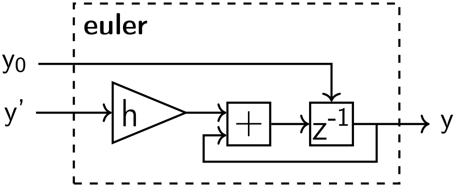
Lustre:[1] example




```
node euler(y0, y': int)
  returns (y: int)
  var h: int;
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"

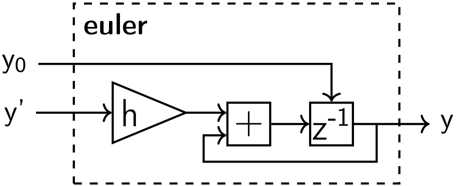
Lustre: example




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

y_0	0
y'	4
y	0
h	2

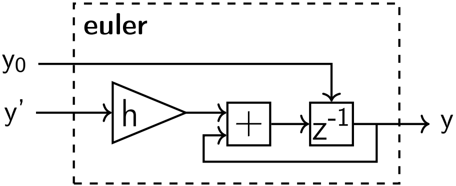
Lustre: example




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

y_0	0	5
y'	4	2
y	0	8
h	2	2

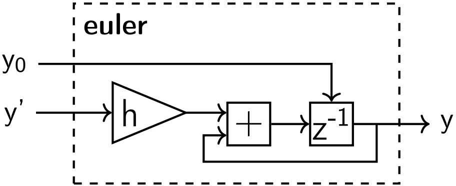
Lustre: example




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

y_0	0	5	10
y'	4	2	1
y	0	8	12
h	2	2	2

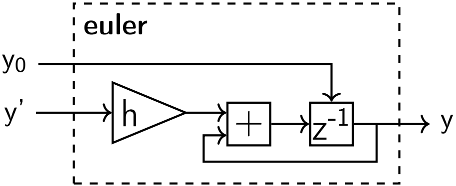
Lustre: example




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

y_0	0	5	10	–
y'	4	2	1	–
y	0	8	12	–
h	2	2	2	–

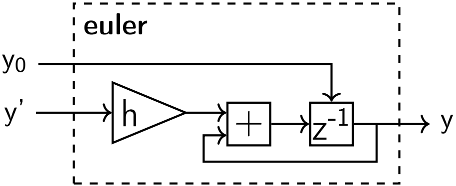
Lustre: example



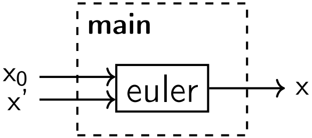

```
node euler(y0, y': int)
  returns (y: int)
  var h: int;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel
```

y_0	0	5	10	–	15	$\cdots$
y'	4	2	1	–	3	$\cdots$
y	0	8	12	–	14	$\cdots$
h	2	2	2	–	2	$\cdots$

Lustre: example

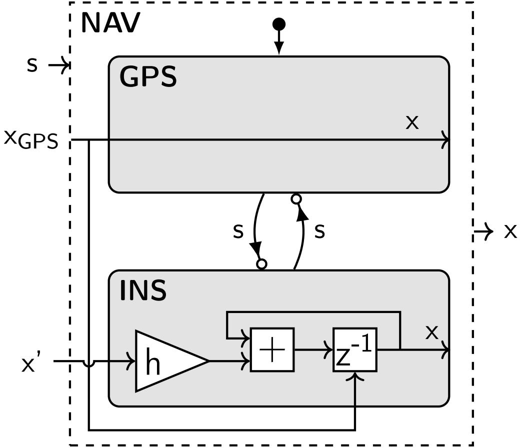



```
node euler(y0, y': int)
  returns (y: int)
  var h: int;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel
```



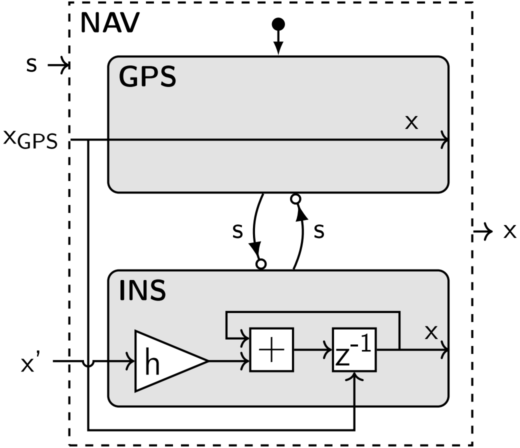

```
node main(x0, x': int)
  returns (x: int)
let
  x = euler(x0, x');
tel
```

Scade-like state machines and reset primitive



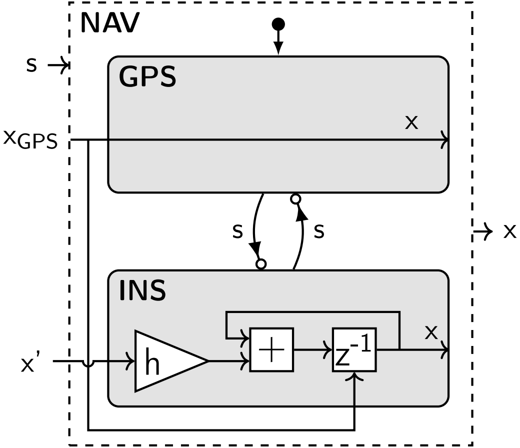
Scade-like state machines and reset primitive

- Can be compiled into Lustre



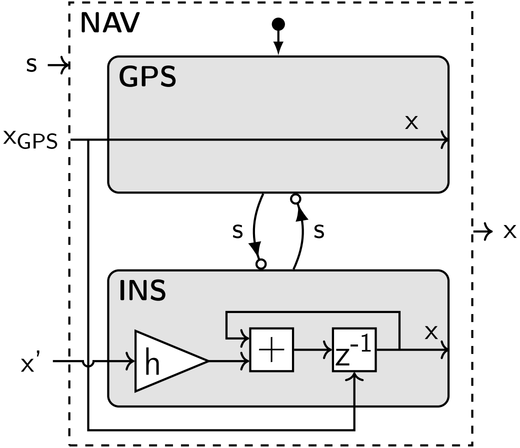
Scade-like state machines and reset primitive

- Can be compiled into Lustre
- *Reset*:
    - Reset the state of a node, ie. reinitialize the *fby*s



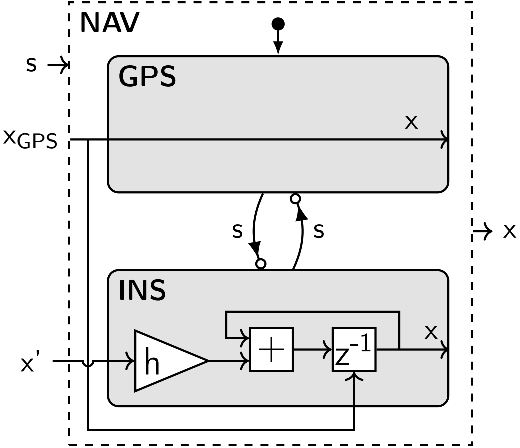
Scade-like state machines and reset primitive

- Can be compiled into Lustre
- *Reset*:
  - Reset the state of a node, ie. reinitialize the *fby*s
  - Useful primitive (not only for state machines)



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?



Non-modular reset

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

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

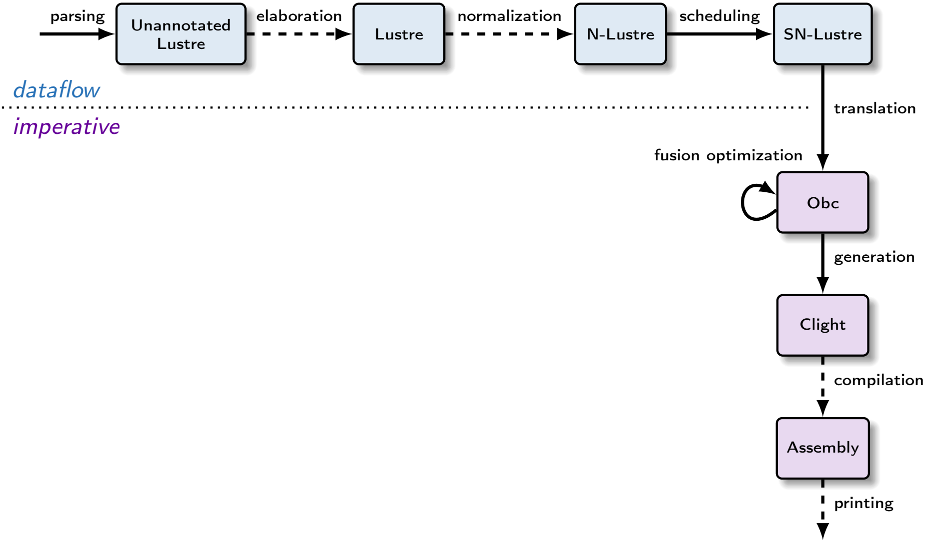
```
node euler(y0, y': int; r: bool)
  returns (y: int)
  var h: int;
let
  y = if r then y0
      else (y0 fby (y + y' * h));
  h = 2;
tel

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

Non-modular reset

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

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

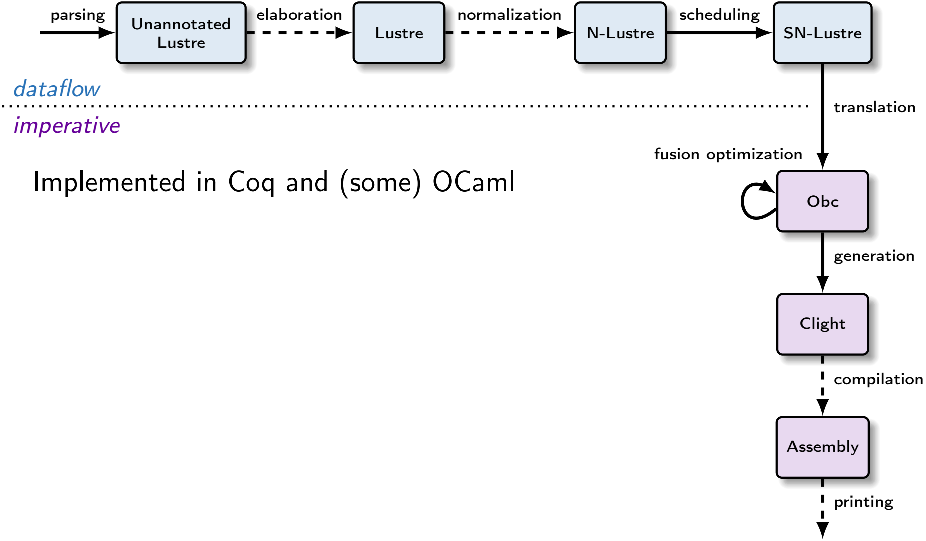
```
node euler(y0, y': int; r: bool)
  returns (y: int)
  var h: int;
let
  y = if r then y0
      else (y0 fby (y + y' * h));
  h = 2;
tel

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

Vélus:[1] a verified compiler



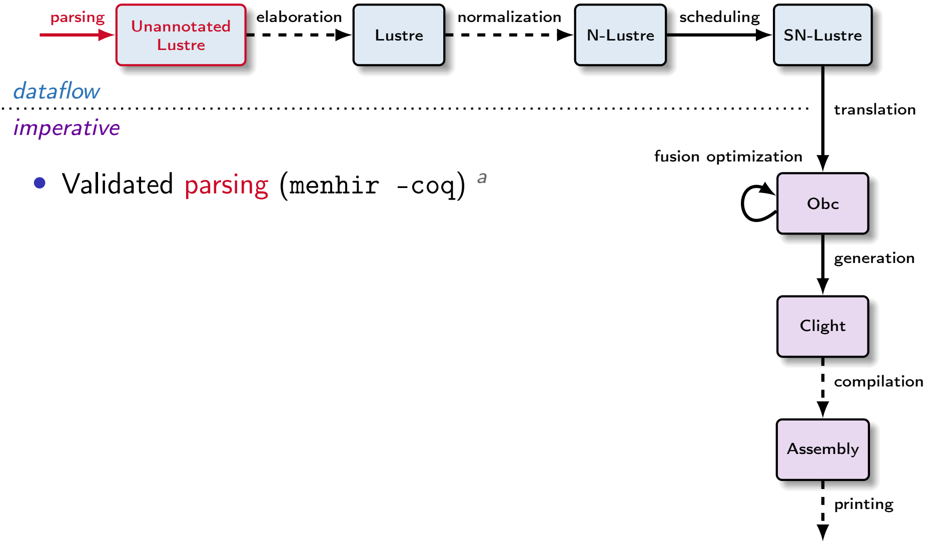
[1] Bourke, Brun, Dagand, Leroy, Pouzet, and Rieg (2017): "A Formally Verified Compiler for Lustre"

Vélus: a verified compiler



Implemented in Coq and (some) OCaml

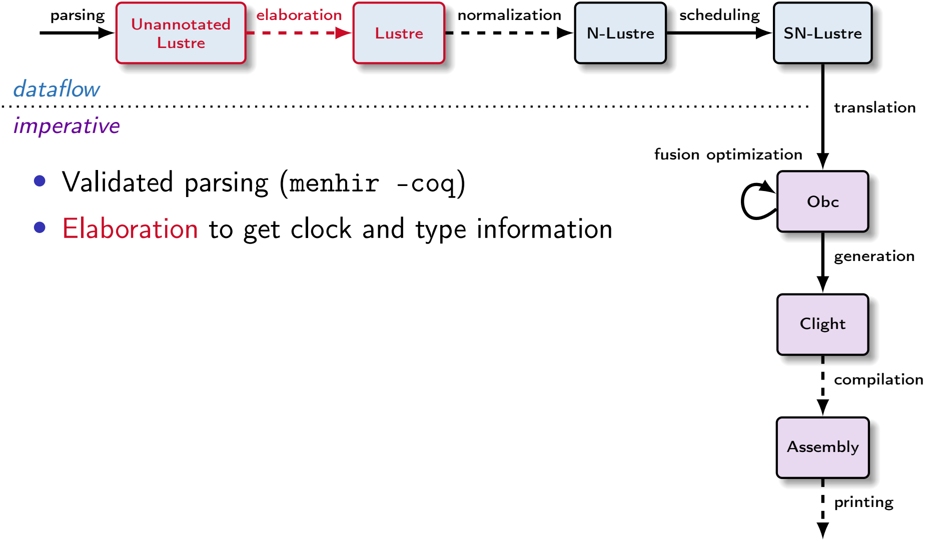
Vélus: a verified compiler



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

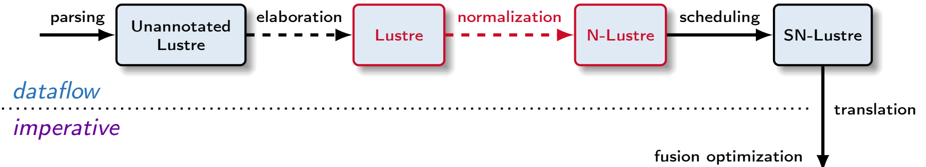
[a] Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"

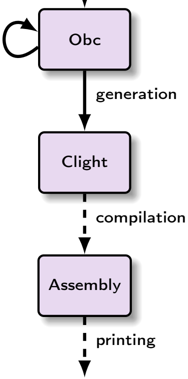
Vélus: a verified compiler



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

Vélus: a verified compiler

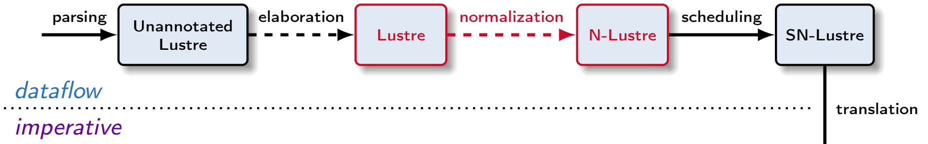


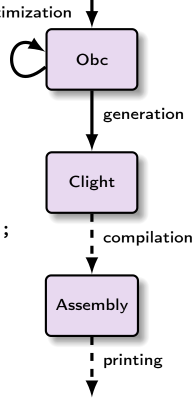


- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type information
- Not yet implemented: normalization[a]

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

Vélus: a verified compiler



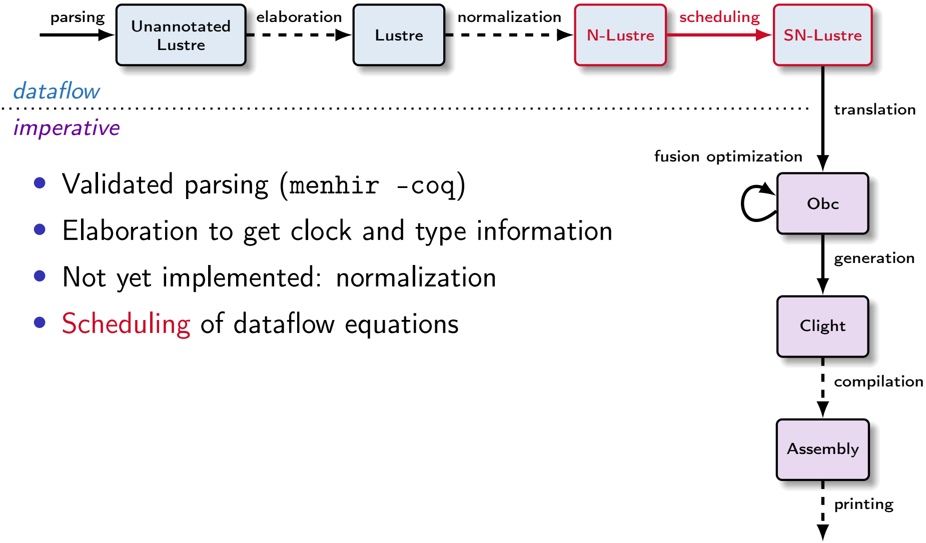



```
node euler(y0, y': int)
  returns (y: int)
  var h: int;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel
```

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

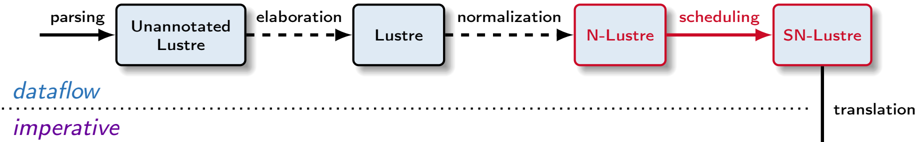
$$e_0 \text{ fby } e \mapsto \text{if (true fby false) then } e_0 \text{ else } (0 \text{ fby } e)$$

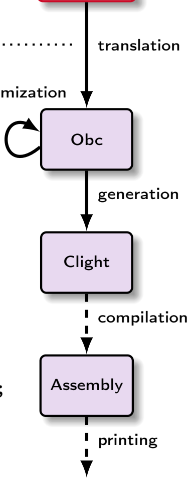
Vélus: a verified compiler



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

Vélus: a verified compiler

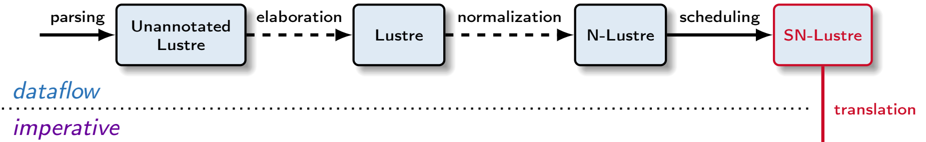




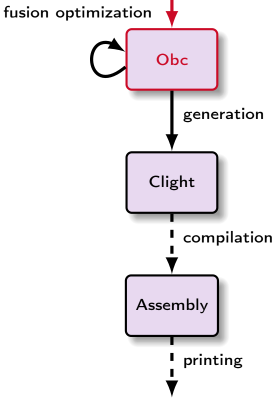

```
node euler(y0, y': int)
  returns (y: int)
  var h, y1: int;
      init: bool;
let
  init = true fby false;
  y1 = 0 fby (y + y' * h);
  y = if init then y0
      else y1;
  h = 2;
tel
```

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

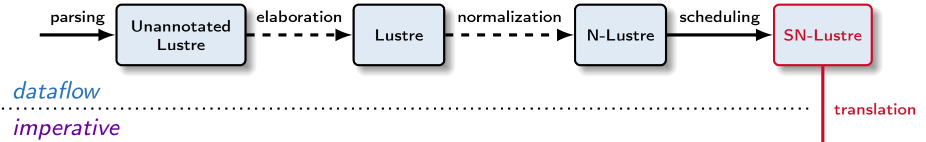
Vélus: a verified compiler

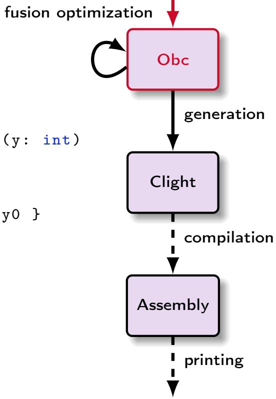


- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type information
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code



Vélus: a verified compiler

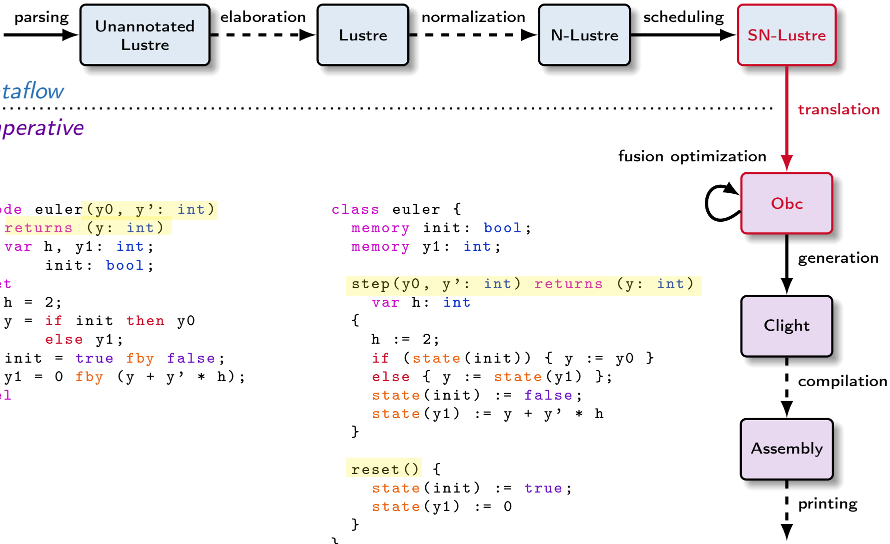





```
node euler(y0, y': int)
  returns (y: int)
  var h, y1: int;
      init: bool;
let
  h = 2;
  y = if init then y0
      else y1;
  init = true fby false;
  y1 = 0 fby (y + y' * h);
tel
```

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

  step(y0, y': int) returns (y: int)
    var h: int
  {
    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
  }
}
```

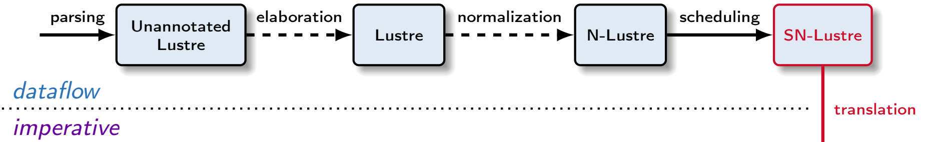
Vélus: a verified compiler

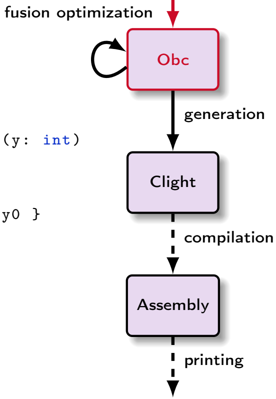



```
node euler(y0, y': int)
  returns (y: int)
  var h, y1: int;
      init: bool;
let
  h = 2;
  y = if init then y0
      else y1;
  init = true fby false;
  y1 = 0 fby (y + y' * h);
tel
```

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

  step(y0, y': int) returns (y: int)
    var h: int
  {
    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

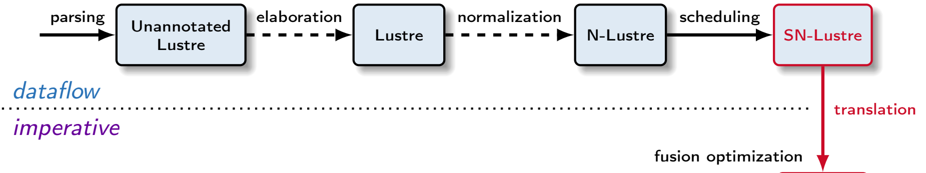





```
node euler(y0, y': int)
  returns (y: int)
  var h, y1: int;
      init: bool;
let
  h = 2;
  y = if init then y0
      else y1;
  init = true fby false;
  y1 = 0 fby (y + y' * h);
tel
```

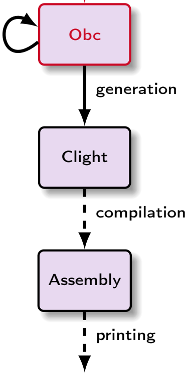
```
class euler {
  memory init: bool;
  memory y1: int;

  step(y0, y': int) returns (y: int)
    var h: int
  {
    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

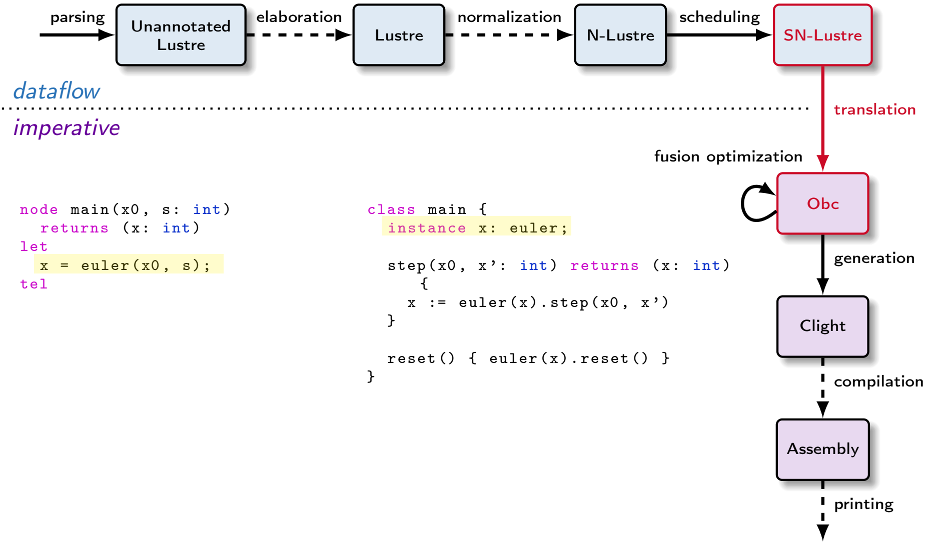



```
node main(x0, s: int)
  returns (x: int)
let
  x = euler(x0, s);
tel
```

```
class main {
  instance x: euler;

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

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



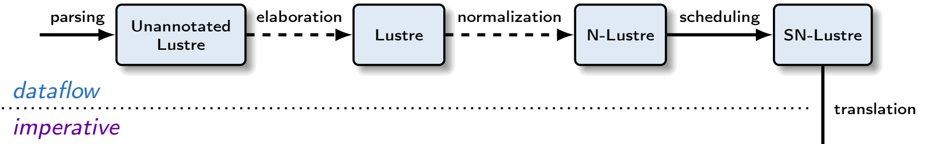
Vélus: a verified compiler

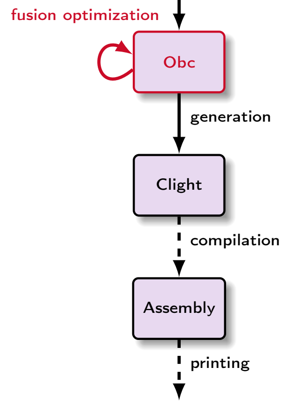



```
node main(x0, s: int)
  returns (x: int)
let
  x = euler(x0, s);
tel
```

```
class main {
  instance x: euler;

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

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

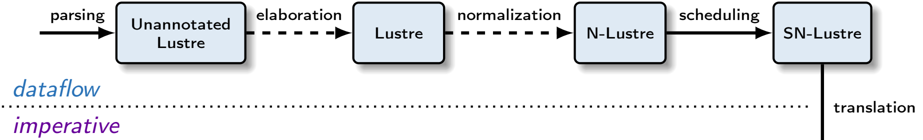
Vélus: a verified compiler



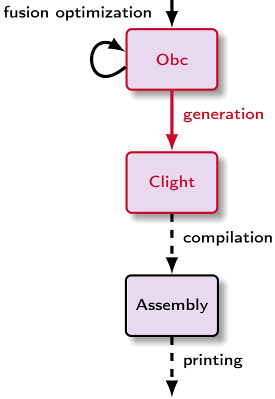


- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type information
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code
- Optimization of intermediate Obc code

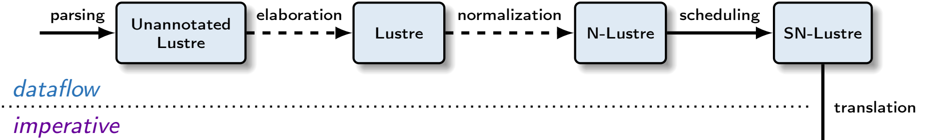
Vélus: a verified compiler

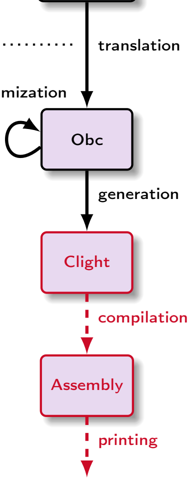


- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type information
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code
- Optimization of intermediate Obc code
- Generation of Clight code



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"

Adding the modular reset

- Node application: $f(\vec{e})$
  call the node f

Adding the modular reset

- Node application: $f(\vec{e})$
  call the node f
- Modular reset: $f(\vec{e})$ every r
  reset the internal state (delays) of f at each tick of r

Adding the modular reset

- Node application: $f(\vec{e})$
  call the node f
- Modular reset: $f(\vec{e})$ every r
  reset the internal state (delays) of f at each tick of r

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

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

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

node main(x0, x': int)
  returns (x: int)
  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 [1]

```
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
```

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

Semantics?

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

```
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
```

Definable 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	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$

Infinitely unrolling the recursion

mask: a cofixpoint written in Coq

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

Infinitely unrolling the recursion

mask: a cofixpoint written in Coq

r	F	F	T	F	F	T	F	T	F	$\cdots$
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	$\cdots$
$\vdots$										
$nat(i)$ 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

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

Formal semantics

Node application

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

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

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

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

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

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

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

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

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

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

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

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

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

Use of an universally quantified relation as a constraint:

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

Formal semantics in Coq

```
Inductive sem_equation : history → clock → equation → Prop :=
...
| SeqApp:
    Forall2 (sem_lexp H b) es ess →
    Forall2 (sem_var H) xs xss →
    sem_node f ess xss →
    sem_equation H b (EqApp xs ck f es None)
| SeqReset:
    Forall2 (sem_lexp H b) es ess →
    Forall2 (sem_var H) xs xss →
    sem_var H r rs →
    sem_reset f (bool_mask rs) ess xss →
    sem_equation H b (EqApp xs ck f es (Some r))
...

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

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

Formal semantics in Coq

```
Inductive sem_equation : history → clock → equation → Prop :=
...
| SeqApp:
    Forall2 (sem_lexp H b) es ess →
    Forall2 (sem_var H) xs xss →
    sem_node f ess xss →
    sem_equation H b (EqApp xs ck f es None)
| SeqReset:
    Forall2 (sem_lexp H b) es ess →
    Forall2 (sem_var H) xs xss →
    sem_var H r rs →
    sem_reset f (bool_mask rs) ess xss →
    sem_equation H b (EqApp xs ck f es (Some r))
...

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

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

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

Formal semantics in Coq

```
Inductive sem_equation : history → clock → equation → Prop :=
...
| SeqApp:
    Forall2 (sem_lexp H b) es ess →
    Forall2 (sem_var H) xs xss →
    sem_node f ess xss →
    sem_equation H b (EqApp xs ck f es None)
| SeqReset:
    Forall2 (sem_lexp H b) es ess →
    Forall2 (sem_var H) xs xss →
    sem_var H r rs →
    sem_reset f (bool_mask rs) ess xss →
    sem_equation H b (EqApp xs ck f es (Some r))
...

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

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

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

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

Formal semantics in Coq

```
Inductive sem_equation : history → clock → equation → Prop :=
...
| SeqApp:
    Forall2 (sem_lexp H b) es ess →
    Forall2 (sem_var H) xs xss →
    sem_node f ess xss →
    sem_equation H b (EqApp xs ck f es None)
| SeqReset:
    Forall2 (sem_lexp H b) es ess →
    Forall2 (sem_var H) xs xss →
    sem_var H r rs →
    sem_reset f (bool_mask rs) ess xss →
    sem_equation H b (EqApp xs ck f es (Some r))
...

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

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

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

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

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

Naive compilation

$y = f(x)$ every r:

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

Naive compilation

$y = f(x)$ every r:

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

Problem with fusion optimization:

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

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

Naive compilation

$y = f(x)$ every r:

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

Problem with fusion optimization:

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

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

Conclusion

Summary

- A verified compiler for Lustre
- Simple semantics for modular reset

Conclusion

Summary

- A verified compiler for Lustre
- Simple semantics for modular reset

Future Work

- Compilation and proof of correctness
- A need for a new intermediate language
- Automata

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.
```

N-Lustre: abstract syntax

$le ::=$		**expression**
$\mid$	k	(constant)
$\mid$	x	(variable)
$\mid$	le `when` x	(when)
$\mid$	$\diamond\ e$	(unary operator)
$\mid$	$e \oplus e$	(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)
$\mid$	$\vec{x} :: c = x(\overrightarrow{le})$	(app)
$\mid$	$\vec{x} :: c = x(\overrightarrow{le})$ `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`

Obc: Abstract Syntax

$e :=$		**expression**
$\mid$	x	(local variable)
$\mid$	$\mathrm{state}(x)$	(state variable)
$\mid$	k	(constant)
$\mid$	$\diamond\ e$	(unary operator)
$\mid$	$e \oplus e$	(binary operator)

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

$cls :=$ **declaration**

$\mid$ class k { (class)

memory $\overrightarrow{x^{ty}}$

instance $\overrightarrow{i^{k}}$

$f(\overrightarrow{x^{ty}})$ returns $(\overrightarrow{x^{ty}})$ [var $\overrightarrow{x^{ty}}$] { s }

}

Separation logic in CompCert

predicate

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

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

Separation logic in CompCert

predicate

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

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

conjonction

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

Separation logic in CompCert

predicate

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

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

conjonction

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

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)

$$\begin{aligned}\text{match\_states} \triangleq\ & \text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\ & * \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\ & * \text{pure}\,(\text{ge}(\text{f\_c}) = co_{out})\end{aligned}$$

self pointer

out pointer

output structure

States correspondence

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

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

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)

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

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

States correspondence

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

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

parameters and local variables $\approx$ temporaries

States correspondence

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

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

subcalls output
structures allocation