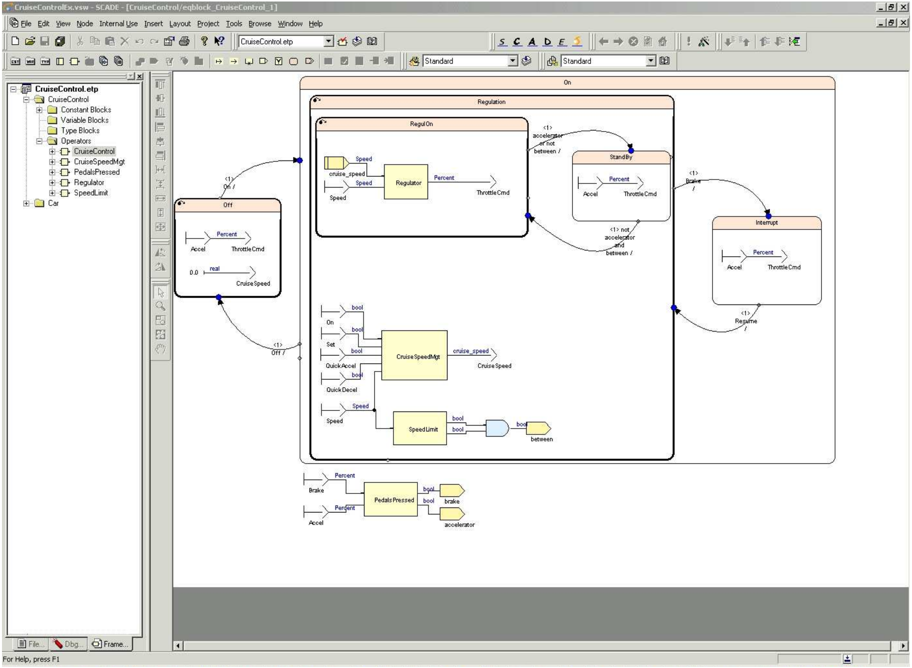
A Formally Verified Compiler for Lustre

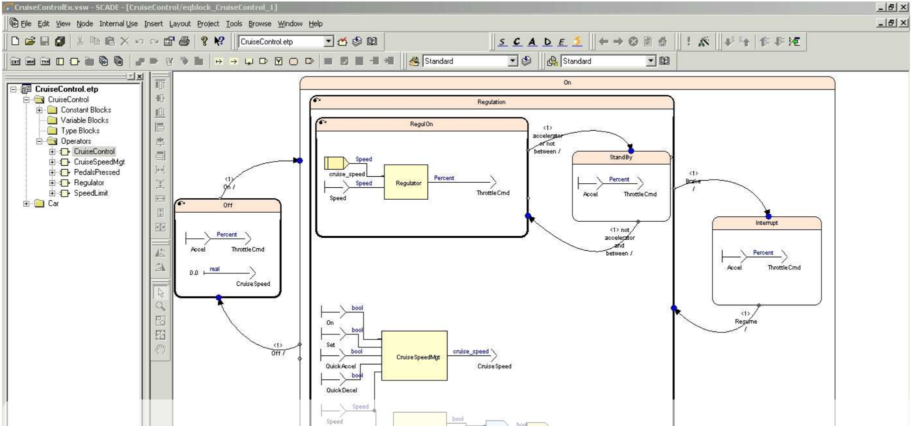
Timothy Bourke[1,2] Lélio Brun[1,2] Pierre-Évariste Dagand[4,3,1]
Xavier Leroy[1] Marc Pouzet[4,2,1] Lionel Rieg[5,6]

1. Inria Paris
2. DI, École normale supérieure
3. CNRS
4. Univ. Pierre et Marie Curie
5. Yale University
6. Collège de France

PLDI, Barcelona—20 June 2017

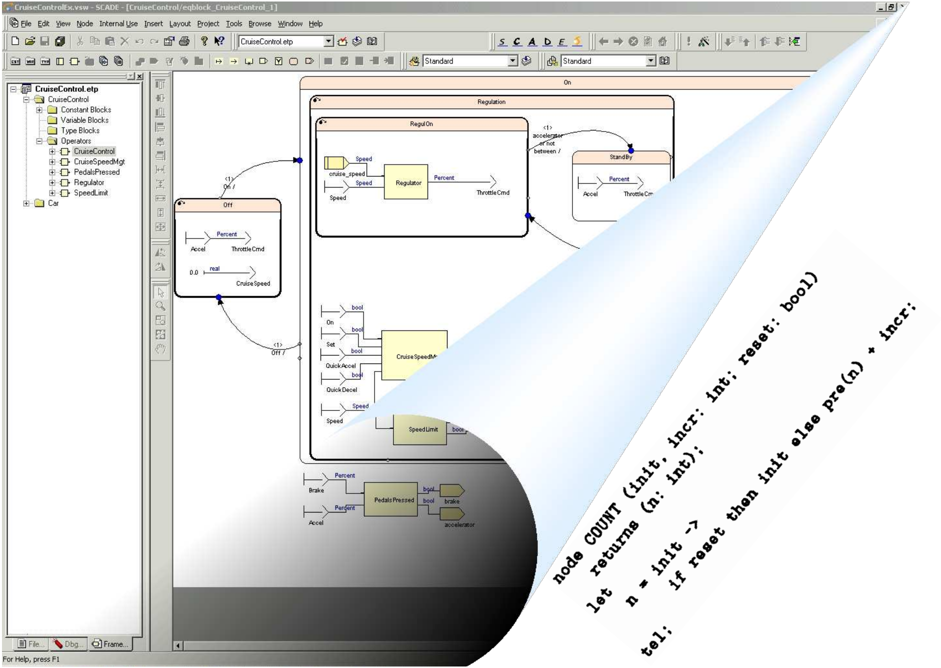


Screenshot from ANSYS/Esterel Techologies SCADE Suite



- Widely used to program safety-critical software:
  - Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear.
  - Airbus (A340, A380), Comac, EADS Astrium, Embraer, Eurocopter, PIAGGIO Aerospace, Pratt & Whitney, Sukhoi, Turbomeca, U.S. Army, Siemens, . . .
- DO-178B level A certified development tool.

Screenshot from ANSYS/Esterel Techologies SCADE Suite



Screenshot from ANSYS/Esterel Techologies SCADE Suite

What did we do?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - Building on 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.
  - Building on 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 (higher-order logic);
  - Tactic-based interactive proof.

What did we do?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - Building on 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 (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.
  - Building on 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 (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)

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

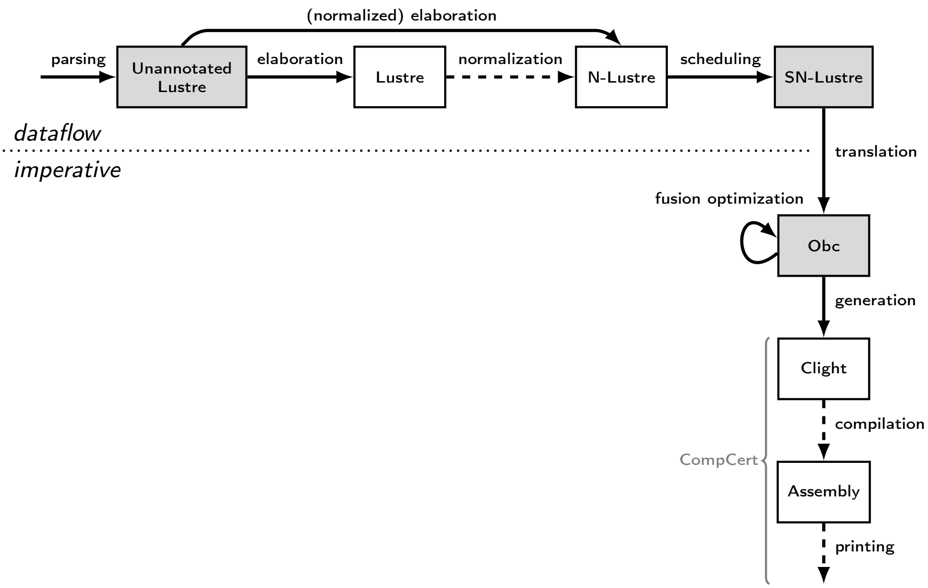
What did we do?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - Building on 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 (higher-order logic);
  - Tactic-based interactive proof.
- Why not use HOL, Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

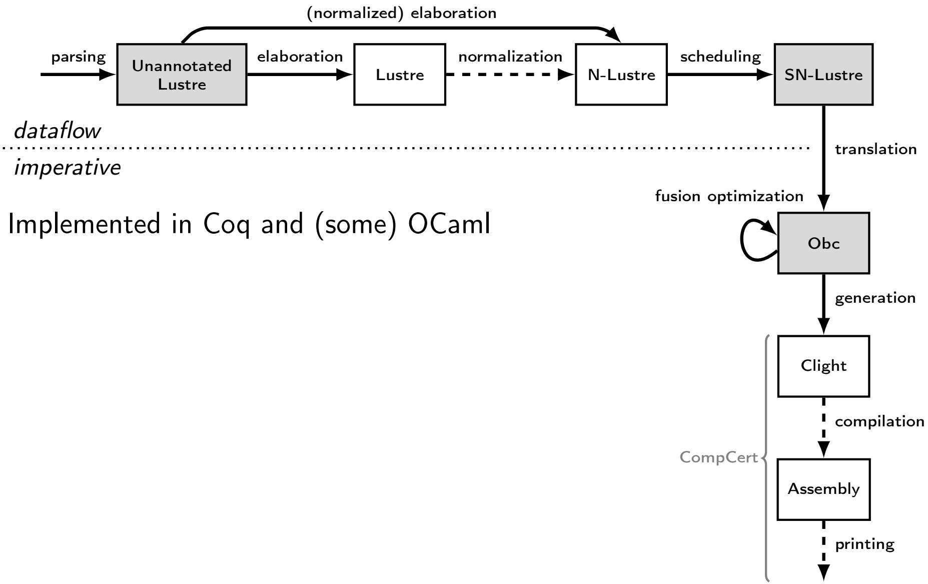
  CompCert: a formal model and compiler for a subset of C
  - A generic machine-level model of execution and memory
  - A verified path to assembly code output (PowerPC, ARM, x86)

  [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

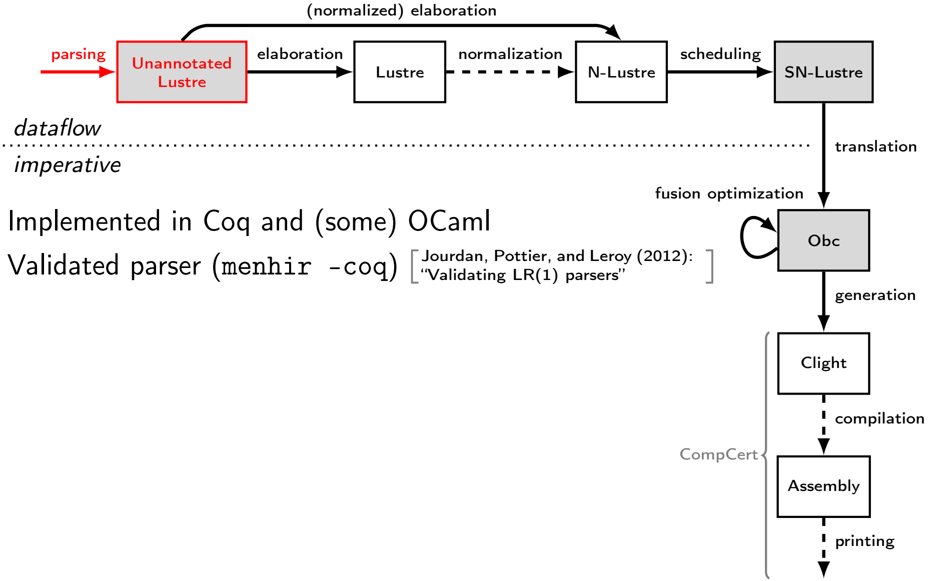


The Vélus Lustre Compiler



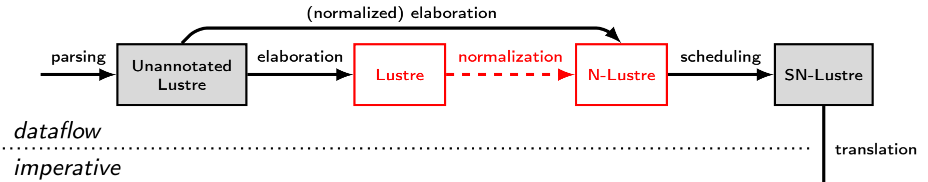
- Implemented in Coq and (some) OCaml

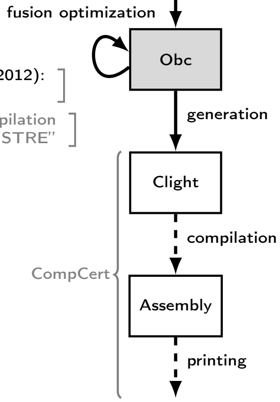
The Vélus Lustre Compiler



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

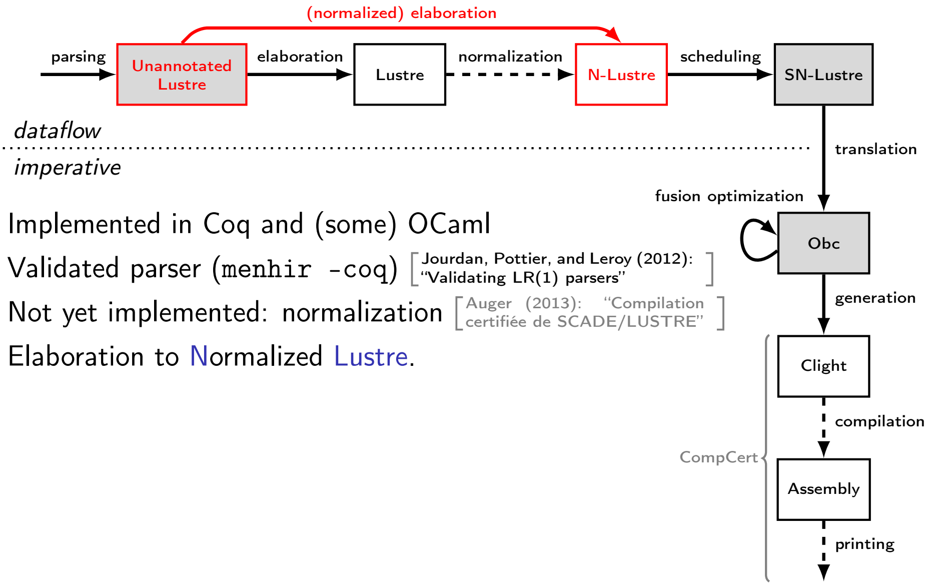
The Vélus Lustre Compiler





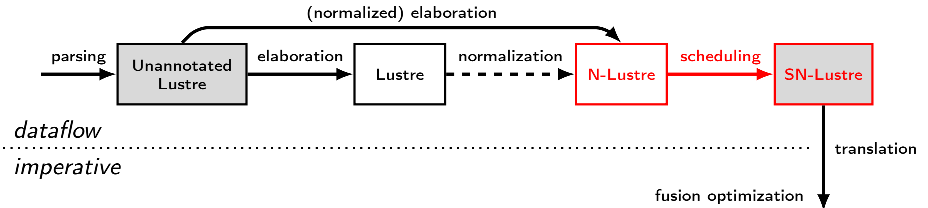
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]

The Vélus Lustre Compiler

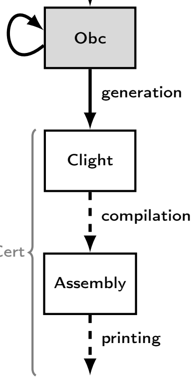


- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.

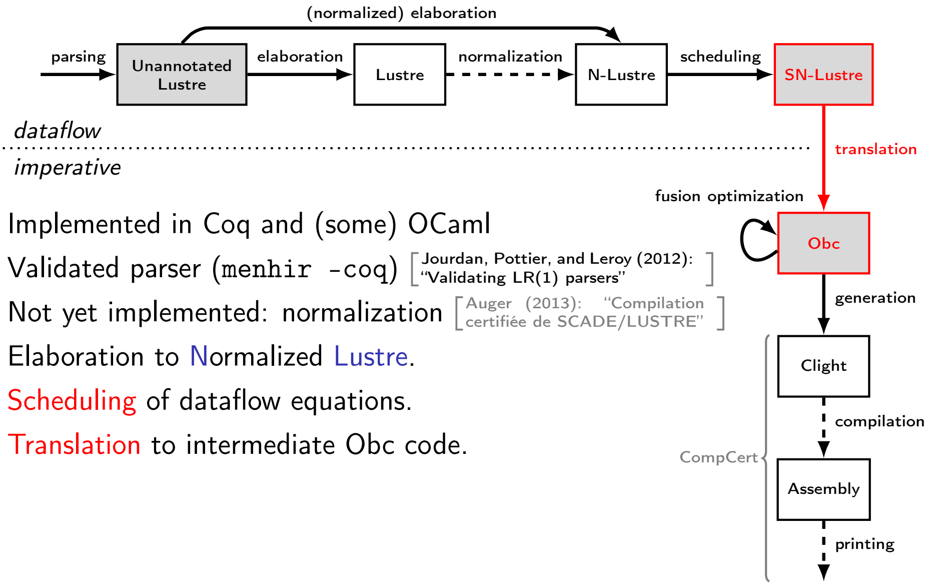
The Vélus Lustre Compiler



- 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.

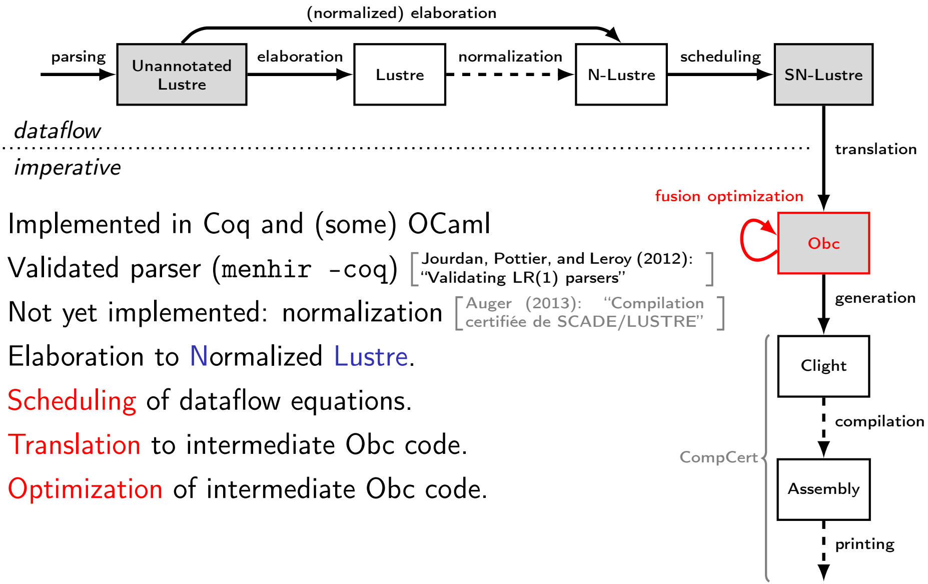


The Vélus Lustre Compiler



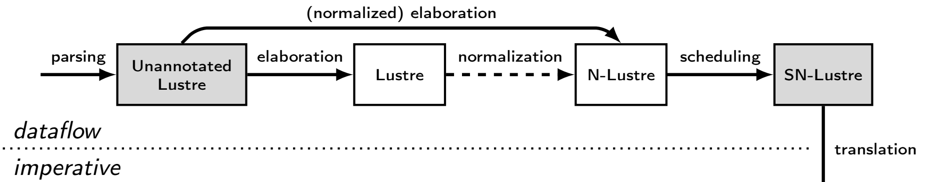
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.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

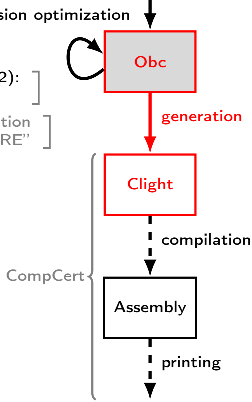
The Vélus Lustre Compiler



- 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.

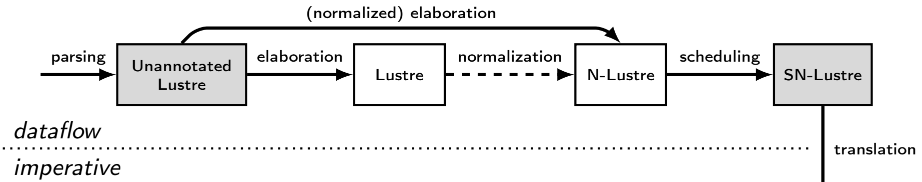
The Vélus Lustre Compiler

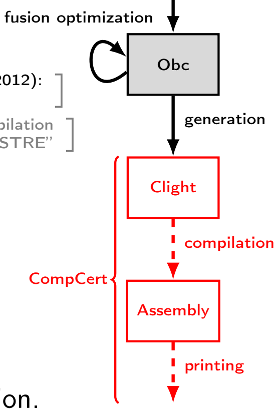




- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.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

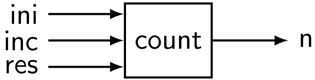
- A language for programming cyclic control software.

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

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

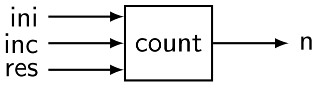
- A restriction of Kahn process networks [Kahn (1974): "The Semantics of a Simple Language for Parallel Programming"], guaranteed to execute in bounded time and space.

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



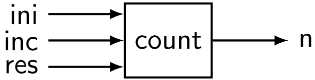
Lustre [Caspi, Pilaud, Halbwachs, and Plaice (1987): "LUSTRE: A declarative language for programming synchronous systems"]

```
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: global) :
  ident → stream (list value) → stream (list value) → Prop :=
| SNode:
    clock_of xss bk →
    find_node f G = Some (mk_node f i o v eqs _ _ _ _ _ _) →
    → same_clock xss → same_clock yss →
    (∃ H,
          sem_vars bk H (map fst i) xss
       ∧ sem_vars bk H (map fst o) yss
       ∧ (∀ n, absent_list xss n ↔ absent_list yss)
       ∧ Forall (sem_equation G bk H) 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: global) :
  ident → stream (list value) → stream (list value) → Prop :=
| SNode:
    clock_of xss bk →
    find_node f G = Some (mk_node f i o v eqs _ _ _ _ _ _) →
    → same_clock xss → same_clock yss →
    (∃ H,
          sem_vars bk H (map fst i) xss
       ∧  sem_vars bk H (map fst o) yss
       ∧  (∀ n, absent_list xss n ↔ absent_list yss)
       ∧  Forall (sem_equation G bk H) eqs) →
    sem_node G f xss yss.
```

sem_node G f xss yss

$f : \text{stream}(T^+) \rightarrow \text{stream}(T^+)$

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
```

→ normalization

```
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
```

normalization ⇒

```
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
```

⇓ scheduling

```
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
  }
}
```

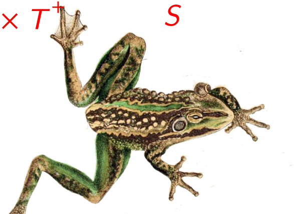
Lustre compilation: translation to imperative code

[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)

$S \times T^+ \rightarrow S \times T^+ \quad S$



```
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	⋯

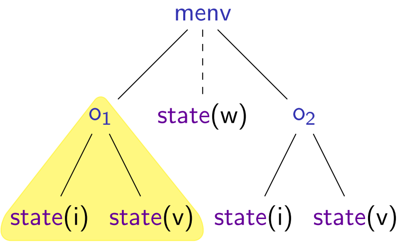
Lustre compilation: translation to clocked imperative code

```
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

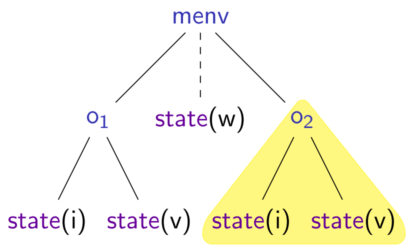
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
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

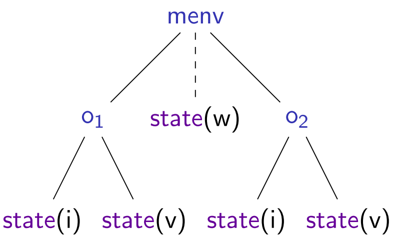
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
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

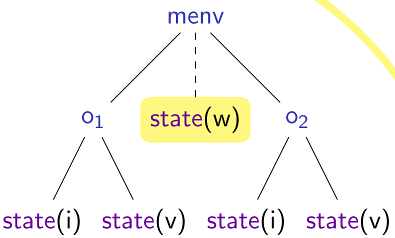
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
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.

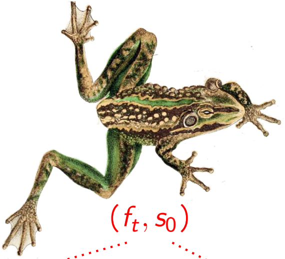
```
Definition tovar (x: ident) : exp :=
  if PS.mem x memories then State x else Var x.

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`

$\text{stream}(T^+) \rightarrow \text{stream}(T^+)$

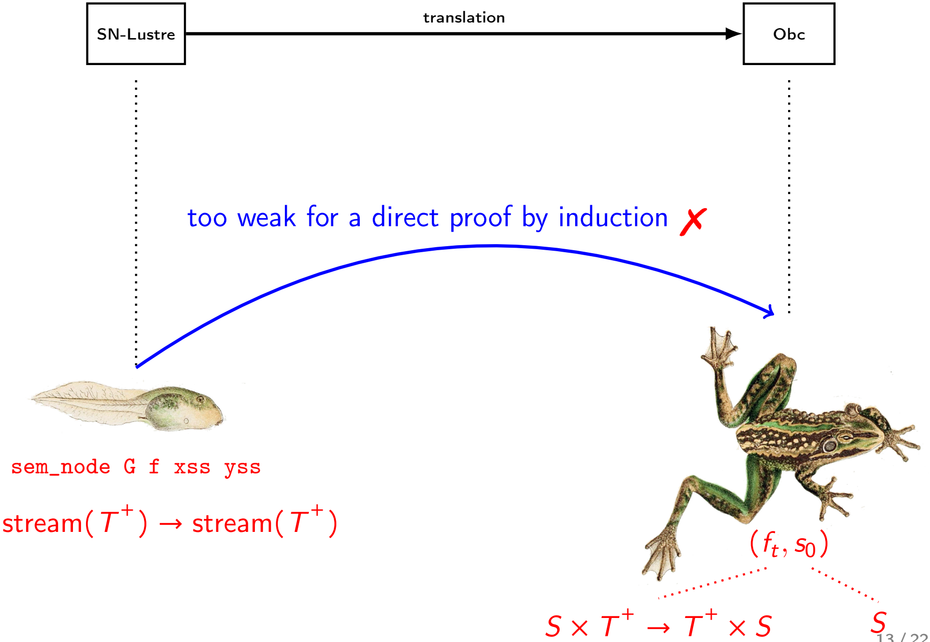


(f_t, s_0)

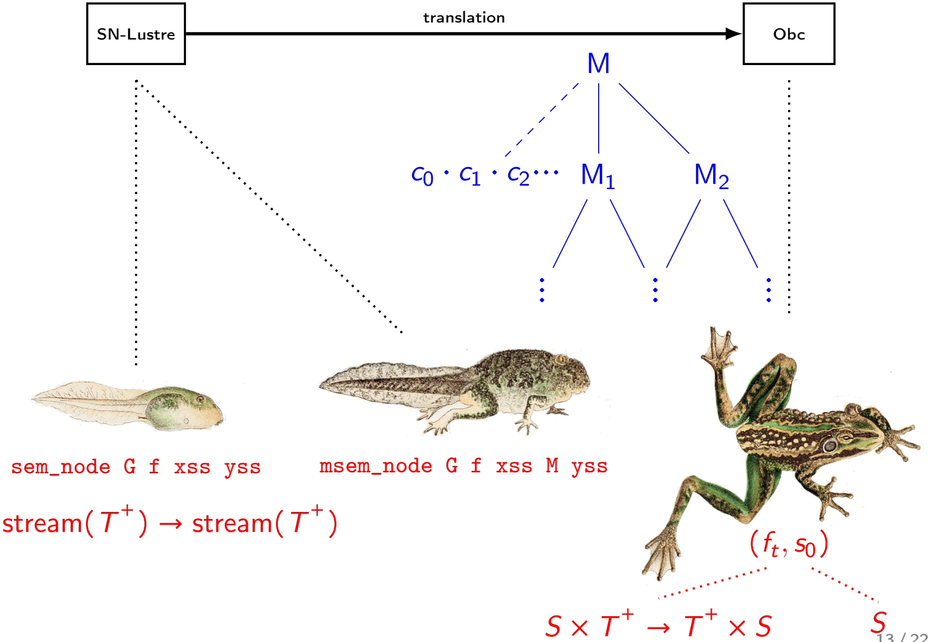
$S \times T^+ \rightarrow T^+ \times S$

S

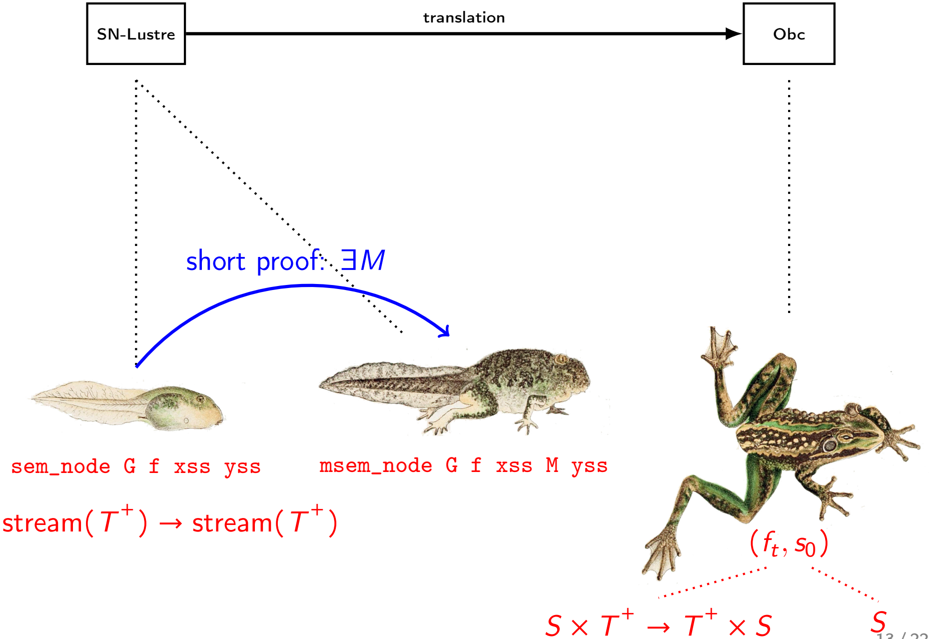
Correctness of translation



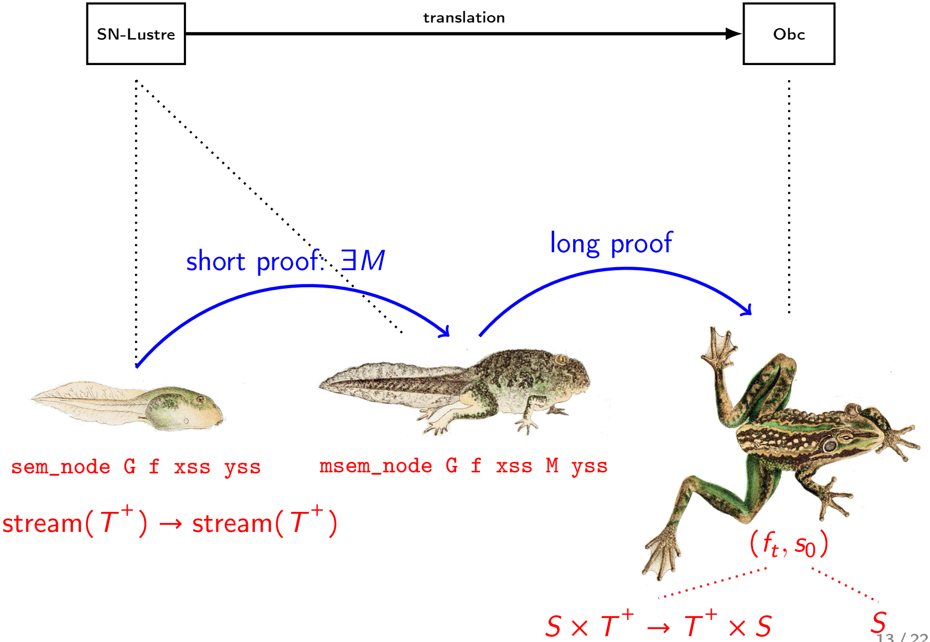
Correctness of translation



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

SN-Lustre — translation → Obc

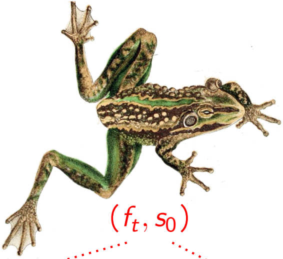
short proof: $\exists M$

long proof

sem_node G f xss yss

msem_node G f xss M yss

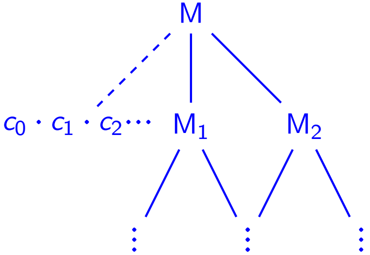
$\text{stream}(T^+) \to \text{stream}(T^+)$

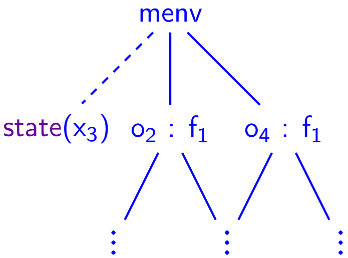


(f_t, s_0)

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

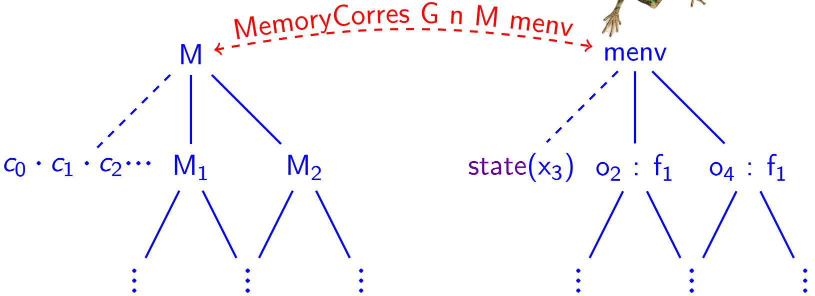
SN-Lustre to Obc: memory correspondence





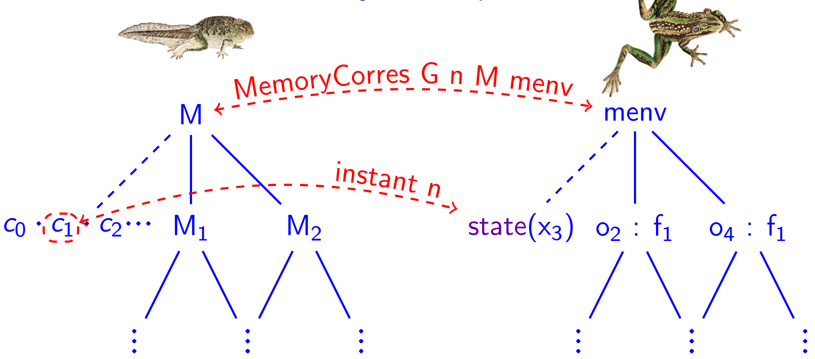
- Memory 'model' does not change between SN-Lustre and Obc.
  - Corresponds at each 'snapshot'.
- The real challenge is in the change of semantic model: from dataflow streams to sequenced assignments

SN-Lustre to Obc: memory correspondence



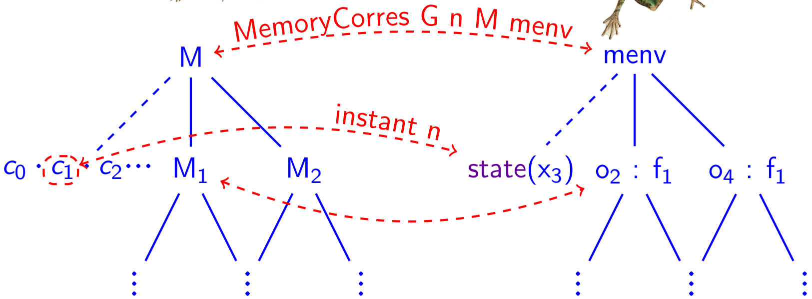
- Memory 'model' does not change between SN-Lustre and Obc.
  - Corresponds at each 'snapshot'.
- The real challenge is in the change of semantic model: from dataflow streams to sequenced assignments

SN-Lustre to Obc: memory correspondence



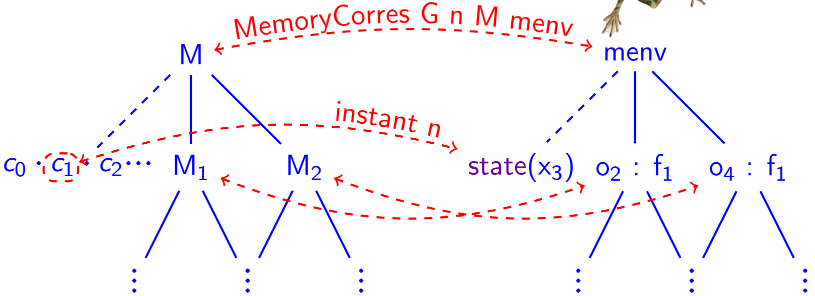
- Memory 'model' does not change between SN-Lustre and Obc.
  - Corresponds at each 'snapshot'.
- The real challenge is in the change of semantic model: from dataflow streams to sequenced assignments

SN-Lustre to Obc: memory correspondence



- Memory ‘model’ does not change between SN-Lustre and Obc.
  - Corresponds at each ‘snapshot’.
- The real challenge is in the change of semantic model: from dataflow streams to sequenced assignments

SN-Lustre to Obc: memory correspondence



- Memory ‘model’ does not change between SN-Lustre and Obc.
  - Corresponds at each ‘snapshot’.
- The real challenge is in the change of semantic model: from dataflow streams to sequenced assignments

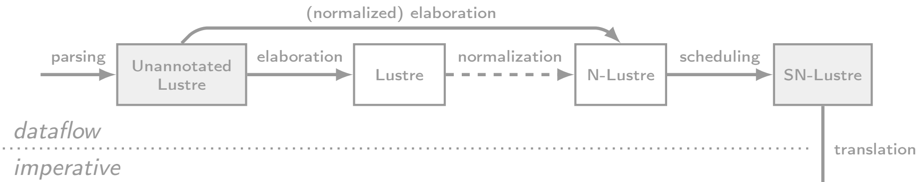
Control structure fusion [Biernacki, Colaço, Hamon, and Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

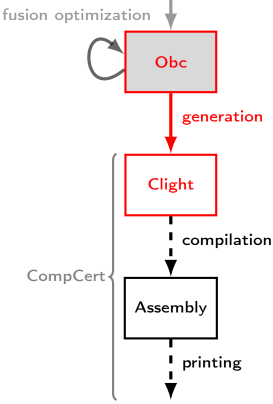
```
step(delta: int, sec: bool)
   returns (v: int) {
  var r, t : int;

  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
}
```

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

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


  state(w) := v
}
```

- Generate control for each equation; splits proof obligation in two.
- Fuse afterward: scheduler places similarly clocked equations together.
- Use whole framework to justify required invariant.
- Easier to reason in intermediate language than in Clight.

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.

```
class count { ... }

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);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
   int w;
   struct count o1;
   struct count o2;
};

struct avgvelocity$step {
   int r;
   int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
   count$reset(&(self→o1));
   count$reset(&(self→o2));
   self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
   register int t, step$n;

   step$n = count$step(&(self→o1), 0, delta, 0);
   out→r = step$n;
   if (sec) {
     step$n = count$step(&(self→o2), 1, 1, 0);
     t = step$n;
     out→v = out→r / t;
   } else {
     out→v = self→w;
   }
   self→w = out→v;
}
```

```
class count { ... }
```

```
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);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }
```

```
struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

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);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

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);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

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);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count ∗self) { ... }
int count$step(struct count ∗self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity ∗self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity ∗self,
                struct avgvelocity$step ∗out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

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);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

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);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

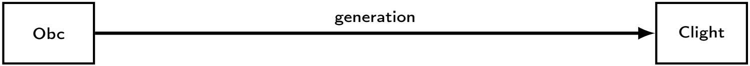
```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
   int w;
   struct count o1;
   struct count o2;
};

struct avgvelocity$step {
   int r;
   int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
   count$reset(&(self→o1));
   count$reset(&(self→o2));
   self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
               struct avgvelocity$step *out, int delta, _Bool sec)
{
   register int t, step$n;

   step$n = count$step(&(self→o1), 0, delta, 0);
   out→r = step$n;
   if (sec) {
     step$n = count$step(&(self→o2), 1, 1, 0);
     t = step$n;
     out→v = out→r / t;
   } else {
     out→v = self→w;
   }
   self→w = out→v;
}
```

```
class count { ... }

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);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

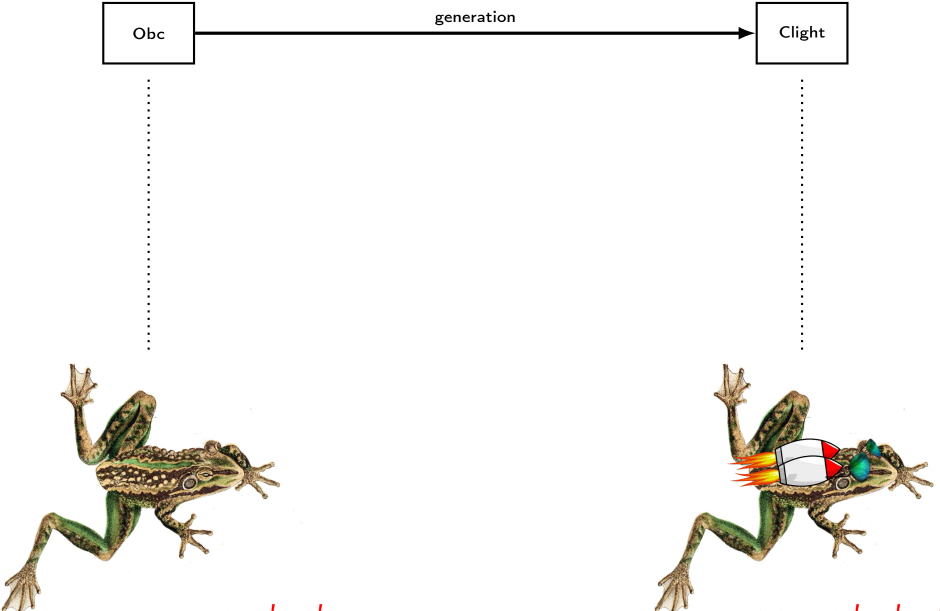
- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

Correctness of generation

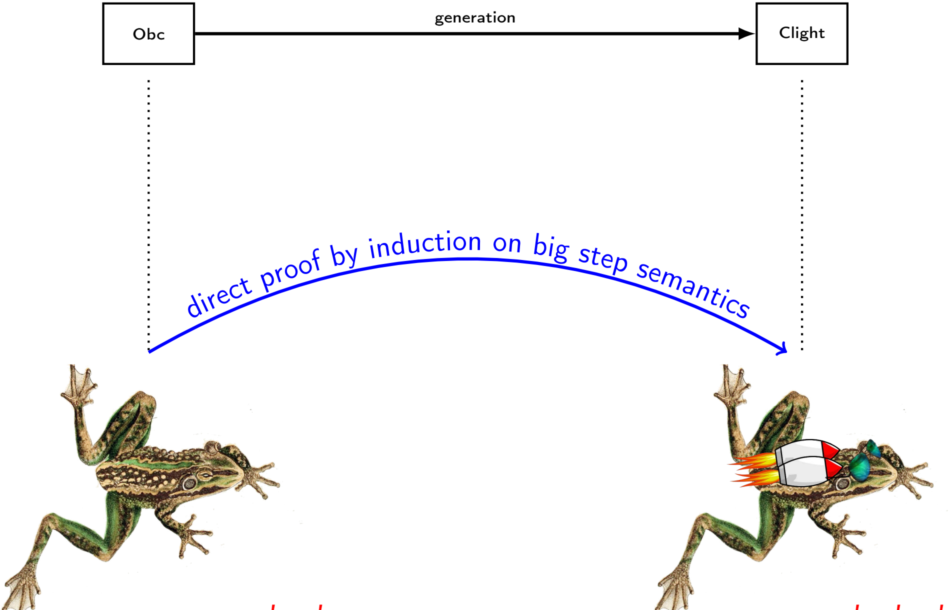


Correctness of generation



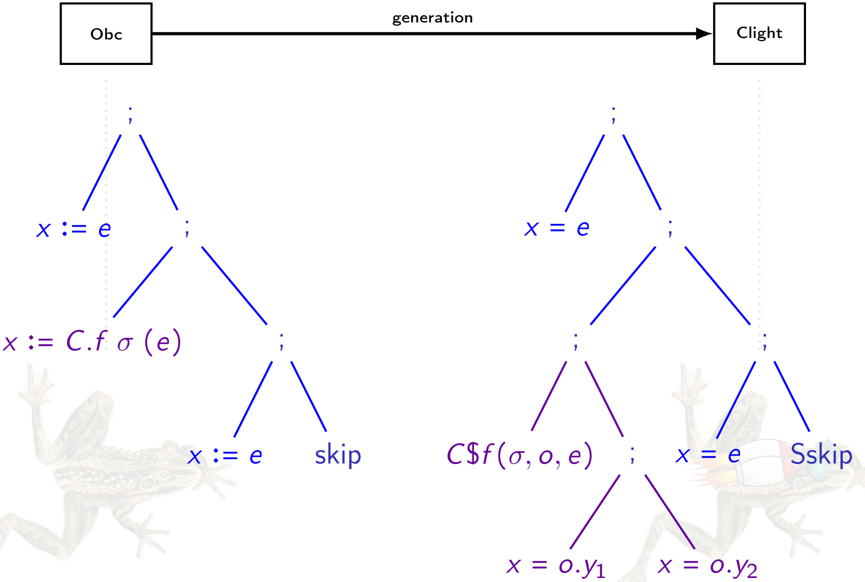
$me, ve \vdash s \Downarrow (me', ve')$ $\quad$ $e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correctness of generation



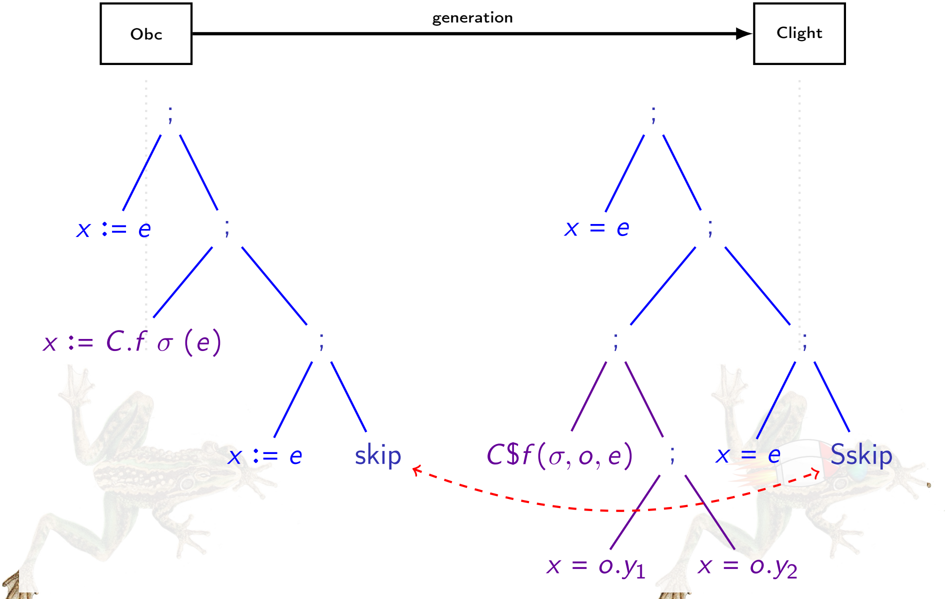
$me, ve \vdash s \Downarrow (me', ve')$ $\quad$ $e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correctness of generation



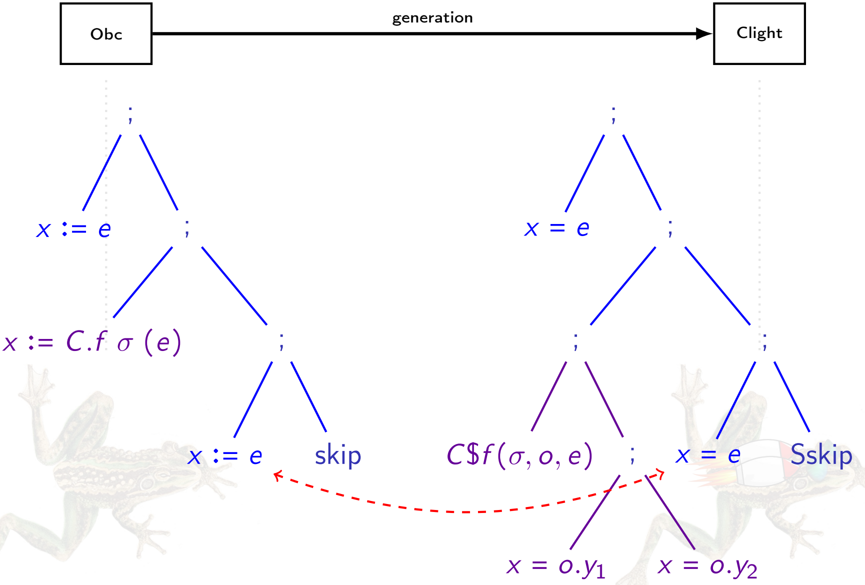
$me, ve \vdash s \Downarrow (me', ve')$ $\qquad$ $e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correctness of generation



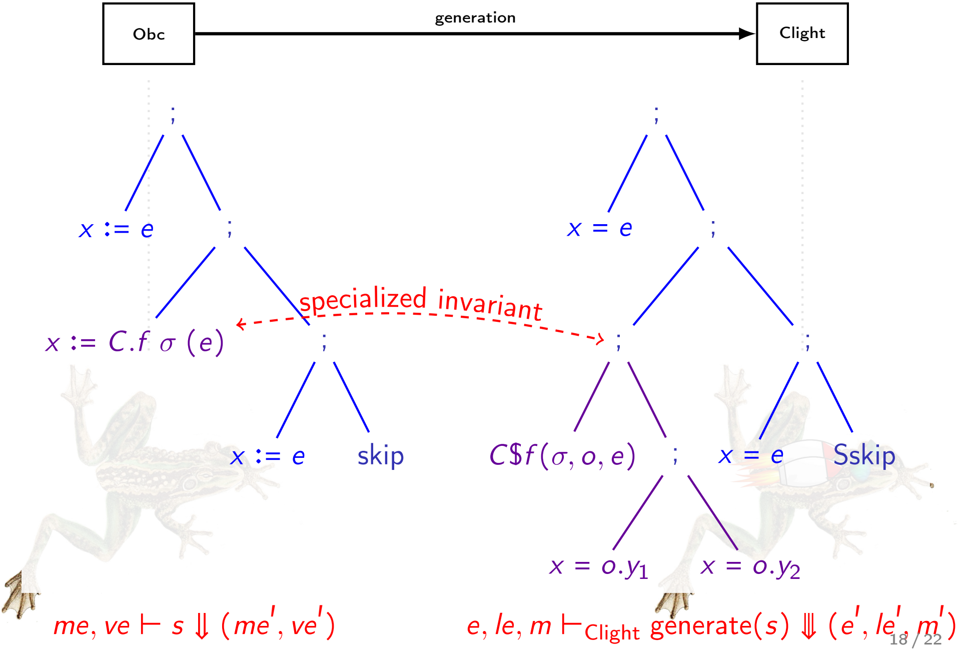
$me, ve \vdash s \Downarrow (me', ve')$ $e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correctness of generation

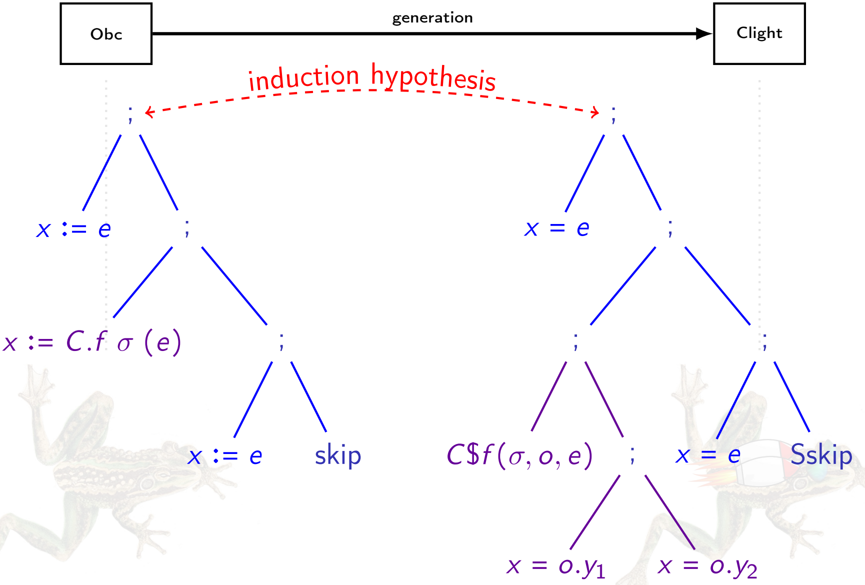


$me, ve \vdash s \Downarrow (me', ve')$ $\quad$ $e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correctness of generation



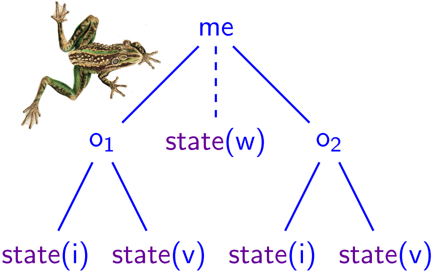
Correctness of generation

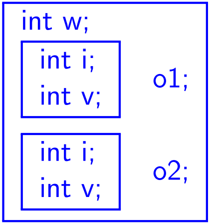


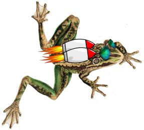
$me, ve \vdash s \Downarrow (me', ve')$

$e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Obc to Clight: memory correspondence

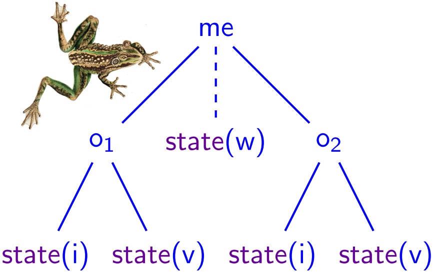


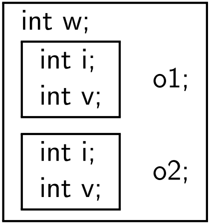


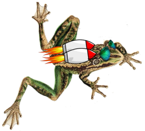


- This time the semantic models are similar (Clight: very detailed)
- The real challenge is to relate the memory models.
  - Obc: tree structure, variable separation is manifest.
  - Clight: block-based, must treat aliasing, alignment, and sizes.

Obc to Clight: memory correspondence







- This time the semantic models are similar (Clight: very detailed)
- The real challenge is to relate the memory models.
  - Obc: tree structure, variable separation is manifest.
  - Clight: block-based, must treat aliasing, alignment, and sizes.

- Extend CompCert's lightweight library of separating assertions: https://github.com/AbsInt/CompCert/common/Separation.v.
- Encode simplicity of source model in richer memory model.
- General (and very useful) technique for interfacing with CompCert.

```
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.
```

```
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,

```
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,

```
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...

```
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

Experimental results

Industrial application

- ≈6 000 nodes
- ≈162 000 equations
- ≈12 MB source file
  (minus comments)
- Modifications:
  - Remove constant lookup tables.
  - Replace calls to assembly code.
- Vélus compilation: ≈1 min 40 s

Experimental results

Industrial application

- ≈6 000 nodes
- ≈162 000 equations
- ≈12 MB source file (minus comments)
- Modifications:
  - Remove constant lookup tables.
  - Replace calls to assembly code.
- Vélus compilation: ≈1 min 40 s

	Vélus	*Hept+CC*	*Hept+gcc*	*Hept+gcci*	*Lus6+CC*	*Lus6+gcc*	*Lus6+gcci*
avgvelocity	315	385 (22%)	265 (-15%)	70 (-77%)	1 150 (265%)	625 (98%)	350 (11%)
count	55	55 (0%)	25 (-54%)	25 (-54%)	300 (445%)	160 (190%)	50 (-9%)
tracker	680	790 (16%)	530 (-22%)	500 (-26%)	2 610 (283%)	1 515 (122%)	735 (8%)
pip_ex	4 415	4 065 (-7%)	2 565 (-41%)	2 040 (-53%)	10 845 (145%)	6 245 (41%)	2 905 (-34%)
mp_longitudinal [16]	5 525	6 465 (17%)	3 465 (-37%)	2 835 (-48%)	11 675 (111%)	6 785 (22%)	3 135 (-43%)
cruise [54]	1 760	1 875 (6%)	1 230 (-30%)	1 230 (-30%)	5 855 (232%)	3 595 (104%)	1 965 (11%)
risingedgeretrigger [19]	285	300 (5%)	190 (-33%)	190 (-33%)	1 440 (405%)	820 (187%)	335 (17%)
chrono [20]	410	425 (3%)	305 (-25%)	305 (-25%)	2 490 (507%)	1 500 (265%)	670 (63%)
watchdog3 [26]	610	575 (-5%)	355 (-41%)	310 (-49%)	2 015 (230%)	1 135 (86%)	530 (-13%)
functionalchain [17]	11 550	13 535 (17%)	8 545 (-26%)	7 525 (-34%)	23 085 (99%)	14 280 (23%)	8 240 (-28%)
landing_gear [11]	9 660	8 475 (-12%)	5 880 (-39%)	5 810 (-39%)	25 470 (163%)	15 055 (55%)	8 025 (-16%)
minus [57]	890	900 (1%)	580 (-34%)	580 (-34%)	2 825 (217%)	1 620 (82%)	800 (-10%)
prodcell [32]	1 020	990 (-2%)	620 (-39%)	410 (-59%)	3 615 (254%)	2 050 (100%)	1 070 (4%)
ums_verif [57]	2 590	2 285 (-11%)	1 380 (-46%)	920 (-64%)	11 725 (352%)	6 730 (159%)	3 420 (32%)

Figure 12. WCET estimates in cycles [4] for step functions compiled for an armv7-a/vfpv3-d16 target with CompCert 2.6 (CC) and GCC 4.4.8 `-O1` without inlining (gcc) and with inlining (gcci). Percentages indicate the difference relative to the first column.

It performs loads and stores of volatile variables to model, respectively, input consumption and output production. The coinductive predicate presented in Section 1 is introduced to relate the trace of these events to input and output streams.

Finally, we exploit an existing CompCert lemma to transfer our results from the big-step model to the small-step one, from whence they can be extended to the generated assembly code to give the property stated at the beginning of the paper. The transfer lemma requires showing that a program does not diverge. This is possible because the body of the main loop always produces observable events.

5. Experimental Results

Our prototype compiler, Vélus, generates code for the platforms supported by CompCert (PowerPC, ARM, and x86). The code can be executed in a 'test mode' that `scanf`s inputs and `printf`s outputs using an alternative (unverified) entry point. The *verified* integration of generated code into a complete system where it would be triggered by interrupts and interact with hardware is the subject of ongoing work.

As there is no standard benchmark suite for Lustre, we adapted examples from the literature and the Lustre v4 distribution [57]. The resulting test suite comprises 14 programs, totaling about 160 nodes and 960 equations. We compared the code generated by Vélus with that produced by the Heptagon 1.03 [23] and Lustre v6 [35, 57] academic compilers. For the example with the deepest nesting of clocks (3 levels), both Heptagon and our prototype found the same optimal schedule. Otherwise, we follow the approach of [23, §6.2] and estimate the Worst-Case Execution Time (WCET) of the generated code using the open-source OTAWA v5 framework [4] with the 'trivial' script and default parameters.[10] For the targeted domain, an over-approximation to the WCET is usually more valuable than raw performance numbers. We compiled with CompCert 2.6 and GCC 4.8.4 (`-O1`) for the `arm-none-eabi` target (armv7-a) with a hardware floating-point unit (vfpv3-d16).

The results of our experiments are presented in Figure 12. The first column shows the worst-case estimates in cycles for the step functions produced by Vélus. These estimates compare favorably with those for generation with either Heptagon or Lustre v6 and then compilation with CompCert. Both Heptagon and Lustre (automatically) re-normalize the code to have one operator per equation, which can be costly for nested conditional statements, whereas our prototype simply maintains the (manually) normalized form. This re-normalization is unsurprising: both compilers must treat a richer input language, including arrays and automata, and both expect the generated code to be post-optimized by a C compiler. Compiling the generated code with GCC but still without any inlining greatly reduces the estimated WCETs, and the Heptagon code then outperforms the Vélus code. GCC applies 'if-conversions' to exploit predicated ARM instructions which avoids branching and thereby improves WCET estimates. The estimated WCETs for the Lustre v6 generated code only become competitive when inlining is enabled because Lustre v6 implements operators, like **pre** and ->, using separate functions. CompCert can perform inlining, but the default heuristic has not yet been adapted for this particular case. We note also that we use the modular compilation scheme of Lustre v6, while the code generator also provides more aggressive schemes like clock enumeration and automaton minimization [29, 56].

Finally, we tested our prototype on a large industrial application (≈6 000 nodes, ≈162 000 equations, ≈12 MB source file without comments). The source code was already normalized since it was generated with a graphical interface,

[10] This configuration is quite pessimistic but suffices for the present analysis.

12

Experimental results

Industrial application

- ≈6 000 nodes
- ≈162 000 equations
- ≈12 MB source file (minus comments)
- Modifications:
  - Remove constant lookup tables.
  - Replace calls to assembly code.
- Vélus compilation: ≈1 min 40 s

	Vélus	Hept+CC	Hept+gcc	Hept+gcci	Lus6+CC	Lus6+gcc	Lus6+gcci
avgvelocity	315	385 (22%)	265 (-15%)	70 (-77%)	1 150 (265%)	625 (98%)	350 (11%)
count	55	55 (0%)	25 (-54%)	25 (-54%)	300 (445%)	160 (190%)	50 (-9%)
tracker	680	790 (16%)	530 (-22%)	500 (-26%)	2 610 (283%)	1 515 (122%)	735 (8%)
pip_ex	4 415	4 065 (-7%)	2 565 (-41%)	2 040 (-53%)	10 845 (145%)	6 245 (41%)	2 905 (-34%)
mp_longitudinal [16]	5 525	6 465 (17%)	3 465 (-37%)	2 835 (-48%)	11 675 (111%)	6 785 (22%)	3 135 (-43%)
cruise [54]	1 760	1 875 (6%)	1 230 (-30%)	1 230 (-30%)	5 855 (232%)	3 595 (104%)	1 965 (11%)
risingedgeretrigger [19]	285	300 (5%)	190 (-33%)	190 (-33%)	1 440 (405%)	820 (187%)	335 (17%)
chrono [20]	410	425 (3%)	305 (-25%)	305 (-25%)	2 490 (507%)	1 500 (265%)	670 (63%)
watchdog3 [26]	610	575 (-5%)	355 (-41%)	310 (-49%)	2 015 (230%)	1 135 (86%)	530 (-13%)
functionalchain [17]	11 550	13 535 (17%)	8 545 (-26%)	7 525 (-34%)	23 085 (99%)	14 280 (23%)	8 240 (-28%)
landing_gear [11]	9 660	8 475 (-12%)	5 880 (-39%)	5 810 (-39%)	25 470 (163%)	15 055 (55%)	8 025 (-16%)
minus [57]	890	900 (1%)	580 (-34%)	580 (-34%)	2 825 (217%)	1 620 (82%)	800 (-10%)
prodcell [32]	1 020	990 (-2%)	620 (-39%)	410 (-59%)	3 615 (254%)	2 050 (100%)	1 070 (4%)
ums_verif [57]	2 590	2 285 (-11%)	1 380 (-46%)	920 (-64%)	11 725 (352%)	6 730 (159%)	3 420 (32%)

Figure 12. WCET estimates in cycles [4] for step functions compiled for an armv7-a/vfpv3-d16 target with CompCert 2.6 (CC) and GCC 4.4.8 -O1 without inlining (gcc) and with inlining (gcci). Percentages indicate the difference relative to the first column.

- Compare WCET of generated code with two academic compilers on smaller examples.

[Ballabriga, Cassé, Rochange, and Sainrat (2010): "OTAWA: An Open Toolbox for Adaptive WCET Analysis"]

- Results depend on C compiler:
  - CompCert: Vélus code same/better
  - gcc -O1 no-inlining: Vélus code slower
  - gcc -O1: Vélus code much slower
- [TODO]: adjust CompCert inlining heuristic.

12

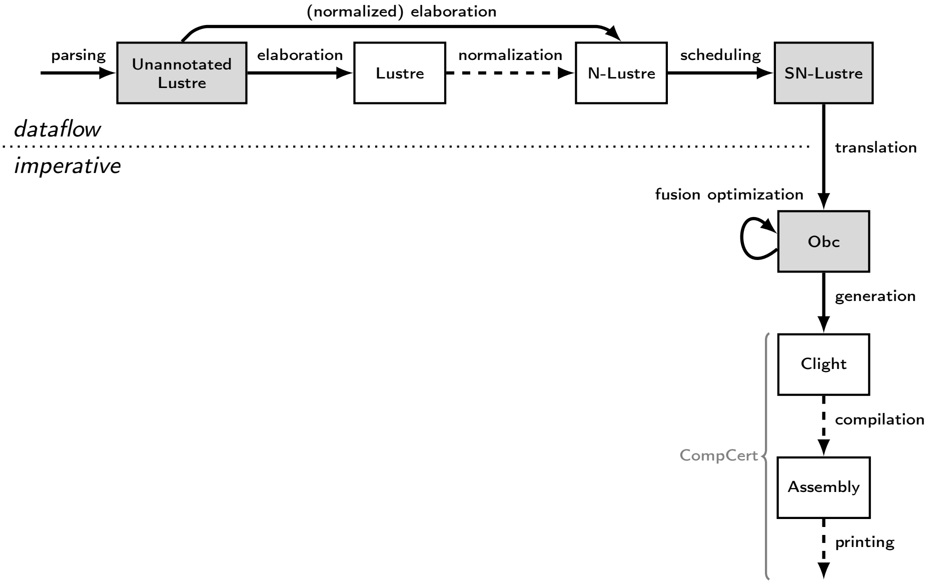
Vélus: A Formally Verified Compiler for Lustre

Results (after 2 years)

- Working compiler from Lustre to assembler in Coq.
- Formally relate dataflow model to imperative code.
- Generate Clight for CompCert; change to richer memory model.

Ongoing work

- Finish normalization pass.
- Prove that a well-typed program has a semantics.
- Combine interactive and automatic proof to verify Lustre programs.


(normalized) elaboration
parsing
Unannotated Lustre
elaboration
Lustre
normalization
N-Lustre
scheduling
SN-Lustre
dataflow
imperative
translation
fusion optimization
Obc
generation
Clight
compilation
CompCert
Assembly
printing

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.

Ballabriga, C., H. Cassé, C. Rochange, and P. Sainrat (2010). "OTAWA: An Open Toolbox for Adaptive WCET Analysis". In: *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*. Vol. 6399. Lecture Notes in Computer Science. Waidhofen/Ybbs, Austria: Springer, pp. 35–46.

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.

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.

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.

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