Inlining in a Verified Synchronous Language Compiler

Balthazar Patiachvili, supervised by Timothy Bourke
in the PARKAS team at INRIA Paris

24/11/2025

Outline

Vélus

Introduction

- Verified compiler in Rocq/OCaml from a synchronous dataflow language to Clight
- Introduced in 2016[1], three theses[2][3][4] in the PARKAS team at INRIA Paris.

[1]T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for Lustre," *SIGPLAN Not.*, vol. 52, no. 6, pp. 586–601, Jun. 2017, doi: 10.1145/3140587.3062358.

[2]L. Brun, "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset," 2020. [Online]. Available: https://www.leliobrun.net/files/thesis.pdf

[3]B. Pesin, "Verified Compilation of a Synchronous Dataglow Language with State Machines," 2023. [Online]. Available: https://velus.inria.fr/phd-pesin/thesis.pdf

[4]P. Jeanmaire, "Une sémantique dénotationnelle pour un compilateur synchrone vérifié," 2024. [Online]. Available: https://velus.inria.fr/phd-jeanmaire/these.pdf

[5]D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed modular code generation for synchronous data-flow languages," *SIGPLAN Not.*, vol. 43, no. 7, pp. 121–130, Jun. 2008, doi: 10.1145/1379023.1375674.

Vélus

Introduction

- Verified compiler in Rocq/OCaml from a synchronous dataflow language to Clight
- Introduced in 2016[1], three theses[2][3][4] in the PARKAS team at INRIA Paris.

- Lustre $\sqsubseteq$ Vélus Lustre $\sqsubseteq$ SCADE 6
- Modular compilation scheme[5]

---

[1]T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for Lustre," *SIGPLAN Not.*, vol. 52, no. 6, pp. 586–601, Jun. 2017, doi: 10.1145/3140587.3062358.

[2]L. Brun, "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset," 2020. [Online]. Available: https://www.leliobrun.net/files/thesis.pdf

[3]B. Pesin, "Verified Compilation of a Synchronous Dataglow Language with State Machines," 2023. [Online]. Available: https://velus.inria.fr/phd-pesin/thesis.pdf

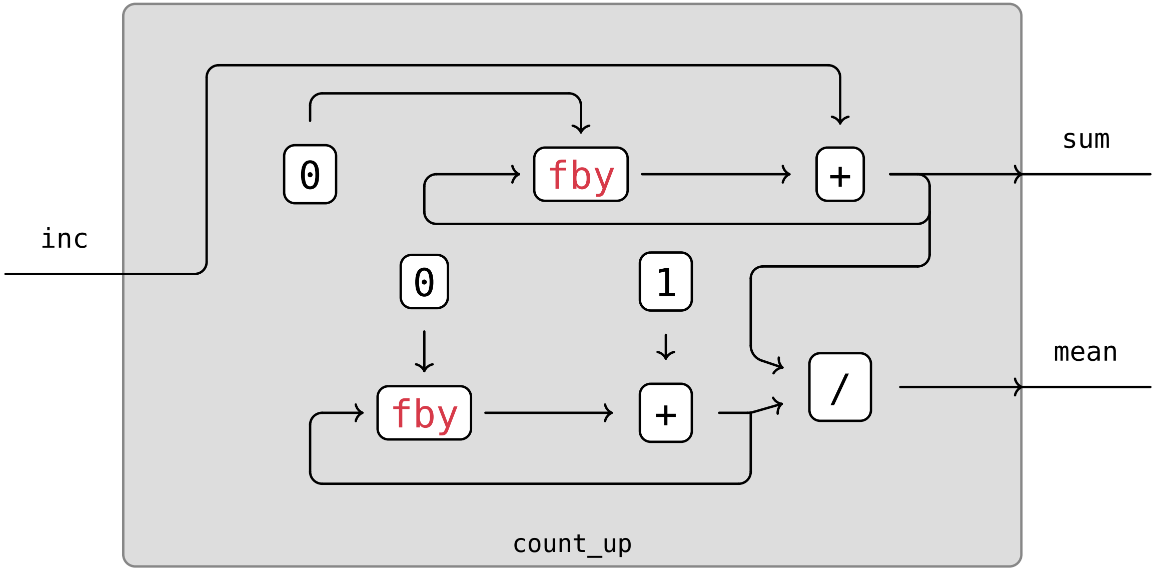
[4]P. Jeanmaire, "Une sémantique dénotationnelle pour un compilateur synchrone vérifié," 2024. [Online]. Available: https://velus.inria.fr/phd-jeanmaire/these.pdf

[5]D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed modular code generation for synchronous data-flow languages," *SIGPLAN Not.*, vol. 43, no. 7, pp. 121–130, Jun. 2008, doi: 10.1145/1379023.1375674.

Vélus

Lustre program / global environment

List of *nodes*, *type* and *external function declarations*.



inc	1	3	5	2	6	4	...
n	1	2	3	4	5	6	...
sum	1	4	9	11	17	21	...
mean	1	2	3	2.75	3.4	3.5	...

Vélus

Lustre program / global environment

List of *nodes*, *type* and *external function declarations.*

```lustre
node count_up(inc : int)
returns (sum : int; mean : float);
var n : int;
let
  mean = sum / n;
  n = (0 fby n) + 1;
  sum = (0 fby sum) + inc;
tel
```

inc	1	3	5	2	6	4	...
n	1	2	3	4	5	6	...
sum	1	4	9	11	17	21	...
mean	1	2	3	2.75	3.4	3.5	...

Vélus

Lustre program / global environment

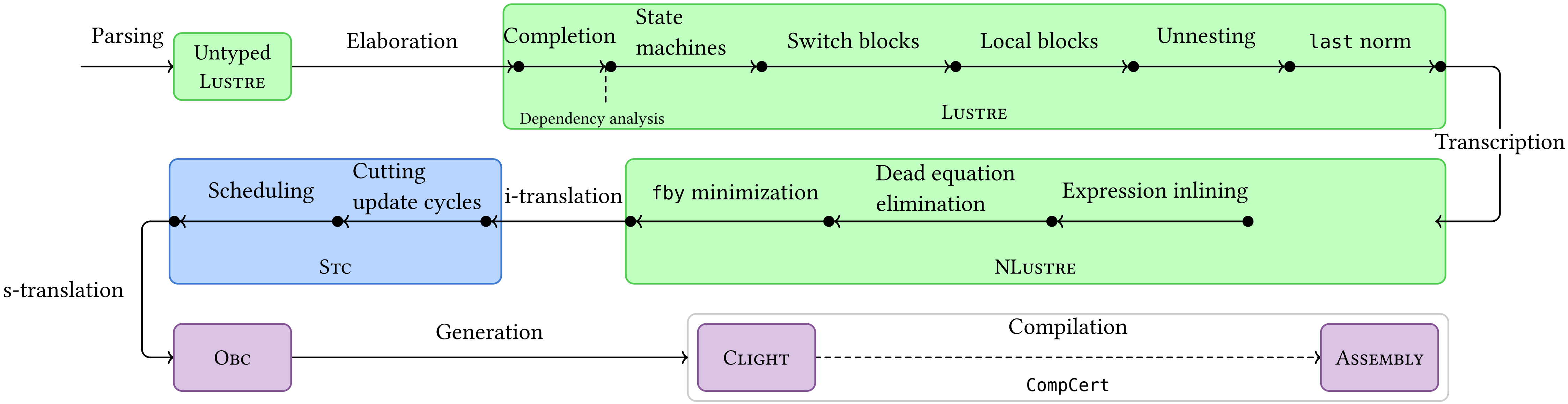
List of *nodes*, *type* and *external function declarations*.

```lustre
node count_up(inc : int)
returns (sum : int; mean : float);
var n : int;
let
  mean = sum / n;
  n = (0 fby n) + 1;
  sum = (0 fby sum) + inc;
tel
```

inc	1	3	5	2	6	4	...
`0 fby n`	0	1	2	3	4	5	...
n	1	2	3	4	5	6	...
`0 fby sum`	0	1	4	9	11	17	...
sum	1	4	9	11	17	21	...
mean	1	2	3	2.75	3.4	3.5	...

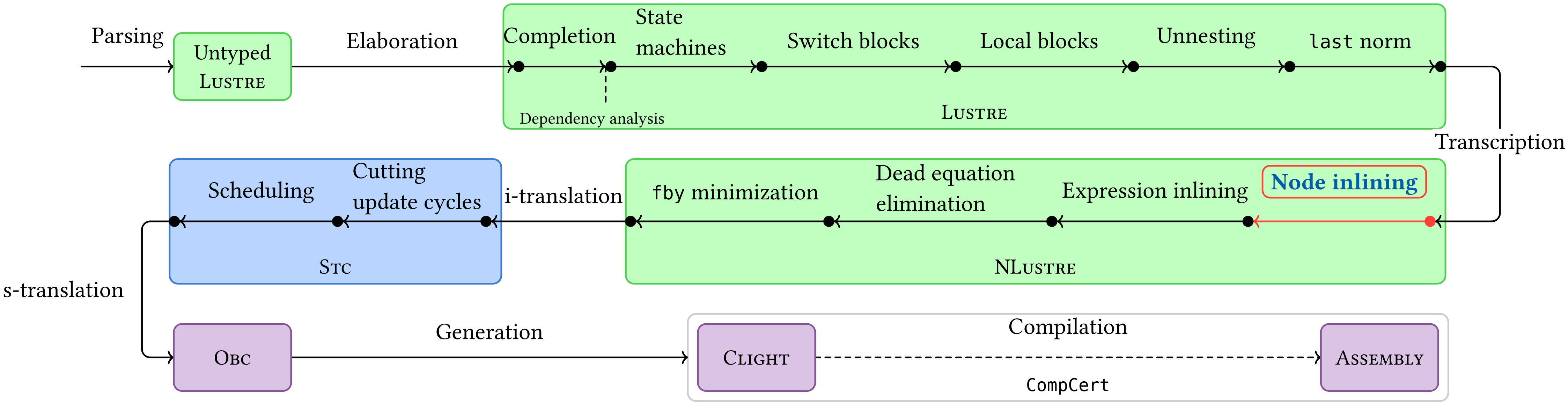
Vélus

Architecture



Vélus

Architecture



NLustre

- Normalized Lustre
- Intermediate language with much simpler syntax than Lustre.

NLustre

- Normalized Lustre
- Intermediate language with much simpler syntax than Lustre.

Equations

$$\begin{aligned} equ ::= {} & x = e & \texttt{(Def)} \\ | {} & x =_{rs} c \texttt{ fby } e & \texttt{(Fby)} \\ | {} & \texttt{last } x =_{rs} c & \texttt{(Last)} \\ | {} & x^{+} =_{rs} f(e^{+}) & \texttt{(App)} \end{aligned}$$

Node

Rocq

```
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 * bool));
  n_eqs     : list equation;
}.
```

NLustre

- Normalized Lustre
- Intermediate language with much simpler syntax than Lustre.

Equations

$$
\begin{aligned}
equ ::= {} & x = e & \texttt{(Def)} \\
\mid {} & x =_{rs} c \ \texttt{fby} \ e & \texttt{(Fby)} \\
\mid {} & \texttt{last} \ x =_{rs} c & \texttt{(Last)} \\
\mid {} & x^{+} =_{rs} f(e^{+}) & \texttt{(App)}
\end{aligned}
$$

```lustre
node count_up(inc : int)
returns (sum : int; mean : float);
var n : int;
let mean = sum / n;
    n = (0 fby n) + 1;
    sum = (0 fby sum) + inc;
tel
```

Node

```rocq
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 * bool));
  n_eqs     : list equation;
}.
```

```
n_name := "count_up"
  n_in := [inc]
 n_out := [sum, mean]
n_vars := [n]
 n_eqs := [EqDef mean (Div sum n), ...]
```

NLustre

Reset signals

- The equations `(Fby)`, `(Last)` and `(App)` “store” actively the current state of its variables.
- It is possible through reset signals to enforce the value of the variables as if it were again the first instant of the equation.

NLustre

Reset signals

- The equations (Fby), (Last) and (App) “store” actively the current state of its variables.
- It is possible through reset signals to enforce the value of the variables as if it were again the first instant of the equation.

```
nlustre
node count_up(inc : int)
returns (sum : int; mean : float);
var n : int;
let mean = sum / n;
    n = (0 fby n) + 1;
    sum = (0 fby sum) + inc;
tel

node foo(x : int; rs : bool)
returns (sum : int; mean : float);
let (sum, mean) = reset count_up(x) every rs;
tel
```

x	1	5	3	2	6	4	8	...
rs	F	F	T	F	T	F	F	...
sum	1	6	3	5	6	10	18	...
mean	1	3	3	2.5	6	5	6	...

NLustre

Reset signals

- The equations (Fby), (Last) and (App) "store" actively the current state of its variables.
- It is possible through reset signals to enforce the value of the variables as if it were again the first instant of the equation.

```nlustre
node count_up(inc : int)
returns (sum : int; mean : float);
var n : int;
let mean = sum / n;
    n = (0 fby n) + 1;
    sum = (0 fby sum) + inc;
tel

node foo(x : int; rs : bool)
returns (sum : int; mean : float);
let (sum, mean) = reset count_up(x) every rs;
tel
```

x	1	5	3	2	6	4	8	...
rs	F	F	T	F	T	F	F	...
mask^0 sum	1	6						...
mask^1 sum			3	5				...
mask^2 sum					6	10	18	...
sum	1	6	3	5	6	10	18	...

Outline

Description

Node inlining

- Inline expansion of nodes
- Manual or compiler optimization that replace a node call with the body of the called node.
- This internship: defined on resettable stream functions.

Description

Node inlining

- Inline expansion of nodes
- Manual or compiler optimization that replace a node call with the body of the called node.
- This internship: defined on resettable stream functions.

Usages

- May reduce worst-case execution time
- Increases the number of programs that can be compiled by Vélus

Node inlining makes more programs compilable

```lustre
node swap(x, y : int) returns (y2, x2 : int);
let x2 = x;
    y2 = y;
tel

node foo(x : int) returns (y : int);
var z : int;
let (y, z) = swap(x, z);
tel
```

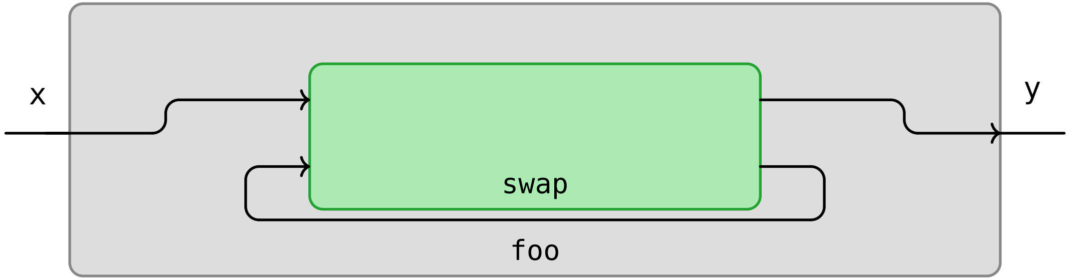
```lustre
node foo(x : int) returns (y : int);
var x', y', x2', y2', z : int;
let x2' = x';
    y2' = y';
    x' = x;
    y' = z;
    y = y2';
    z = x2';
tel
```

Node inlining makes more programs compilable

```lustre
node swap(x, y : int) returns (y2, x2 : int);
let x2 = x;
    y2 = y;
tel

node foo(x : int) returns (y : int);
var z : int;
let (y, z) = swap(x, z);
tel
```

```lustre
node foo(x : int) returns (y : int);
var x', y', x2', y2', z : int;
let x2' = x';
    y2' = y';
    x' = x;
    y' = z;
    y = y2';
    z = x2';
tel
```

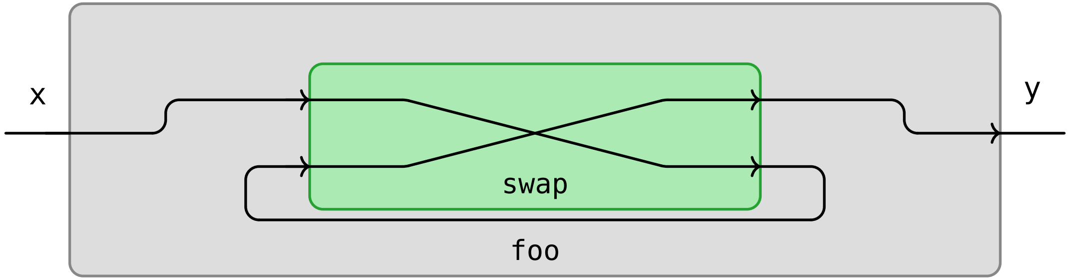


Node inlining makes more programs compilable

lustre
```
node swap(x, y : int) returns (y2, x2 : int);
let x2 = x;
    y2 = y;
tel

node foo(x : int) returns (y : int);
var z : int;
let (y, z) = swap(x, z);
tel
```

lustre
```
node foo(x : int) returns (y : int);
var x', y', x2', y2', z : int;
let x2' = x';
    y2' = y';
    x' = x;
    y' = z;
    y = y2';
    z = x2';
tel
```

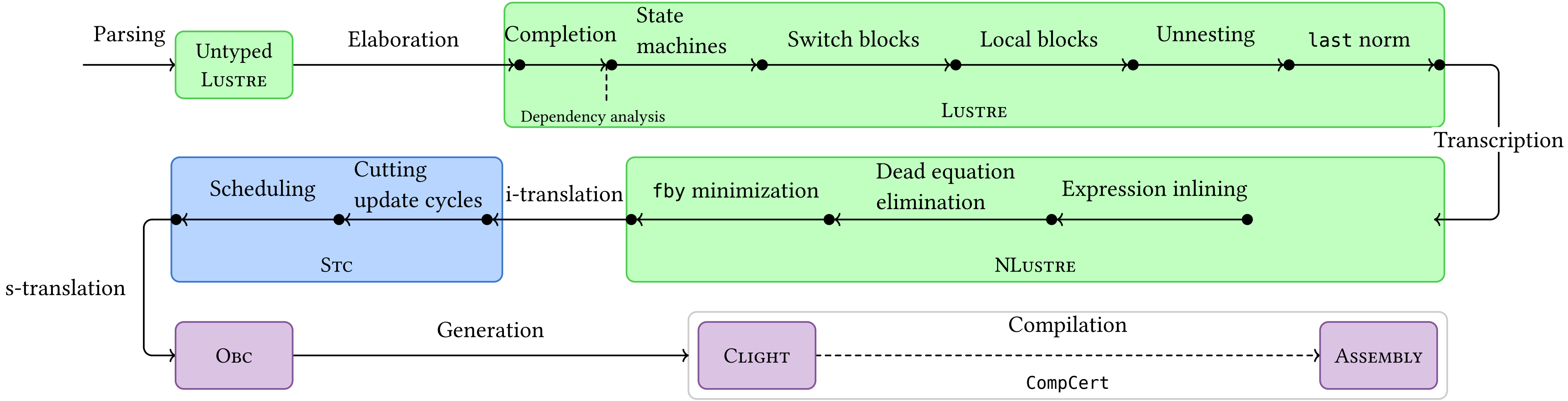


Implementation

Where?

Two possibilities:

- In Lustre, between the compilation of switch blocks and local blocks
- In NLustre, before the expression inlining

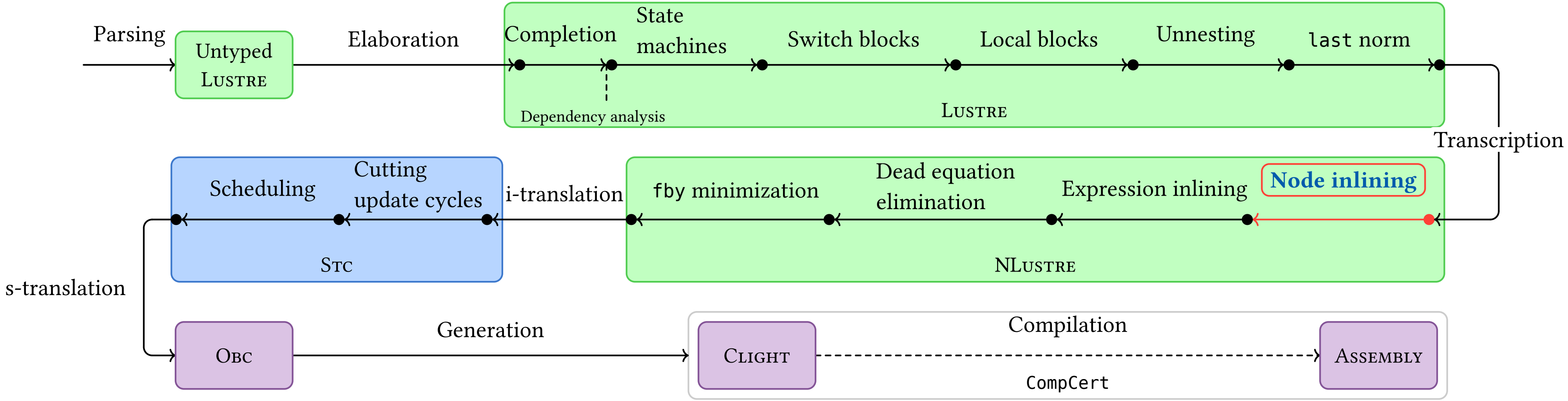


Implementation

Where?

Two possibilities:

- In Lustre, between the compilation of switch blocks and local blocks
- In NLustre, before the expression inlining



Choice: NLustre for the small number of expressions

Implementation

```
node foo(a: int; b: int)                                nlustre
returns (c: int; d: int);
var t: bool;
let
  t = true fby (not t);
  c = a + b;
  d = reset (0 fby (d + c)) every t;
tel

node bar(x, y: int; rs: bool)
returns (z: int);
var t, u: int;
let
  (u, t) = reset foo(x + 1, y * 3) every rs;
  z = t + u;
tel
```

```
node bar(x, y: int; rs: bool)                           nlustre
returns (z: int);
var t, u: int;
    a', b', c', d', t': int;
let
  t' = reset true fby (not t') every rs;          } Renamed
  c' = a' + b';                                   } equations
  d' = reset 0 fby (d' + c') every t', rs;
  a' = x + 1;                                     } Input
  b' = y * 3;                                     } equations
  u = c';                                         } Output
  t = d';                                         } equations
  z = t + u;                                      } Kept equation
tel
```

Outline

Introduction

Goal

Reestablish compiler correctness with node inlining pass

Introduction

Goal

Reestablish compiler correctness with node inlining pass

Required properties

- Syntactic properties
- Typing preservation
- Clock-typing preservation
- Semantics preservation

Syntactic properties

Presentation

- Syntactic properties are contained in a node definition
- Only hypotheses available in the proofs: syntactic properties of the old node

↳ Semantic properties cannot be used

Rocq

```
Record node {prefs : PS.t} : Type := mk_node {
  n_name     : ident;                                  (* name *)
  n_in       : list (ident * (type * clock));           (* inputs *)
  n_out      : list (ident * (type * clock));           (* outputs *)
  n_vars     : list (ident * (type * clock * bool));    (* local variables *)
  n_eqs      : list equation;                           (* equations *)
  ... (* Syntactic properties *)
}.
```

Syntactic properties

Properties

```
  n_ingt0 :  0 < length n_in
 n_outgt0 :  0 < length n_out
   n_defd : Permutation (vars_defined n_eqs) (n_vars ++ n_out)
 n_lastd1 : Permutation (last_defined n_eqs) (filter islast n_vars)
 n_lastd2 : ∀ x, x ∈ n_vars ⟹ islast x ⟹ x ∈ vars_defined (filter is_def_cexp n_eqs)
   n_vout : ∀ out, out ∈ n_out ⟹ out ∉ vars_defined (filter is_fby n_eqs)
  n_nodup : NoDup (n_in ++ n_vars ++ n_out)
   n_good : ∀ x ∈ (n_in ++ n_vars ++ n_out), AtomOrGensym prefs x ∧ atom n_name
```

Syntactic properties

Properties

```
  n_ingt0 :  0 < length n_in
 n_outgt0 :  0 < length n_out
   n_defd : Permutation (vars_defined n_eqs) (n_vars ++ n_out)
 n_lastd1 : Permutation (last_defined n_eqs) (filter islast n_vars)
 n_lastd2 : ∀ x, x ∈ n_vars ⟹ islast x ⟹ x ∈ vars_defined (filter is_def_cexp n_eqs)
   n_vout : ∀ out, out ∈ n_out ⟹ out ∉ vars_defined (filter is_fby n_eqs)
  n_nodup : NoDup (n_in ++ n_vars ++ n_out)
   n_good : ∀ x ∈ (n_in ++ n_vars ++ n_out), AtomOrGensym prefs x ∧ atom n_name
```

Proofs

- `n_ingt0`, `n_outgt0`: immediate
- `n_defd`, `n_lastd1`, `n_lastd2`, `n_nodup`: proof for one equation then by induction on `n_eqs`
- `n_vout`: induced by `n_nodup`
- `n_good`: immediate after a little change in the definition of `node`

Typing preservation

Presentation

- $\Gamma \vdash_{\text{wt}} e$ (*e is well-typed in* Γ) iff every variable x of type τ in e satisfies $(x, \tau) \in \Gamma$.
- A node n is well-typed iff:
  - every equation is well-typed in $\Gamma :=$ `n_in` $++$ `n_out` $++$ `n_vars` and in its program;
  - each type of `n_in`, `n_out` or `n_vars` is `int`, `float` or an existing enumeration;

Typing preservation

Presentation

- $\Gamma \vdash_{\text{wt}} e$ (*e is well-typed in* Γ) iff every variable x of type τ in e satisfies $(x, \tau) \in \Gamma$.
- A node n is well-typed iff:
  - every equation is well-typed in $\Gamma :=$ `n_in` $+\!\!+$ `n_out` $+\!\!+$ `n_vars` and in its program;
  - each type of `n_in`, `n_out` or `n_vars` is `int`, `float` or an existing enumeration;
  - each variable of each clock type of `n_in`, `n_out` or `n_vars` is well-typed.

Typing preservation

Presentation

- $\Gamma \vdash_{\text{wt}} e$ (*e is well-typed in* Γ) iff every variable x of type τ in e satisfies $(x, \tau) \in \Gamma$.
- A node n is well-typed iff:
  - every equation is well-typed in $\Gamma :=$ `n_in` $++$ `n_out` $++$ `n_vars` and in its program;
  - each type of `n_in`, `n_out` or `n_vars` is `int`, `float` or an existing enumeration;
  - each variable of each clock type of `n_in`, `n_out` or `n_vars` is well-typed.
- A program is well-typed if every node is well-typed with no duplicate name.

Typing preservation

Presentation

- $\Gamma \vdash_{\text{wt}} e$ (*e is well-typed in Γ*) iff every variable x of type τ in e satisfies $(x, \tau) \in \Gamma$.
- A node n is well-typed iff:
  - every equation is well-typed in $\Gamma :=$ `n_in` ++ `n_out` ++ `n_vars` and in its program;
  - each type of `n_in`, `n_out` or `n_vars` is `int`, `float` or an existing enumeration;
  - each variable of each clock type of `n_in`, `n_out` or `n_vars` is well-typed.
- A program is well-typed if every node is well-typed with no duplicate name.

Goal: program well-typed before inlining $\Longrightarrow$ program well-typed after inlining

Typing preservation

Proofs

- Renaming variables preserves typing:

$$\frac{\Gamma \vdash_{\mathrm{wt}} e}{\sigma(\Gamma) \vdash_{\mathrm{wt}} \sigma(e)}$$

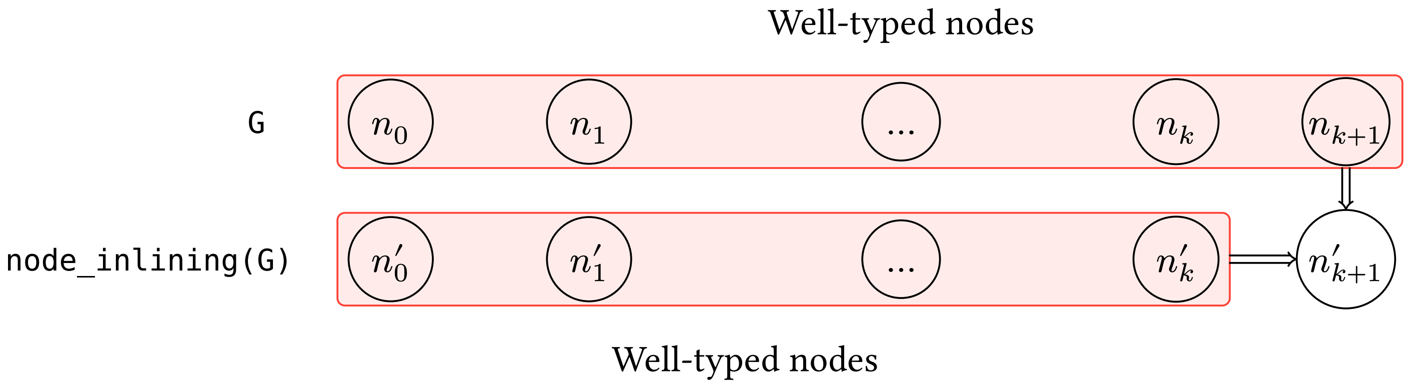
Typing preservation

Proofs

- Renaming variables preserves typing:

$$\frac{\Gamma \vdash_{\text{wt}} e}{\sigma(\Gamma) \vdash_{\text{wt}} \sigma(e)}$$

- Complete proof scheme: induction on the list of nodes $[n_0, ..., n_k]$ of the program G



Semantics preservation

Streams

Two equivalent[1] definitions: indexed and coinductive.

Here, only the indexed definition is used.

```
Definition stream (A : Type) := nat -> A. (* Indexed streams *)
```

Rocq

[1]L. Brun, “Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset,” 2020. [Online]. Available: https://www.leliobrun.net/files/thesis.pdf

Semantics preservation

Streams

Two equivalent[1] definitions: indexed and coinductive.

Here, only the indexed definition is used.

Rocq

```
Definition stream (A : Type) := nat -> A. (* Indexed streams *)
```

Environment and histories

Rocq

```
Inductive svalue : Type := absent | present (v : value).
Definition env : Type := ident -> option svalue.
Definition history : Type := stream env.
```

[1]L. Brun, “Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset,” 2020. [Online]. Available: https://www.leliobrun.net/files/thesis.pdf

Semantics preservation

Streams

Two equivalent[1] definitions: indexed and coinductive.

Here, only the indexed definition is used.

```
Definition stream (A : Type) := nat -> A. (* Indexed streams *)
```
Rocq

Environment and histories

```
Inductive svalue : Type := absent | present (v : value).
Definition env : Type := ident -> option svalue.
Definition history : Type := stream env.
```
Rocq

inc	1	3	5	2	6	4	...
n	1	2	3	4	5	6	...
sum	1	4	9	11	17	21	...
mean	1	2	3	2.75	3.4	3.5	...

[1]L. Brun, "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset," 2020. [Online]. Available: https://www.leliobrun.net/files/thesis.pdf

Semantics preservation

Semantics of a node

The semantics of an equation/a node are mutually recursive.

```
Inductive sem_node :=
  | SNode (* ... *) : (* ... *) ->
    find_node f G = n ->
    Forall (sem_equation (clock_of ins) H) n.(n_eqs) ->
    sem_node G f ins outs.
```

Rocq

```
Inductive sem_equation :=
  | SEqApp (* ... *) : (* ... *) ->
    (∀ k, sem_node f (mask k rs ins) (mask k rs outs)) ->
    sem_equation G bk H (xs =ck reset f(args) every rs)
```

Rocq

Semantics preservation

Semantics of a node

The semantics of an equation/a node are mutually recursive.

Rocq

```
Inductive sem_node :=
  | SNode (* ... *) : (* ... *) ->
    find_node f G = n ->
    Forall (sem_equation (clock_of ins) H) n.(n_eqs) ->
    sem_node G f ins outs.
```

Rocq

```
Inductive sem_equation :=
  | SEqApp (* ... *) : (* ... *) ->
    (∀ k, sem_node f (mask k rs ins) (mask k rs outs)) ->
    sem_equation G bk H (xs =ck reset f(args) every rs)
```

Rocq

```
Theorem node_inlining_sem (* ...*) :
  wt_global G ->
  sem_node G f ins outs ->
  sem_node (node_inlining G) f ins outs.
```

Semantics preservation

lustre

```
node f(wi, xi : int) returns (wo, xo : int);
let
  wo = wi + 1;
  xo = wi * 4;
tel

node g(x : int) returns (y : int);
var z : int;
let (y, z) = f(x + 2, z + 3);
tel
```

lustre

```
node g(x : int) returns (y : int);
var wi', xi', wo', xo', z : int;
let
  wo' = wi' + 1;
  xo' = wi' * 4;
  wi' = x + 2;
  xi' = z + 3;
  y = xo';
  z = wo';
tel
```

Rocq

```
Theorem node_inlining_sem (* ...*) :
  wt_global G ->
  sem_node G f ins outs ->
  sem_node (node_inlining G) f ins outs.
```

H : [x, y, z]

H': [wi', xi', wo', xo']

Semantics preservation

Lift reset

```
Inductive sem_node :=
  | SNode (* ... *) : (* ... *) ->
    find_node f G = n ->
    Forall (sem_equation (clock_of ins) H) n.(n_eqs) ->
    sem_node G f ins outs.
```

```
Inductive sem_equation :=
  | SEqApp (* ... *) : (* ... *) ->
    (∀ k, sem_node f (mask k rs ins) (mask k rs outs)) ->
    sem_equation G bk H (xs =ck reset f(args) every rs)
```

x	1	5	3	2	6	4	8	...
rs	F	F	T	F	T	F	F	...
mask^0 sum	1	6						...
mask^1 sum			3	5				...
mask^2 sum					6	10	18	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	...
sum	1	6	3	5	6	10	18	...

Semantics preservation

Lift reset

Rocq
```
Inductive sem_node :=
  | SNode (* ... *) : (* ... *) ->
    find_node f G = n ->
    Forall (sem_equation (clock_of ins) H) n.(n_eqs) ->
    sem_node G f ins outs.
```

Rocq
```
Inductive sem_equation :=
  | SEqApp (* ... *) : (* ... *) ->
    (∀ k, sem_node f (mask k rs ins) (mask k rs outs)) ->
    sem_equation G bk H (xs =ck reset f(args) every rs)
```

Rocq
```
Lemma sem_node_unmask (*...*) :
  (∀ (k: nat), sem_node G f (mask k rs ins) (mask k rs outs)) ->
  ∃ (H' : history), (* ... *)
  ∧ Forall (sem_equation G bk H') (map (lift_reset rs) n.(n_eqs)).
```

x	1	5	3	2	6	4	8	...
rs	F	F	T	F	T	F	F	...
mask^0 sum	1	6						...
mask^1 sum			3	5				...
mask^2 sum					6	10	18	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	...
sum	1	6	3	5	6	10	18	...

Semantics preservation

Rocq

```
Definition lift_reset (rs : list ident) (eq : equation) : equation :=
  match eq with
  | EqDef x ck e => EqDef x ck e
  | EqLast x ty ck c rs' => EqLast x ty ck c (rs ++ rs')
  | EqApp xs ck f args rs => EqApp xs ck f args (rs ++ rs')
  | EqFby x ck c e rs => EqFby x ck c e (rs ++ rs')
  end.
```

Rocq

```
Lemma sem_node_unmask (*...*) :
  (∀ (k: nat), sem_node G f (mask k rs ins) (mask k rs outs)) ->
    ∃ (H' : history), (* ... *) ∧ Forall (sem_equation G bk H') (map (lift_reset rs) n.(n_eqs)).
```

Semantics preservation

Semantics of an equation with a `lift_reset`

```
Lemma sem_equation_lift_reset (* ... *) : (* ... *) ->
  (∀ (k : nat), sem_equation G (mask k rs bk) H' eq) ->
    sem_equation G bk (H ⊎ H') (lift_reset rs eq).
```

Semantics preservation

Semantics of an equation with a `lift_reset`

Rocq

```
Lemma sem_equation_lift_reset (* ... *) : (* ... *) ->
  (∀ (k : nat), sem_equation G (mask k rs bk) H' eq) ->
    sem_equation G bk (H ⊎ H') (lift_reset rs eq).
```

lustre

```
reset {
  xs = reset exp every rs'
} every rs
```

Semantics preservation

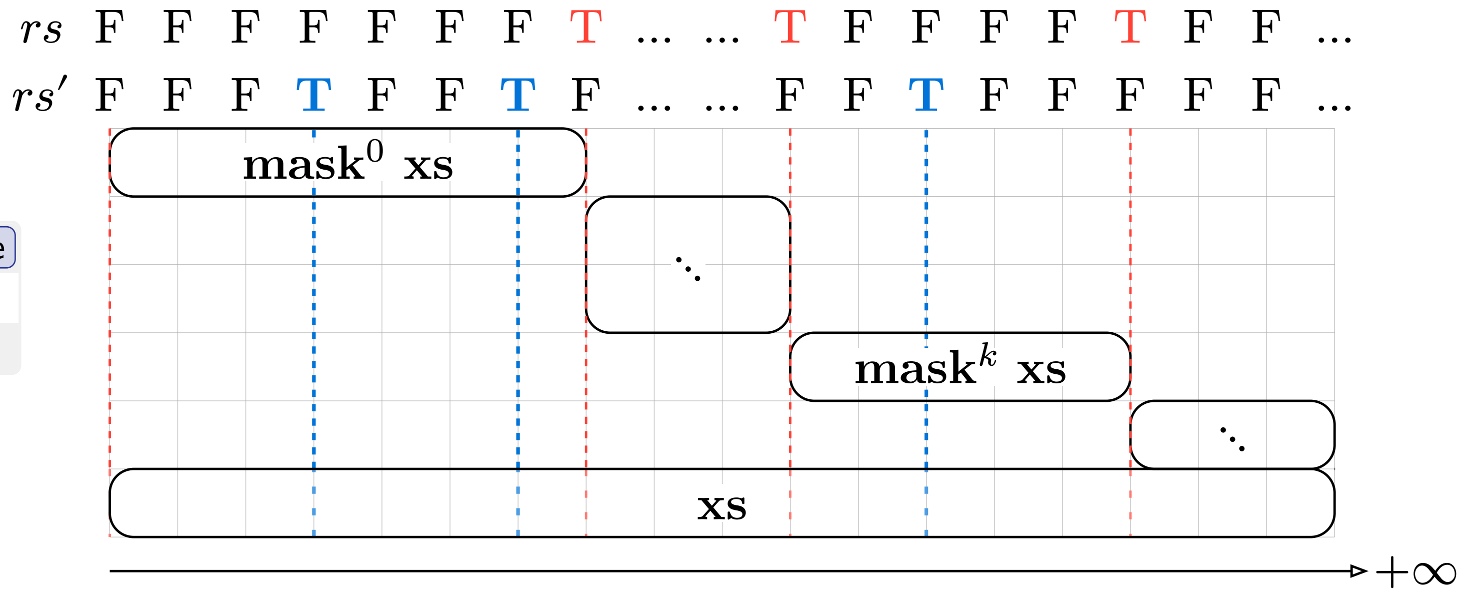
Semantics of an equation with a `lift_reset`

```coq
Lemma sem_equation_lift_reset (* ... *) : (* ... *) ->
  (∀ (k : nat), sem_equation G (mask k rs bk) H' eq) ->
    sem_equation G bk (H ⊎ H') (lift_reset rs eq).
```

Rocq

```lustre
reset {
  xs = reset exp every rs'
} every rs
```

lustre



Outline

Conclusion

What has been done

- Implementation of node inlining compilation pass
- Syntactic proofs and typing preservation complete
- Big part of semantics preservation done

Conclusion

What has been done

- Implementation of node inlining compilation pass
- Syntactic proofs and typing preservation complete
- Big part of semantics preservation done

What is remaining

- Would it have been easier in Lustre?
- Reestablish compiler correctness
- Improve dependency analysis to be able to compile more Lustre programs