Verifying a Lustre Compiler (Part 1)

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

1. INRIA Paris
2. DI, École normale supérieure
3. CNRS
4. Univ. Pierre et Marie Curie
5. Yale University

SYNCHRON Workshop, Bamberg—December 2016

What are we doing?

- Implementing a Lustre compiler in the Coq Interactive Theorem Prover
- Proving that the generated code implements the dataflow semantics

(Part of the ITEA 3 14014 ASSUME Project.)

What are we doing?

- Implementing a Lustre compiler in the Coq Interactive Theorem Prover
- Proving that the generated code implements the dataflow semantics

(Part of the ITEA 3 14014 ASSUME Project.)

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 Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

What are we doing?

- Implementing a Lustre compiler in the Coq Interactive Theorem Prover
- Proving that the generated code implements the dataflow semantics

(Part of the ITEA 3 14014 ASSUME Project.)

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 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

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

What are we doing?

- Implementing a Lustre compiler in the Coq Interactive Theorem Prover
- Proving that the generated code implements the dataflow semantics

(Part of the ITEA 3 14014 ASSUME Project.)

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 Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

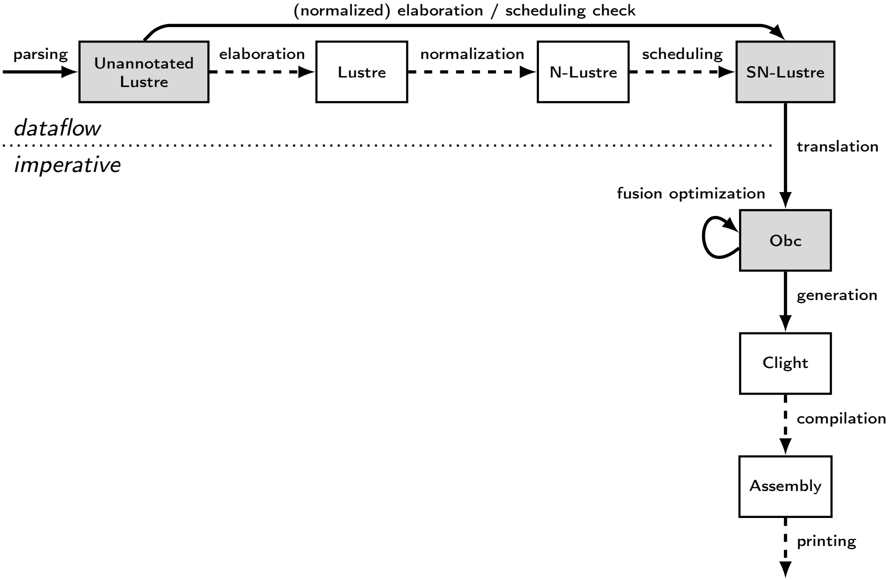
  CompCert: a formal model and compiler for a subset of C

  - A generic machine-level model of execution and memory
  - A verified path to assembly code

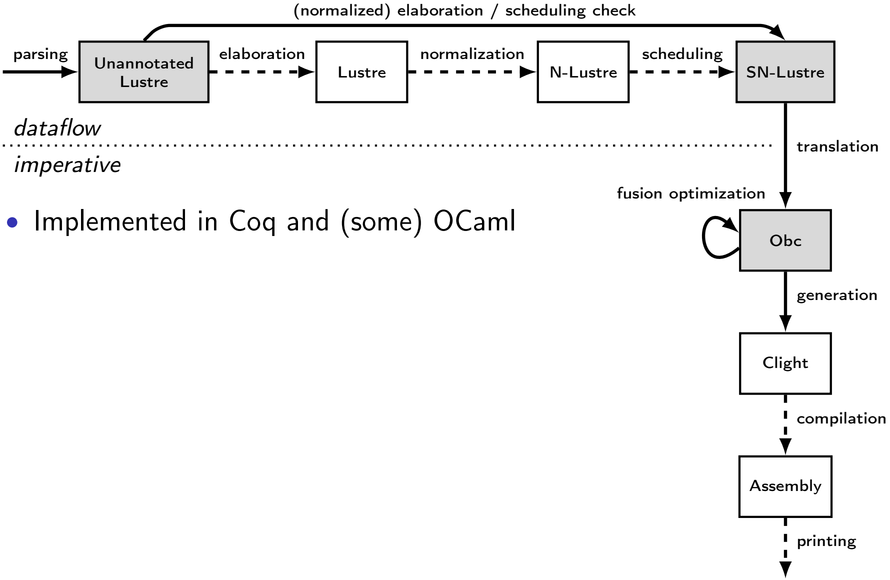
  [Blazy, Dargaye, and Leroy (2006): "Formal Verification of a C Compiler Front-End"] [Leroy (2009): "Formal verification of a realistic compiler"]

- Computer assistance is all but essential for such detailed models.

The Vélus Lustre Compiler

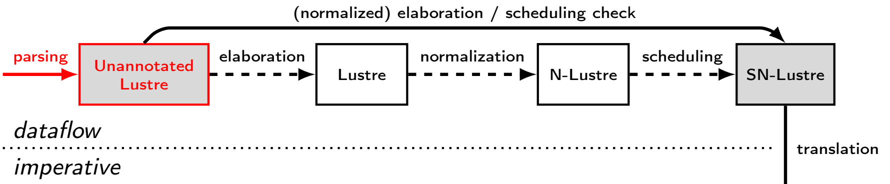


The Vélus Lustre Compiler



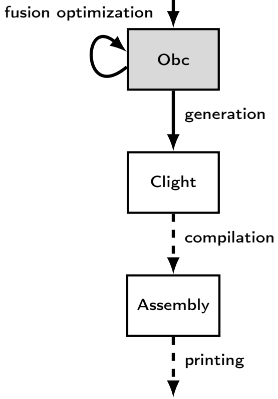
- Implemented in Coq and (some) OCaml

The Vélus Lustre Compiler

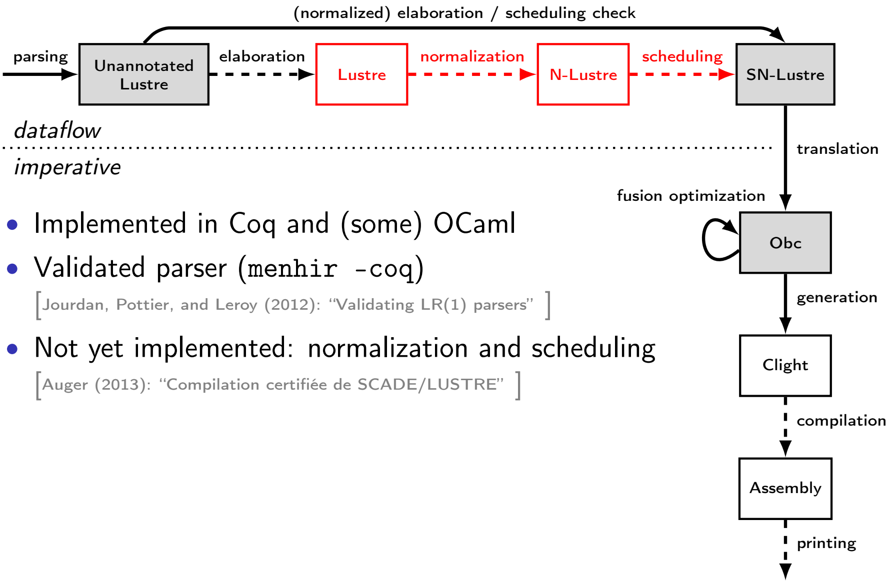


- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`)

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



The Vélus Lustre Compiler

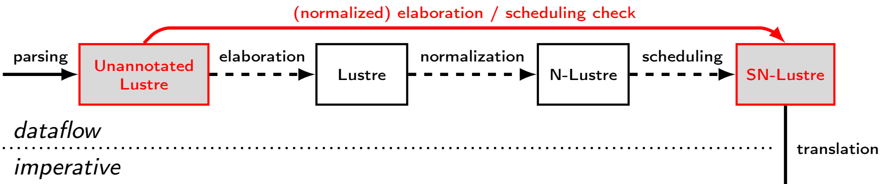


- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`)

  [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization and scheduling

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

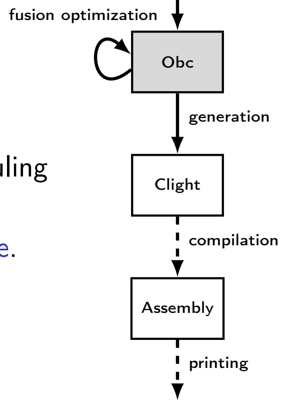
The Vélus Lustre Compiler



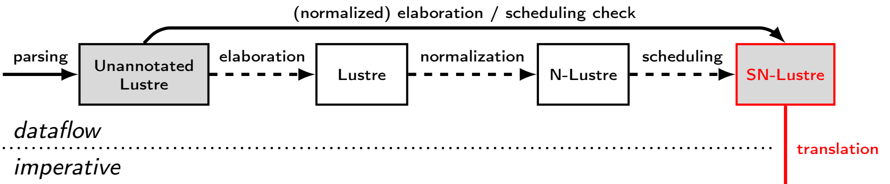
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`)

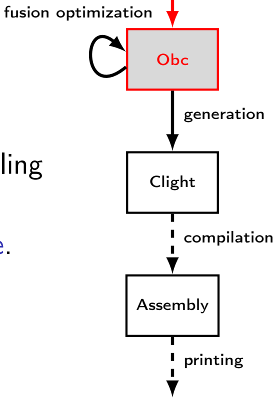
  [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization and scheduling

  [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Scheduled and Normalized Lustre.



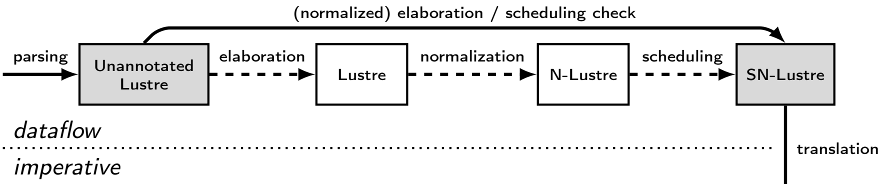
The Vélus Lustre Compiler



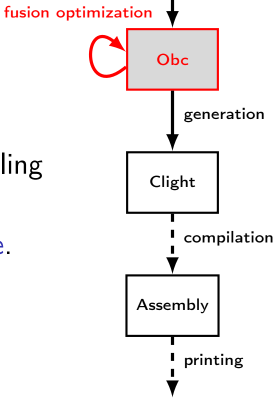


- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`)
  [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization and scheduling
  [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Scheduled and Normalized Lustre.
-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

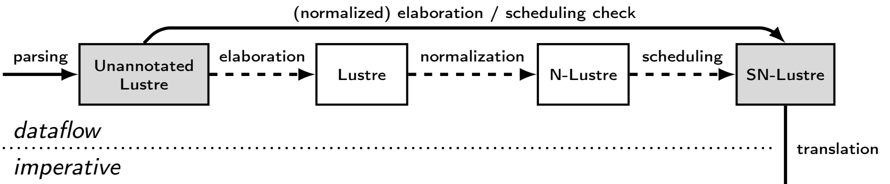
The Vélus Lustre Compiler

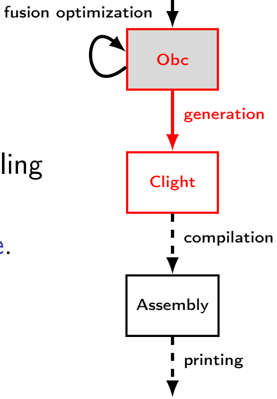


- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`)
  [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]
- Not yet implemented: normalization and scheduling
  [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Scheduled and Normalized Lustre.
-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



The Vélus Lustre Compiler





- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`)

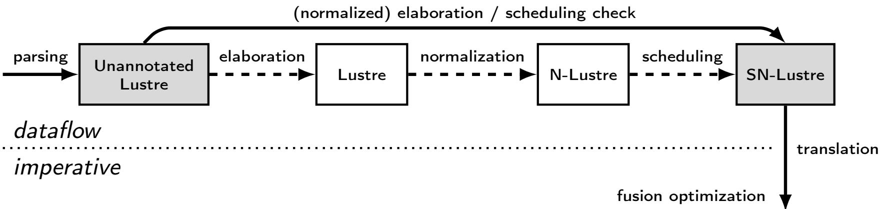
  [Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"]

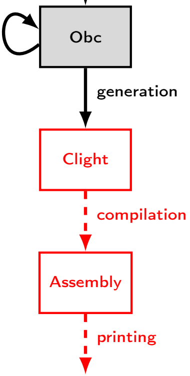
- Not yet implemented: normalization and scheduling

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

- Elaboration to Scheduled and Normalized Lustre.
- 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 and scheduling
  [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Elaboration to Scheduled and Normalized Lustre.
- Translation to intermediate Obc code.
- Optimization of intermediate Obc code.
- Generation of CompCert Clight code.
- Rely on CompCert for compilation.

Lustre 30 years later? [Caspi et al. (1987): "LUSTRE: A declarative language for programming synchronous systems"]

Not quite...

- No `pre`: use `fby`, avoid initialization analysis for now
- No sub-clocking on inputs or outputs
- No `current`: use (binary) `merge`
- No external calls

Lustre 30 years later? [Caspi et al. (1987): "LUSTRE: A declarative language for programming synchronous systems"]

Not quite. . .

- No `pre`: use `fby`, avoid initialization analysis for now
- No sub-clocking on inputs or outputs
- No `current`: use (binary) `merge`
- No external calls

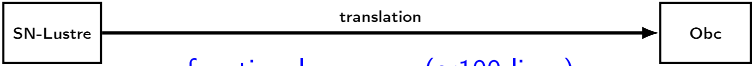
Two talks

1. Tim:
   - Overview
   - Translation correctness: SN-Lustre to Obc (recap)
   - Control-fusion optimization
   - Integration of Clight operators
2. Lélio:
   - Obc to Clight
   - Demo

Outline

Translation of SN-Lustre to Obc

Translation of SN-Lustre to Obc



functional program (≈100 lines)

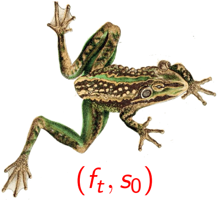
Translation of SN-Lustre to Obc

SN-Lustre —translation→ Obc

functional program (≈100 lines)

`sem_node G f xss yss`

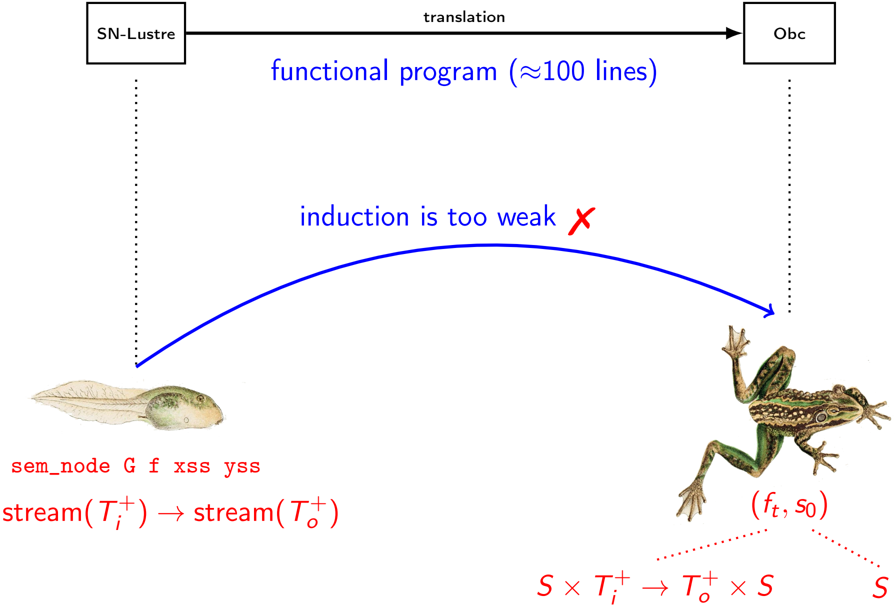
$\text{stream}(T_i^+) \rightarrow \text{stream}(T_o^+)$



(f_t, s_0)

$S \times T_i^+ \rightarrow T_o^+ \times S$ $\qquad S$

Translation of SN-Lustre to Obc



Translation of SN-Lustre to Obc

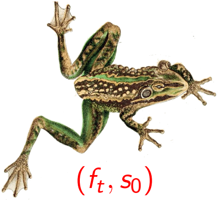
SN-Lustre —translation→ Obc

functional program (≈100 lines)

`sem_node G f xss yss`

$\text{stream}(T_i^+) \rightarrow \text{stream}(T_o^+)$

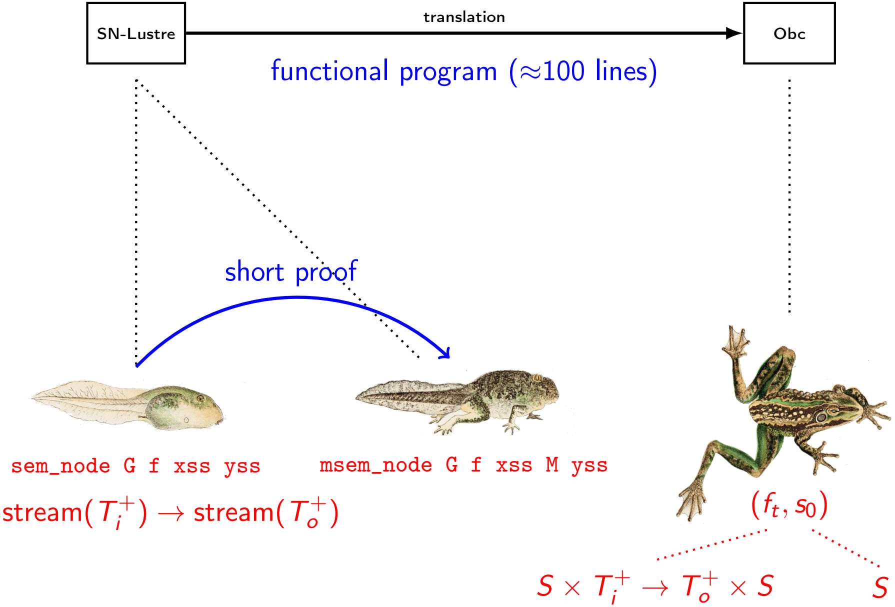
`msem_node G f xss M yss`



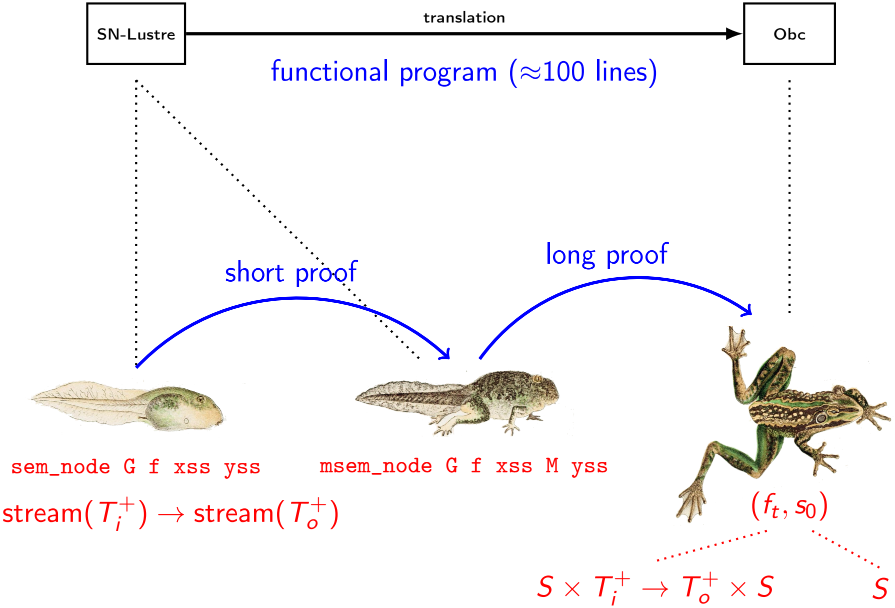
(f_t, s_0)

$S \times T_i^+ \rightarrow T_o^+ \times S$ $\qquad S$

Translation of SN-Lustre to Obc



Translation of SN-Lustre to Obc

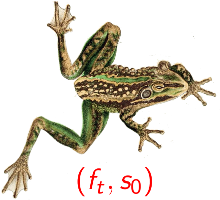


Translation of SN-Lustre to Obc

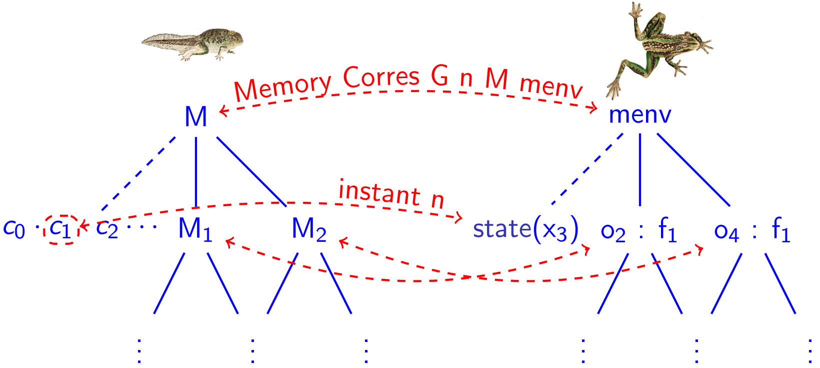
induction n

- 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 to Obc: main invariant



- 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

Outline

Fusion of control structures [Biernacki et al. (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 := mem(w)
  };
  mem(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 := mem(w)
  };


  mem(w) := v
}
```

- Generate control for each equation (simpler to implement and prove).
- Afterward fuse control structures together.
- Effective if scheduler places similarly clocked equations together.

Fusion of control structures [Biernacki et al. (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 := mem(w)
  };
  mem(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 := mem(w)
  };


  mem(w) := v
}
```

- Generate control for each equation (simpler to implement and prove).
- Afterward fuse control structures together.
- Effective if scheduler places similarly clocked equations together.

We also define the function $Join(.,.)$ which merges two control structures gathered by the same guards:

$$\begin{aligned}
&Join(\texttt{ case } (x)\ \{C_1 : S_1; ...; C_n : S_n\},\\
&\qquad \texttt{case } (x)\ \{C_1 : S'_1; ...; C_n : S'_n\})\\
&\quad = \texttt{case } (x)\ \{C_1 : Join(S_1, S'_1); ...; C_n : Join(S_n, S'_n)\}\\
&Join(S_1, S_2) = S_1; S_2
\end{aligned}$$

$$\begin{aligned}
&JoinList(S) = S\\
&JoinList(S_1, ..., S_n) = Join(S_1, JoinList(S_2, ..., S_n))
\end{aligned}$$

[Biernacki et al. (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

We also define the function $Join(.,.)$ which merges two control structures gathered by the same guards:

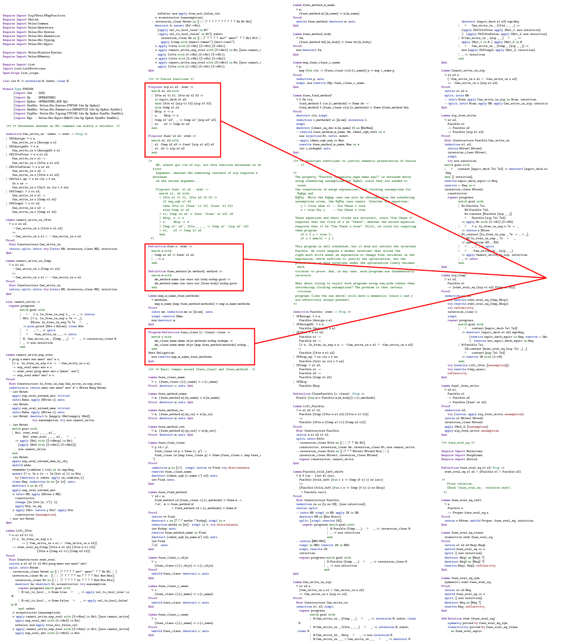
$$
\begin{aligned}
&Join(\ \texttt{case}\ (x)\ \{C_1 : S_1; ...; C_n : S_n\},\\
&\qquad \texttt{case}\ (x)\ \{C_1 : S'_1; ...; C_n : S'_n\})\\
&\quad = \texttt{case}\ (x)\ \{C_1 : Join(S_1, S'_1); ...; C_n : Join(S_n, S'_n)\}\\
&Join(S_1, S_2) = S_1; S_2
\end{aligned}
$$

$$
\begin{aligned}
&JoinList(S) = S\\
&JoinList(S_1, ..., S_n) = Join(S_1, JoinList(S_2, ..., S_n))
\end{aligned}
$$

```
Fixpoint zip s1 s2 : stmt :=
  match s1, s2 with
  | Ifte e1 t1 f1, Ifte e2 t2 f2 ⇒
    if equiv_decb e1 e2
    then Ifte e1 (zip t1 t2) (zip f1 f2)
    else Comp s1 s2
  | Skip, s ⇒ s
  | s,    Skip ⇒ s
  | Comp s1' s2', _ ⇒ Comp s1' (zip s2' s2)
  | s1,   s2 ⇒ Comp s1 s2
  end.

Fixpoint fuse' s1 s2 : stmt :=
  match s1, s2 with
  | s1, Comp s2 s3 ⇒ fuse' (zip s1 s2) s3
  | s1, s2 ⇒ zip s1 s2
  end.

Definition fuse s : stmt :=
  match s with
  | Comp s1 s2 ⇒ fuse' s1 s2
  | _ ⇒ s
  end.
```



```
Fixpoint zip s1 s2 : stmt :=
  match s1, s2 with
  | Ifte e1 t1 f1, Ifte e2 t2 f2 ⇒
    if equiv_decb e1 e2
    then Ifte e1 (zip t1 t2) (zip f1 f2)
    else Comp s1 s2
  | Skip, s ⇒ s
  | s,    Skip ⇒ s
  | Comp s1' s2', _ ⇒ Comp s1' (zip s2' s2)
  | s1,    s2 ⇒ Comp s1 s2
  end.

Fixpoint fuse' s1 s2 : stmt :=
  match s1, s2 with
  | s1, Comp s2 s3 ⇒ fuse' (zip s1 s2) s3
  | s1, s2 ⇒ zip s1 s2
  end.

Definition fuse s : stmt :=
  match s with
  | Comp s1 s2 ⇒ fuse' s1 s2
  | _ ⇒ s
  end.
```

```
Require Import Coq.FSets.FMapPositive.
Require Import PArith.
Require Import Velus.Common.
Require Import Velus.Operators.
Require Import Velus.Obc.Syntax.
Require Import Velus.Obc.Semantics.
Require Import Velus.Obc.Typing.
Require Import Velus.Obc.Equiv.

Require Import Velus.NLustre.Syntax.
Require Import Velus.RMemory.

Require Import List.
Import List.ListNotations.
Open Scope list_scope.

Ltac inv H := inversion H; subst; clear H.

Module Type FUSION
       (Import Ids  : IDS)
       (Import Op   : OPERATORS)
       (Import OpAux : OPERATORS_AUX Op)
       (Import SynObc: Velus.Obc.Syntax.SYNTAX Ids Op OpAux)
       (Import SemObc: Velus.Obc.Semantics.SEMANTICS Ids Op OpAux SynObc)
       (Import TypObc: Velus.Obc.Typing.TYPING Ids Op OpAux SynObc SemObc)
       (Import Equ  : Velus.Obc.Equiv.EQUIV Ids Op OpAux SynObc SemObc).

  (** ** Determine whether an Obc command can modify a variable. *)

  Inductive Can_write_in : ident → stmt → Prop :=
  | CWIAssign: ∀ x e,
      Can_write_in x (Assign x e)
  | CWIAssignSt: ∀ x e,
      Can_write_in x (AssignSt x e)
  | CWIIfteTrue: ∀ x e s1 s2,
      Can_write_in x s1 →
      Can_write_in x (Ifte e s1 s2)
  | CWIIfteFalse: ∀ x e s1 s2,
      Can_write_in x s2 →
      Can_write_in x (Ifte e s1 s2)
  | CWICall_ap: ∀ x xs cls i f es,
      In x xs →
      Can_write_in x (Call xs cls i f es)
  | CWIComp1: ∀ x s1 s2,
      Can_write_in x s1 →
      Can_write_in x (Comp s1 s2)
  | CWIComp2: ∀ x s1 s2,
      Can_write_in x s2 →
      Can_write_in x (Comp s1 s2).

  Lemma cannot_write_in_Ifte:
    ∀ x e s1 s2,
      ¬ Can_write_in x (Ifte e s1 s2)
      ↔
      ¬ Can_write_in x s1 ∧ ¬ Can_write_in x s2.
  Proof.
    Hint Constructors Can_write_in.
    intros; split; intro; try (intro HH; inversion_clear HH); intuition.
  Qed.

[illegible]

  (** ** Fusion functions *)

  Fixpoint zip s1 s2 : stmt :=
    match s1, s2 with
    | Ifte e1 t1 f1, Ifte e2 t2 f2 =>
      if equiv_decb e1 e2
      then Ifte e1 (zip t1 t2) (zip f1 f2)
      else Comp s1 s2
    | Skip, s => s
    | s,    Skip => s
    | Comp s1' s2',  _ => Comp s1' (zip s2' s2)
    | s1,    s2 => Comp s1 s2
    end.

  Fixpoint fuse' s1 s2 : stmt :=
    match s1, s2 with
    | s1, Comp s2 s3 => fuse' (zip s1 s2) s3
    | s1, s2 => zip s1 s2
    end.

  (*
    NB: almost got rid of zip, but this function decreases on the first
    argument, whereas the remaining instance of zip requires a decrease
    on the second argument...

    Fixpoint fuse' s1 s2 : stmt :=
      match s1, s2 with
[illegible]
          then Ifte e1 (fuse' t1 t2) (fuse' f1 f2)
          else Comp s1 s2
      | s1, Comp s2 s3 => fuse' (fuse' s1 s2) s3
      | Skip, s => s
      | s,    Skip => s
      | Comp s1' s2', Ifte _ _ _ => Comp s1' (zip s2' s2)
      | s1,   s2 => Comp s1 s2
      end.
  *)

  Definition fuse s : stmt :=
    match s with
    | Comp s1 s2 => fuse' s1 s2
    | _ => s
    end.

  Definition fuse_method (m: method): method :=
    match m with
      mk_method name ins vars out body nodup good =>
      mk_method name ins vars out (fuse body) nodup good
    end.

  Lemma map_m_name_fuse_methods:
    ∀ methods,
      map m_name (map fuse_method methods) = map m_name methods.
  Proof.
    intro ms; induction ms as [|m ms]; auto.
    simpl. rewrite IHms.
    now destruct m.
  Qed.

  Program Definition fuse_class (c: class): class :=
    match c with
      mk_class name mems objs methods nodup nodupm =>
      mk_class name mems objs (map fuse_method methods) nodup _
    end.
  Next Obligation.
    now rewrite map_m_name_fuse_methods.
  Qed.

  (** ** Basic lemmas around [fuse_class] and [fuse_method]. *)

  Lemma fuse_class_name:
    ∀ c, (fuse_class c).(c_name) = c.(c_name).
  Proof. destruct c; auto. Qed.

  Lemma fuse_method_name:
    ∀ m, (fuse_method m).(m_name) = m.(m_name).

[illegible]

    trickier to prove. And, in any case, such programs are fundamentally
    incorrect.

    What about trying to reject such programs using sem_node rather than
    introducing clocking assumptions? The problem is that certain
     circular
    programs (like the one above) still have a semantics (since x and y
    are effectively always present).
  *)

  Inductive Fusible : stmt → Prop :=
  | IFAssign: ∀ x e,
      Fusible (Assign x e)
  | IFAssignSt: ∀ x e,
      Fusible (AssignSt x e)
  | IFIfte: ∀ e s1 s2,
      Fusible s1 →
      Fusible s2 →
      (∀ x, Is_free_in_exp x e → ¬Can_write_in x s1 ∧ ¬Can_write_in x s2)
     →
      Fusible (Ifte e s1 s2)
  | IFStep_ap: ∀ xs cls i f es,
      Fusible (Call xs cls i f es)
  | IFComp: ∀ s1 s2,
      Fusible s1 →
      Fusible s2 →
      Fusible (Comp s1 s2)
  | IFSkip:
      Fusible Skip.

  Definition ClassFusible (c: class) : Prop :=
    Forall (fun m=> Fusible m.(m_body)) c.(c_methods).

  Lemma lift_Fusible:
    ∀ e s1 s2 t1 t2,
      Fusible (Comp (Ifte e s1 s2) (Ifte e t1 t2))
      ↔
      Fusible (Ifte e (Comp s1 t1) (Comp s2 t2)).
  Proof.
    Hint Constructors Fusible.
    intros e s1 s2 t1 t2.
    split; intro Hifv.

[illegible]
```

Fusion of control structures: requires invariant

```
if e then {s1} else {s2};
if e then {t1} else {t2}
```

⇒

```
if e then {s1; t1} else {s2; t2};
```

Fusion of control structures: requires invariant

```
if e then {s1} else {s2};
if e then {t1} else {t2}
```

➡ `if e then {s1; t1} else {s2; t2};`

```
if x then {x := false} else {x := true};
if x then {t1} else {t2}
```

✗

Fusion of control structures: requires invariant

```
if e then {s1} else {s2};
if e then {t1} else {t2}
```

➠ `if e then {s1; t1} else {s2; t2};`

```
if x then {x := false} else {x := true};
if x then {t1} else {t2}
```

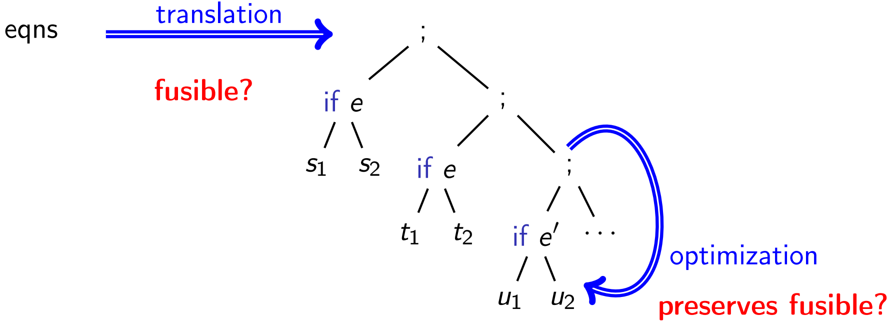
✗

$$\frac{\text{fusible}(s_1) \qquad \text{fusible}(s_2) \qquad \forall x \in \text{free}(e), \neg\text{maywrite } x\ s_1 \wedge \neg\text{maywrite } x\ s_2}{\text{fusible}(\text{if } e \text{ then } \{s_1\} \text{ else } \{s_2\})}$$

$$\frac{\text{fusible}(s_1) \quad \text{fusible}(s_2)}{\text{fusible}(s_1; s_2)}$$

...

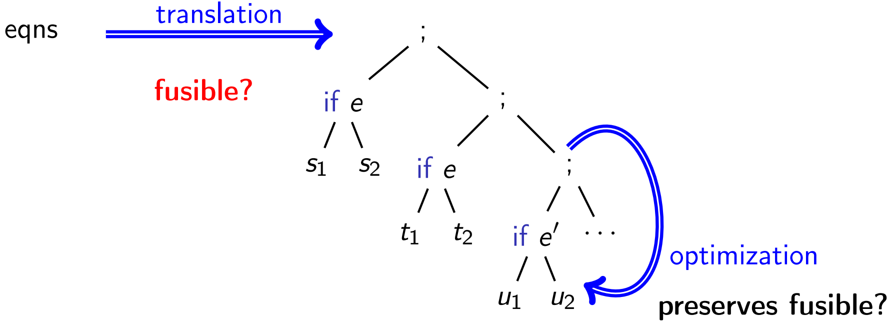
Fusion of control structures: correctness



General Schema

- Implement optimization as a function on code.
- Find invariant under which the semantics is preserved:
  - Satisfied by the generated code.
  - Preserved by (components of) the optimization.

Fusion of control structures: correctness

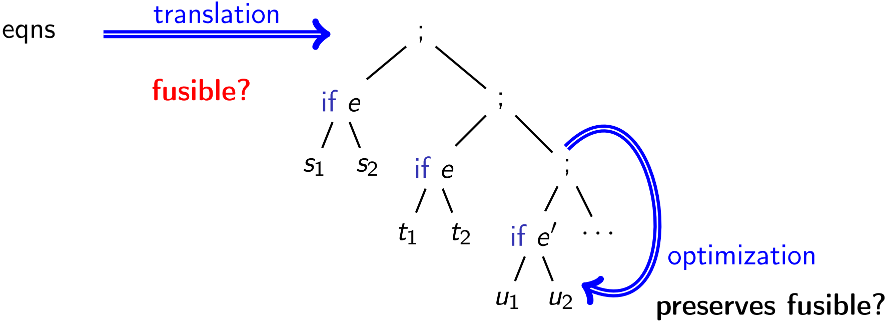


$x = (\text{merge b e1 e2})^{\text{base on ck}}$

```
if ck then {
  if b then {
    x := e1
  } else {
    x := e2
  }
}
```

- In a well scheduled dataflow program it is not possible to read x before writing it.
- Compiling $x = (ce)^{ck}$ and $x = (f\ le)^{ck}$ gives fusible imperative code.

Fusion of control structures: correctness

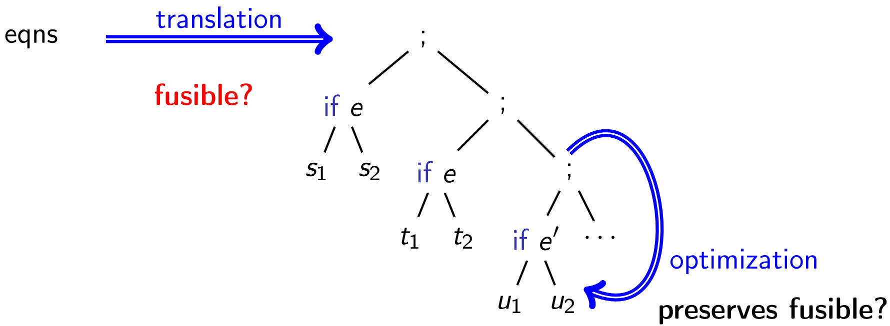


$\mathsf{x} = (0 \text{ fby } (\mathsf{x} + 1))^{\text{base on ck}}$

```
if ck then {
  mem(x) := mem(x) + 1
}
```

- But for fby equations, we must read x before writing it.
- A different invariant?
  Once we write x, we never read it again.
  Trickier to express. Trickier to work with.

Fusion of control structures: correctness



$y = (\text{true when } x)^{\text{base on } x}$
$x = (\text{true fby } y)^{\text{base on } x}$

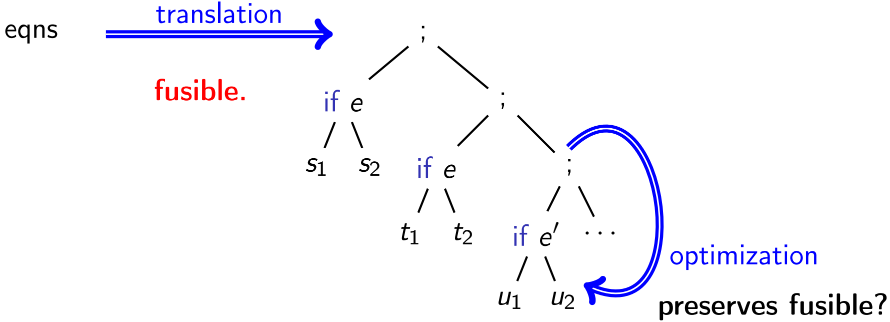
```
if mem(x) then {
  y := true
}
if mem(x) then {
  mem(x) := y
}
```

- Happily, such programs are not well clocked.

$$\frac{C \vdash \mathit{true} :: \mathit{base} \qquad C \vdash x :: \mathit{base}}{C \vdash \mathit{true}\ \texttt{when}\ x :: \mathit{base}\ \text{on}\ (x = T)}$$

$$C \vdash x :: \mathit{base}\ \text{on}\ (x = T)$$

Fusion of control structures: correctness



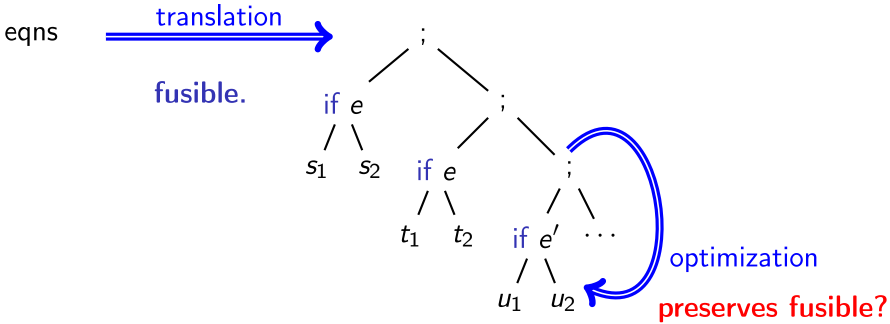
$y = (\text{true when } x)^{\text{base on } x}$

$x = (\text{true fby } y)^{\text{base on } x}$

```
if mem(x) then {
  y := true
}
if mem(x) then {
  mem(x) := y
}
```

- Happily, such programs are not well clocked.
- Show that a variable x is never free in its own clock in a well clocked program: $C \not\vdash x :: \mathit{base} \text{ on } \cdots \text{ on } x \text{ on } \cdots$
- Compiling $x = (\text{v0 fby le})^{ck}$ also gives fusible imperative code.

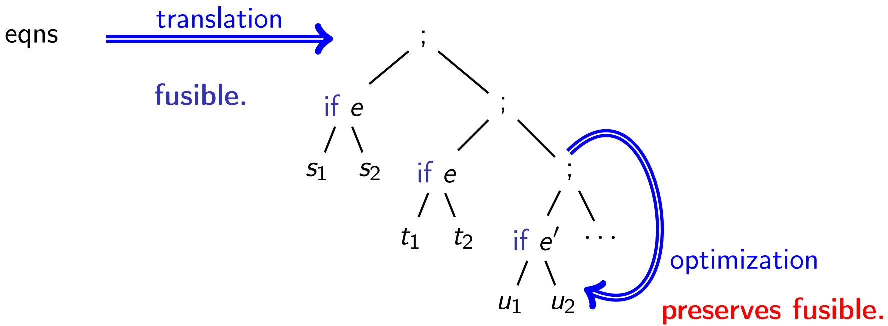
Fusion of control structures: correctness



- Define $s_1 \approx_{eval} s_2$

```
Definition stmt_eval_eq s1 s2: Prop :=
  ∀ prog menv env menv' env',
    stmt_eval prog menv env s1 (menv', env')
    ↔
    stmt_eval prog menv env s2 (menv', env').
```

Fusion of control structures: correctness



- Define $s_1 \approx_{eval} s_2$
- Define $s_1 \approx_{fuse} s_2$ as $s_1 \approx_{eval} s_2 \wedge \text{fusible}(s_1) \wedge \text{fusible}(s_2)$
- Show congruence for ;/fuse/fuse'/zip.
- Proofs by rewriting to get: $$\frac{\text{fusible}(s)}{\text{fuse}\,(s) \approx_{eval} s}$$

Outline

Verifying Lustre compilation in Coq

Translation correctness: SN-Lustre to Obc

Fusion of control structures

Integrating Clight operators into N-Lustre and Obc

Conclusion

- Introduce an abstract interface for values, types, and operators.
    - Define SN-Lustre and Obc syntax and semantics against this interface.
    - Likewise for the SN-Lustre to Obc translation and proof.
- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.
```

- Introduce an abstract interface for values, types, and operators.
  - Define SN-Lustre and Obc syntax and semantics against this interface.
  - Likewise for the SN-Lustre to Obc translation and proof.
- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
 Axiom true_not_false_val :
   true_val <> false_val.
```

- Introduce an abstract interface for values, types, and operators.
  - Define SN-Lustre and Obc syntax and semantics against this interface.
  - Likewise for the SN-Lustre to Obc translation and proof.
- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
 Axiom true_not_false_val :
   true_val <> false_val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.
```

- Introduce an abstract interface for values, types, and operators.
    - Define SN-Lustre and Obc syntax and semantics against this interface.
    - Likewise for the SN-Lustre to Obc translation and proof.
- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
 Axiom true_not_false_val :
   true_val <> false_val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.

 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)
End OPERATORS.
```

- Introduce an abstract interface for values, types, and operators.
  - Define SN-Lustre and Obc syntax and semantics against this interface.
  - Likewise for the SN-Lustre to Obc translation and proof.

- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
 Axiom true_not_false_val :
   true_val <> false_val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.

 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)
End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.
```

```
Inductive val: Type :=
  | Vundef  : val
  | Vint    : int → val
  | Vlong   : int64 → val
  | Vfloat  : float → val
  | Vsingle : float32 → val
  | Vptr    : block → int → val.
```

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
 Axiom true_not_false_val :
   true_val <> false_val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.

 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
       → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)
End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

 Inductive type : Type :=
 | Tint   : intsize → signedness → type
 | Tlong  : signedness → type
 | Tfloat : floatsize → type.

 Inductive signedness : Type :=
   | Signed   : signedness
   | Unsigned : signedness.

 Inductive intsize : Type :=
   | I8    : intsize    (* char  *)
   | I16   : intsize    (* short *)
   | I32   : intsize    (* int   *)
   | IBool : intsize.   (* bool  *)

 Inductive floatsize : Type :=
   | F32 : floatsize    (* float  *)
   | F64 : floatsize.   (* double *)
```

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
 Axiom true_not_false_val :
   true_val <> false_val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.

 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)
End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

 Inductive type : Type :=
 | Tint   : intsize → signedness → type
 | Tlong  : signedness → type
 | Tfloat : floatsize → type.

 Inductive const : Type :=
 | Cint    : int → intsize → signedness → const
 | Clong   : int64 → signedness → const
 | Cfloat  : float → const
 | Csingle : float32 → const.
```

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
 Axiom true_not_false_val :
   true_val <> false_val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.

 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)
End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

 Inductive type : Type :=
 | Tint   : intsize → signedness → type
 | Tlong  : signedness → type
 | Tfloat : floatsize → type.

 Inductive const : Type :=
 | Cint    : int → intsize → signedness → const
 | Clong   : int64 → signedness → const
 | Cfloat  : float → const
 | Csingle : float32 → const.

 Definition true_val  := Vtrue.  (* Vint Int.one  *)
 Definition false_val := Vfalse. (* Vint Int.zero *)

 Lemma true_not_false_val: true_val <> false_val.
 Proof. discriminate. Qed.

 Definition bool_type : type := Tint IBool Signed.
```

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
 Axiom true_not_false_val :
   true_val <> false_val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.

 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
       → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)
End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

 Inductive type : Type :=
 | Tint   : intsize → signedness → type
 | Tlong  : signedness → type
 | Tfloat : floatsize → type.

 Inductive const : Type :=
 | Cint    : int → intsize → signedness → const
 | Clong   : int64 → signedness → const
 | Cfloat  : float → const
 | Csingle : float32 → const.

 Definition true_val  := Vtrue.  (* Vint Int.one  *)
 Definition false_val := Vfalse. (* Vint Int.zero *)

 Lemma true_not_false_val: true_val <> false_val.
 Proof. discriminate. Qed.

 Definition bool_type : type := Tint IBool Signed.

 Inductive unop : Type :=
 | UnaryOp: Cop.unary_operation → unop
 | CastOp:  type → unop.

 Definition binop := Cop.binary_operation.

 Definition sem_unop (uop: unop) (v: val) (ty: type) : option val
 := match uop with
    | UnaryOp op ⇒ sem_unary_operation op v (cltype ty) Mem.empty
    | CastOp ty' ⇒ sem_cast v (cltype ty) (cltype ty') Mem.empty
    end.

 (* ... *)
End Op.
```

Operator types

- $(+) : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$

Operator types

- $(+) : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$
- $(+) : \mathsf{double} \to \mathsf{double} \to \mathsf{double}$

Operator types

- $(+) : \text{int} \to \text{int} \to \text{int}$
- $(+) : \text{double} \to \text{double} \to \text{double}$
- $(+) : \text{unsigned char} \to \text{unsigned char} \to ?$

Operator types

- $(+) : \text{int} \to \text{int} \to \text{int}$
- $(+) : \text{double} \to \text{double} \to \text{double}$
- $(+) : \text{unsigned char} \to \text{unsigned char} \to \text{int}$

Operator types

- $(+) : \text{int} \to \text{int} \to \text{int}$
- $(+) : \text{double} \to \text{double} \to \text{double}$
- $(+) : \text{unsigned char} \to \text{unsigned char} \to \text{int}$
- $(+) : \text{double} \to \text{unsigned short} \to \text{double}$

Operator types

- $(+)$: int $\to$ int $\to$ int
- $(+)$: double $\to$ double $\to$ double
- $(+)$: unsigned char $\to$ unsigned char $\to$ int
- $(+)$: double $\to$ unsigned short $\to$ double

Obc

```
var x : uint8,
    y : int;

x := y
```

Clight

```
unsigned char x;
int y;

x = y;
```

Operator types

- $(+)$: int $\rightarrow$ int $\rightarrow$ int
- $(+)$: double $\rightarrow$ double $\rightarrow$ double
- $(+)$: unsigned char $\rightarrow$ unsigned char $\rightarrow$ int
- $(+)$: double $\rightarrow$ unsigned short $\rightarrow$ double

Obc

```
var x : uint8,
    y : int;

x := y
```

Clight

```
unsigned char x;
int y;

x = y;
```

implicit cast: x = (**unsigned char**) y

Operator types

- $(+) : \text{int} \to \text{int} \to \text{int}$
- $(+) : \text{double} \to \text{double} \to \text{double}$
- $(+) : \text{unsigned char} \to \text{unsigned char} \to \text{int}$
- $(+) : \text{double} \to \text{unsigned short} \to \text{double}$

Obc

```
var x : uint8,
    y : int;

x := (y : uint8)
```

Clight

```
unsigned char x;
int y;

x = y;
```

implicit cast: x = (**unsigned char**) y

- No implicit casting in Obc.
- Simple relation in simulation proof (equality of values).
- Explicit casts simplify substitution (referential transparency).

Operator types: bool

- _Bool: a special kind of integer that is normally 0 or 1.
- x = (_Bool)7

Operator types: bool

- _Bool: a special kind of integer that is normally 0 or 1.
- x = (_Bool)7   puts 1 into x.

Operator types: bool

- _Bool: a special kind of integer that is normally 0 or 1.
- x = (_Bool)7 puts 1 into x.

Obc

```
var x, y : bool;

x := y
```

explicit cast not mandated

Clight

```
_Bool x, y

x = y;
```

implicit cast: x = (_Bool) y

Operator types: bool

- _Bool: a special kind of integer that is normally 0 or 1.
- x = (_Bool)7 puts 1 into x.

Obc

```
var x, y : bool;

x := y
```

explicit cast not mandated

Clight

```
_Bool x, y

x = y;
```

implicit cast: x = (_Bool) y

- The Clight type system is not strong enough for our purposes:
  - There is no typing invariant on the memory.
  - A _Bool is stored in 8-bits.
  - E.g., no way to know that y does not contain 7.

Operator types: bool

- _Bool: a special kind of integer that is normally 0 or 1.
- x = (_Bool)7 puts 1 into x.

Obc

```
var x, y : bool;

x := y
```

explicit cast not mandated

Clight

```
_Bool x, y

x = y;
```

implicit cast: x = (_Bool) y

- The Clight type system is not strong enough for our purposes:
  - There is no typing invariant on the memory.
  - A _Bool is stored in 8-bits.
  - E.g., no way to know that y does not contain 7.
- We refine the types of operators and use a typing invariant.

$$(<) : \text{int} \to \text{int} \to \text{int} \quad \Longrightarrow \quad (<) : \text{int} \to \text{int} \to \text{bool}$$

$$(\&) : \text{bool} \to \text{bool} \to \text{int} \quad \Longrightarrow \quad (\&) : \text{bool} \to \text{bool} \to \text{bool}$$

Operator domains

- Partial operators:
  - integer division/modulo x/y, $x \% y$: $y = 0 \vee (x = \text{MIN\_INT} \wedge y = -1)$
  - shifts $x \ll y$, $x \gg y$: $y < 0 \vee y \geq 32$
- 'Dynamic' precondition in the existence proof.
  ($\forall i$, `sem_binop_i <> None`)
- Alternative OPERATORS implementation and translation:
  x / y becomes **if** x != MIN_INT && y != 0 then x / y **else** 0

Outline

Verifying Lustre compilation in Coq

Translation correctness: SN-Lustre to Obc

Fusion of control structures

Integrating Clight operators into N-Lustre and Obc

Conclusion

What does it cost?

- Elaborator/type checker:
  - 475 lines of Coq (monadic checks) + 80 lines of AST.
  - Rapid development and proof: 2 weeks.
- N-Lustre Syntax: 82 lines of Coq (inductive datatypes)
- Obc Syntax: 50 lines of Coq (inductive datatypes)
- Translation function: 110 lines of Coq (functional definitions)
  - Almost direct from [Biernacki et al. (2008): "Clock-directed modular code generation for synchronous data-flow languages"]
  - 3 semantic models, auxiliary definitions, lemmas, etc.
  - Correctness proof: several months
  - Learning Coq / discovering proof strategy
  - Many elements are useful for other analyses
  - Still quite a complicated proof
- Generation of Clight: Lélio's talk

What does it cost?

- Elaborator/type checker:
  - 475 lines of Coq (monadic checks) + 80 lines of AST.
  - Rapid development and proof: 2 weeks.
- N-Lustre Syntax: 82 lines of Coq (inductive datatypes)
- Obc Syntax: 50 lines of Coq (inductive datatypes)
- Translation function: 110 lines of Coq (functional definitions)
  - Almost direct from [Biernacki et al. (2008): "Clock-directed modular code generation for synchronous data-flow languages"]
  - 3 semantic models, auxiliary definitions, lemmas, etc.
  - Correctness proof: several months
  - Learning Coq / discovering proof strategy
  - Many elements are useful for other analyses
  - Still quite a complicated proof
- Generation of Clight: Lélio's talk

Not cheap.

What's it worth?

What's it worth?

Intrinsic challenge: work out how to do it (simply and efficiently)

What's it worth?

Intrinsic challenge: work out how to do it (simply and efficiently)

Vision: verify Lustre program, get proof about assembly code

- Treat machine representations and arithmetic.
- Integrated verification of external host code.
- More abstract models: parameters and timing properties.

What's it worth?

Intrinsic challenge: work out how to do it (simply and efficiently)

Vision: verify Lustre program, get proof about assembly code

- Treat machine representations and arithmetic.
- Integrated verification of external host code.
- More abstract models: parameters and timing properties.

'Digitized' formal models of Lustre and its compilation

- A form of precise and executable documentation.
- A base for other projects:
  - Trickier features: modular reset and automata;
  - Formal analysis of more optimization passes.

What's it worth?

Intrinsic challenge: work out how to do it (simply and efficiently)

Vision: verify Lustre program, get proof about assembly code

- Treat machine representations and arithmetic.
- Integrated verification of external host code.
- More abstract models: parameters and timing properties.

'Digitized' formal models of Lustre and its compilation

- A form of precise and executable documentation.
- A base for other projects:
  - Trickier features: modular reset and automata;
  - Formal analysis of more optimization passes.

Open question: what is the usefulness in practice?

- **Not** a replacement for SCADE Suite.
- Can we facilitate certification?
- Can we help developers of industrial tools?

References I

Auger, C. (2013). "Compilation certifiée de SCADE/LUSTRE". PhD thesis. Orsay, France: Univ. Paris Sud 11.

Biernacki, D. et al. (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.

Caspi, P. et al. (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.

References II

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.

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

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