Un compilateur vérifié pour Lustre

Timothy Bourke[1,2] Lélio Brun[1,2] Pierre-Évariste Dagand[4,3,1]

Xavier Leroy[1] Marc Pouzet[4,2,1] Lionel Rieg[5,6]

1. Inria Paris

2. DI, École normale supérieure

3. CNRS

4. Univ. Pierre et Marie Curie

5. Yale University

6. Collège de France

Journée LaMHA/LTP - Automne 2017, LIP6—11 octobre 2017

What did we do?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2013): "A Formalization and Proof of a Modular Lustre Code Generator"].
- Prove that the generated code implements the dataflow semantics.

What did we do?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2013): "A Formalization and Proof of a Modular Lustre Code Generator"].
- Prove that the generated code implements the dataflow semantics.
- Coq? [The Coq Development Team (2016): *The Coq proof assistant reference manual*]
  - A functional programming language;
  - 'Extraction' to OCaml programs;
  - A specification language (a form of higher-order logic);
  - Tactic-based interactive proof.

What did we do?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2013): "A Formalization and Proof of a Modular Lustre Code Generator"].
- Prove that the generated code implements the dataflow semantics.
- Coq? [The Coq Development Team (2016): *The Coq proof assistant reference manual*]
  - A functional programming language;
  - 'Extraction' to OCaml programs;
  - A specification language (a form of higher-order logic);
  - Tactic-based interactive proof.
- Why not use HOL, Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

What did we do?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2013): "A Formalization and Proof of a Modular Lustre Code Generator"].
- Prove that the generated code implements the dataflow semantics.
- Coq? [The Coq Development Team (2016): *The Coq proof assistant reference manual*]
  - A functional programming language;
  - 'Extraction' to OCaml programs;
  - A specification language (a form of higher-order logic);
  - Tactic-based interactive proof.
- Why not use HOL, Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

  CompCert: a formal model and compiler for a subset of C
  - A generic machine-level model of execution and memory
  - A verified path to assembly code output (PowerPC, ARM, x86)
  - No need for a garbage-collected runtime.

  [Blazy, Dargaye, and Leroy (2006): "Formal Verification of a C Compiler Front-End"] [Leroy (2009): "Formal verification of a realistic compiler"]

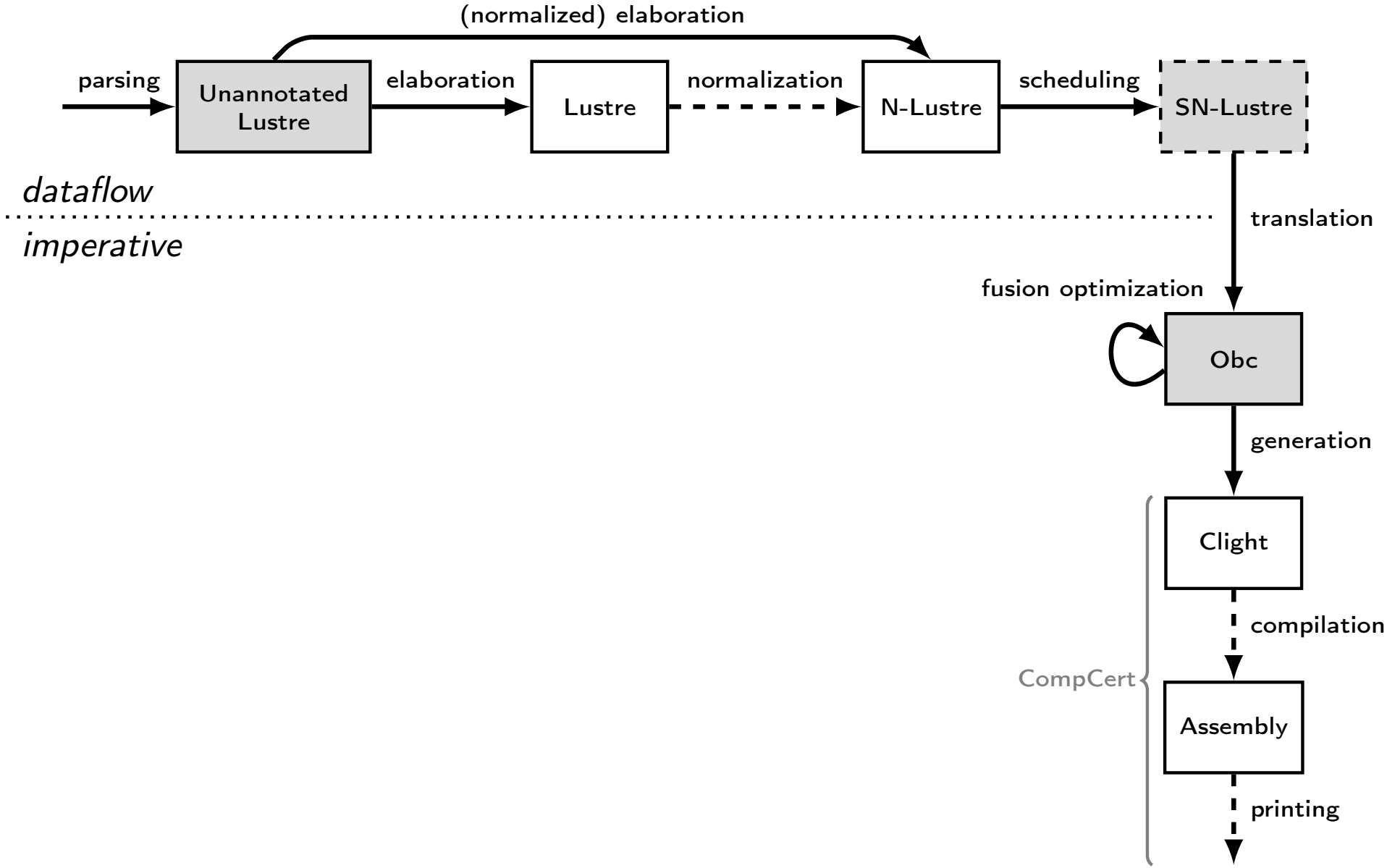
What did we do?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2013): "A Formalization and Proof of a Modular Lustre Code Generator"].
- Prove that the generated code implements the dataflow semantics.
- Coq? [The Coq Development Team (2016): *The Coq proof assistant reference manual*]
  - A functional programming language;
  - 'Extraction' to OCaml programs;
  - A specification language (a form of higher-order logic);
  - Tactic-based interactive proof.
- Why not use HOL, Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

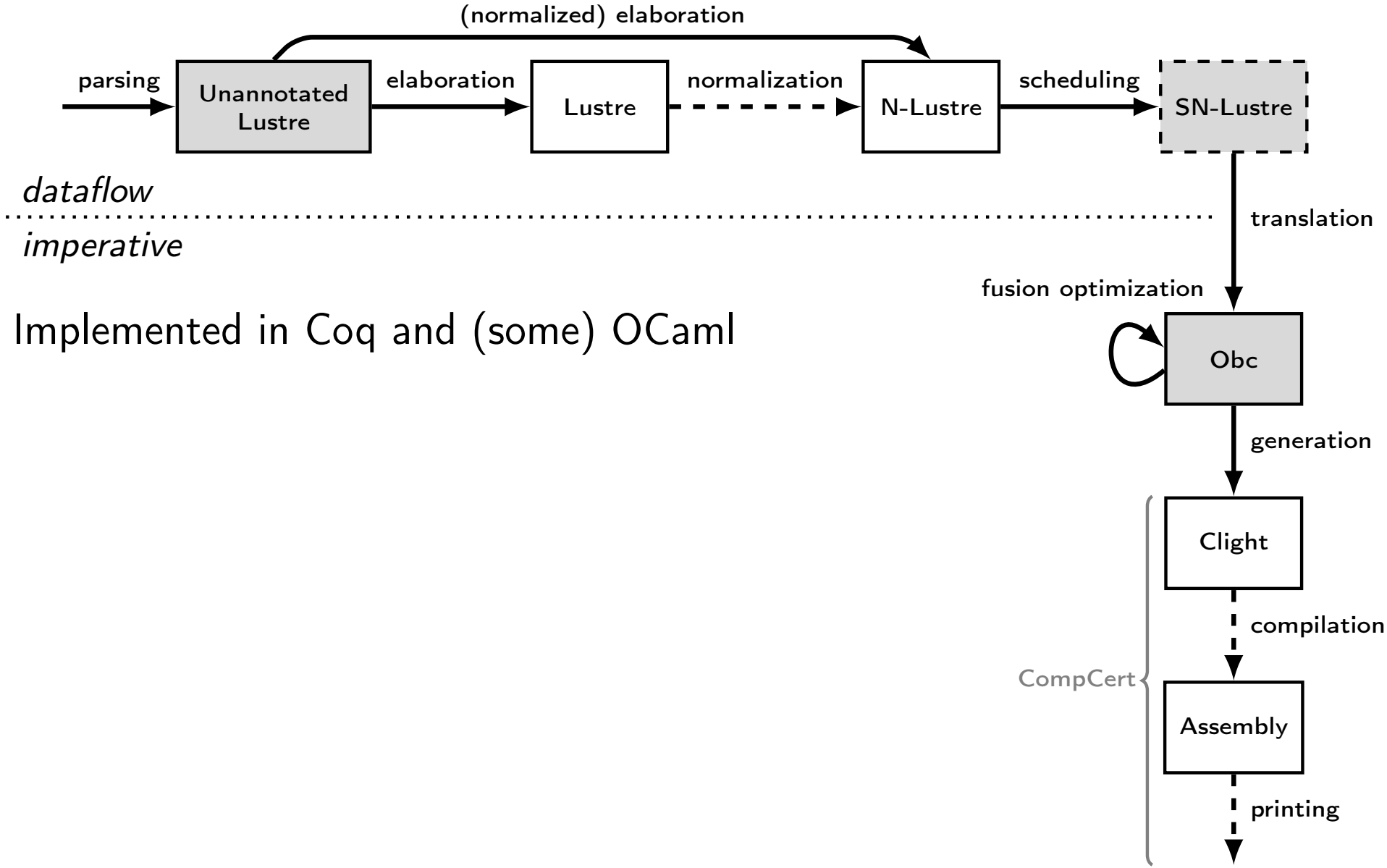
  CompCert: a formal model and compiler for a subset of C
  - A generic machine-level model of execution and memory
  - A verified path to assembly code output (PowerPC, ARM, x86)
  - No need for a garbage-collected runtime.

  [Blazy, Dargaye, and Leroy (2006): "Formal Verification of a C Compiler Front-End"] [Leroy (2009): "Formal verification of a realistic compiler"]
- Computer assistance is all but essential for such detailed models.

The Vélus Lustre Compiler

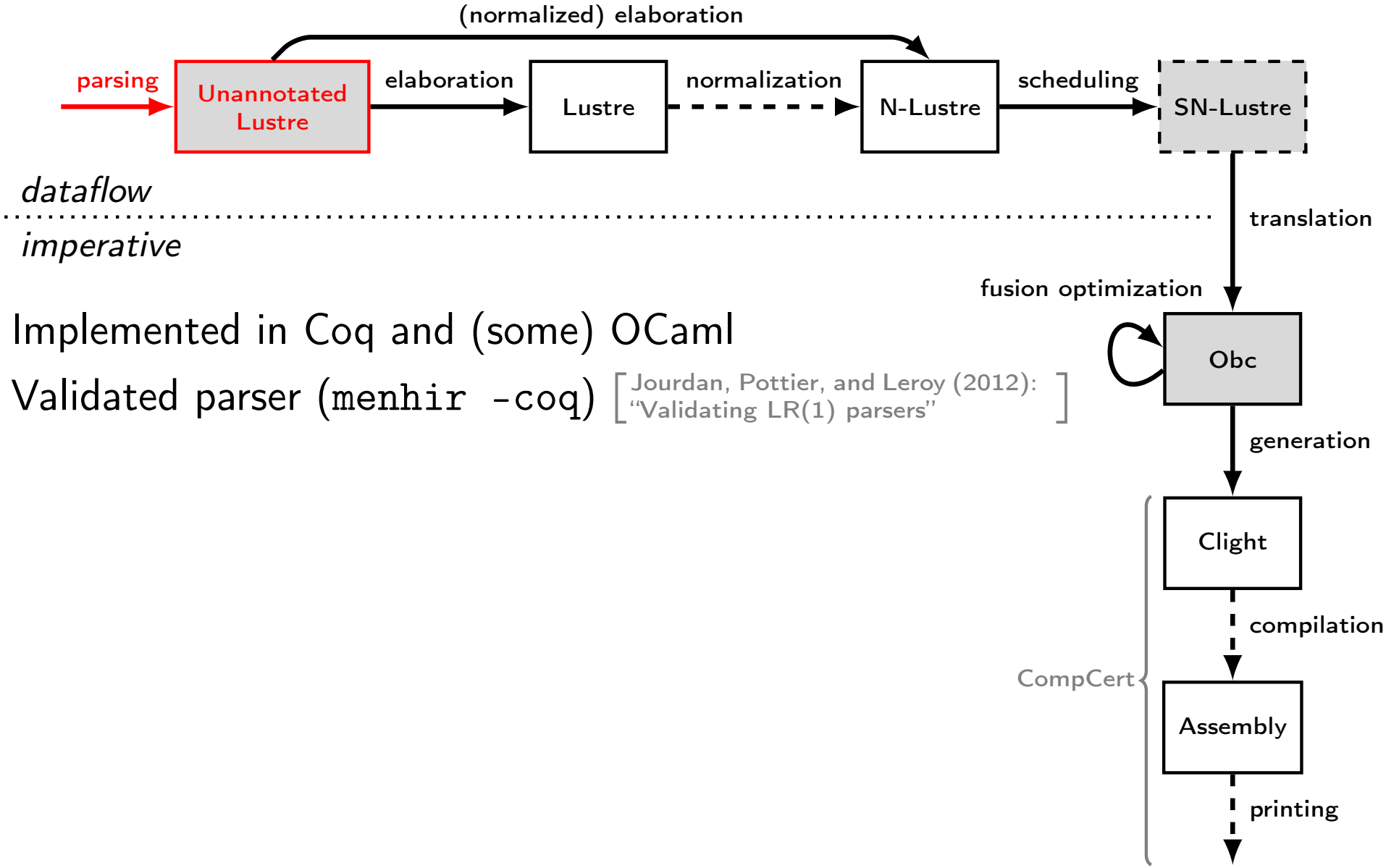


The Vélus Lustre Compiler



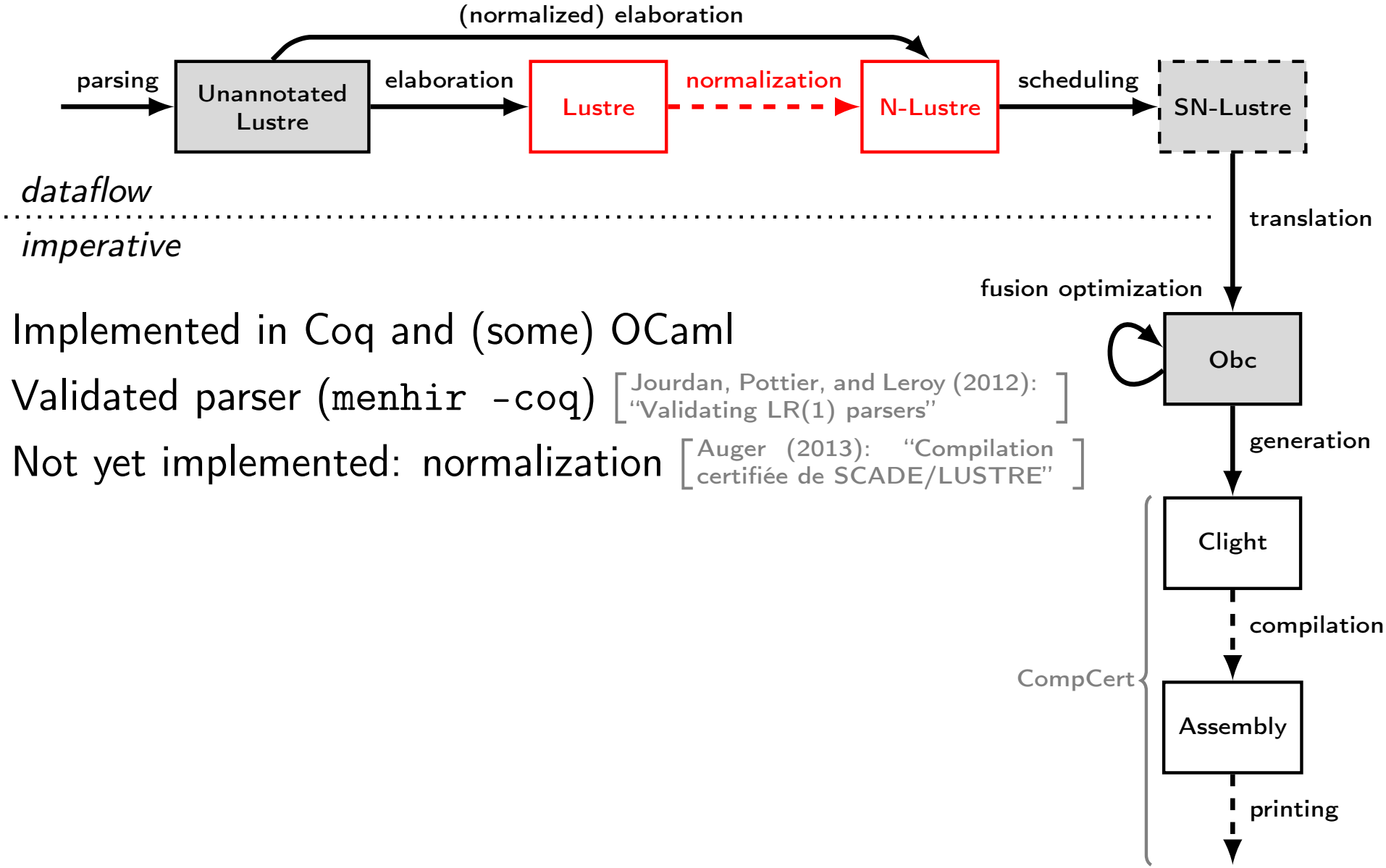
- Implemented in Coq and (some) OCaml

The Vélus Lustre Compiler



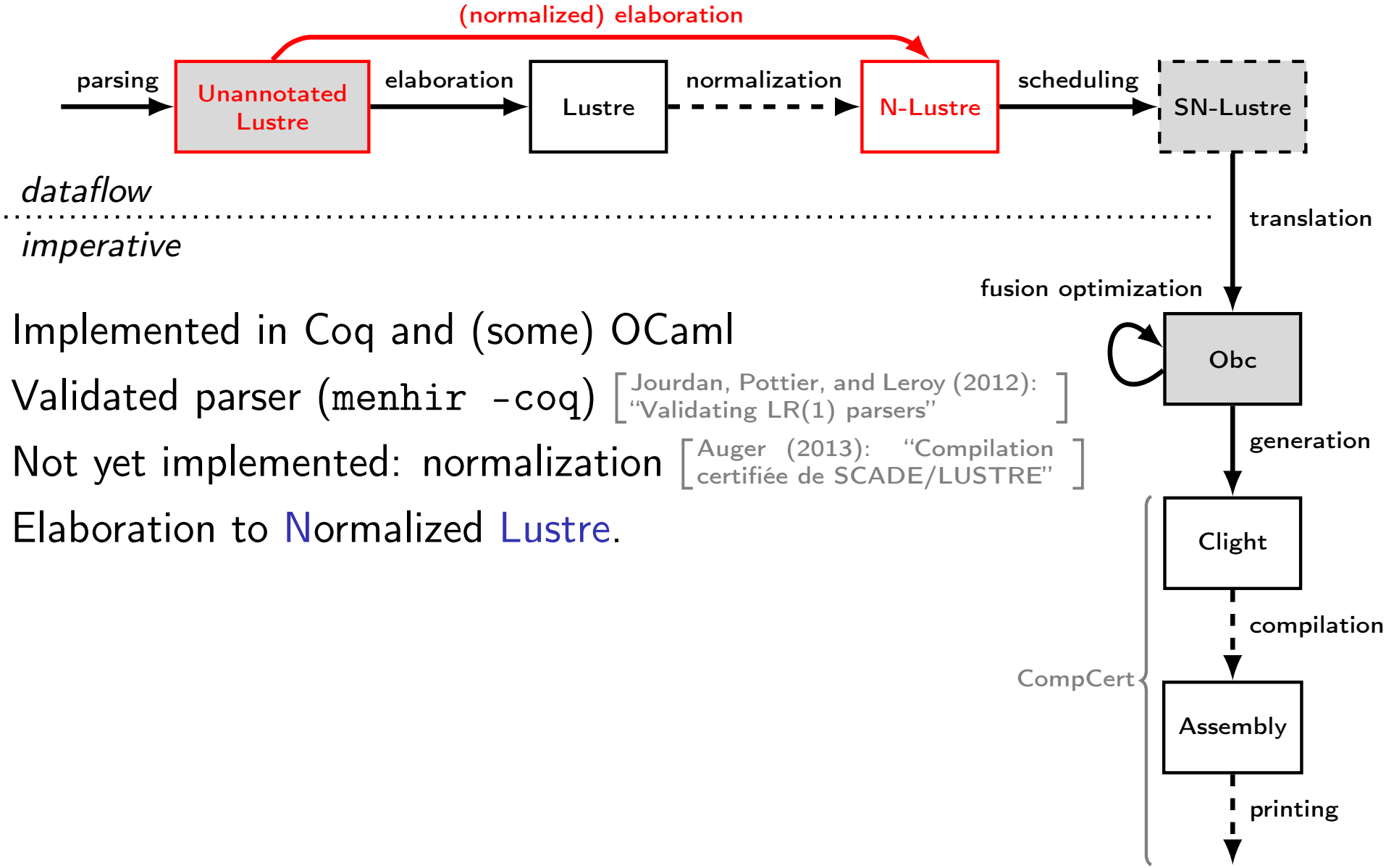
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]

The Vélus Lustre Compiler



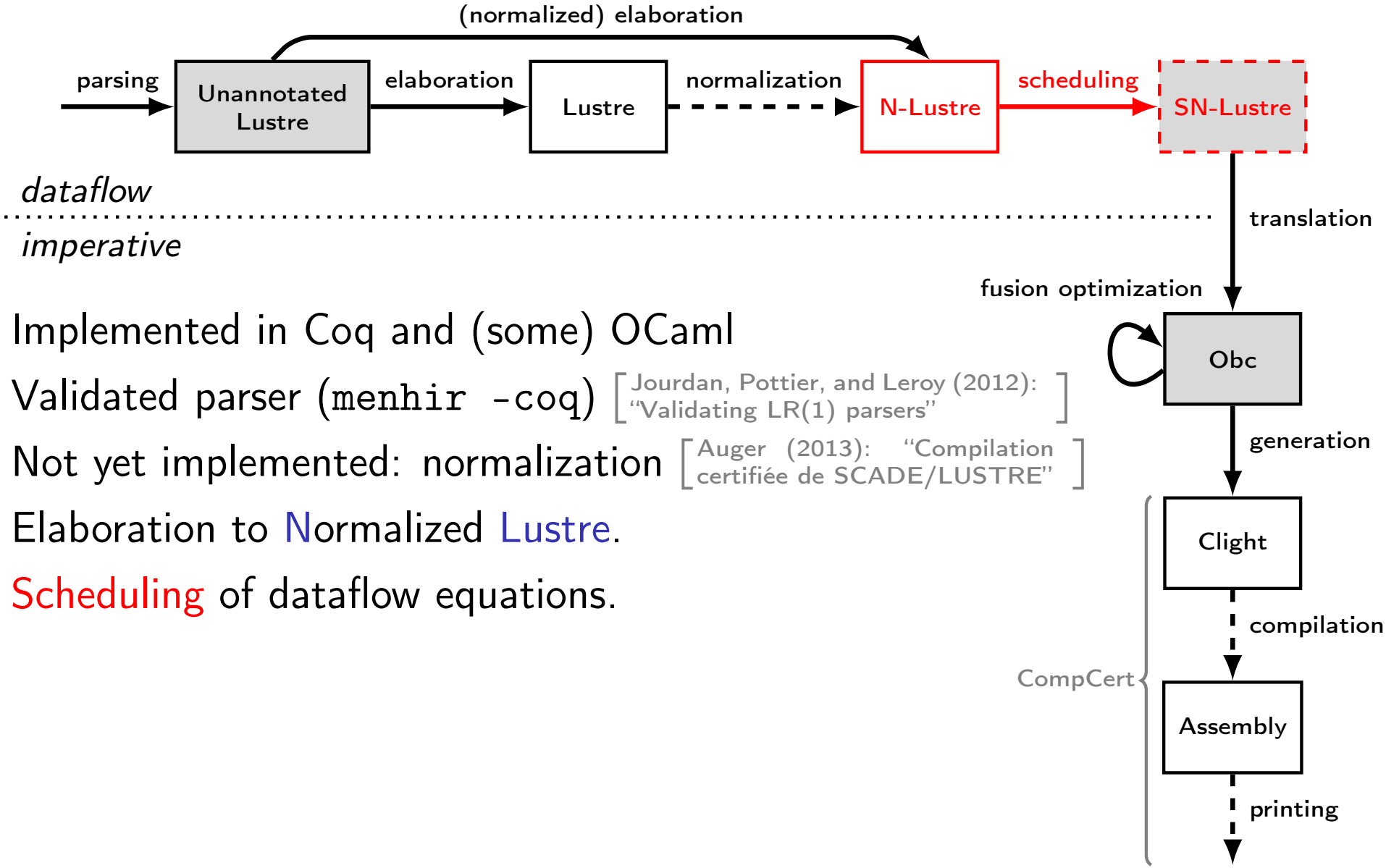
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]

The Vélus Lustre Compiler



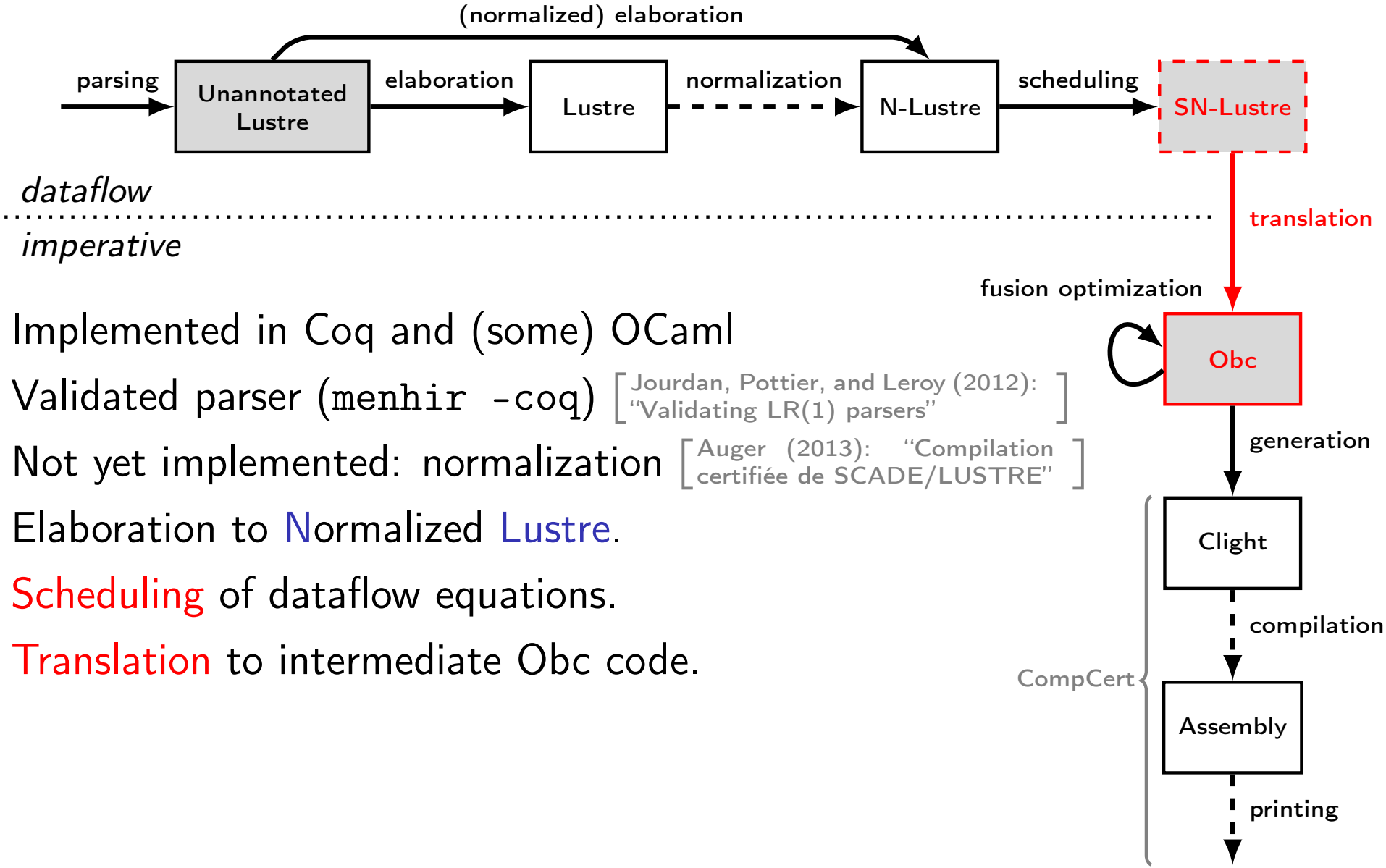
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.

The Vélus Lustre Compiler



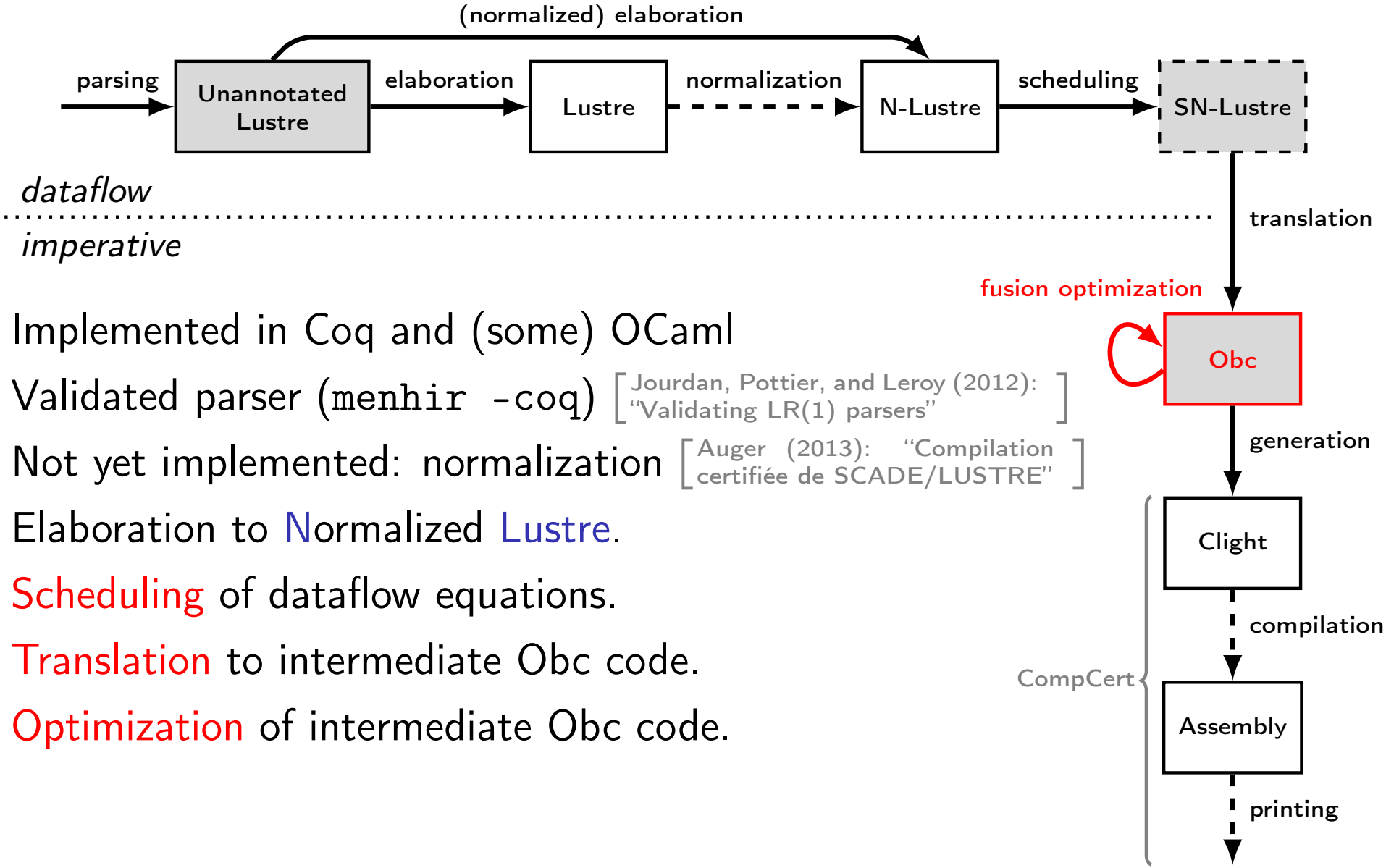
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.

The Vélus Lustre Compiler



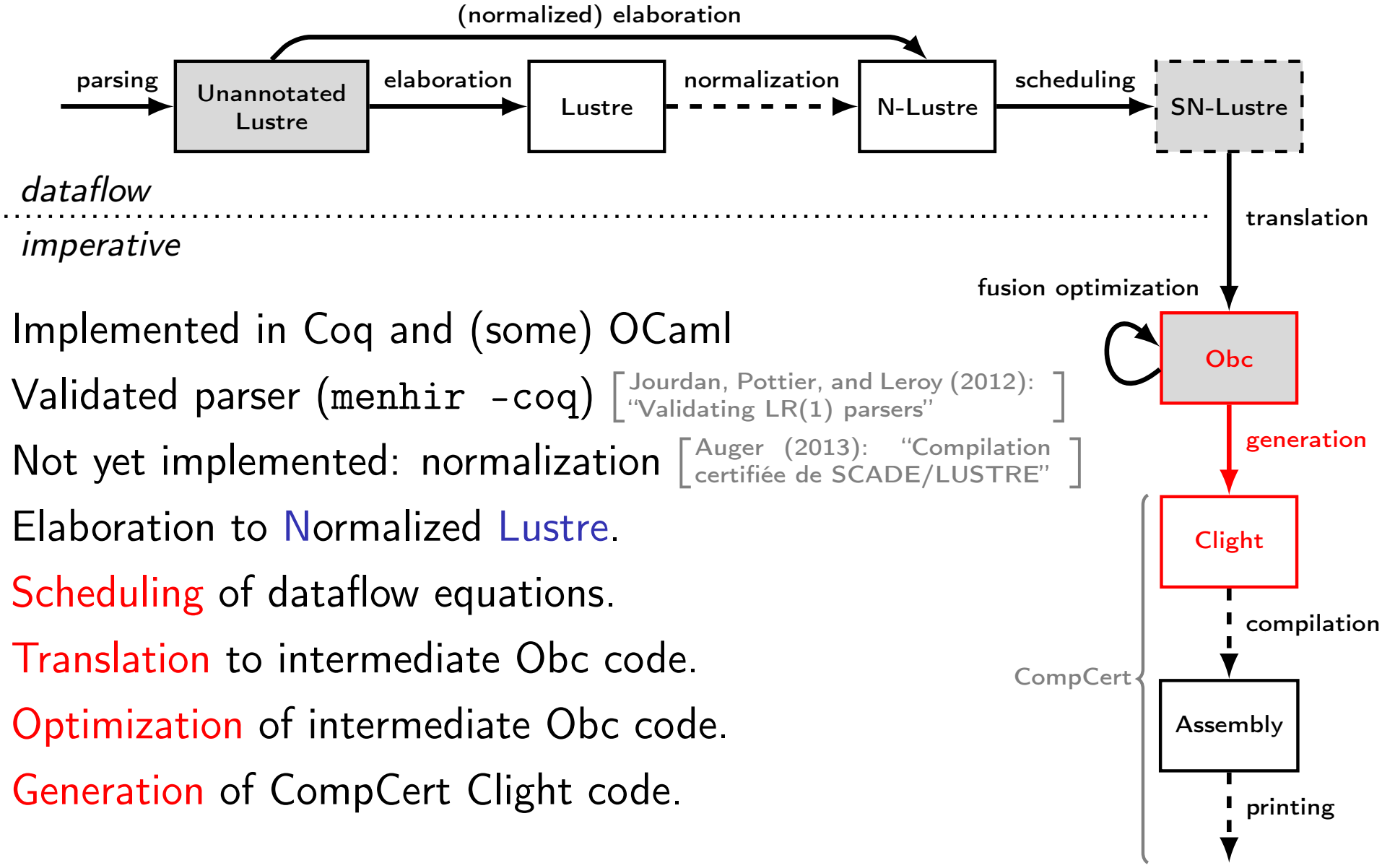
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.
- Translation to intermediate Obc code.

The Vélus Lustre Compiler



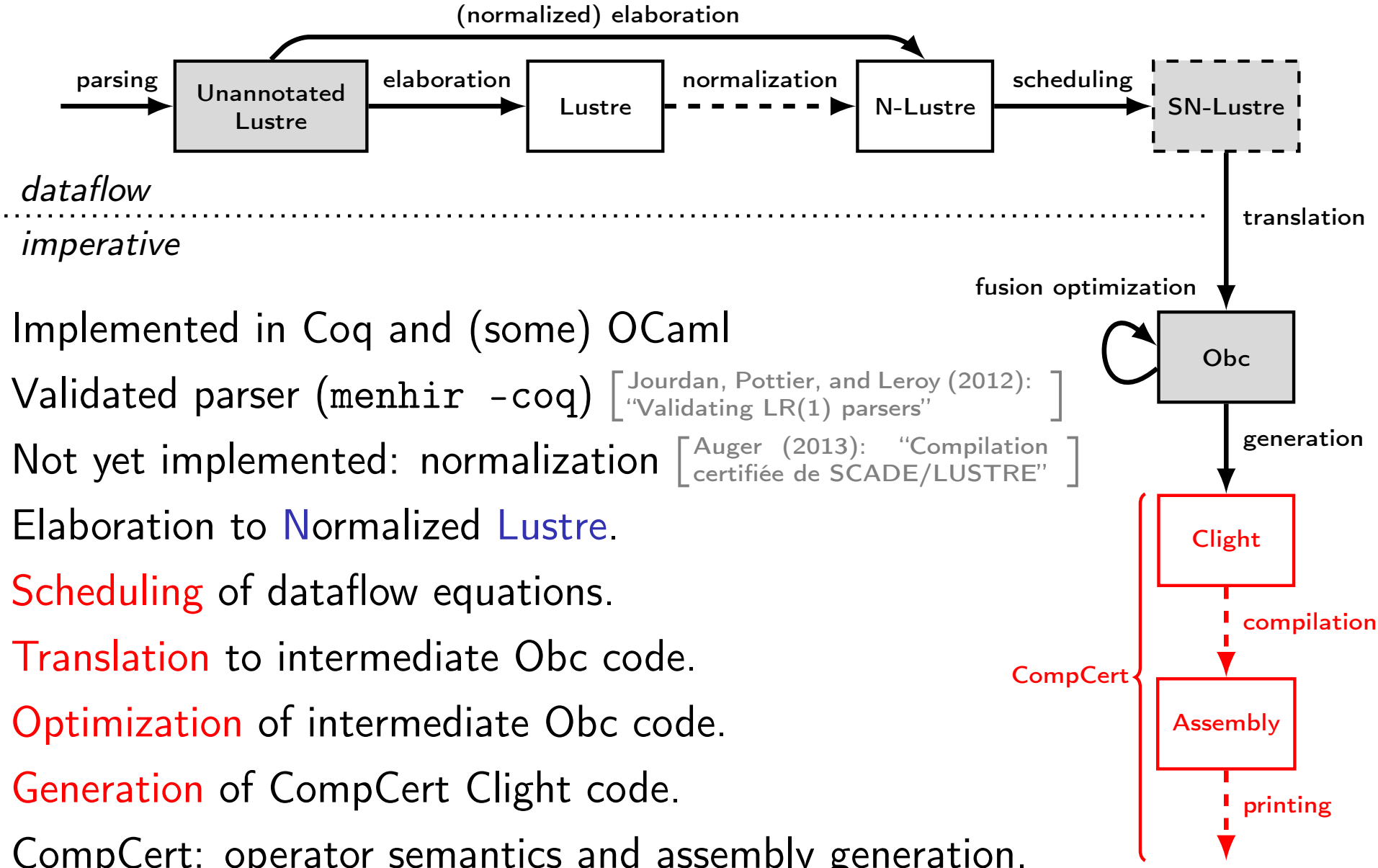
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.
- Translation to intermediate Obc code.
- Optimization of intermediate Obc code.

The Vélus Lustre Compiler



- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.
- Translation to intermediate Obc code.
- Optimization of intermediate Obc code.
- Generation of CompCert Clight code.

The Vélus Lustre Compiler



- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.
- Translation to intermediate Obc code.
- Optimization of intermediate Obc code.
- Generation of CompCert Clight code.
- CompCert: operator semantics and assembly generation.

What is Lustre?

- A language for programming cyclic control software.

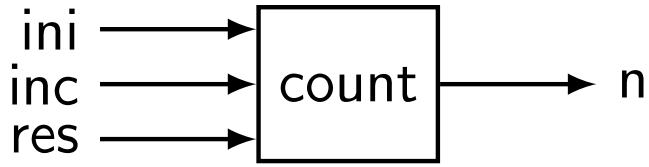
```
every trigger {
  read inputs;
  calculate; // and update internal state
  write outputs;
}
```

- A language for *programming* transition systems
  - ···+ functional abstraction
  - ···+ conditional activations
  - ···+ efficient (modular) compilation

- A restriction of Kahn process networks [Kahn (1974): "The Semantics of a Simple Language for Parallel Programming"], guaranteed to execute in bounded time and space.

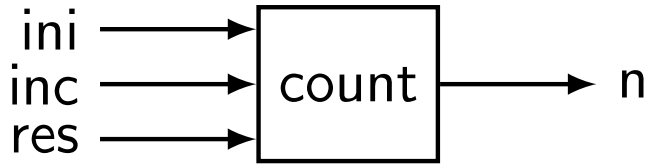
Lustre

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



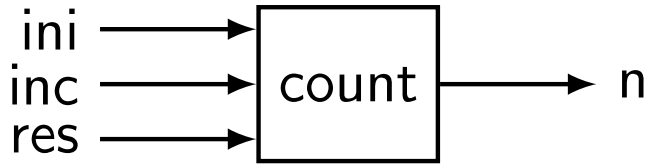
Lustre [Caspi, Pilaud, Halbwachs, and Plaice (1987): "LUSTRE: A declarative language for programming synchronous systems"]

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```



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

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```



ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

- Node: set of causal equations (variables at left).
- Semantic model: synchronized streams of values.
- A node defines a function between input and output streams.

Lustre: syntax and semantics

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

ini	0	0	0	0	0	0	0	⋯
inc	0	1	2	1	2	3	0	⋯
res	F	F	F	F	T	F	F	⋯
true fby false	T	F	F	F	F	F	F	⋯
0 fby n	0	0	1	3	4	0	3	⋯
n	0	1	3	4	0	3	3	⋯

Lustre: syntax and semantics

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

```
Inductive clock : Set :=
| Cbase : clock
| Con   : clock → ident → bool → clock.

Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

Inductive cexp : Type :=
| Emerge : ident → cexp → cexp → cexp
| Eite   : lexp → cexp → cexp → cexp
| Eexp   : lexp → cexp.

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;

  n_defd : Permutation (vars_defined n_eqs)
                       (map fst (n_vars ++ n_out));
  n_nodup : NoDupMembers (n_in ++ n_vars ++ n_out);
  ... }.
```

Lustre: syntax and semantics

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

```
Inductive clock : Set :=
| Cbase : clock
| Con   : clock → ident → bool → clock.

Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

Inductive cexp : Type :=
| Emerge : ident → cexp → cexp → cexp
| Eite   : lexp → cexp → cexp → cexp
| Eexp   : lexp → cexp.

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;

  n_defd : Permutation (vars_defined n_eqs)
                       (map fst (n_vars ++ n_out));
  n_nodup : NoDupMembers (n_in ++ n_vars ++ n_out);
  ... }.
```

```
inductive sem_node (G: list node) : ident →
  stream (list value) → stream (list value) → Prop :=
| SNode:
    find_node f G = Some n →
    clock_of xss bk →
    sem_vars bk H (map fst n.(n_in)) xss →
    sem_vars bk H (map fst n.(n_out)) yss →
    sem_clocked_vars bk H (idck n.(n_in)) →
    Forall (sem_equation G bk H) n.(n_eqs) →
    sem_node G f xss yss.
```

Lustre: syntax and semantics

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

```
Inductive clock : Set :=
| Cbase : clock
| Con   : clock → ident → bool → clock.

Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

Inductive cexp : Type :=
| Emerge : ident → cexp → cexp → cexp
| Eite   : lexp → cexp → cexp → cexp
| Eexp   : lexp → cexp.

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;

  n_defd : Permutation (vars_defined n_eqs)
                       (map fst (n_vars ++ n_out));
  n_nodup : NoDupMembers (n_in ++ n_vars ++ n_out);
  ... }.
```

```
inductive sem_node (G: list node) : ident →
  stream (list value) → stream (list value) → Prop :=
| SNode:
    find_node f G = Some n →
    clock_of xss bk →
    sem_vars bk H (map fst n.(n_in)) xss →
    sem_vars bk H (map fst n.(n_out)) yss →
    sem_clocked_vars bk H (idck n.(n_in)) →
    Forall (sem_equation G bk H) n.(n_eqs) →
    sem_node G f xss yss.
```

sem_node G f xss yss

$f : \mathrm{stream}(T^+) \to \mathrm{stream}(T^+)$

Lustre Compilation: normalization and scheduling

```
node count (ini, inc: int; res: bool)
    returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

Lustre Compilation: normalization and scheduling

```
node count (ini, inc: int; res: bool)
    returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```


```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  f = true fby false;
  c = 0 fby n;
  n = if f or res then ini else c + inc;
tel
```

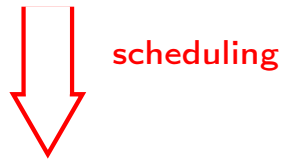
Normalization

- Rewrite to put each fby in its own equation.
- Introduce fresh variables using the substitution principle.

Lustre Compilation: normalization and scheduling

```
node count (ini, inc: int; res: bool)
    returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```


```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  f = true fby false;
  c = 0 fby n;
  n = if f or res then ini else c + inc;
tel
```




```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

Scheduling

- The semantics is independent of equation ordering; but not the correctness of imperative code translation.
- Reorder so that
  - 'Normals' variables are written before being read, . . . and
  - 'fby' variables are read before being written.

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre compilation: translation to imperative code

Biernacki, Colaço, Hamon, and Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

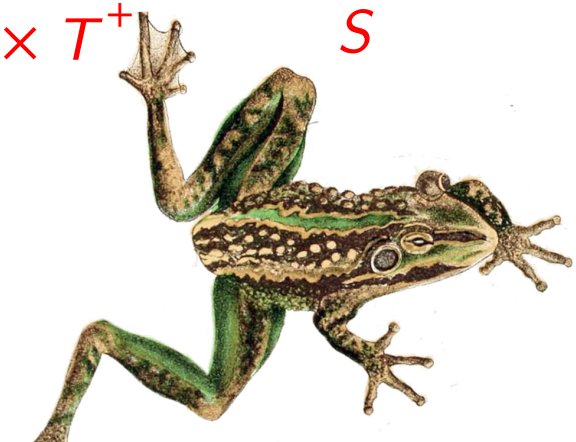
Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

(f_t, s_0)

$S \times T^+ \to S \times T^+$ $\qquad S$



```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯

Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯

Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯

Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯

Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯
0 fby v	0	0	0	0	4	4	4	3	⋯
(0 fby v) when not sec	0	0	0		4			3	⋯

Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯
0 fby v	0	0	0	0	4	4	4	3	⋯
(0 fby v) when not sec	0	0	0		4			3	⋯
v	0	0	0	4	4	4	3	3	⋯

Lustre: instantiation and sampling

Semantic model

- History environment maps identifiers to streams.
- Maps from natural numbers: `Notation stream A := nat → A`
- Model absence: `Inductive value := absent | present v`

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯
0 fby v	0	0	0	0	4	4	4	3	⋯
(0 fby v) when not sec	0	0	0		4			3	⋯
v	0	0	0	4	4	4	3	3	⋯

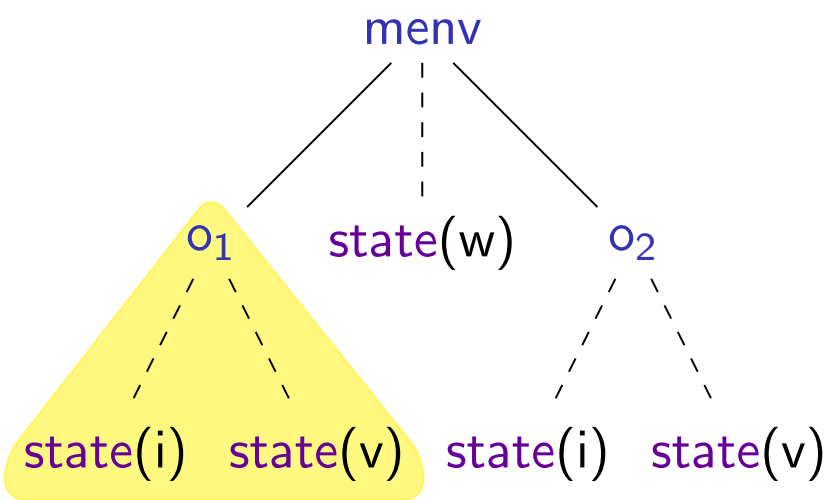
Lustre compilation: translation to clocked imperative code

```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```

```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

Lustre compilation: translation to clocked imperative code

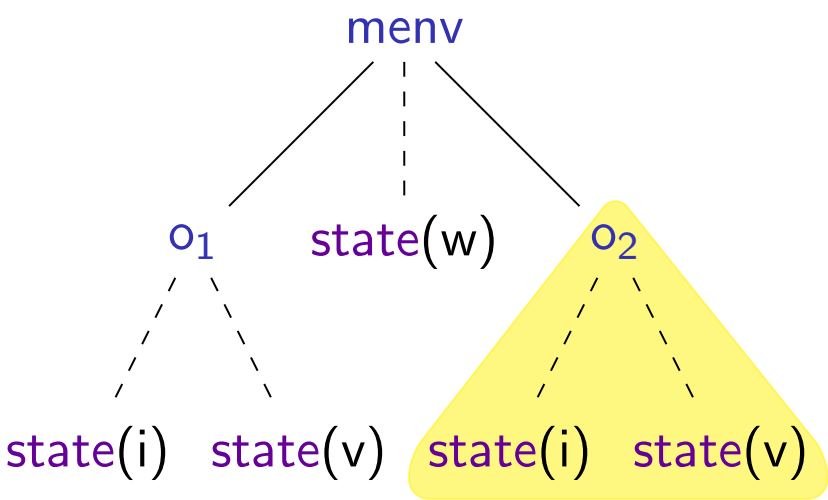
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

Lustre compilation: translation to clocked imperative code

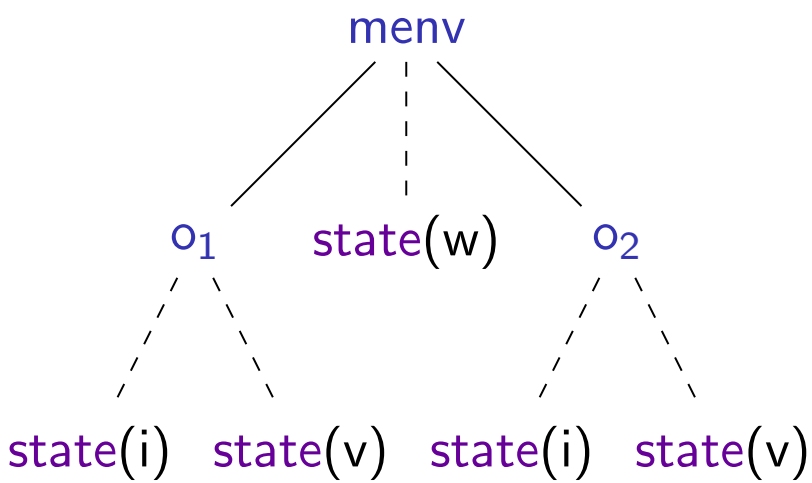
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

Lustre compilation: translation to clocked imperative code

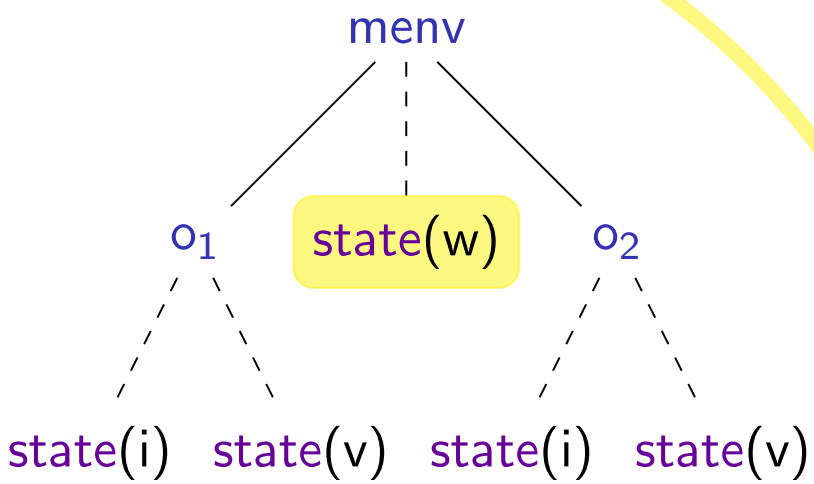
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

Lustre compilation: translation to clocked imperative code

```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

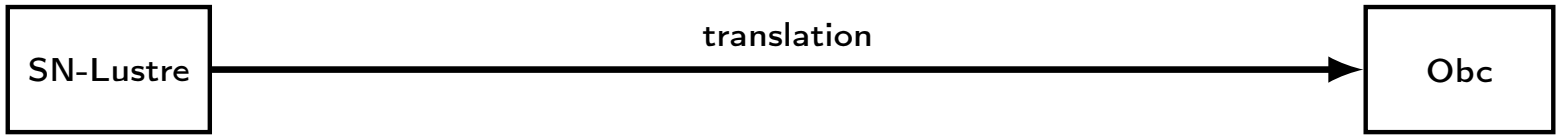
Implementation of translation

- Translation pass: small set of functions on abstract syntax.
- Challenge: going from one semantic model to another.

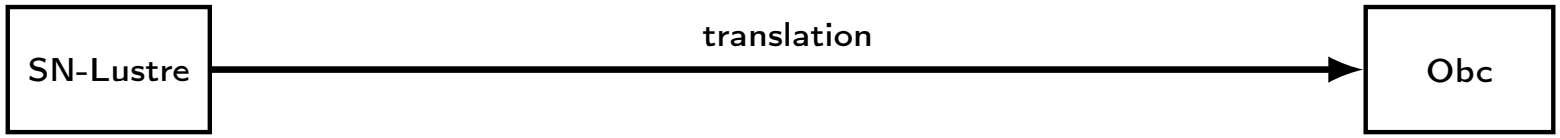
```
Definition tovar (x: ident) : exp :=
  if PS.mem x memories then State x else Var x.

Fixpoint Control (ck: clock) (s: stmt) : stmt :=
  match ck with
  | Cbase ⇒ s
  | Con ck x true  ⇒ Control ck (Ifte (tovar x) s Skip)
  | Con ck x false ⇒ Control ck (Ifte (tovar x) Skip s)
  end.

Fixpoint translate_lexp (e : lexp) : exp :=
  match e with
  | Econst c ⇒ Const c
  | Evar x ⇒ tovar x
  | Ewhen e c x ⇒ translate_lexp e
  | Eop op es ⇒ Op op (map translate_lexp es)
  end.

Fixpoint translate_cexp (x: ident) (e: cexp) : stmt :=
  match e with
  | Emerge y t f ⇒ Ifte (tovar y) (translate_cexp x t)
                                  (translate_cexp x f)
  | Eexp l ⇒ Assign x (translate_lexp l)
  end.

Definition translate_eqn (eqn: equation) : stmt :=
  match eqn with
  | EqDef x ck ce   ⇒ Control ck (translate_cexp x ce)
  | EqApp x ck f les ⇒ Control ck (Step_ap x f x (map translate_lexp les))
  | EqFby x ck v le ⇒ Control ck (AssignSt x (translate_lexp le))
  end.
```

```
Definition translate_eqns (eqns: list equation) : stmt :=
  fold_left (fun i eq ⇒ Comp (translate_eqn eq) i) eqns Skip.

Definition translate_reset_eqn (s: stmt) (eqn: equation) : stmt :=
  match eqn with
  | EqDef _ _ _ ⇒ s
  | EqFby x _ v0 _ ⇒ Comp (AssignSt x (Const v0)) s
  | EqApp x _ f _ ⇒ Comp (Reset_ap f x) s
  end.

Definition translate_reset_eqns (eqns: list equation): stmt :=
  fold_left translate_reset_eqn eqns Skip.

Definition ps_from_list (l: list ident) : PS.t :=
  fold_left (fun s i⇒PS.add i s) l PS.empty.

Definition translate_node (n: node): class :=
  let names := gather_eqs n.(n_eqs) in
  let mems := ps_from_list (fst names) in
  mk_class n.(n_name) n.(n_input) n.(n_output)
           (fst names) (snd names)
           (translate_eqns mems n.(n_eqs))
           (translate_reset_eqns n.(n_eqs)).

Definition translate (G: global) : program := map translate_node G.
```

Correctness of translation

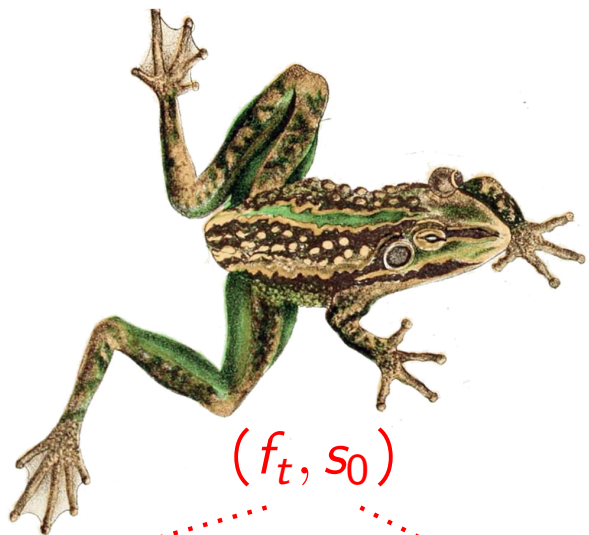


Correctness of translation



`sem_node G f xss yss`

$\mathsf{stream}(T^+) \to \mathsf{stream}(T^+)$

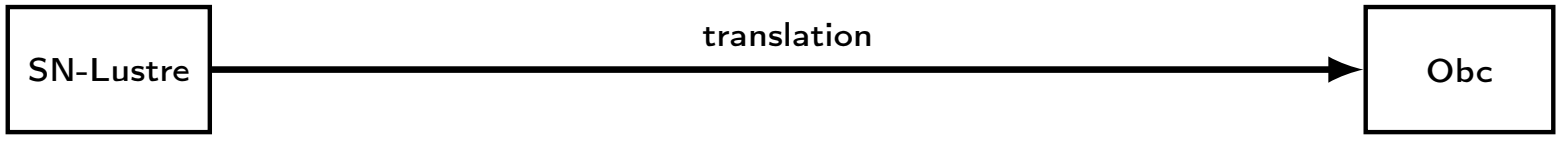


(f_t, s_0)

$S \times T^+ \to T^+ \times S$

S

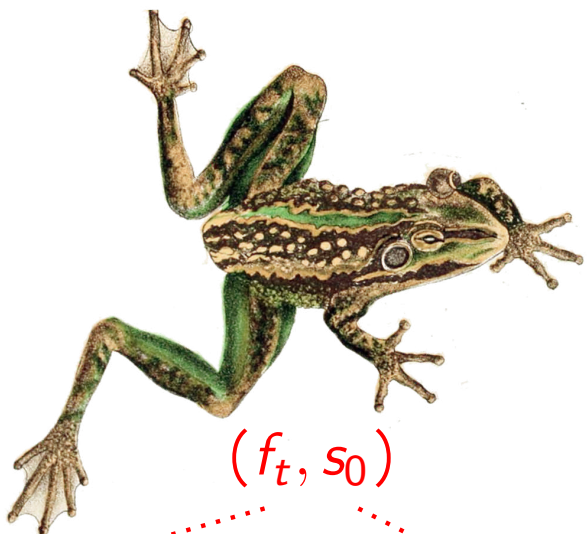
Correctness of translation



too weak for a direct proof by induction ✗

`sem_node G f xss yss`

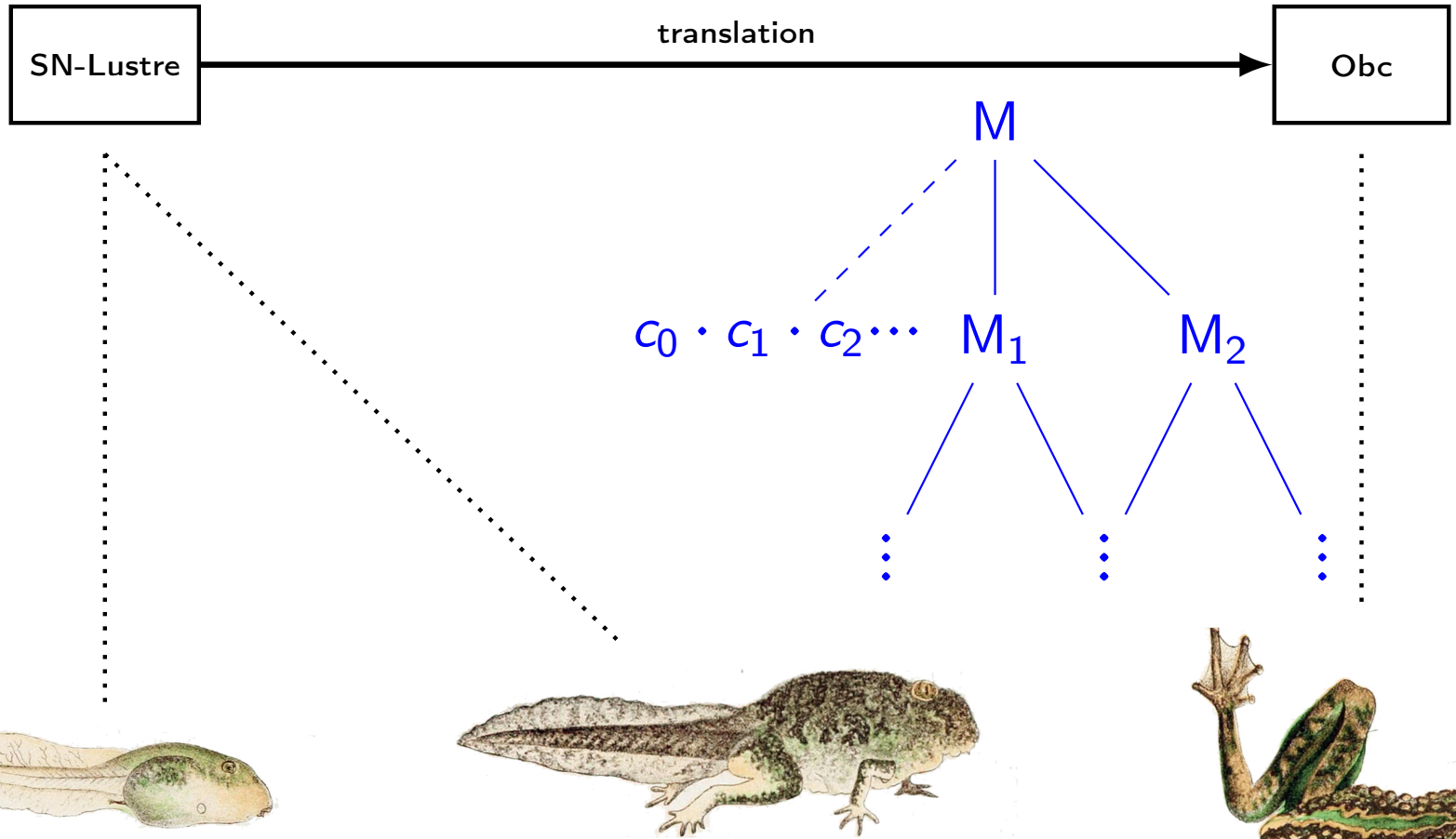
$\mathsf{stream}(T^+) \to \mathsf{stream}(T^+)$



(f_t, s_0)

$S \times T^+ \to T^+ \times S$ $\qquad$ S

Correctness of translation



`sem_node G f xss yss`

`msem_node G f xss M yss`

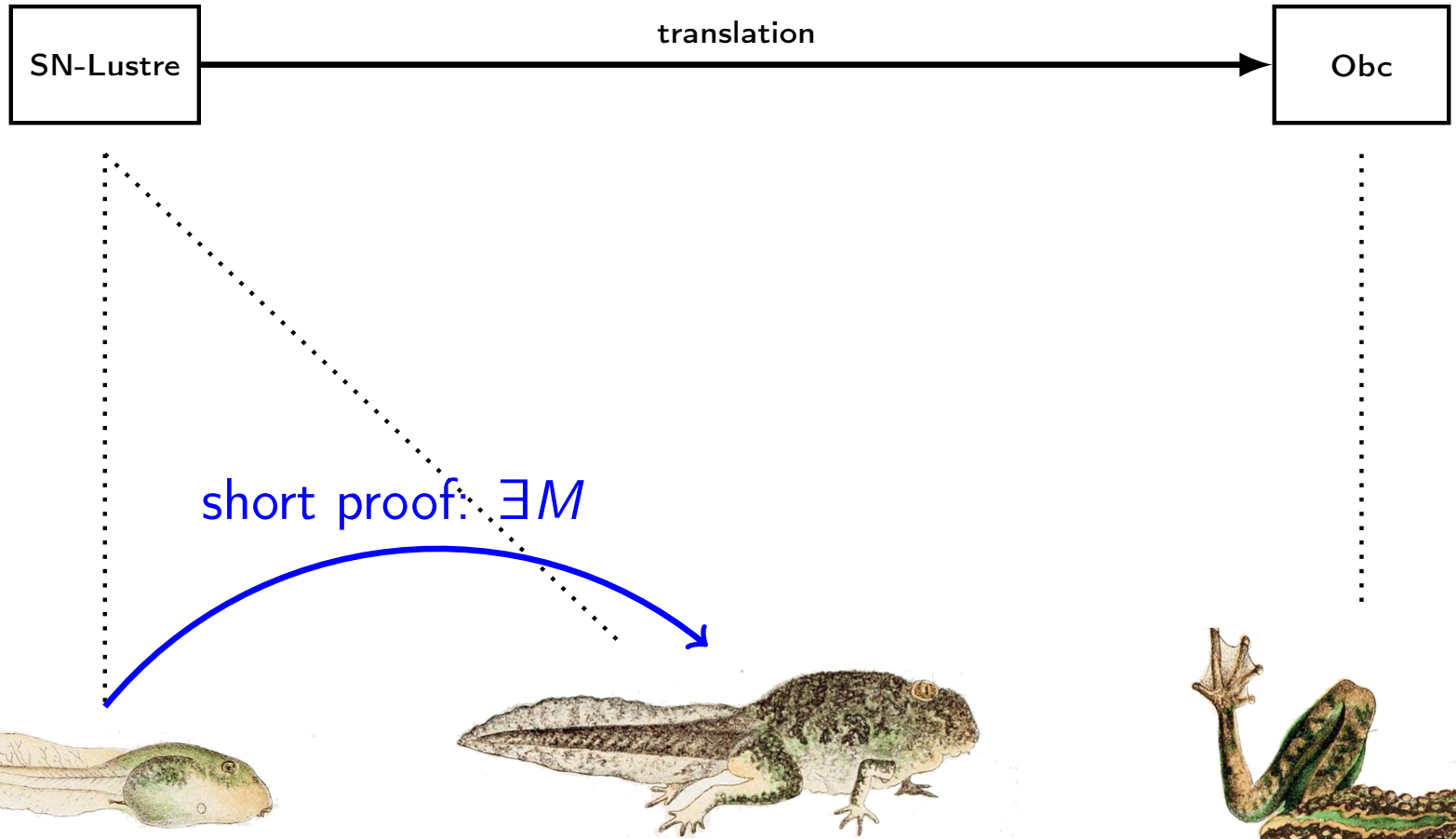
$\mathsf{stream}(T^+) \to \mathsf{stream}(T^+)$

(f_t, s_0)

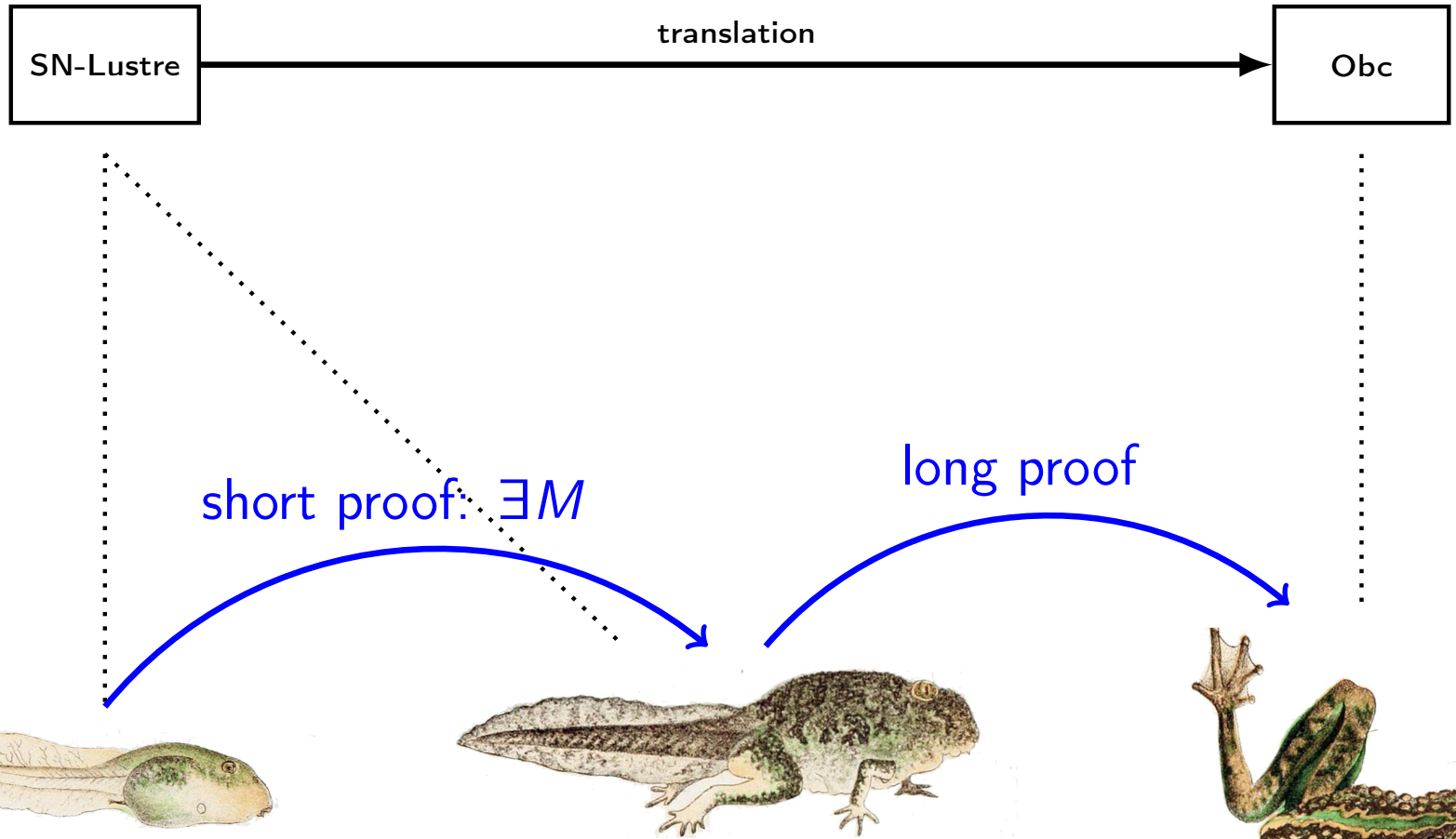
$S \times T^+ \to T^+ \times S$

S

Correctness of translation



Correctness of translation



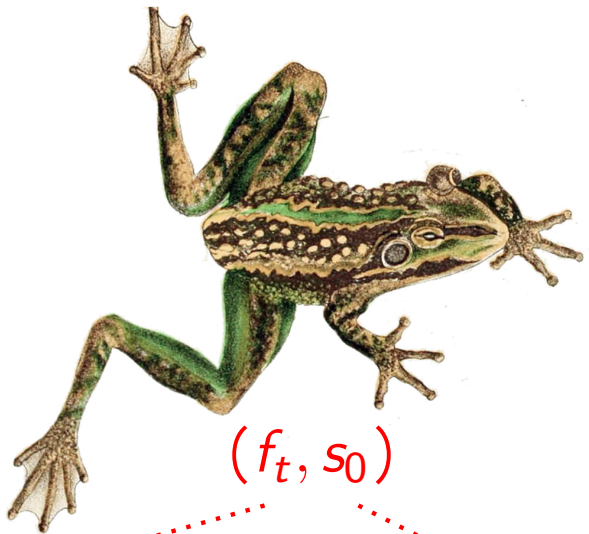
Correctness of translation

- Tricky proof full of technical details.
- ≈100 lemmas
- Several iterations to find the right definitions.
- The intermediate model is central.

induction n
- induction G
  - induction eqs
    - case: $x = (ce)^{ck}$
      - case: present
      - case: absent
    - case: $x = (f\ e)^{ck}$
      - case: present
      - case: absent
    - case: $x = (k\ \texttt{fby}\ e)^{ck}$
      - case: present
      - case: absent

long proof

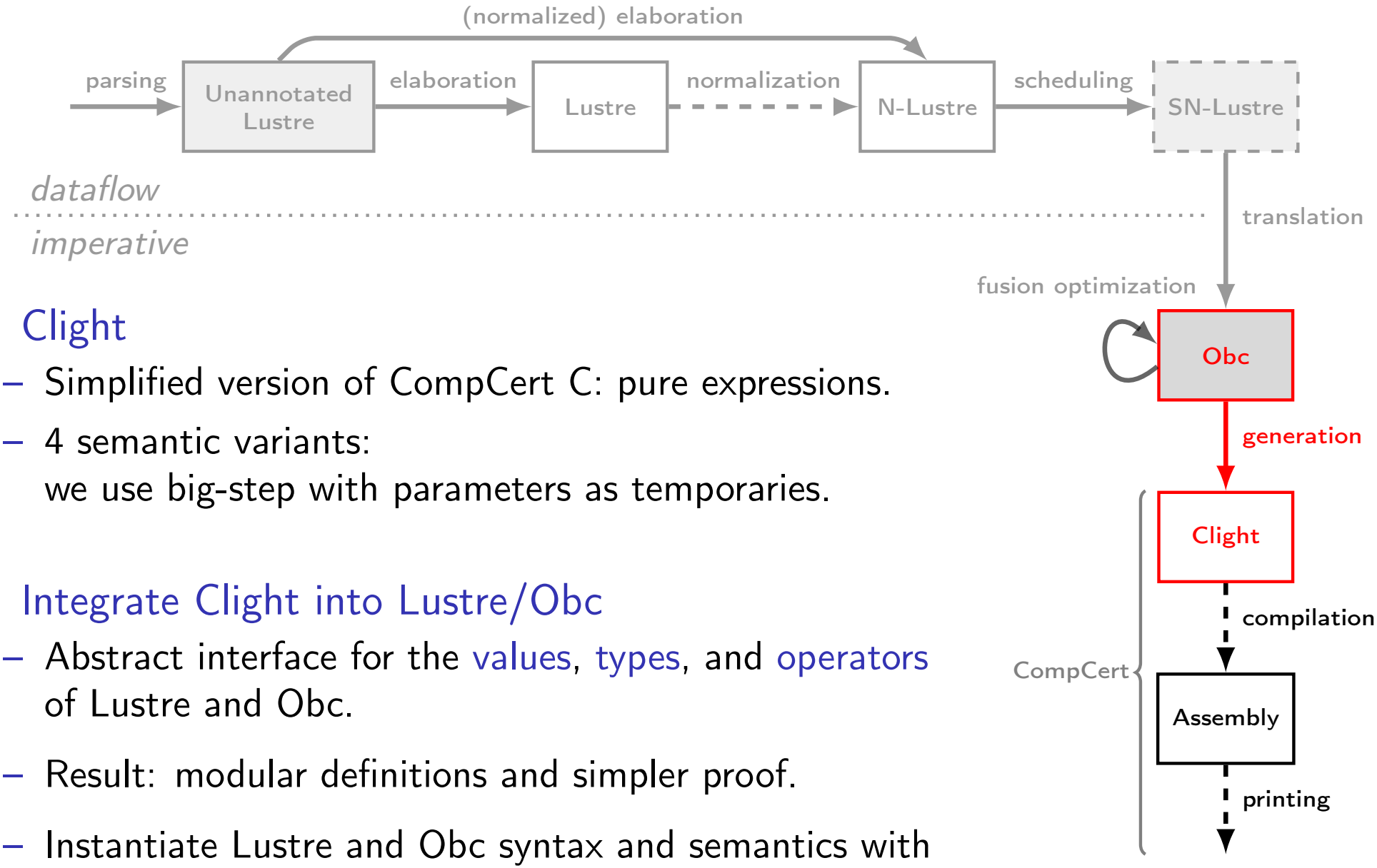
`f xss M yss`



(f_t, s_0)

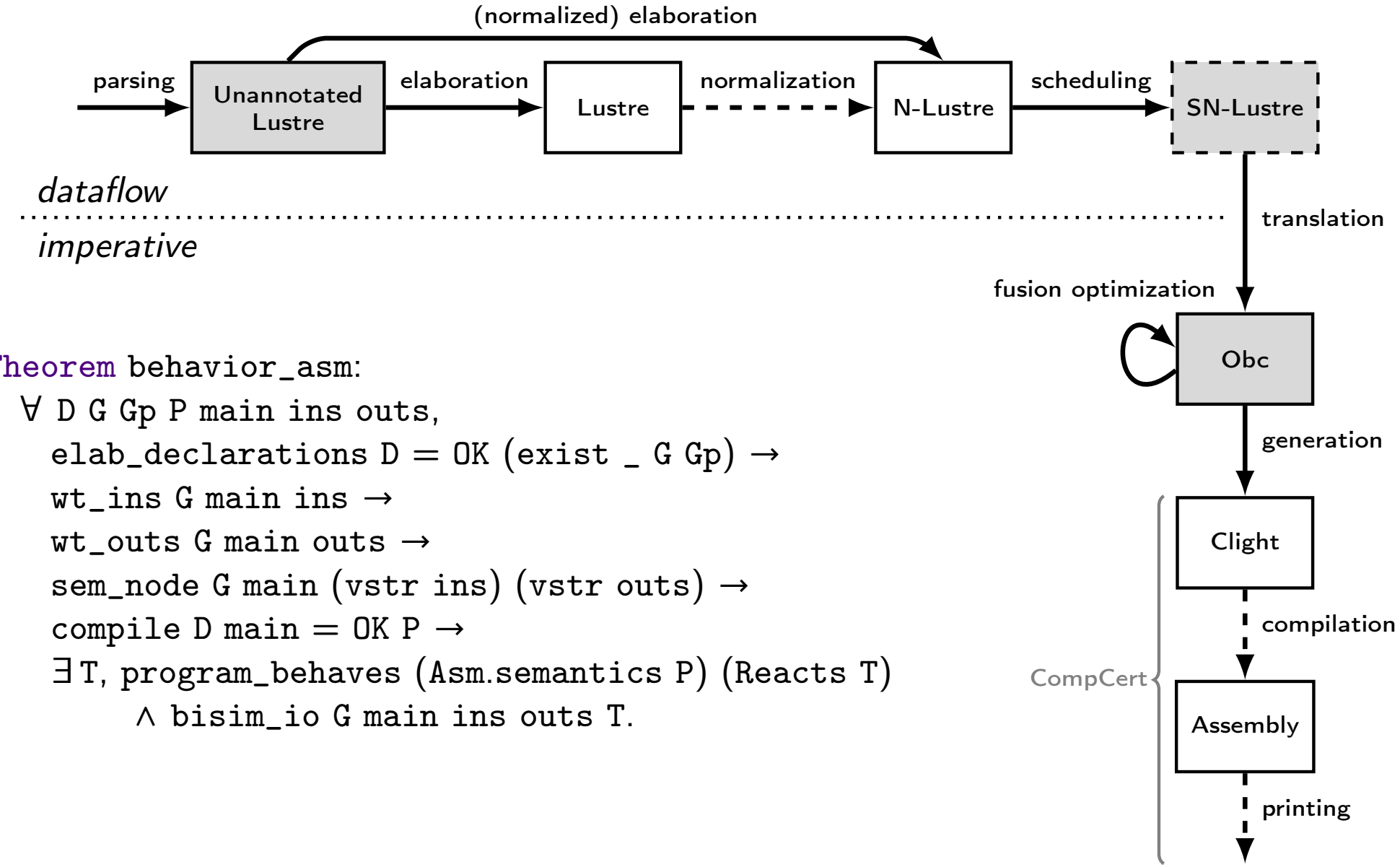
$S \times T^+ \to T^+ \times S$ S

Generation: Obc to Clight

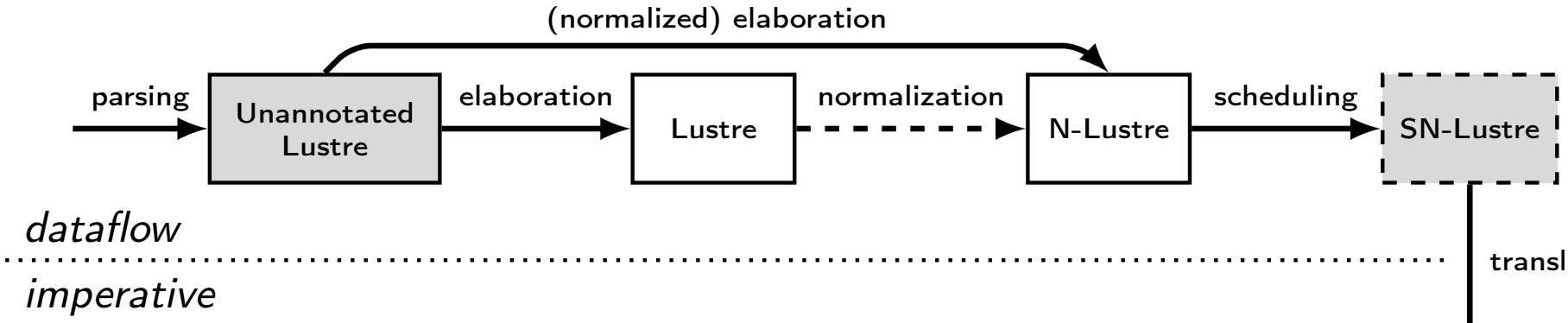


- Clight
  - Simplified version of CompCert C: pure expressions.
  - 4 semantic variants:
    we use big-step with parameters as temporaries.

- Integrate Clight into Lustre/Obc
  - Abstract interface for the values, types, and operators of Lustre and Obc.
  - Result: modular definitions and simpler proof.
  - Instantiate Lustre and Obc syntax and semantics with CompCert definitions.

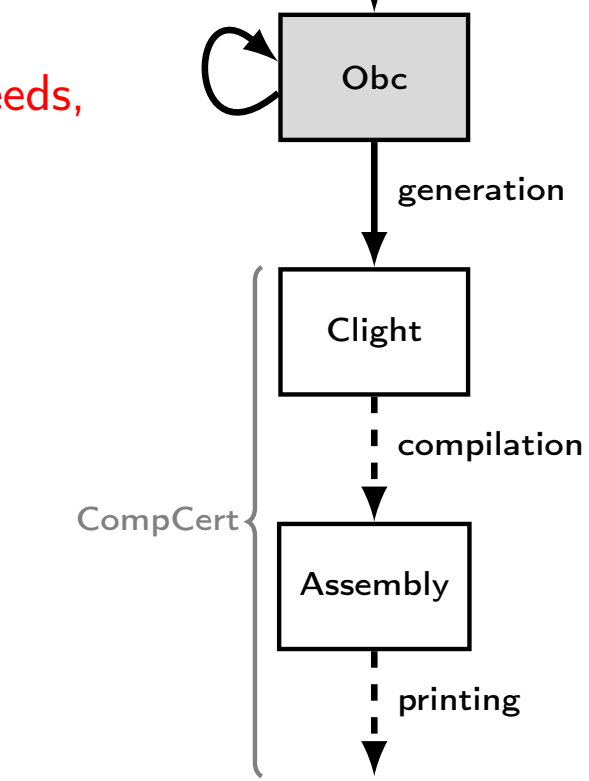


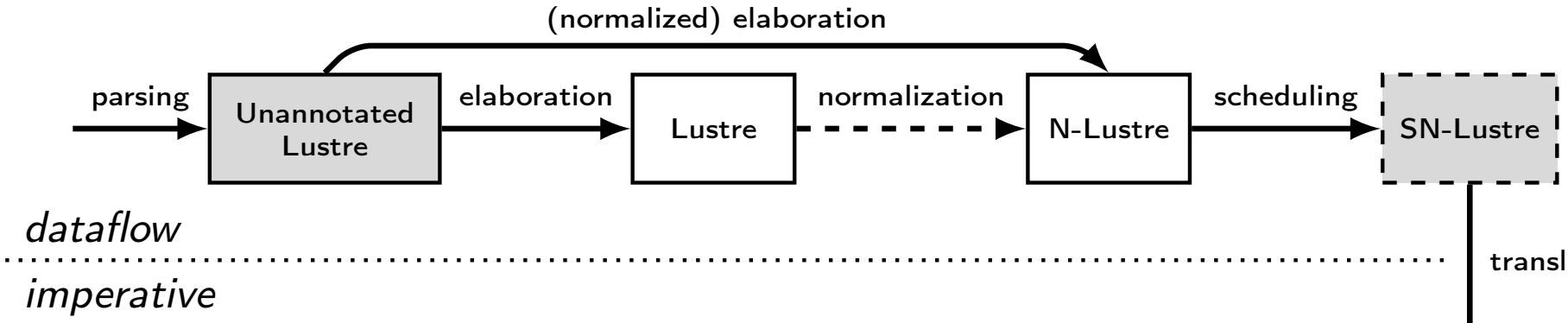

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```




```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

typing/elaboration succeeds,

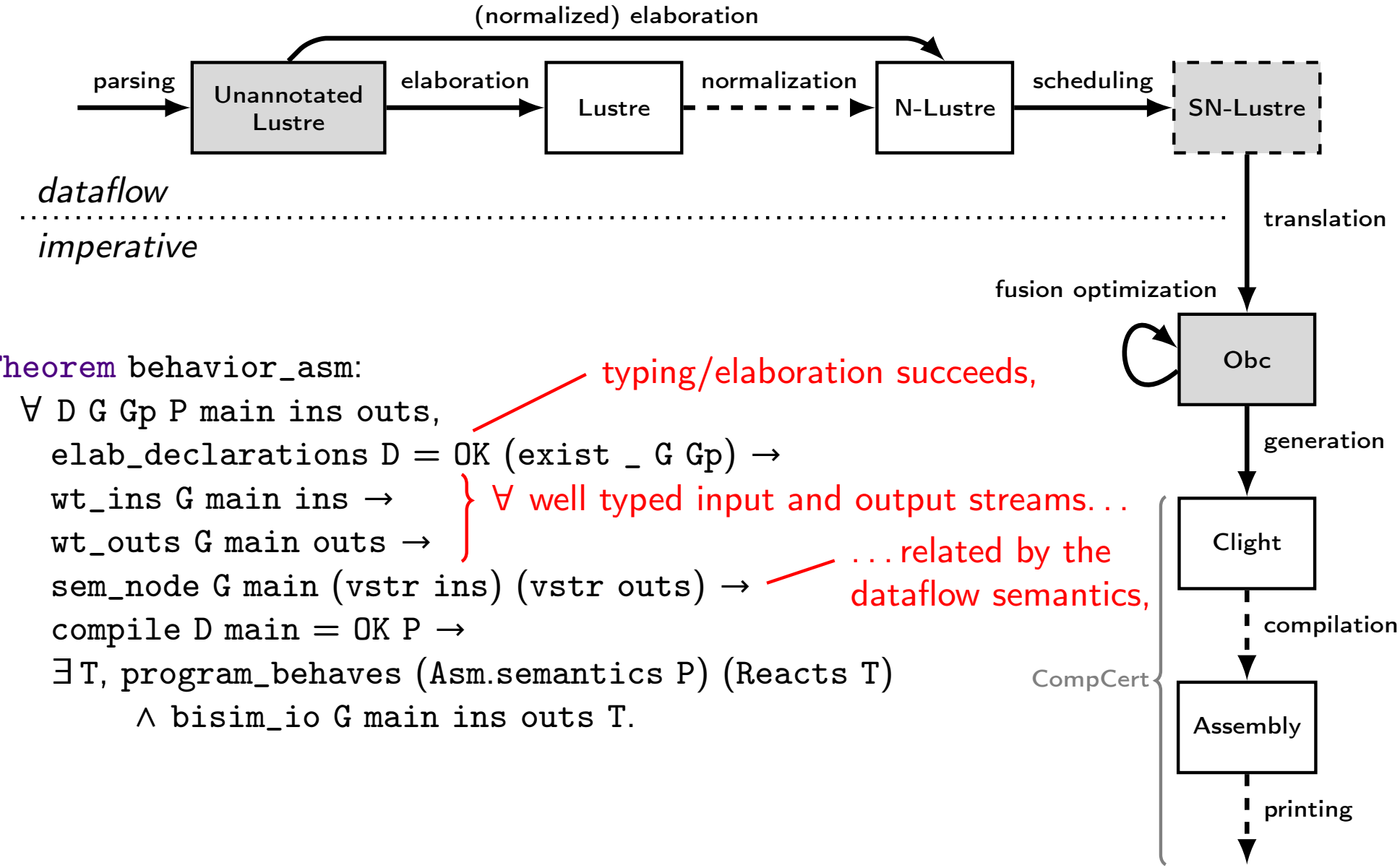





```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

typing/elaboration succeeds,

∀ well typed input and output streams. . .

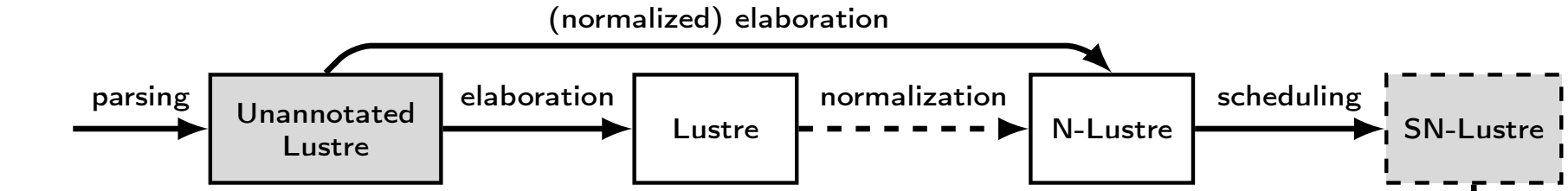



```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

typing/elaboration succeeds,

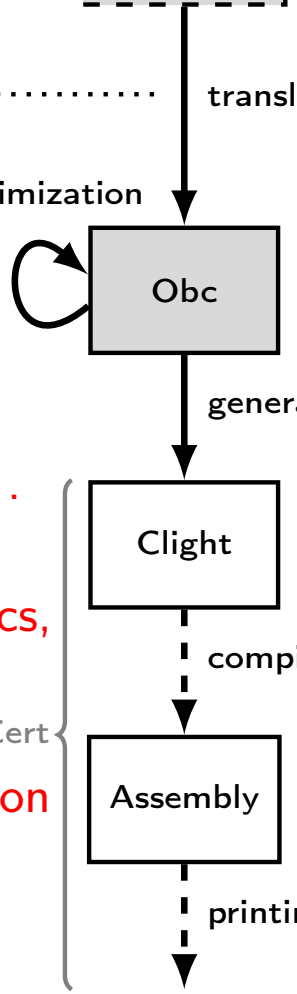
∀ well typed input and output streams. . .

. . . related by the dataflow semantics,



dataflow

imperative



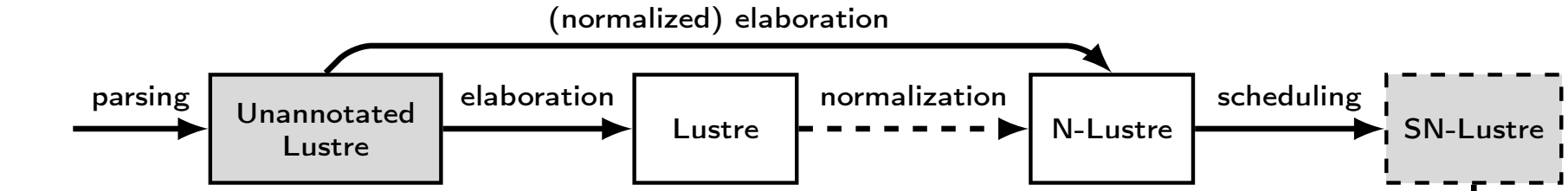

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

typing/elaboration succeeds,

∀ well typed input and output streams. . .

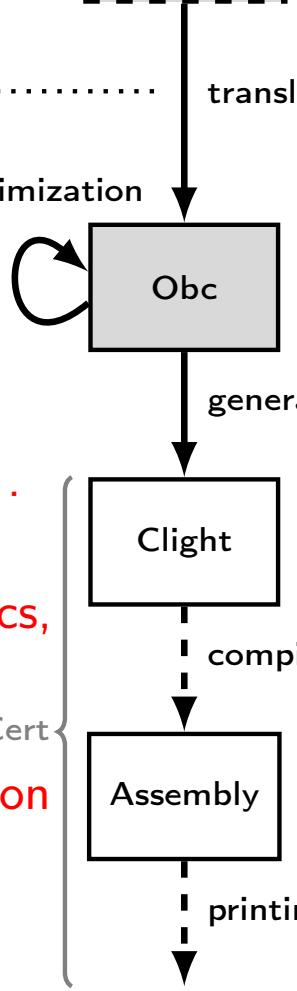
. . . related by the dataflow semantics,

if compilation succeeds,



dataflow

imperative




```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

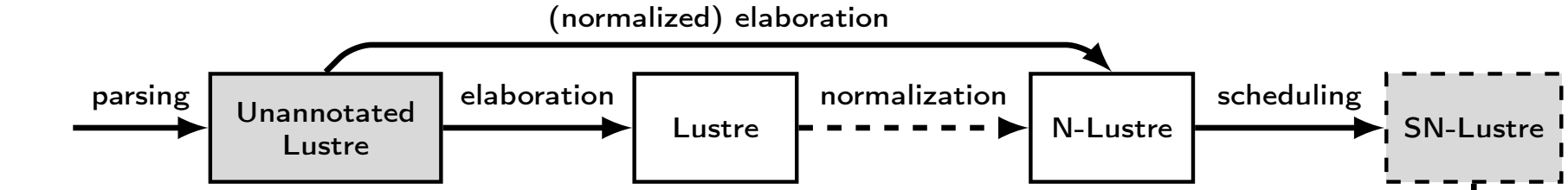
typing/elaboration succeeds,

∀ well typed input and output streams...

...related by the dataflow semantics,

if compilation succeeds,

then, the generated assembly produces an infinite trace...

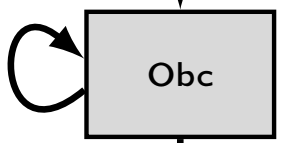


dataflow

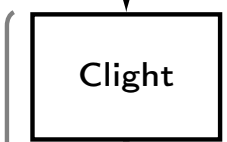
imperative

translation

fusion optimization

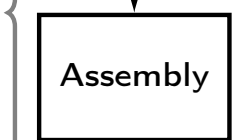


generation



compilation

CompCert



printing

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

typing/elaboration succeeds,

∀ well typed input and output streams...

...related by the dataflow semantics,

if compilation succeeds,

then, the generated assembly produces an infinite trace...

...that corresponds to the dataflow model.

Work in Progress. . .

1. Semantics of (not-yet-normalized) Lustre
2. Clocking of node arguments
3. Treatment of modular reset (Lélio Brun's thesis topic)

NLustre: syntax

```
Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.
```

le when x / le when not x

```
Inductive cexp : Type :=
| Emerge : ident → cexp → cexp → cexp
| Eite   : lexp → cexp → cexp → cexp
| Eexp   : lexp → cexp.
```

merge x ce1 ce2

```
Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : list ident → clock → ident → list lexp → equation
| EqFby : ident → clock → const → lexp → equation.
```

x = ce

y,...,y = f(le, ..., le)

x = c fby le

```
Record node : Type :=
  mk_node {
    n_name : ident;
    n_in   : list (ident * (type * clock));
    n_out  : list (ident * (type * clock));
    n_vars : list (ident * (type * clock));
    n_eqs  : list equation;

    n_ingt0 : 0 < length n_in;
    n_outgt0 : 0 < length n_out;
    n_defd  : Permutation (vars_defined n_eqs)
              (map fst (n_vars ++ n_out));
    n_vout  : ∀ out, In out (map fst n_out) →
              ¬ In out (vars_defined (filter is_fby n_eqs));
    n_nodup : NoDupMembers (n_in ++ n_vars ++ n_out);
    n_good  : Forall NotReserved (n_in ++ n_vars ++ n_out)
  }.
```

node f(x,...,x) returns (y,...,y);
var w, ..., w;
let x = ...; ... tel

Lustre: syntax

```
Definition ann : Type := (type * nclock).
Definition lann : Type := (list type * nclock).

Inductive exp : Type :=
| Econst : const → exp
| Evar   : ident → ann → exp
| Eunop  : unop → exp → ann → exp
| Ebinop : binop → exp → exp → ann → exp

| Ewhen  : list exp → ident → bool → lann → exp
| Emerge : ident → list exp → list exp → lann → exp
| Eite   : exp → list exp → list exp → lann → exp
| Efby   : list exp → list exp → list ann → exp

| Eapp   : ident → list exp → list ann → exp.

Definition equation : Type := (list ident * list exp).

Record node : Type :=
  mk_node {
    n_name     : ident;
    n_in       : list (ident * (type * clock));
    n_out      : list (ident * (type * clock));
    n_vars     : list (ident * (type * clock));
    n_eqs      : list equation;

    n_ingt0    : 0 < length n_in;
    n_outgt0   : 0 < length n_out;
    n_defd     : Permutation (vars_defined n_eqs)
                             (map fst (n_vars ++ n_out));
    n_nodup    : NoDupMembers (n_in ++ n_vars ++ n_out);
    n_good     : Forall NotReserved (n_in ++ n_vars ++ n_out)
  }.
```

```
(* No tuples. `Lists of flows' are flattened: *)
node shuffle (a, b, c, d : bool)
returns (w, x, y, z : bool);

(w, x, y, z) = shuffle(((a, (b, (c)), d)));
```

(e,...,e) when x / (e,...,e) when not x

merge x (e,...,e) (e,...,e)

(e,...,e) fby (e,...,e)

f(e, ..., e)

x,...,x = e,...,e

node f(x,...,x) returns (y,...,y);
var w, ..., w;
let x = ...; ... tel

NLustre: semantics 1/3

```
Notation stream A := (nat → A).

Definition history := PM.t (stream value).
Definition R := PM.t value.

Section InstantSemantics.

  Variable base : bool.
  Variable R : R.

  Inductive sem_var_instant (x: ident) (v: value): Prop :=
  | Sv:
      PM.find x R = Some v →
      sem_var_instant x v.

End InstantSemantics.

Section LiftSemantics.

  Variable bk : stream bool.

  Definition restr H (n: nat): R :=
    PM.map (fun xs ⇒ xs n) H.

  Definition lift (sem: bool → R → A → B → Prop) H x (ys: stream B): Prop :=
    ∀ n, sem (bk n) (restr H n) x (ys n).

  Definition sem_var H (x: ident)(xs: stream value): Prop :=
    lift (fun base ⇒ sem_var_instant) H x xs.

End LiftSemantics.
```

NLustre: semantics 2/3

```
Inductive sem_lexp_instant: lexp → value → Prop :=
| Sconst:
    v = (if base then present (sem_const c) else absent) →
    sem_lexp_instant (Econst c) v

| Svar:
    sem_var_instant x v →
    sem_lexp_instant (Evar x ty) v

| Swhen_abs:
    sem_lexp_instant s absent →
    sem_var_instant x absent →
    sem_lexp_instant (Ewhen s x b) absent

| Swhen_eq:
    sem_lexp_instant s (present sc) →
    sem_var_instant x (present xc) →
    val_to_bool xc = Some b →
    sem_lexp_instant (Ewhen s x b) (present sc)

| Swhen_abs1:
    sem_lexp_instant s (present sc) →
    sem_var_instant x (present xc) →
    val_to_bool xc = Some b →
    sem_lexp_instant (Ewhen s x (negb b)) absent

...

Definition sem_lexp H (e: lexp) (xs: stream value) : Prop :=
  lift sem_lexp_instant H e xs.
```

$$
\begin{aligned}
i^{\#}[\epsilon] &= \epsilon \\
i^{\#}[true.cl] &= v.i^{\#}[cl] \\
i^{\#}[false.cl] &= abs.i^{\#}[cl]
\end{aligned}
$$

$$
\begin{aligned}
\texttt{when}^{\#}(s_1, s_2) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\texttt{when}^{\#}(abs.xs, abs.cs) &= abs.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, true.cs) &= x.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, false.cs) &= abs.\texttt{when}^{\#}(xs, cs)
\end{aligned}
$$

[Colaço and Pouzet (2003): "Clocks as First Class Abstract Types"]

NLustre: semantics 3/3

```
Inductive sem_equation G : stream bool → history → equation → Prop :=
| SEqDef:
    sem_var bk H x xs →
    sem_caexp bk H ck ce xs →
    sem_equation G bk H (EqDef x ck ce)

| SEqApp:
    sem_lexps bk H arg ls →
    sem_vars bk H x xs →
    sem_clock bk H ck cks →
    clock_of ls cks →
    sem_node G f ls xs →
    sem_equation G bk H (EqApp x ck f arg)

| SEqFby:
    sem_laexp bk H ck le ls →
    sem_var bk H x xs →
    (∀ n, xs n = fby (sem_const c0) ls n) →
    sem_equation G bk H (EqFby x ck c0 le)

with sem_node G :
  ident → stream (list value) → stream (list value)
    → Prop :=
| SNode:
    find_node f G = Some n →
    clock_of xss bk →
    sem_vars bk H (map fst n.(n_in)) xss →
    sem_vars bk H (map fst n.(n_out)) yss →
    sem_clocked_vars bk H (idck n.(n_in)) →
    Forall (sem_equation G bk H) n.(n_eqs) →
    sem_node G f xss yss.
```

```
Inductive sem_laexp_instant
    : clock → lexp → value → Prop:=
| SLtick:
    sem_lexp_instant ce (present c) →
    sem_clock_instant ck true →
    sem_laexp_instant ck ce (present c)
| SLabs:
    sem_clock_instant ck false →
    sem_laexp_instant ck ce absent.
```

```
Fixpoint hold
    (v0: val) (xs: stream value) (n: nat) : val :=
  match n with
  | 0 ⇒ v0
  | S m ⇒ match xs m with
          | absent ⇒ hold v0 xs m
          | present hv ⇒ hv
          end
  end.

Definition fby
    (v0: val) (xs: stream value) (n: nat) : value :=
  match xs n with
  | absent ⇒ absent
  | _ ⇒ present (hold v0 xs n)
  end.
```

Lustre: semantics 1/3

```
Definition history := PM.t (Stream value).

Notation sem_var H := (fun (x: ident) (s: Stream value) ⇒ PM.MapsTo x s H).

Inductive sem_exp : history → Stream bool → exp → list (Stream value) → Prop :=
| Sconst:
    sem_exp H b (Econst c) [const c b]

| Svar:
    sem_var H x s →
    sem_exp H b (Evar x ann) [s]

| Swhen:
    Forall2 (sem_exp H b) es ss →
    sem_var H x s →
    Forall2 (when k s) (concat ss) os →
    sem_exp H b (Ewhen es x k lann) os

| ...
```

```
CoFixpoint const (c: const) (b: Stream bool) : Stream value :=
  match b with
  | true  ::: b' ⇒ present (sem_const c) ::: const c b'
  | false ::: b' ⇒ absent ::: const c b'
  end.
```

$$
\begin{aligned}
i^{\#}[\epsilon] &= \epsilon \\
i^{\#}[true.cl] &= v.i^{\#}[cl] \\
i^{\#}[false.cl] &= abs.i^{\#}[cl]
\end{aligned}
$$

```
CoInductive when (k: bool)
   : Stream value → Stream value → Stream value → Prop :=
| WhenA:
    when k xs cs rs →
    when k (absent ::: cs) (absent ::: xs) (absent ::: rs)

| WhenPA:
    when k xs cs rs →
    val_to_bool c = Some (negb k) →
    when k (present c ::: cs) (present x ::: xs) (absent ::: rs)

| WhenPP:
    when k xs cs rs →
    val_to_bool c = Some k →
    when k (present c ::: cs) (present x ::: xs) (present x ::: rs)
```

$$
\begin{aligned}
\texttt{when}^{\#}(s_1, s_2) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\texttt{when}^{\#}(abs.xs, abs.cs) &= abs.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, true.cs) &= x.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, false.cs) &= abs.\texttt{when}^{\#}(xs, cs)
\end{aligned}
$$

Lustre: semantics 2/3

```
Inductive sem_exp : history → Stream bool → exp → list (Stream value) → Prop :=
| Sconst:
    sem_exp H b (Econst c) [const c b]

| ...

| Sfby:
    Forall2 (sem_exp H b) e0s s0ss →
    Forall2 (sem_exp H b) es sss →
    Forall3 fby (concat s0ss) (concat sss) os →
    sem_exp H b (Efby e0s es anns) os
```

$$
\begin{aligned}
\texttt{fby}^{\#}(\epsilon, ys) &= \epsilon \\
\texttt{fby}^{\#}(abs.xs, abs.ys) &= abs.\texttt{fby}^{\#}(xs, ys) \\
\texttt{fby}^{\#}(x.xs, y.ys) &= x.\texttt{fby1}^{\#}(y, xs, ys) \\
\texttt{fby1}^{\#}(v, \epsilon, ys) &= \epsilon \\
\texttt{fby1}^{\#}(v, abs.xs, abs.ys) &= abs.\texttt{fby1}^{\#}(v, xs, ys) \\
\texttt{fby1}^{\#}(v, w.xs, s.ys) &= v.\texttt{fby1}^{\#}(s, xs, ys)
\end{aligned}
$$

```
CoInductive fby
| FbyA:
      fby xs ys rs →
      fby (absent ::: xs) (absent ::: ys) (absent ::: rs)

| FbyP:
      fby1 y xs ys rs →
      fby (present x ::: xs) (present y ::: ys) (present x ::: rs).

CoInductive fby1
| Fby1A:
      fby1 v xs ys rs →
      fby1 v (absent ::: xs) (absent ::: ys) (absent ::: rs)

| Fby1P:
      fby1 s xs ys rs →
      fby1 v (present w ::: xs) (present s ::: ys) (present v ::: rs).
```

Lustre: semantics 3/3

```
Inductive sem_exp : history → Stream bool → exp → list (Stream value) → Prop :=
| Sconst:
    sem_exp H b (Econst c) [const c b]

| ...

| Sapp:
    Forall2 (sem_exp H b) es ss →
    sem_node f (concat ss) os →
    sem_exp H b (Eapp f es lann) os

| ...

with sem_equation: history → Stream bool → equation → Prop :=
| Seq:
    Forall2 (sem_exp H b) es ss →
    Forall2 (sem_var H) xs (concat ss) →
    sem_equation H b (xs, es)

with sem_node: ident → list (Stream value) → list (Stream value) → Prop :=
| Snode:
    find_node f G = Some n →
    Forall2 (sem_var H) (idents n.(n_in)) ss →
    Forall2 (sem_var H) (idents n.(n_out)) os →
    Forall (sem_equation H b) n.(n_eqs) →
    b = sclocksof ss →
    sem_node f ss os.
```

```
CoFixpoint sclocksof (ss: list (Stream value)) : Stream bool :=
  existsb (fun s⇒ hd s <>b absent) ss ::: sclocksof (List.map tl ss).
```

Prior work on Lustre semantics in Coq

- [Boulmé and Hamon (2001): *A clocked denotational semantics for Lucid-Synchrone in Coq*]
  - Shallow embedding of Lucid Synchrone into Coq.
  - Embed the clocking rules into the Coq type system.
  - Use clocks (boolean streams) to control rhythms.
  - Denotational semantics using co-fixpoints.
  - Values: present, absent, and *fail*.

Prior work on Lustre semantics in Coq

- [Boulmé and Hamon (2001): *A clocked denotational semantics for Lucid-Synchrone in Coq*]
  - Shallow embedding of Lucid Synchrone into Coq.
  - Embed the clocking rules into the Coq type system.
  - Use clocks (boolean streams) to control rhythms.
  - Denotational semantics using co-fixpoints.
  - Values: present, absent, and *fail*.

- [Paulin-Mohring (2009): "A constructive denotational semantics for Kahn networks in Coq"]
  - Denotational semantics of Kahn process networks.
  - ```
    CoInductive Str (A:Type) : Type :=
    | Eps: StrA → Str A
    | cons: A → StrA → Str A
    ```
  - Least element: $\texttt{Eps}^{\infty}$
  - Shallow embedding of programs.
    ```

Prior work on Lustre semantics in Coq

- [Boulmé and Hamon (2001): *A clocked denotational semantics for Lucid-Synchrone in Coq*]
  - Shallow embedding of Lucid Synchrone into Coq.
  - Embed the clocking rules into the Coq type system.
  - Use clocks (boolean streams) to control rhythms.
  - Denotational semantics using co-fixpoints.
  - Values: present, absent, and *fail*.

- [Paulin-Mohring (2009): "A constructive denotational semantics for Kahn networks in Coq"]
  - Denotational semantics of Kahn process networks.
  - ```
    CoInductive Str (A:Type) : Type :=
    | Eps: StrA → Str A
    | cons: A → StrA → Str A
    ```
  - Least element: $\texttt{Eps}^{\infty}$
  - Shallow embedding of programs.

- [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
  - Streams as (backward) finite lists.
  - Deep embedding of programs.
  - Relational semantics linking programs to lists.
    ```

Semantic model: design constraints

- Judging the (informal) 'correctness' of the development.
- Proving properties of programs.
- Proving correctness of compilation.
- Proving properties of the language
  - Existence of semantics
  - Determinism of language
  - Correctness of the clocking system

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

1. w :: α on u

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

1. w :: α on u
2. g returns (β on x)

approximate base clock: $\beta = \alpha$

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

1. w :: α on u
2. g returns (β on x)

   approximate base clock: $\beta = \alpha$
3. g(y :: β on x on z , z :: β on x , x :: β)

   | f | u

   approximate input mapping: $\{x \mapsto u\}$

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

1. w :: α on u
2. g returns (β on x)
   approximate base clock: $\beta = \alpha$
3. g(y :: β on x on z , z :: β on x , x :: β)
   | f | u
   approximate input mapping: $\{x \mapsto u\}$
4. *instantiated*: g(α on u on ?, α on u, α)

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

1. w :: α on u
2. g returns (β on x)
   approximate base clock: $\beta = \alpha$
3. g(y :: β on x on z , z :: β on x , x :: β)
   | f | u
   approximate input mapping: $\{x \mapsto u\}$
4. *instantiated*: g(α on u on ?, α on u, α)
5. f returns (1 : γ on 2 , 2 : γ)
   approximate base clock: $\gamma = \alpha$ on u

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

1. w :: α on u
2. g returns (β on x)
   approximate base clock: $\beta = \alpha$
3. g(y :: β on x on z , z :: β on x , x :: β)
   | f | u
   approximate input mapping: $\{x \mapsto u\}$
4. *instantiated*: g(α on u on ?, α on u, α)
5. f returns (1 : γ on 2 , 2 : γ)
   approximate base clock: $\gamma = \alpha$ on u
6. f(a) *approx. input map*: {}
   true

Inferring whens

```
node f(a: bool) returns (b: bool when c; c: bool);
let
  c = false fby (not c);
  b = a when c;
tel

node g(y: bool when z; z: bool when x; x: bool) returns (o: bool when x)
let
  o = false fby (not z);
tel

node h(u: bool) returns (v: bool);
var w: bool when u;
let
  w = g(f(true), u);
  v = merge u w false;
tel
```

w = g(f(true when u), u)

1. w :: α on u
2. g returns (β on x) — *approximate base clock*: $\beta = \alpha$
3. g(y :: β on x on z , z :: β on x , x :: β) — f — u — *approximate input mapping*: $\{x \mapsto u\}$
4. *instantiated*: g(α on u on ?, α on u, α)
5. f returns (1 : γ on 2 , 2 : γ) — *approximate base clock*: $\gamma = \alpha$ on u
6. f(a) *approx. input map*: {} — true
7. *instantiated:* f(α on u)

Conclusion

First results

- Working compiler from Lustre to assembler in Coq.
- Formally relate dataflow model to imperative code.
- Generate Clight for CompCert; change to richer memory model.
- Intermediate language and separation predicates were decisive.

Ongoing work

- Finish normalization pass, add resets, add automata...
- Prove that a well-typed program has a semantics.
- Combine interactive and automatic proof to verify Lustre programs.
  - Can verify reactive models in Isabelle. [Bourke, Glabbeek, and Höfner (2016): "Mechanizing a Process Algebra for Network Protocols"]
  - Can compile reactive programs in Coq.
  - What's the best way to do both at the same time?
- Treat side-effects in dataflow model and integrate C code.

References I

- Auger, C. (2013). "Compilation certifiée de SCADE/LUSTRE". PhD thesis. Orsay, France: Univ. Paris Sud 11.
- Auger, C., J.-L. Colaço, G. Hamon, and M. Pouzet (2013). "A Formalization and Proof of a Modular Lustre Code Generator". Draft.
- Biernacki, D., J.-L. Colaço, G. Hamon, and M. Pouzet (2008). "Clock-directed modular code generation for synchronous data-flow languages". In: *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. ACM. Tucson, AZ, USA: ACM Press, pp. 121–130.
- Blazy, S., Z. Dargaye, and X. Leroy (2006). "Formal Verification of a C Compiler Front-End". In: *Proc. 14th Int. Symp. Formal Methods (FM 2006)*. Vol. 4085. Lecture Notes in Comp. Sci. Hamilton, Canada: Springer, pp. 460–475.
- Boulmé, S. and G. Hamon (2001). *A clocked denotational semantics for Lucid-Synchrone in Coq*. Tech. rep. LIP6.
- Bourke, T., R. J. van Glabbeek, and P. Höfner (2016). "Mechanizing a Process Algebra for Network Protocols". In: *J. Automated Reasoning* 56.3, pp. 309–341.

References II


Caspi, P., D. Pilaud, N. Halbwachs, and J. Plaice (1987). "LUSTRE: A declarative language for programming synchronous systems". In: *Proc. 14th ACM SIGPLAN-SIGACT Symp. Principles Of Programming Languages (POPL 1987)*. ACM. Munich, Germany: ACM Press, pp. 178–188.


Colaço, J.-L. and M. Pouzet (2003). "Clocks as First Class Abstract Types". In: *Proc. 3rd Int. conf. Embedded Software (EMSOFT 2003)*. Ed. by R. Alur and I. Lee. Vol. 2855. Philadelphia, Pennsylvania, USA, pp. 134–155.


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


Kahn, G. (1974). "The Semantics of a Simple Language for Parallel Programming". In: ed. by J. L. Rosenfeld. North-Holland, pp. 471–475. ISBN: 0-7204-2803-3.


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


McCoy, F. (1885). *Natural history of Victoria: Prodromus of the Zoology of Victoria*. Frog images.

References III

Paulin-Mohring, C. (2009). "A constructive denotational semantics for Kahn networks in Coq". In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*. Ed. by Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin. Cambridge University Press, pp. 383–413.

The Coq Development Team (2016). *The Coq proof assistant reference manual*. Version 8.5. Inria. URL: http://coq.inria.fr.