A Formally Verified Compiler for Lustre

Timothy Bourke¹,²  Lélio Brun¹,²  Pierre-Évariste Dagand⁴,³,¹
Xavier Leroy¹  Marc Pouzet⁴,²,¹  Lionel Rieg⁵,⁶

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
• 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 Technologies SCADE Suite
node COUNT (init: int; incr: int; reset: bool):
returns (n: int):
let n = init ->
if reset then init else pre(n) + incr:
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.

  - 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.

  - 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.

  – 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
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 Formal-izition and Proof of a Modular Lustre Code Generator”].

- Prove that the generated code implements the dataflow semantics.

  - 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)

- Computer assistance is all but essential for such detailed models.
The Vélus Lustre Compiler

Dataflow

Imperative

(normalized) elaboration

parsing

elaboration

normalization

scheduling

Unannotated Lustre

Lustre

N-Lustre

SN-Lustre

translation

fusion optimization

generation

Clight

Assembly

CompCert

CompCert4 / 22
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.

---

**Dataflow**: parsing → Unannotated Lustre → elaboration → Lustre → normalization → N-Lustre → scheduling → SN-Lustre

**Imperative**: (normalized) elaboration → fusion optimization → generation → Clight → compilation → CompCert → Assembly → printing
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.

---

Unannotated Lustre → (normalized) elaboration → Lustre → normalization → N-Lustre → scheduling → SN-Lustre

- Parsing
- Elaboration
- Normalization
- Scheduling

---

(dataflow)

(imperative)

---

- Translation
- Fusion optimization
- Generation
- Compilation
- Printing
The Vélus Lustre Compiler

- Implemented in Coq and (some) OCaml
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.
- Translation to intermediate Obc code.

---

**Diagram:**

- Parsing Unannotated Lustre → (normalized) elaboration → Lustre → normalization → N-Lustre → scheduling → SN-Lustre

**Steps:**

- Dataflow
- Imperative

**Flowchart:**

- Unannotated Lustre
- Parsing
- Elaboration
- Normalization
- Scheduling
- SN-Lustre
- Translation
- Fusion optimization
- Obc
- Generation
- Compilation
- Assembly
- Printing
- CompCert
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.

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

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

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)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel

- 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   | 0   | ...
|------|------|------|------|------|------|------|------|------|------
| inc  | 0    | 1    | 2    | 1    | 2    | 3    | 0    | 0    | ...
| res  | F    | F    | F    | F    | T    | F    | F    | F    | ...
| true fby false | T    | F    | F    | F    | F    | F    | F    | F    | ...
| 0 fby n       | 0    | 0    | 1    | 3    | 4    | 0    | 3    | 3    | ...
| n             | 0    | 1    | 3    | 4    | 0    | 3    | 3    | 3    | ...

true fby false: True if fby is false, False otherwise.
0 fby n: 0 fby n is a sequence of 0, 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

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   : ident → clock → cexp → equation
| EqFby   : ident → clock → const → cexp → 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 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>inc</td>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>res</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

|     | true fby false | T | F | F | F | F | F | F | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 fby n</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>...</td>
</tr>
</tbody>
</table>

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 → lexs → 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

Inductive clock : Set :=
| Cbase : clock
| Cons : 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 : stream( T⁺ ) → stream( T⁺ )
Lustre Compilation: normalization and scheduling

```luster
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;
```

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

• 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

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.

```plaintext
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
```

```plaintext
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
  }
}
```
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


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
  }
}
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
}
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


definition 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

\[(f_t, s_0)\]

\[S \times T^+ \rightarrow S \times T^+ \rightarrow S\]
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

<table>
<thead>
<tr>
<th>delta</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>...</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>delta</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>...</td>
</tr>
<tr>
<td>r</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>...</td>
</tr>
<tr>
<td>(c₁)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>...</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>delta</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>...</td>
</tr>
<tr>
<td>r</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>...</td>
</tr>
<tr>
<td>(c₁)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>...</td>
</tr>
<tr>
<td>r when sec</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 | ...
| (c₁)  | 0 | 0 | 1 | 3 | 4 | 6 | 9 | 9 | ...
| r when sec | | | | | | | | | |
| t     | 1 | 2 | 3 | | | | | | |
| (c₂)  | 0 | 1 | 2 | | | | | | |
Lustre: instantiation and sampling

```plaintext
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);
  in
```

<table>
<thead>
<tr>
<th>delta</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>...</td>
</tr>
<tr>
<td>r</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>...</td>
</tr>
<tr>
<td>(c₁)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>...</td>
</tr>
<tr>
<td>r when sec</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>t</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>(c₂)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>0 fby v</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>(0 fby v) when not sec</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>...</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Lustre: instantiation and sampling

```plaintext
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 | T | F | ...
| r     | 0 | 1 | 3 | 4 | 6 | 9 | 9 | 12 | ...
| (c₁)  | 0 | 0 | 1 | 3 | 4 | 6 | 9 | 9 | ...
| (c₂)  | 0 | 1 | 2 | 3 |   |   |   |   |   |
| 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 | ...
| (c₁)  | 0 | 0 | 1 | 3 | 4 | 6 | 9 | 9 | ...
| r when sec |     | 4 | 9 | 9 |     |     |     |     |     |
| t     |     | 1 | 2 | 3 |     |     |     |     |     |
| (c₂)  |     | 0 | 1 | 2 |     |     |     |     |     |
| 0 fby v  |     | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 3 | ...
| (0 fby v) when not sec |     | 0 | 0 | 0 | 4 | 4 | 4 | 3 | ...
| v     |     | 0 | 0 | 0 | 4 | 4 | 4 | 3 | ...

\[ (c_1) \]
\[ (c_2) \]
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
    }
}
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
}
}
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)
| 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 f0 _ => Control ck (AssignSt x (translate_lexp f0))
end.

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

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

Definition `translate_Reset_eqns (eqns: list equation): stmt :=` fold_left (fun i eq => Comp (translate_reset_eqn eq) i) eqns Skip.

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

SN-Lustre  \[\text{translation}\]  Obc
Correctness of translation

\[ \text{sem_node G f xss yss} \]

\[ \text{stream}(T^+) \rightarrow \text{stream}(T^+) \]
Correctness of translation

\[ \text{sem_node } G \ f \ xss \ yss \]

\[ \text{stream}(T^+) \rightarrow \text{stream}(T^+) \]

too weak for a direct proof by induction

\[ (f_t, s_0) \]

\[ S \times T^+ \rightarrow T^+ \times S \]
Correctness of translation

SN-Lustre $\xrightarrow{\text{translation}}$ Obc

$msem_node\ G\ f\ xss\ M\ yss

\text{sem_node}\ G\ f\ xss\ yss$

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

$(f_t, s_0)$

$S \times T^+ \rightarrow T^+ \times S$
Correctness of translation

\[
\text{sem}_\text{node} \ G \ f \ xss \ yss \quad \Rightarrow \quad \text{msem}_\text{node} \ G \ f \ xss \ M \ yss
\]

stream\((T^+)\) \(\rightarrow\) stream\((T^+)\)

\[S \times T^+ \rightarrow T^+ \times S\]

\((f_t, s_0)\)
Correctness of translation

$$\text{SN-Lustre} \xrightarrow{\text{translation}} \text{Obc}$$

short proof: $\exists M$

long proof

sem_node $G f xss yss$

msem_node $G f xss M yss$

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

$$(f_t, s_0)$$

$$S \times T^+ \rightarrow T^+ \times S$$
Correctness of translation

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

**Short proof:**

$$\exists M$$

**Long proof**

- Case: $x = (ce)^{ck}$
  - Case: present
  - Case: absent
- Case: $x = (f e)^{ck}$
  - Case: present
  - Case: absent
- Case: $x = (k \ fby \ e)^{ck}$
  - Case: present
  - Case: absent

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

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

$$(f_t, s_0)$$
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
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


```
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.
• **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.
- Standard technique for encapsulating state.
- Each detail entails complications in the proof.
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.
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;
};

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;
}
Standard technique for encapsulating state.

Each detail entails complications in the proof.
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.
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
    }
}

void count$reset(struct count *self) {
}

int count$step(struct count *self, int ini, int inc, _Bool res) {
}

struct count { _Bool f; int c; }

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;
}

• Standard technique for encapsulating state.
• Each detail entails complications in the proof.
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.
Correctness of generation
Correctness of generation

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

$e, le, m \leftarrow \text{Clight} \ \text{generate}(s) \Downarrow (e', le', m')$
Correctness of generation

\[ \text{Obc} \xrightarrow{\text{generation}} \text{Clight} \]

direct proof by induction on big step semantics

\[ me, ve \vdash s \Downarrow (me', ve') \quad e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m') \]
Correctness of generation

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

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

```
x := e ;
x := C.f σ (e) ;
x := e  ;  skip
```

```
x = e ;
C$f (σ, o, e) ;
x = e
Sskip
```

```
x = o.y1
x = o.y2
```

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

```
e, le, m \vdash_{Clight} generate(s) \Downarrow (e', le', m')
```
Correctness of generation

\[ x := e \]

\[ x := C.f (e) \]

\[ C$f (\sigma, o, e) \]

\[ x = o.y_1 \]

\[ x = o.y_2 \]

\[ me, ve \leftarrow s \downarrow (me', ve') \]

\[ e, le, m \leftarrow Clight \ generate(s) \downarrow (e', le', m') \]
Correctness of generation

\[ x := e \]
\[ x := C.f \sigma (e) \]
\[ x := e \]
\[ x := e \]
\[ x = e \]
\[ x = e \]
\[ x = o \cdot y_1 \]
\[ x = o \cdot y_2 \]

me, ve \(\vdash\) s \(\downarrow\) (me', ve')
e, le, m \(\vdash\) Clight generate(s) \(\downarrow\) (e', le', m')
Correctness of generation

\[ x := e \]

\[ x := C.f \sigma (e) \]

\[ x := e \quad \text{skip} \]

\[ C \$ f (\sigma, o, e) \]

\[ x = o \cdot y_1 \]

\[ x = o \cdot y_2 \]

\[ me, ve \xrightarrow{\Downarrow} (me', ve') \]

\[ e, le, m \xrightarrow{\text{Clight}} \text{generate}(s) \xrightarrow{\Downarrow} (e', le', m') \]
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:

\( \forall D G Gp P \text{ main ins outs,} \)
\[ \text{elab_declarations D = OK } (\text{exist } _ G Gp) \rightarrow \]
\[ \text{wt_ins G main ins } \rightarrow \]
\[ \text{wt_outs G main outs } \rightarrow \]
\[ \text{sem_node G main } (\text{vstr ins})(\text{vstr outs}) \rightarrow \]
\[ \text{compile D main } = \text{OK } P \rightarrow \]
\[ \exists T, \text{program_behaves} (\text{Asm.semantics P})(\text{Reacts T}) \]
\[ \land \text{bisim_io G main ins outs T}. \]
Theorem behavior_asm:
\[\forall D G Gp P \text{ main } \text{ ins } \text{ outs},\]
\[\text{elab_declarations } D = \text{ OK } (\text{exist } \_ G Gp) \rightarrow\]
\[\text{wt}_\text{ins } G \text{ main } \text{ ins } \rightarrow\]
\[\text{wt}_\text{outs } G \text{ main } \text{ outs } \rightarrow\]
\[\text{sem}_\text{node } G \text{ main } (\text{vstr } \text{ ins})(\text{vstr } \text{ outs}) \rightarrow\]
\[\text{compile } D \text{ main } = \text{ OK } P \rightarrow\]
\[\exists T, \text{ program}\_\text{behaves} (\text{Asm}\_\text{semantics } P)(\text{Reacts } T)\]
\[\land \text{bisim}_\text{io } G \text{ main } \text{ ins } \text{ outs } T.\]
Theorem behavior_asm:

\[\forall D G Gp P \text{ main ins outs},\]
\[\text{elab_declarations } D = \text{OK (exist } _G Gp) \rightarrow\]
\[\text{wt_ins } G \text{ main ins }\rightarrow\]
\[\text{wt Outs } G \text{ main outs }\rightarrow\]
\[\text{sem_node } G \text{ main (vstr ins) (vstr outs)} \rightarrow\]
\[\text{compile } D \text{ main } = \text{OK P }\rightarrow\]
\[\exists T, \text{ program_behaves (Asm.semantics } P) (\text{Reacts T})\]
\[\land \text{ bisim_io } G \text{ main ins outs } T.\]
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.
**Theorem** behavior_asm:

\[
\forall D G Gp P \text{ main ins outs,} \\
\text{elab_declarations } D = \text{OK (exist } G Gp) \rightarrow \\
\text{wt_ins } G \text{ main ins } \rightarrow \\
\text{wt_outs } G \text{ main outs } \rightarrow \\
\text{sem_node } G \text{ main (vstr ins)(vstr outs) } \rightarrow \\
\text{compile } D \text{ main = OK } P \rightarrow \\
\exists T, \text{ program_behaves (Asm.semantics } P) (\text{Reacts } T) \\
\wedge \text{bisim_io } G \text{ main ins outs } T.
\]
Theorem behavior_asm:
\[ \forall D \ G \ Gp \ P \ \text{main} \ \text{ins} \ \text{outs}, \]
\[ \text{elab_declarations} \ D = \text{OK} (\exists \ G \ Gp) \rightarrow \]
\[ \text{wt_ins} \ G \ \text{main} \ \text{ins} \rightarrow \]
\[ \text{wt_outs} \ G \ \text{main} \ \text{outs} \rightarrow \]
\[ \text{sem_node} \ G \ \text{main} \ (\text{vstr} \ \text{ins}) (\text{vstr} \ \text{outs}) \rightarrow \]
\[ \text{compile} \ D \ \text{main} = \text{OK} \ P \rightarrow \]
\[ \exists \ T, \ \text{program_behaves} (\text{Asm.semantics} \ P) (\text{Reacts} \ T) \]
\[ \wedge \ \text{bisim_io} \ G \ \text{main} \ \text{ins} \ \text{outs} \ T. \]

typing/elaboration succeeds,

\[ \forall \ \text{well typed input and output streams} \ldots \]

...related by the dataflow semantics,

if compilation succeeds,

then, the generated assembly produces an infinite trace...
Theorem `behavior_asm`:
\[ \forall D G Gp P \text{ main } \text{ ins} \text{ outs}, \]
\[ \text{elab_declarations } D = \text{OK} (\text{exist } _G Gp) \rightarrow \]
\[ \text{wt_ins } G \text{ main } \text{ ins} \rightarrow \]
\[ \text{wt_outs } G \text{ main } \text{ outs} \rightarrow \]
\[ \text{sem_node } G \text{ main } (\text{vstr } \text{ins}) (\text{vstr } \text{outs}) \rightarrow \]
\[ \text{compile } D \text{ main } = \text{OK} P \rightarrow \]
\[ \exists T, \text{ program_behaves } (\text{Asm.semantics } P) (\text{Reacts } T) \]
\[ \land \text{bisim_io } G \text{ main } \text{ ins} \text{ outs} T. \]

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
**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**

<table>
<thead>
<tr>
<th>Vélus</th>
<th>Heptagon+CC</th>
<th>Heptagon+gcc</th>
<th>Lus6+CC</th>
<th>Lus6+gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>avgvelocity</td>
<td>315</td>
<td>265</td>
<td>300</td>
<td>250</td>
</tr>
<tr>
<td>count</td>
<td>55</td>
<td>55 (6%)</td>
<td>25 (4%)</td>
<td>25 (4%)</td>
</tr>
<tr>
<td>tracker</td>
<td>680</td>
<td>790 (16%)</td>
<td>530 (25%)</td>
<td>500 (20%)</td>
</tr>
<tr>
<td>pip_ex</td>
<td>4,415</td>
<td>4,065 (7%)</td>
<td>2,565 (45%)</td>
<td>2,040 (44%)</td>
</tr>
<tr>
<td>mp_longitudinal [16]</td>
<td>5,525</td>
<td>6,646 (17%)</td>
<td>3,465 (57%)</td>
<td>2,835 (80%)</td>
</tr>
<tr>
<td>cruise [54]</td>
<td>1,760</td>
<td>1,875 (6%)</td>
<td>1,230 (69%)</td>
<td>1,230 (69%)</td>
</tr>
<tr>
<td>risingged/gegerger [19]</td>
<td>285</td>
<td>300 (4%)</td>
<td>190 (65%)</td>
<td>190 (65%)</td>
</tr>
<tr>
<td>chrono [20]</td>
<td>410</td>
<td>425 (3%)</td>
<td>305 (72%)</td>
<td>305 (72%)</td>
</tr>
<tr>
<td>watchdog3 [26]</td>
<td>610</td>
<td>575 (5%)</td>
<td>355 (41%)</td>
<td>310 (40%)</td>
</tr>
<tr>
<td>functionalchain [17]</td>
<td>11,550</td>
<td>13,515 (17%)</td>
<td>8,545 (68%)</td>
<td>7,525 (58%)</td>
</tr>
<tr>
<td>landing_gear [11]</td>
<td>9,660</td>
<td>8,475 (12%)</td>
<td>5,880 (57%)</td>
<td>5,810 (56%)</td>
</tr>
<tr>
<td>minus [57]</td>
<td>890</td>
<td>900 (1%)</td>
<td>580 (64%)</td>
<td>580 (64%)</td>
</tr>
<tr>
<td>prodcell [32]</td>
<td>1,020</td>
<td>990 (2%)</td>
<td>620 (60%)</td>
<td>410 (69%)</td>
</tr>
<tr>
<td>ums_verif [57]</td>
<td>2,590</td>
<td>2,285 (13%)</td>
<td>1,380 (48%)</td>
<td>920 (66%)</td>
</tr>
</tbody>
</table>

**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 (gccci). 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 conduction 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 scans inputs and prints 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 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 (≈60 000 nodes, ≈162 000 equations, ≈12 MB source file without comments). The source code was already normalized since it was generated with a graphical interface, usually more valuable than raw performance numbers. We compiled with CompCert 2.6 and GCC 4.4.8 -O1 (for the arm-none-eabi target) with a hardware floating-point unit (vfp/3-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 prod 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].
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

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


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.
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.
Unannotated Lustre → elaboration → Lustre → normalization → N-Lustre → scheduling → SN-Lustre

(normalized) elaboration

dataflow
imperative

translation
fusion optimization

Obc

Clight
CompCert
Assembly

generation
compilation
printing


References II


