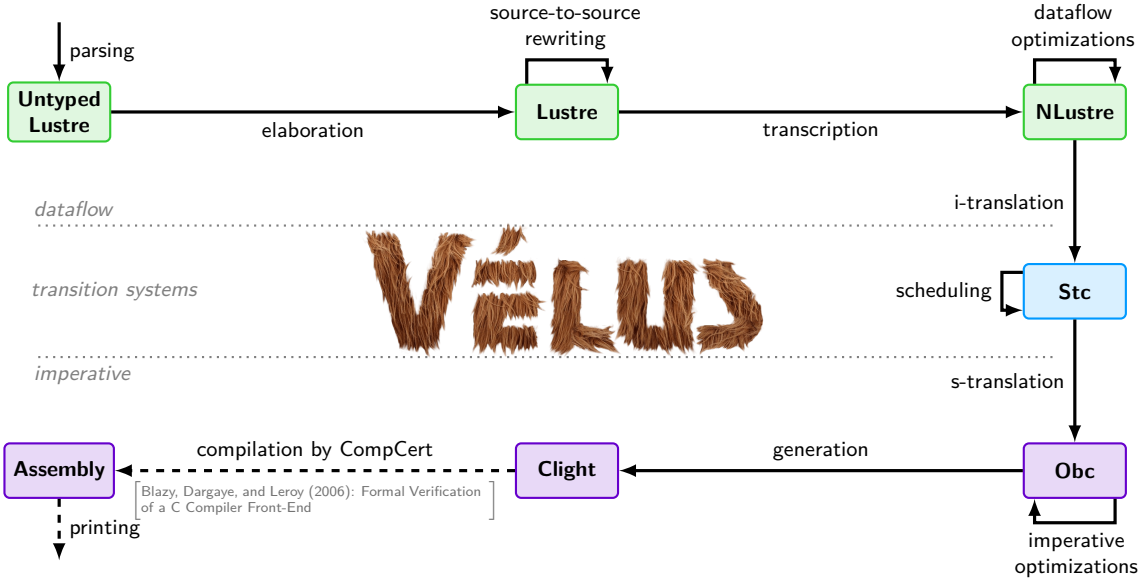
Formal verification of Vélus programs with SMTCoq

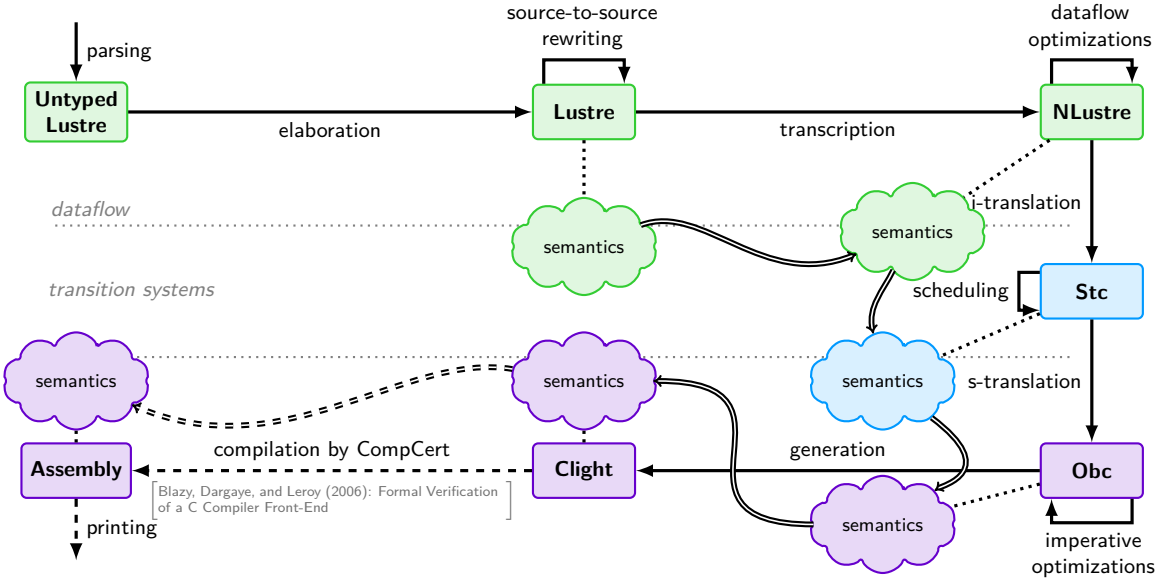
Basile Pesin
Fédération ENAC ISAE-SUPAERO ONERA, Université de Toulouse, France

SYNCHRON 2025

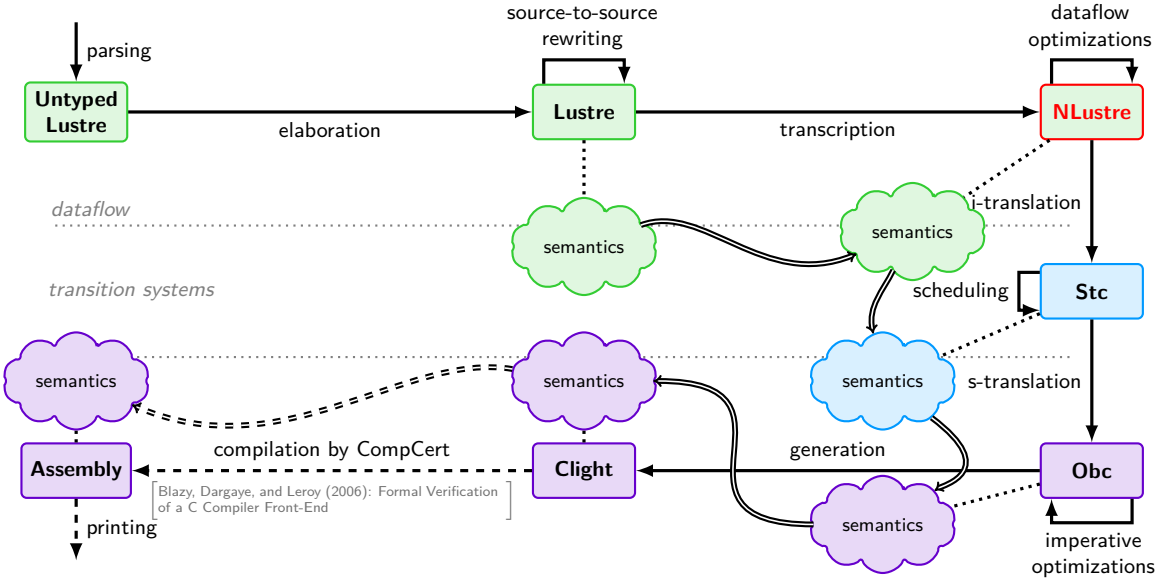
The Vélus compiler



The Vélus compiler



The Vélus compiler



Running Example

```
node sum(i: int; r: bool) returns (c: int)
let
  c = i + (if r then 0 else (0 fby c))
tel
```

Computes the sum of `i`, resets every `r`.

Running Example

```
node sum(i: int; r: bool) returns (c: int)
let
  c = i + (if r then 0 else (0 fby c))
tel
```

Computes the sum of `i`, resets every `r`.

```
node since(b1, b2: bool) returns (b: bool)
var pb : bool;
let
  pb = true fby b;
  b = b2 and (b1 or pb);
tel
```

`b2 = true` since last time `b1 = true`.

Normalized form (`fby` in its own equation).
Looks like NLustre.

```
node obs(i: int; r: bool) returns (ok: bool)
let
  ok = if since(r, i >= 0)
       then sum(i, r) >= 0 else true
tel
```

If `i` non-negative since the last reset, then `sum` is non-negative.

Dataflow Semantics in Vélus

```
Inductive sem_exp:
| Svar:
  sem_var H x s →
  sem_exp H (Evar x) s
| Sbinop:
  sem_exp H e1 s1 →
  sem_exp H e2 s2 →
  lift2 op s1 s2 os →
  sem_exp H b (Ebinop op e1 e2) os [...]
with sem_equation:
| Seq:
  sem_exp H e s →
  sem_var H x s →
  sem_equation H (EqDef x e)
| Sfby:
  sem_exp H e es →
  os = fby (sem_const c0) es →
  sem_var H (Var x) os →
  sem_equation H (EqFby x c0 e) [...]
```

```
node since(b1, b2: bool) returns (b: bool)
var pb : bool;
let
  pb = true fby b;
  b = b2 and (b1 or pb);
tel
```

b1	‹F›	‹T›	‹F›	‹F›	‹›	‹F›	‹T›	…
b2	‹F›	‹T›	‹T›	‹F›	‹›	‹T›	‹T›	…
pb	‹T›	‹F›	‹T›	‹T›	‹›	‹F›	‹F›	…
b	‹F›	‹T›	‹T›	‹F›	‹›	‹F›	‹T›	…

```
with sem_node:
| Snode:
  find_node f G = Some n →
  sem_vars H n.(n_in) iss →
  sem_vars H n.(n_out) oss →
  Forall (sem_equation H n.(n_eqs)) →
  sem_node f iss oss
```

Dataflow Semantics in Vélus

```
Inductive sem_exp:
| Svar:
  sem_var H x s →
  sem_exp H (Evar x) s
| Sbinop:
  sem_exp H e1 s1 →
  sem_exp H e2 s2 →
  lift2 op s1 s2 os →
  sem_exp H b (Ebinop op e1 e2) os [...]
with sem_equation:
| Seq:
  sem_exp H e s →
  sem_var H x s →
  sem_equation H (EqDef x e)
| Sfby:
  sem_exp H e es →
  os = fby (sem_const c0) es →
  sem_var H (Var x) os →
  sem_equation H (EqFby x c0 e) [...]
```

```
node since(b1, b2: bool) returns (b: bool)
var pb : bool;
let
  pb = true fby b;
  b = b2 and (b1 or pb);
tel
```

b1	⟨F⟩	⟨T⟩	⟨F⟩	⟨F⟩	⟨⟩	⟨F⟩	⟨T⟩	...
b2	⟨F⟩	⟨T⟩	⟨T⟩	⟨F⟩	⟨⟩	⟨T⟩	⟨T⟩	...
pb	⟨T⟩	⟨F⟩	⟨T⟩	⟨T⟩	⟨⟩	⟨F⟩	⟨F⟩	...
b	⟨F⟩	⟨T⟩	⟨T⟩	⟨F⟩	⟨⟩	⟨F⟩	⟨T⟩	...

```
with sem_node:
| Snode:
  find_node f G = Some n →
  sem_vars H n.(n_in) iss →
  sem_vars H n.(n_out) oss →
  Forall (sem_equation H n.(n_eqs)) →
  sem_node f iss oss
```

Dataflow Semantics in Vélus

```
Inductive sem_exp:
| Svar:
  sem_var H x s →
  sem_exp H (Evar x) s
| Sbinop:
  sem_exp H e1 s1 →
  sem_exp H e2 s2 →
  lift2 op s1 s2 os →
  sem_exp H b (Ebinop op e1 e2) os [...]
with sem_equation:
| Seq:
  sem_exp H e s →
  sem_var H x s →
  sem_equation H (EqDef x e)
| Sfby:
  sem_exp H e es →
  os = fby (sem_const c0) es →
  sem_var H (Var x) os →
  sem_equation H (EqFby x c0 e) [...]
```

```
node since(b1, b2: bool) returns (b: bool)
var pb : bool;
let
  pb = true fby b;
  b = b2 and (b1 or pb);
tel
```

b1	‹F›	‹T›	‹F›	‹F›	‹›	‹F›	‹T›	...
b2	‹F›	‹T›	‹T›	‹F›	‹›	‹T›	‹T›	...
pb	‹T›	‹F›	‹T›	‹T›	‹›	‹F›	‹F›	...
b	‹F›	‹T›	‹T›	‹F›	‹›	‹F›	‹T›	...

```
with sem_node:
| Snode:
  find_node f G = Some n →
  sem_vars H n.(n_in) iss →
  sem_vars H n.(n_out) oss →
  Forall (sem_equation H n.(n_eqs)) →
  sem_node f iss oss
```

Dataflow Semantics in Vélus

```
Inductive sem_exp:
| Svar:
  sem_var H x s →
  sem_exp H (Evar x) s
| Sbinop:
  sem_exp H e1 s1 →
  sem_exp H e2 s2 →
  lift2 op s1 s2 os →
  sem_exp H b (Ebinop op e1 e2) os [...]
with sem_equation:
| Seq:
  sem_exp H e s →
  sem_var H x s →
  sem_equation H (EqDef x e)
| Sfby:
  sem_exp H e es →
  os = fby (sem_const c0) es →
  sem_var H (Var x) os →
  sem_equation H (EqFby x c0 e) [...]
```

```
node since(b1, b2: bool) returns (b: bool)
var pb : bool;
let
  pb = true fby b;
  b = b2 and (b1 or pb);
tel
```

b1	‹F›	‹T›	‹F›	‹F›	‹›	‹F›	‹T›	…
b2	‹F›	‹T›	‹T›	‹F›	‹›	‹T›	‹T›	…
pb	‹T›	‹F›	‹T›	‹T›	‹›	‹F›	‹F›	…
b	‹F›	‹T›	‹T›	‹F›	‹›	‹F›	‹T›	…

```
with sem_node:
| Snode:
  find_node f G = Some n →
  sem_vars H n.(n_in) iss →
  sem_vars H n.(n_out) oss →
  Forall (sem_equation H n.(n_eqs)) →
  sem_node f iss oss
```

Dataflow Semantics in Vélus

```
Inductive sem_exp:
| Svar:
  sem_var H x s →
  sem_exp H (Evar x) s
| Sbinop:
  sem_exp H e1 s1 →
  sem_exp H e2 s2 →
  lift2 op s1 s2 os →
  sem_exp H b (Ebinop op e1 e2) os [...]
with sem_equation:
| Seq:
  sem_exp H e s →
  sem_var H x s →
  sem_equation H (EqDef x e)
| Sfby:
  sem_exp H e es →
  os = fby (sem_const c0) es →
  sem_var H (Var x) os →
  sem_equation H (EqFby x c0 e) [...]
```

```
node since(b1, b2: bool) returns (b: bool)
var pb : bool;
let
  pb = true fby b;
  b = b2 and (b1 or pb);
tel
```

b1	‹F›	‹T›	‹F›	‹F›	‹›	‹F›	‹T›	…
b2	‹F›	‹T›	‹T›	‹F›	‹›	‹T›	‹T›	…
pb	‹T›	‹F›	‹T›	‹T›	‹›	‹F›	‹F›	…
b	‹F›	‹T›	‹T›	‹F›	‹›	‹F›	‹T›	…

```
with sem_node:
| Snode:
  find_node f G = Some n →
  sem_vars H n.(n_in) iss →
  sem_vars H n.(n_out) oss →
  Forall (sem_equation H n.(n_eqs)) →
  sem_node f iss oss
```

Dataflow Semantics in Vélus

```
Inductive sem_exp:
| Svar:
  sem_var H x s →
  sem_exp H (Evar x) s
| Sbinop:
  sem_exp H e1 s1 →
  sem_exp H e2 s2 →
  lift2 op s1 s2 os →
  sem_exp H b (Ebinop op e1 e2) os [...]
with sem_equation:
| Seq:
  sem_exp H e s →
  sem_var H x s →
  sem_equation H (EqDef x e)
| Sfby:
  sem_exp H e es →
  os = fby (sem_const c0) es →
  sem_var H (Var x) os →
  sem_equation H (EqFby x c0 e) [...]
```

```
node since(b1, b2: bool) returns (b: bool)
var pb : bool;
let
  pb = true fby b;
  b = b2 and (b1 or pb);
tel
```

b1	‹F›	‹T›	‹F›	‹F›	‹›	‹F›	‹T›	…
b2	‹F›	‹T›	‹T›	‹F›	‹›	‹T›	‹T›	…
pb	‹T›	‹F›	‹T›	‹T›	‹›	‹F›	‹F›	…
b	‹F›	‹T›	‹T›	‹F›	‹›	‹F›	‹T›	…

```
with sem_node:
| Snode:
  find_node f G = Some n →
  sem_vars H n.(n_in) iss →
  sem_vars H n.(n_out) oss →
  Forall (sem_equation H n.(n_eqs)) →
  sem_node f iss oss
```

Proving a program property

- Goal: prove that `ok` is always true
- Against Vélus' semantic model
- Carries to the generated C-code

In Rocq (simplified):

```
Lemma sum_obs_spec : ∀ xs ys,
  sem_node G "obs" xs ys →
  ∀ n, ys n = [vtrue].
```

- How should we prove it ?

```
node obs(i: int; r: bool) returns (ok: bool)
let
  ok = if since(r, i >= 0)
       then sum(i, r) >= 0 else true
tel
```

Proving a program property

- Goal: prove that `ok` is always true
- Against Vélus' semantic model
- Carries to the generated C-code

```
node obs(i: int; r: bool) returns (ok: bool)
let
  ok = if since(r, i >= 0)
       then sum(i, r) >= 0 else true
tel
```

In Rocq (simplified):

```
Lemma sum_obs_spec : ∀ xs ys,
  sem_node G "obs" xs ys →
  ∀ n, ys n = [vtrue].
```

- How should we prove it ?
- Invert `sem_node` hypothesis ⇒ get unknown "history" with some hypotheses
- Now what ?

Proving a program property

- Goal: prove that `ok` is always true
- Against Vélus' semantic model
- Carries to the generated C-code

```
node obs(i: int; r: bool) returns (ok: bool)
let
  ok = if since(r, i >= 0)
       then sum(i, r) >= 0 else true
tel
```

In Rocq (simplified):

```
Lemma sum_obs_spec : ∀ xs ys,
  sem_node G "obs" xs ys →
  ∀ n, ys n = [vtrue].
```

- How should we prove it ?
- Invert `sem_node` hypothesis ⇒ get unknown "history" with some hypotheses
- Now what ?
- Either do some complex reasoning by hand, maybe using Jeanmaire's semantics [Bourke, Jeanmaire, and Pouzet (2025): Functional Stream Semantics for a Synchronous Block-Diagram Compiler], or...

Proving a program property

- Goal: prove that `ok` is always true
- Against Vélus' semantic model
- Carries to the generated C-code

```
node obs(i: int; r: bool) returns (ok: bool)
let
  ok = if since(r, i >= 0)
       then sum(i, r) >= 0 else true
tel
```

In Rocq (simplified):

```
Lemma sum_obs_spec : ∀ xs ys,
  sem_node G "obs" xs ys →
  ∀ n, ys n = [vtrue].
```

- How should we prove it ?
- Invert `sem_node` hypothesis ⇒ get unknown "history" with some hypotheses
- Now what ?
- Either do some complex reasoning by hand, maybe using Jeanmaire's semantics [Bourke, Jeanmaire, and Pouzet (2025): Functional Stream Semantics for a Synchronous Block-Diagram Compiler], or...
- Use model-checking to do it automatically

Proving a program property – Automatic approach

- Kind [Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]: automatic verif of Lustre programs
- Uses SMT-solving (Z3) + K-induction/Bounded Model Checking
- Let's do the same, but in Rocq :)

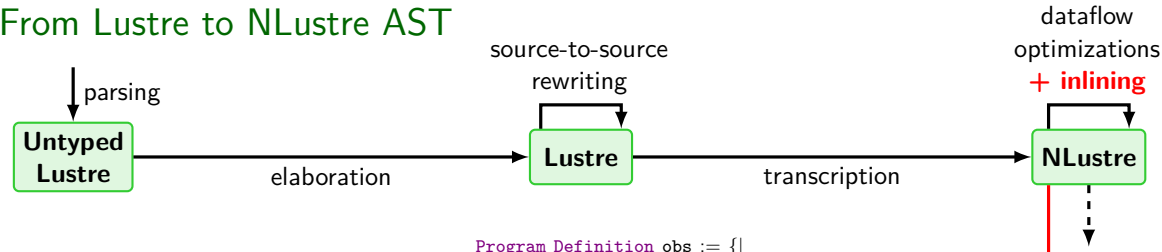
Proving a program property – Automatic approach

- Kind [Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]: automatic verif of Lustre programs
- Uses SMT-solving (Z3) + K-induction/Bounded Model Checking
- Let's do the same, but in Rocq :)

Our approach:

- Compile a Vélus source program into NLustre, print its AST into a .v file
- Generates a parameterized boolean formula for the AST's semantics
- Apply k-induction principle to specialize this formula
- Discharge the specialized formula with SMTCoq
  [Armand, Faure, Grégoire, Keller, Théry, and Werner (2011): A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses]

From Lustre to NLustre AST




```
node sum(i: int; r: bool) returns (c: int)
let
  c = i + (if r then 0 else (0 fby c))
tel

node since(b1, b2: bool) returns (b: bool)
var pb : bool;
let
  pb = true fby b;
  b = b2 and (b1 or pb);
tel

node obs(i: int; r: bool) returns (ok: bool)
let
  ok = if since(r, i >= 0)
       then sum(i, r) >= 0 else true
tel
```

```
Program Definition obs := {|
  n_in := [(_i, (tyint, Cbase)); (_r, (tybool, Cbase))];
  n_out := [(_ok, (tybool, Cbase))];
  n_vars := [...];
  n_eqs := [
    EqFby v5 Cbase (Const 0) (Evar v3);
    EqDef v4 (Ecase (Evar _r) [Evar v5; Econst 0])
    EqDef v3 (Ebinop Oadd (Evar _i) (Evar v4));
    EqFby v2 Cbase (Enum 1%nat) (Evar v1);
    EqDef v1
      (Ebinop Oand
         (Ebinop Oge (Evar _i) (Econst 0))
         (Ebinop Oor (Evar _r) (Evar v2)));
    EqDef _ok
      (Ecase (Evar v1)
         [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)]);
  ]
|}.
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
```

```
(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
```

```
(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  denot_cexp (H n) (H n _ok) (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
Fixpoint denot_bool_exp (E: env) (e: exp) : bool * bool.
```

```
(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  ifb (fst (denot_bool_exp (Evar v1)))
    (ifb (snd (denot_bool_exp (Evar v1)))
      (denot_cexp (H n) (H n _ok) (Ebinop Oge (Evar v3) (Econst 0)))
      (denot_cexp (H n) (H n _ok) (Eenum 1)))
    (negb (fst (H n _ok)))
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
Fixpoint denot_bool_exp (E: env) (e: exp) : bool * bool.
```

```
(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  ifb (fst (H n v1))
    (ifb (snd (H n v1))
      (denot_cexp (H n) (H n _ok) (Ebinop Oge (Evar v3) (Econst 0)))
      (denot_cexp (H n) (H n _ok) (Eenum 1)))
    (negb (fst (H n _ok)))
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
Fixpoint denot_bool_exp (E: env) (e: exp) : bool * bool.
Fixpoint denot_int_exp (E: env) (e: exp) : bool * Z.


(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  ifb (fst (H n v1))
    (ifb (snd (H n v1))
      (let v := denot_int_exp (H n) (Ebinop Oge (Evar v3) (Econst 0)) in
       (fst (H n _ok) <---> fst v) && (snd v <---> (snd (H n _ok) =? 1)))
      (denot_cexp (H n) (H n _ok) (Eenum 1)))
    (negb (fst (H n _ok)))
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
Fixpoint denot_bool_exp (E: env) (e: exp) : bool * bool.
Fixpoint denot_int_exp (E: env) (e: exp) : bool * Z.
```

```
(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  ifb (fst (H n v1))
    (ifb (snd (H n v1))
      (let v := (let v1 := denot_int_exp (H n) (Evar v3) in
                 fst v1, snd v1 >=? snd (denot_int_exp (H n) (Econst 0))) in
       (fst (H n _ok) <---> fst v) && (snd v <---> (snd (H n _ok) =? 1)))
      (denot_cexp (H n) (H n _ok) (Eenum 1)))
    (negb (fst (H n _ok)))
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
Fixpoint denot_bool_exp (E: env) (e: exp) : bool * bool.
Fixpoint denot_int_exp (E: env) (e: exp) : bool * Z.


(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  ifb (fst (H n v1))
    (ifb (snd (H n v1))
      (let v := (let v1 := H n v3 in
                 fst v1, snd v1 >=? snd (true, 0))) in
       (fst (H n _ok) <---> fst v) && (snd v <---> (snd (H n _ok) =? 1)))
      (denot_cexp (H n) (H n _ok) (Eenum 1)))
    (negb (fst (H n _ok)))
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
Fixpoint denot_bool_exp (E: env) (e: exp) : bool * bool.
Fixpoint denot_int_exp (E: env) (e: exp) : bool * Z.


(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  ifb (fst (H n v1))
    (ifb (snd (H n v1))
      (let v := (fst (H n v3), snd (H n v3) <=? 0)) in
       (fst (H n _ok) <---> fst v) && (snd v <---> (snd (H n _ok) =? 1)))
      (denot_cexp (H n) (H n _ok) (Eenum 1)))
    (negb (fst (H n _ok)))
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
Fixpoint denot_bool_exp (E: env) (e: exp) : bool * bool.
Fixpoint denot_int_exp (E: env) (e: exp) : bool * Z.
```

```
(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  ifb (fst (H n v1))
    (ifb (snd (H n v1))
      ((fst (H n _ok) <---> fst (H n v3))
       && ((snd (H n v3) <=? 0) <---> (snd (H n _ok) =? 1)))
      (denot_cexp (H n) (H n _ok) (Eenum 1)))
    (negb (fst (H n _ok)))
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_cexp (E: env) (v: bool * Z) (e: cexp) : bool.
Fixpoint denot_bool_exp (E: env) (e: exp) : bool * bool.
Fixpoint denot_int_exp (E: env) (e: exp) : bool * Z.
```

```
(* Example (combinatorial) *)
denot_equ H n (EqDef _ok (Ecase (Evar v1) [Eenum 1; Ebinop Oge (Evar v3) (Econst 0)])) =
  ifb (fst (H n v1))
    (ifb (snd (H n v1))
      ((fst (H n _ok) <---> fst (H n v3))
       && (snd (H n v3) <=? 0) <---> (snd (H n _ok) =? 1)))
      ((fst (H n _ok) <---> true) && (snd (H n _ok) =? 1)))
    (negb (fst (H n _ok)))
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_clock (E: env) (ck: clock) : bool.
```

```
(* Example (fby) *)
denot_equ H n (EqFby _ok Cbase (Const 0) (Evar v3)) =
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_clock (E: env) (ck: clock) : bool.
```

```
(* Example (fby) *)
denot_equ H n (EqFby _ok Cbase (Const 0) (Evar v3)) =
  ifb (denot_clock (H n) Cbase)
    (fst (H n _ok)
     && hold
          (fun m => denot_clock (H m) Cbase)
          (snd (H n _ok) =? 0)
          (fun m ⇒ denot_exp (H m) (H n _ok) (Evar v3))
          n)
    (negb (fst (H n _ok)) && snd (H n _ok) =? 0)
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_clock (E: env) (ck: clock) : bool.
```

```
(* Example (fby) *)
denot_equ H n (EqFby _ok Cbase (Const 0) (Evar v3)) =
  ifb (true)
    (fst (H n _ok)
     && hold
          (fun m => denot_clock (H m) Cbase)
          (snd (H n _ok) =? 0)
          (fun m ⇒ denot_exp (H m) (H n _ok) (Evar v3))
          n)
    (negb (fst (H n _ok)) && snd (H n _ok) =? 0)
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_clock (E: env) (ck: clock) : bool.
```

```
(* Example (fby) *)
denot_equ H n (EqFby _ok Cbase (Const 0) (Evar v3)) =
  fst (H n _ok)
  && hold
       (fun m => denot_clock (H m) Cbase)
       (snd (H n _ok) =? 0)
       (fun m ⇒ denot_exp (H m) (H n _ok) (Evar v3))
       n
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_clock (E: env) (ck: clock) : bool.
```

```
(* Example (fby) *)
denot_equ H n (EqFby _ok Cbase (Const 0) (Evar v3)) =
  fst (H n _ok)
  && (fix hold m :=
        match n with
        | O ⇒ snd (H n _ok) =? 0
        | S m ⇒ Bool.ifb (denot_clock (H m) Cbase)
                  (denot_exp (H m) (H n _ok) (Evar v3))
                  (hold m)
        end) n
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_clock (E: env) (ck: clock) : bool.
```

```
(* Example (fby) *)
denot_equ H n (EqFby _ok Cbase (Const 0) (Evar v3)) =
  fst (H n _ok)
  && (fix hold m :=
        match n with
        | O ⇒ snd (H n _ok) =? 0
        | S m ⇒ denot_exp (H m) (H n _ok) (Evar v3)
        end) n
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
Fixpoint denot_clock (E: env) (ck: clock) : bool.
```

```
(* Example (fby) *)
denot_equ H n (EqFby _ok Cbase (Const 0) (Evar v3)) =
  fst (H n _ok)
  && match n with
      | 0 ⇒ snd (H n _ok) =? 0
      | S n ⇒ denot_exp (H n) (H (S n) _ok) (Evar v3)
      end
```

Boolean denotation for NLustre equations

```
Definition env := ident → (bool * Z). (* clock, value *)
Definition hist : nat → env.
(* Denotes the constraint induced by [eq] at cycle [n] on [H] *)
Definition denot_equ (H: hist) (eq: equation) (n: nat) : bool.
```

```
(* Correctness lemma *)
Lemma denot_equation_safe :
    wt_equation G Γ equ →
    sem_equation G H equ →
    ∀ n, denot_equation H equ n = true.
```

Automated goal and k-induction

Node-specific goal to prove automatically:

```
Lemma obs_denot_spec : ∀ H,
  (∀ n, (forallb (fun e ⇒ denot_equ H e n) (n_eqs nd))) →  (* H respects semantics *)
  ∀ n, (snd (H n _ok) =? 1). (* output is always true *)
```

Difficulties:

- SMT solvers cannot deal with `(∀ n, ...) → (∀ n, ...)`
- for combinatorial programs, we can prove

  ```
  ∀ n,
    (forallb (fun e ⇒ denot_equ H e n) (n_eqs nd)) →
    (snd (H n _ok) =? 1)
  ```

- for stateful nodes (with `fby`), we need to add a (k-)induction hypothesis.

K-induction tactic

$$\Delta_0 \wedge \Delta_1 \wedge \cdots \wedge \Delta_k \models_{\mathcal{IL}} P_0 \wedge P_1 \wedge \cdots \wedge P_k$$

$$\Delta_n \wedge \Delta_{n+1} \wedge \cdots \wedge \Delta_{n+(k+1)} \wedge P_n \wedge P_{n+1} \wedge \cdots \wedge P_{n+k} \models_{\mathcal{IL}} P_{n+(k+1)}$$

[Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]

with

$\Delta_n :=$ `forallb (fun e ⇒ denot_equ H e n) (n_eqs nd)`

$P_n :=$ `snd (H n _ok) =? 1`

Automatic tactic, but k must be specified by the user

K-induction tactic

$$\Delta_0 \wedge \Delta_1 \wedge \cdots \wedge \Delta_k \models_{\mathcal{IL}} P_0 \wedge P_1 \wedge \cdots \wedge P_k$$
$$\Delta_n \wedge \Delta_{n+1} \wedge \cdots \wedge \Delta_{n+(k+1)} \wedge P_n \wedge P_{n+1} \wedge \cdots \wedge P_{n+k} \models_{\mathcal{IL}} P_{n+(k+1)}$$

[Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]

with

$\Delta_n :=$ `forallb (fun e ⇒ denot_equ H e n) (n_eqs nd)`

$P_n :=$ `snd (H n _ok) =? 1`

First, apply the following lemma:

```
Lemma k_ind (k: nat) : ∀ (P: nat → Prop),
  (∀ n, (n < k) → P n) →
  (∀ n, (∀ m, m < k → P (n + m)) → P (k + n)) →
  ∀ n, P n.
```

Generates 2 goals

- First goal is split in k subgoals ($n = 0, 1, ..., k-1$)
- Hypothesis of the second is specialized ($m = 0, 1, ..., k-1$)

Automatic tactic, but k must be specified by the user

Solving boolean goals with SMTCoq

K-induction has generated $k + 1$ boolean subgoals

One of them (for our example) looks like this:

```
ifb (fst (N n _i) && (fst (N n _r) && true))
  (ifb
     (ifb (fst (N n var48))
        (ifb (snd (N n var48) =? 1)%Z
           ((fst (N n _ok) <~~> fst (N n var52)) &&
            (fst (N n var52) && (snd (N n var52) >=? 0)%Z <~~>
             (snd (N n _ok) =? 1)%Z))
           ((fst (N n _ok) <~~> true) &&
            (true <~~> (snd (N n _ok) =? 1)%Z)))
        (negb (fst (N n _ok)) && (snd (N n _ok) =? 0)%Z) &&
     ((fst (N n var48) <~~> fst (N n _i)) &&
      (fst (N n _i) && fst (N n _r) &&
       (fst (N n _i) && (snd (N n _i) >=? 0)%Z &&
        (fst (N n _r) && fst (N n var49) &&
         ((snd (N n _r) =? 1)%Z || (snd (N n var49) =? 1)%Z))) <~~>
       (snd (N n var48) =? 1)%Z) && fst (N n var49) && b n) &&
     ((fst (N n var52) <~~> fst (N n _i)) &&
      ifb (fst (N n _i))
        (snd (N n var52) =?
         snd (N n _i) + snd (N n var53))%Z
        (snd (N n var52) =? 0)%Z) &&
     ifb (fst (N n _r))
       (ifb (snd (N n _r) =? 1)%Z
          ((fst (N n var53) <~~> true) && (snd (N n var53) =? 0)%Z)
          ((fst (N n var53) <~~> fst (N n var54)) &&
           ifb (fst (N n var54))
             (snd (N n var53) =? snd (N n var54))%Z
             (snd (N n var53) =? 0)%Z))
       (negb (fst (N n var53)) && (snd (N n var53) =? 0)%Z) &&
  (fst (N n var54) && b0 n))
  (ifb (snd (N n _ok) =? 1)%Z
     (ifb (fst (N (S n) _i) && (fst (N (S n) _r) && true))
        (ifb
           (ifb (fst (N (S n) var48))
              (ifb (snd (N (S n) var48) =? 1)%Z
                 ((fst (N (S n) _ok) <~~> fst (N (S n) var52)) &&
                  (fst (N (S n) var52) &&
                   (snd (N (S n) var52) >=? 0)%Z <~~>
                   (snd (N (S n) _ok) =? 1)%Z))
                 ((fst (N (S n) _ok) <~~> true) &&
                  (true <~~> (snd (N (S n) _ok) =? 1)%Z)))
              (negb (fst (N (S n) _ok)) &&
               (snd (N (S n) _ok) =? 0)%Z) &&
           ((fst (N (S n) var48) <~~> fst (N (S n) _i)) &&
            (fst (N (S n) _i) && fst (N (S n) _r) &&
             (fst (N (S n) _i) && (snd (N (S n) _i) >=? 0)%Z &&
              (fst (N (S n) _r) && fst (N (S n) var49) &&
               ((snd (N (S n) _r) =? 1)%Z
                || (snd (N (S n) var49) =? 1)%Z))) <~~>
             (snd (N (S n) var48) =? 1)%Z)) &&
           (fst (N (S n) var49) &&
            ((fst (N (S n) var49) <~~> fst (N n var48)) &&
             ((snd (N n var48) =? 1)%Z <~~>
              (snd (N (S n) var49) =? 1)%Z))) &&
           ((fst (N (S n) var52) <~~> fst (N (S n) _i)) &&
            ifb (fst (N (S n) _i))
              (snd (N (S n) var52) =?
               snd (N (S n) _i) + snd (N (S n) var53))%Z
              (snd (N (S n) var52) =? 0)%Z) &&
           ifb (fst (N (S n) _r))
             (ifb (snd (N (S n) _r) =? 1)%Z
                ((fst (N (S n) var53) <~~> true) &&
                 (snd (N (S n) var53) =? 0)%Z)
                ((fst (N (S n) var53) <~~> fst (N (S n) var54)) &&
                 ifb (fst (N (S n) var54))
                   (snd (N (S n) var53) =? snd (N (S n) var54))%Z
                   (snd (N (S n) var53) =? 0)%Z))
             (negb (fst (N (S n) var53)) &&
              (snd (N (S n) var53) =? 0)%Z) &&
           (fst (N (S n) var54) &&
            ((fst (N (S n) var54) <~~> fst (N n var52)) &&
             ifb (fst (N n var52))
               (snd (N (S n) var54) =? snd (N n var52))%Z
               (snd (N (S n) var54) =? 0)%Z)))
        (snd (N (S n) _ok) =? 1)%Z true) true) true) true = true
```

Solving boolean goals with SMTCoq

K-induction has generated $k + 1$ boolean subgoals

One of them (for our example) looks like this:

To solve it: `verit` tactic from SMTCoq

- Translates the goal into SMT
- Calls the veriT solver

  [Bouton, Caminha B. de Oliveira, Déharbe, and Fontaine (2009): veriT: an open, trustable and efficient SMT-solver]
- Reconstructs a Rocq proof from the unsat certificate

  [Armand, Faure, Grégoire, Keller, Théry, and Werner (2011): A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses]
- `Qed.` :)

Demonstration

Let's hope the property is true !

Limitations and Future Work

- Support for machine integers/floating point numbers ?
  - The denotation/proof is done on arbitrary size integers

    ```
    Fixpoint denot_int_exp (E: env) (e: exp) : bool * Z.
    ```

  - Define a new version of the denotation
  - Use cvc4/Z3 ?

Limitations and Future Work

- Support for machine integers/floating point numbers ?
  - The denotation/proof is done on arbitrary size integers

```
Fixpoint denot_int_exp (E: env) (e: exp) : bool * Z.
```

  - Define a new version of the denotation
  - Use cvc4/Z3 ?
- Invariant discovery
  - k-induction is not always sufficient (ex: gilbreath shuffle)
  - but they can be strengthened to become 1-inductive

```
node main (...) returns (ok: bool);
var prop, inv : bool;
let (* Body of the program *)
  prop = ...; (* Property of interest *)
  inv = ...; (* Additional invariant *)
  ok = prop and inv;
tel;
```

  - add automatically ? complex heuristics [Champion, Mebsout, Sticksel, and Tinelli (2016): The Kind 2 Model Checker]
  - get invariant by calling Kind2
  - prove (once-and-forall) that adding the invariant can only lead to "less" true