Towards the verified compilation of Lustre

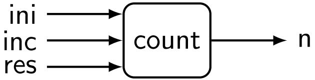
Timothy Bourke[1,2] Pierre-Évariste Dagand[3] Marc Pouzet[4,2,1]
Lionel Rieg[5]

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

```
          S E M I N A I R E
      __
     / _`  _   /  /  o
    /   ) __) /  /  / /  / /\/|
  (___/ (_/ (_ (_ / (__/ /    |
```

April 2016

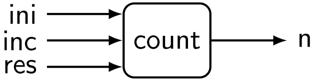
Lustre [Caspi et al. (1987): "LUSTRE: A declarative language for programming synchronous systems"]




```
node count (ini, inc: int; res: bool)
    returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

```
val count(ini : int :: .; inc : int :: .; res : bool :: .)
returns (n : int :: .)
```

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



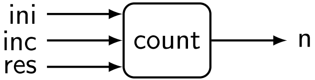

```
node count (ini, inc: int; res: bool)
   returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

val count(ini : int :: .; inc : int :: .; res : bool :: .)
returns (n : int :: .)

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
b	T	F	F	F	F	F	F	···
c	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

- Semantic model: discrete streams.
- Nodes define a (functional) relation between input and output streams.
- Sets of 'causal' equations/definitions (always variable at left).

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

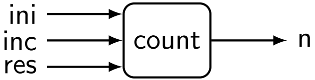



```
node count (ini, inc: int; res: bool)
   returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

```
val count(ini : int :: .; inc : int :: .; res : bool :: .)
returns (n : int :: .)
```

```
node COUNT (init, incr: int; reset: bool)
   returns (n: int);
let
   n = init ->
       if reset then init else pre(n) + incr;
tel;
```

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




```
node count (ini, inc: int; res: bool)
   returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

val count(ini : int :: .; inc : int :: .; res : bool :: .)
returns (n : int :: .)

```
node count (ini, inc: int) returns (n: int)
let
  n = if (true fby false) then ini else (0 fby n) + inc;
tel
```

```
node nats (res: bool) returns (n: int)
let
  reset
    n = count(0, 1)
  every res
tel
```

Sampling and Merging

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
r	0	1	3	4	6	9	9	12	...
(c_1)	0	0	1	3	4	6	9	9	...
r when sec				4		9	9		...
t				1		2	3		...
(c_2)				0		1	2		...
0 fby v	0	0	0	0	4	4	4	3	...
(0 fby v) whenot sec	0	0	0		4			3	...
v	0	0	0	4	4	4	3	3	...

Sampling and Merging

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

delta	0	1	2	1	2	3	0	3	···
sec	F	F	F	T	F	T	T	F	···
r	0	1	3	4	6	9	9	12	···
(c_1)	0	0	1	3	4	6	9	9	···
r when sec				4		9	9		···
t				1		2	3		···
(c_2)				0		1	2		···
0 fby v	0	0	0	0	4	4	4	3	···
(0 fby v) whenot sec	0	0	0		4			3	···
v	0	0	0	4	4	4	3	3	···

Sampling and Merging

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
r	0	1	3	4	6	9	9	12	...
(c_1)	0	0	1	3	4	6	9	9	...
r when sec				4		9	9		...
t				1		2	3		...
(c_2)				0		1	2		...
0 fby v	0	0	0	0	4	4	4	3	...
(0 fby v) whenot sec	0	0	0		4			3	...
v	0	0	0	4	4	4	3	3	...

Sampling and Merging

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯
0 fby v	0	0	0	0	4	4	4	3	⋯
(0 fby v) whenot sec	0	0	0		4			3	⋯
v	0	0	0	4	4	4	3	3	⋯

Sampling and Merging

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

delta	0	1	2	1	2	3	0	3	···
sec	F	F	F	T	F	T	T	F	···
r	0	1	3	4	6	9	9	12	···
(c_1)	0	0	1	3	4	6	9	9	···
r when sec				4		9	9		···
t				1		2	3		···
(c_2)				0		1	2		···
0 fby v	0	0	0	0	4	4	4	3	···
(0 fby v) whenot sec	0	0	0		4			3	···
v	0	0	0	4	4	4	3	3	···

Sampling and Merging

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
r	0	1	3	4	6	9	9	12	...
(c_1)	0	0	1	3	4	6	9	9	...
r when sec				4		9	9		...
t				1		2	3		...
(c_2)				0		1	2		...
0 fby v	0	0	0	0	4	4	4	3	...
(0 fby v) whenot sec	0	0	0		4			3	...
v	0	0	0	4	4	4	3	3	...

Static Clocking

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

sec	base	F	F	F	T	F	T	···
r	base	0	1	3	4	6	9	···
t	base on (sec = T)				1		2	···
(0 fby v) whenot sec	base on (sec = F)	0	0	0		4		···
v	base	0	0	0	4	4	4	···

Static Clocking

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

sec	base	F	F	F	T	F	T	⋯
r	base	0	1	3	4	6	9	⋯
t	base on (sec = T)				1		2	⋯
(0 fby v) whenot sec	base on (sec = F)	0	0	0		4		⋯
v	base	0	0	0	4	4	4	⋯

$$\frac{C \vdash e :: ck \qquad C \vdash x :: ck}{C \vdash e\ \texttt{when}\ x :: ck\ \text{on}\ (x = T)}$$

$$\frac{C \vdash x :: ck \qquad C \vdash e_t :: ck\ \text{on}\ (x = T) \qquad C \vdash e_f :: ck\ \text{on}\ (x = F)}{C \vdash \texttt{merge}\ x\ e_t\ e_f :: ck}$$

Static Clocking

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) whenot sec);
tel
```

val avgvelocity(delta : int :: .; sec : bool :: .) returns (v : int :: .)

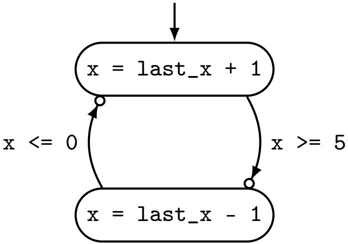
sec	base	F	F	F	T	F	T	···
r	base	0	1	3	4	6	9	···
t	base on (sec = T)				1		2	···
(0 fby v) whenot sec	base on (sec = F)	0	0	0		4		···
v	base	0	0	0	4	4	4	···

- Static inference/verification of clocking.
- "Clocks in the source language are transformed into control structures in the target language." [Biernacki et al. (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

Sampling and merging: what for?

- Provide a means of conditional activation.
- Programming directly with them can be tricky.
- Serve as a target for more complicated structures.

```
node main (go : bool)
   returns (x : int)
  var last_x : int;
let
  last_x = 0 fby x;

  automaton
  state Up
    do x = last_x + 1
    until x >= 5 then Down

  state Down
    do x = last_x − 1
    until x <= 0 then Up
  end;
tel
```



Sampling and merging: what for?

- Provide a means of conditional activation.
- Programming directly with them can be tricky.
- Serve as a target for more complicated structures.

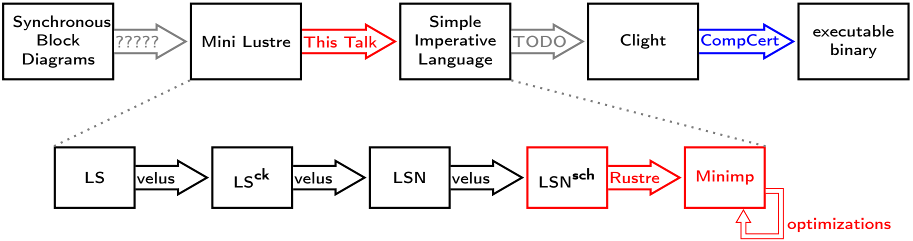
```
node main (go : bool)
   returns (x : int)
  var last_x : int;
let
  last_x = 0 fby x;

  automaton
  state Up
    do x = last_x + 1
    until x >= 5 then Down

  state Down
    do x = last_x − 1
    until x <= 0 then Up
  end;
tel
```

```
type st = St_Up | St_Down

(* ... *)

last_x = 0 fby x

x_St_Down = (last_x when St_Down(ck)) − 1
x_St_Up = (last_x when St_Up(ck)) + 1
x = merge ck (St_Down: x_St_Down)
             (St_Up: x_St_Up);

ck = St_Up fby ns
ns = ...
```

[Colaço, Pagano, and Pouzet (2005): "A Conservative Extension of Synchronous Data-flow with State Machines"]

Verifying Lustre compilation in Coq

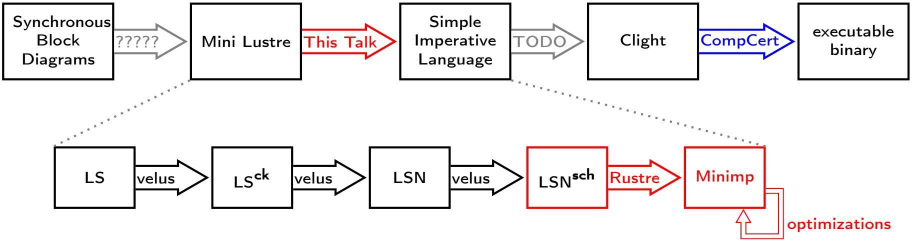
Verifying Lustre compilation in Coq



The Velus project (2008–2010, 2010–2013)

- Pouzet, Hamon, Auger, and others. [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Much formalized in Coq, some pen-and-paper proofs.
- Succession of source-to-source passes.
- Streams modelled as lists.

Verifying Lustre compilation in Coq

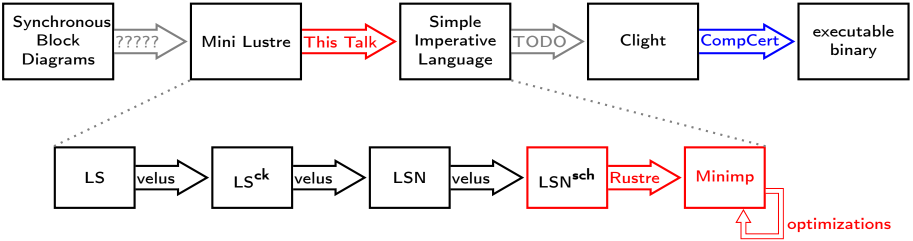


The Rustre project (2015–)

- *A bit less*: tupling ✗, modular resets ✗, n-ary merge ✗
- *A bit more*: imperative code generation and optimization in Coq.
- *A bit different*: streams as functions from nats to values.

- Complete: code generation and optimization.
- Ongoing:
  - refine semantic model / proof of existence
  - incorporate Velus passes into a complete tool chain
  - integrate with CompCert. . .

Verifying Lustre compilation in Coq



Integration with CompCert

[Blazy, Dargaye, and Leroy (2006): "Formal Verification of a C Compiler Front-End"] [Leroy (2009): "Formal verification of a realistic compiler"] [Bedin França et al. (2011): "Towards Formally Verified Optimizing Compilation in Flight Control Software"]

- Incorporate types and operations from CompCert into Lustre.
- Generate Clight from Minimp and show correctness.

- Part of ITEA 3 14014 ASSUME Project.
- Summer internship of Lélio Brun.

- What about external functions? external nodes? external types?

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

```
node avgvelocity(delta: int; sec: bool)
   returns (v: int)
  var r, t: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                ((0 fby v) whenot sec);

tel
```

normalize ⇒

```
node avgvelocity(delta: int; sec: bool)
   returns (v: int)
  var r, t, w: int;
let
  w = 0 fby v;
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w whenot sec);
tel
```

- Rewrite to give each fby and node instantiation its own equation.
- Group merges at tops of equations.
- Introduce fresh variables; exploit referential transparency.

- Proof by validation
  - External program produces w = e; eqs2.
  - Coq validator checks that substituting w = e into eqs2 gives eqs.

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

```
node avgvelocity(delta: int; sec: bool)
   returns (v: int)
 var r, t, w: int;
let
 w = 0 fby v;
 r = count(0, delta, false);
 t = count((1, 1, false) when sec);
 v = merge sec ((r when sec) / t)
                 (w whenot sec);
tel
```

schedule →

```
node avgvelocity(delta: int; sec: bool)
   returns (v: int)
 var r, t, w: int;
let
 r = count(0, delta, false);
 t = count((1, 1, false) when sec);
 v = merge sec ((r when sec) / t)
                 (w whenot sec);
 w = 0 fby v;
tel
```

- Equations semantics are independent of order;
  but correct compilation will depend on order.
- Rewrite to define most variables *before* they are read,
  ...and fby variables *after* they are read; optimize adjacencies.
- Proof by validation
  - External program generates a sequence of permutations.
  - Verified Coq function applies them successively.

Outline

A simple program in Lustre

Verifying Lustre compilation in Coq

Dataflow language: syntax and semantics

Imperative language: syntax and semantics

Relating the Dataflow and Imperative models

Optimization of control structures

Conclusion

Rustre: dataflow language

Expressions

e ::=		x	variable
	|	k	constant
	|	$e \oplus e$	operator
	|	e when $(x = k)$	sub-sampling
ce ::=		merge x ce_t ce_f	binary merge
	|	e	non-control expression

Equations

$$eq ::= x = (ce)^{ck} \mid x = (k_0 \text{ fby } e)^{ck} \mid x = (f\ e)^{ck}$$

Clocks

$$ck ::= \text{base} \mid ck \text{ on } (x = k)$$

(Scheduled) Nodes

node f $(x : \tau)$ returns $(x : \tau)$
var $x : \tau, \ldots, x : \tau$
let $eq; \cdots ; eq$ tel

```
Inductive clock : Set :=
| Cbase : clock
| Con : clock → ident → bool → clock.

Inductive lexp : Type :=
| Econst : const → lexp
| Evar : ident → lexp
| Ewhen : lexp → ident → bool → lexp.

Inductive laexp : Type :=
| LAexp : clock → lexp → laexp.

Inductive cexp : Type :=
| Emerge : ident → cexp → cexp → cexp
| Eexp : lexp → cexp.

Inductive caexp : Type :=
| CAexp : clock → cexp → caexp.

Inductive equation : Type :=
| EqDef : ident → caexp → equation
| EqApp : ident → ident → laexp → equation
| EqFby : ident → const → laexp → equation.
```

Semantics: Dataflow models

	1	2	3	4	5	6	7	...	
`n`	1	2	3	4	5	6	7	...	base
`ck`	F	F	T	F	T	T	F	...	base
`x = n when ck`			3		5	6		...	base on (ck = T)
`y = 0 fby x`			0		3	5		...	base on (ck = T)
`z = 9 whenot ck`	9	9		9			9	...	base on (ck = F)
`w = merge ck y z`	9	9	0	9	3	5	9	...	base

- Model absence

  `Inductive value := absent | present (v : const).`

  [Boulmé and Hamon (2001): *A clocked denotational semantics for Lucid-Synchrone in Coq*]

  [Paulin-Mohring (2009): "A constructive denotational semantics for Kahn networks in Coq"]

- Lists: $1 :: (2 :: (3 :: (4 :: [])))$ *or* $(((\epsilon \cdot 1) \cdot 2) \cdot 3) \cdot 4$
- Coinductive streams?
- Functions from natural numbers to values

  `Notation stream A := (nat → A).`

Rustre: semantic model

```
Definition global := list node.
Definition history := Coq.FSets.FMapPositive.PositiveMap.t (stream value).

Inductive sem_equation (G: global)
  : history → equation → Prop :=

| (SEqDef:)
    sem_var H x xs →
    sem_cexp H ce xs →
  ─────────────────────────────
    sem_equation G H (EqDef x ck ce)
```

$x = (ce)^{ck}$

```
| (SEqApp:)
    sem_lexp H le ls →
    sem_var H x xs →
    sem_node G f ls xs →
  ─────────────────────────────
    sem_equation G H (EqApp x ck f le)
```

$x = (f\ le)^{ck}$

```
| (SEqFby:)
    sem_lexp H le ls →
    sem_var H x xs →
    xs = fby v0 ls →
  ─────────────────────────────
    sem_equation G H (EqFby x ck v0 le)
```

$x = (v0\ fby\ le)^{ck}$

```
with sem_node (G: global)
  : ident
    → stream value
    → stream value
    → Prop :=

| (SNode:)
    find_node f G
      = Some (mk_node f i o eqs) →
    (∃ H, sem_var H i xs
      ∧ sem_var H o ys
      ∧ ...
      ∧ Forall (sem_equation G H) eqs)
  ─────────────────────────────
    → sem_node G f xs ys.
```

Rustre: semantic model

```
Definition global := list node.
Definition history := Coq.FSets.FMapPositive.PositiveMap.t (stream value).
```

```
Inductive sem_equation (G: global)
  : history → equation → Prop :=

| (SEqDef:)
    sem_var H x xs →
    sem_cexp H ce xs →
    ─────────────────────────────
    sem_equation G H (EqDef x ck ce)
```

$x = (ce)^{ck}$

```
| (SEqApp:)
    sem_lexp H le ls →
    sem_var H x xs →
    sem_node G f ls xs →
    ─────────────────────────────
    sem_equation G H (EqApp x ck f le)
```

$x = (f\ le)^{ck}$

```
| (SEqFby:)
    sem_lexp H le ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ─────────────────────────────
    sem_equation G H (EqFby x ck v0 le)
```

$x = (v0\ fby\ le)^{ck}$

$f : \text{stream}(T) \to \text{stream}(T')$

```
with sem_node (G: global)
  : ident
    → stream value
    → stream value
    → Prop :=

| (SNode:)
    find_node f G
       = Some (mk_node f i o eqs) →
    (∃ H, sem_var H i xs
      ∧ sem_var H o ys
      ∧ ...
      ∧ Forall (sem_equation G H) eqs)
    ─────────────────────────────
    → sem_node G f xs ys.
```

Rustre: semantic model: fby

$$\texttt{fby}^{\#}_{v_0}(v.s) = v_0.\texttt{fby}^{\#}_{v}(s)$$
$$\texttt{fby}^{\#}_{v_0}(abs.s) = abs.\texttt{fby}^{\#}_{v_0}(s)$$
$$\texttt{fby}^{\#}_{v_0}(\epsilon) = \epsilon$$

```
Fixpoint hold (v0: const) (xs: stream value) (n: nat) : const :=
  match n with
  | 0 ⇒ v0
  | S m ⇒ match xs m with
          | absent ⇒ hold v0 xs m
          | present hv ⇒ hv
          end
  end.

Definition fby (v0: const) (xs: stream value) (n: nat) : value :=
  match xs n with
  | absent ⇒ absent
  | _ ⇒ present (hold v0 xs n)
  end.
```

Outline

A simple program in Lustre

Verifying Lustre compilation in Coq

Dataflow language: syntax and semantics

Imperative language: syntax and semantics

Relating the Dataflow and Imperative models

Optimization of control structures

Conclusion

Minimp: imperative language

$c ::=$	x	variable
$\mid$	$\mathsf{mem}(x)$	memory
$\mid$	k	constant
$\mid$	$e \oplus e$	operator
$s ::=$	$x := c$	variable assignment
$\mid$	$\mathsf{mem}(x) := c$	memory assignment
$\mid$	$x := f.\mathsf{step}\ o\ (x)$	node transition assignment (and update)
$\mid$	$f.\mathsf{reset}\ o$	initialize node memory
$\mid$	$\mathsf{if}\ c\ \{s\}\ \mathsf{else}\ \{s\}$	conditional branching
$\mid$	$s;\ s$	sequential composition
$\mid$	skip	nop

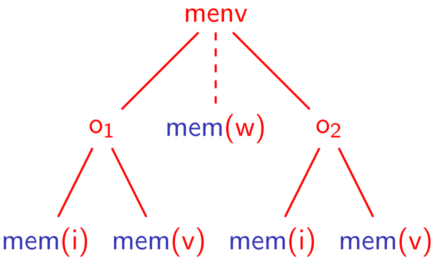
Generation of imperative code [Biernacki et al. (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

```
node avgvelocity(delta: int; sec: bool)
    returns (v: int)
  var r, t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w whenot sec);
  w = 0 fby v;
tel
```

```
memory w;
instance o1, o2;

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

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

reset() returns () {
  count.reset o1;
  count.reset o2;
  mem(w) := 0
}
```

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

```
node avgvelocity(delta: int; sec: bool)
    returns (v: int)
  var r, t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w whenot sec);
  w = 0 fby v;
tel
```




```
memory w;
instance o1, o2;

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

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

reset() returns () {
  count.reset o1;
  count.reset o2;
  mem(w) := 0
}
```

Minimp: semantic model

```
Inductive stmt_eval :
  program → heap → stack → stmt → heap * stack → Prop :=

| Iassign:
      exp_eval menv env e v →
      PM.add x v env = env' →
    ------------------------------------------------
      stmt_eval prog menv env (Assign x e) (menv, env')
```

x := e

```
| Iassignst:
      exp_eval menv env e v →
      madd_mem x v menv = menv' →
    ------------------------------------------------
      stmt_eval prog menv env (AssignSt x e) (menv', env)
```

mem(x) := e

```
| Istep:
      exp_eval menv env e v →
      mfind_inst o menv = Some(omenv) →
      stmt_step_eval prog omenv fcls v omenv' rv →
      madd_obj o omenv' menv = menv' →
      PM.add x rv env  = env' →
    ------------------------------------------------
      stmt_eval prog menv env (Step_ap x fcls o e) (menv', env')
⋮
```

x := fcls.step o (e)

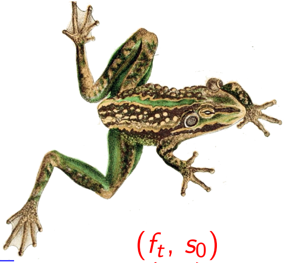
Minimp: semantic model

```
Inductive stmt_eval :
  program → heap → stack → stmt → heap ∗ stack → Prop :=

| Iassign:
    exp_eval menv env e v →
    PM.add x v env = env' →
    ------------------------------------------------
    stmt_eval prog menv env (Assign x e) (menv, env')
```

x := e

```
| Iassignst:
    exp_eval menv env e v →
    madd_mem x v menv = menv' →
    ------------------------------------------------
    stmt_eval prog menv env (AssignSt x e) (menv', env)
```

mem(x) := e



(f_t, s_0)

$S \times T \to T' \times S \qquad S$

```
| Istep:
    exp_eval menv env e v →
    mfind_inst o menv = Some(omenv) →
    stmt_step_eval prog omenv fcls v omenv' rv →
    madd_obj o omenv' menv = menv' →
    PM.add x rv env  = env' →
    ------------------------------------------------
    stmt_eval prog menv env (Step_ap x fcls o e) (menv', env')
⋮
```

x := fcls.step o (e)

Outline

A simple program in Lustre

Verifying Lustre compilation in Coq

Dataflow language: syntax and semantics

Imperative language: syntax and semantics

Relating the Dataflow and Imperative models

Optimization of control structures

Conclusion

Is well scheduled

```
Variables (mems : PS.t) (arg: Nelist.nelist ident).

Inductive Is_well_sch : list equation → Prop :=

| WSchNil: Is_well_sch []

| WSchEq:
    Is_well_sch eqs →
    (∀ i, Is_free_in_eq i eq →
              (¬PS.In i mems → Is_variable_in i eqs
                                ∨ Nelist.In i arg)
            ∧ (PS.In i mems → ¬Is_defined_in i eqs)) →
    (∀ i, Is_defined_in_eq i eq → ¬Is_defined_in i eqs) →
    Is_well_sch (eq :: eqs).

| ...
```

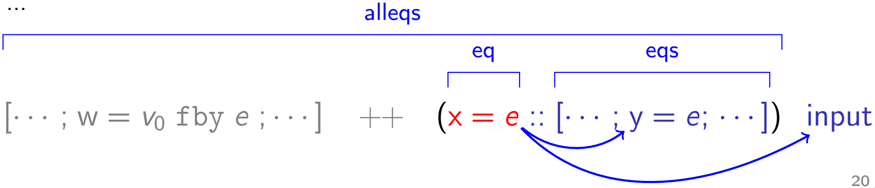
alleqs

eq eqs

$[\cdots ; w = v_0 \text{ fby } e ; \cdots] \quad +\!+ \quad (x = e :: [\cdots ; y = e; \cdots])$ input

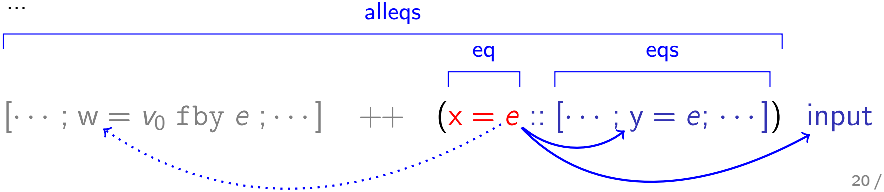
Is well scheduled

```
Variables (mems : PS.t) (arg: Nelist.nelist ident).

Inductive Is_well_sch : list equation → Prop :=

| WSchNil: Is_well_sch []

| WSchEq:
      Is_well_sch eqs →
      (∀ i, Is_free_in_eq i eq →
              (¬PS.In i mems → Is_variable_in i eqs
                                  ∨ Nelist.In i arg)
            ∧ (PS.In i mems → ¬Is_defined_in i eqs)) →
      (∀ i, Is_defined_in_eq i eq → ¬Is_defined_in i eqs) →
      Is_well_sch (eq :: eqs).

| ...
```



Is well scheduled

```
Variables (mems : PS.t) (arg: Nelist.nelist ident).

Inductive Is_well_sch : list equation → Prop :=

| WSchNil: Is_well_sch []

| WSchEq:
    Is_well_sch eqs →
    (∀ i, Is_free_in_eq i eq →
            (¬PS.In i mems → Is_variable_in i eqs
                                ∨ Nelist.In i arg)
          ∧ (PS.In i mems → ¬Is_defined_in i eqs)) →
    (∀ i, Is_defined_in_eq i eq → ¬Is_defined_in i eqs) →
    Is_well_sch (eq :: eqs).

| ...
```



Translation: definition

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

Fixpoint Control (ck: clock) (s: stmt) : stmt :=
  match ck with
  | Cbase ⇒ s
  | Con ck x true  ⇒ Control ck (Ifte (tovar x) s Skip)
  | Con ck x false ⇒ Control ck (Ifte (tovar x) Skip s)
  end.

Fixpoint translate_cexp (x: ident) (e : cexp) {struct e} : stmt :=
  match e with
  | Emerge y t f ⇒ Ifte (tovar y) (translate_cexp x t) (translate_cexp x f)
  | Eexp l ⇒ Assign x (translate_lexp l)
  end.

Definition translate_eqn (eqn: equation) : stmt :=
  match eqn with
  | EqDef x (CAexp ck ce) ⇒   Control ck (translate_cexp x ce)
  | EqApp x f (LAexp ck le) ⇒ Control ck (Step_ap x f x (translate_lexp le))
  | EqFby x v (LAexp ck le) ⇒ Control ck (AssignSt x (translate_lexp le))
  end.
```

Translation: definition

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

Fixpoint Control (ck: clock) (s: stmt) : stmt :=
  match ck with
  | Cbase ⇒ s
  | Con ck x true  ⇒ Control ck (Ifte (tovar x) s Skip)
  | Con ck x false ⇒ Control ck (Ifte (tovar x) Skip s)
  end.

Fixpoint translate_cexp (x: ident)(e : cexp) {struct e} : stmt := ...

Definition translate_eqn (eqn: equation) : stmt :=
  match eqn with
  | EqDef x (CAexp ck ce) ⇒   Control ck (translate_cexp x ce)
  | EqApp x f (LAexp ck le) ⇒ Control ck (Step_ap x f x (translate_lexp le))
  | EqFby x v (LAexp ck le) ⇒ Control ck (AssignSt x (translate_lexp le))
  end.

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

Correctness theorem

```
Variables (G     : global)
          (Hwdef : Welldef_global G).
```

```
Theorem is_event_loop_correct:
  sem_node G main xss ys →
  ∀ n, ∃ menv env,
     step (S n) (translate G) r main obj css menv env
     ∧ (∀ co, ys n ≑ present co ↔ PM.find r env = Some co).
```

$\longleftarrow co_0 \cdot co_1 \cdot co_2 \cdots = f(xss_0 \cdot xss_1 \cdot xss_2 \cdots)$

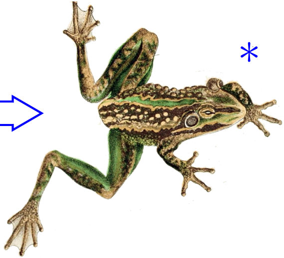
clock-directed translation

```
Fixpoint step (n: nat) P r main obj css menv' env': Prop :=
  match n with
    | 0 ⇒ stmt_eval P hempty sempty (Reset_ap main obj) (menv', env')
    | S n ⇒ let xss := Nelist.map Const (css n) in
      ∃ menvN envN,
        step n P r main obj css menv env
        ∧ stmt_eval P menv env (Step_ap r main obj xss) (menv', env')
  end.
```

```
f.reset obj;
repeat (n + 1) {
    r := f.step obj (css n)
}
```

Induction on n: (internal) memories

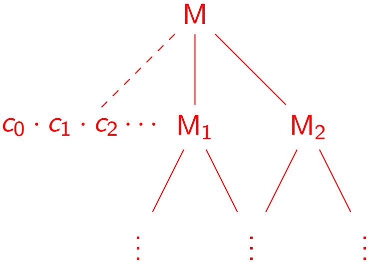
sem_node G f xs ys



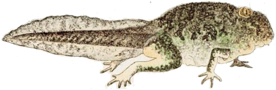
*

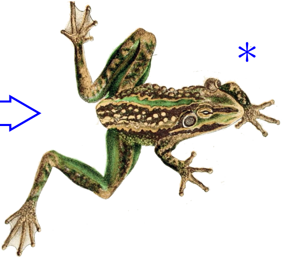
stmt_eval (translate G) menv env (r := f.step obj (ci)) (menv', env')

Induction on *n*: (internal) memories



∃M, msem_node G f xs M ys





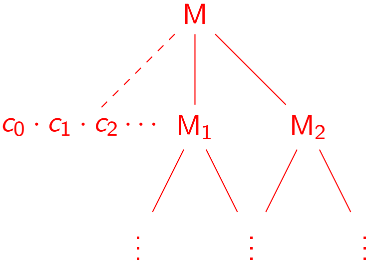
*

sem_node G f xs ys

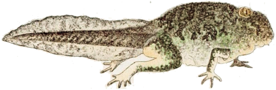
stmt_eval (translate G) menv env (r := f.step obj (ci)) (menv', env')

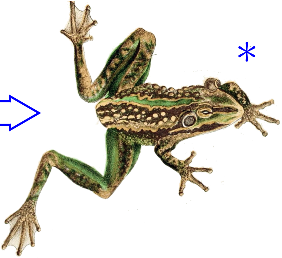
Induction on n: (internal) memories

```
Inductive memory (A: Type): Type := mk_memory {
  mm_values : PM.t A;
  mm_instances : PM.t (memory A)
}.

Definition memory := memory (stream const).
```



∃M, msem_node G f xs M ys





sem_node G f xs ys

stmt_eval (translate G) menv env (r := f.step obj (ci)) (menv', env')

```
Inductive sem_equation (G: global)
  : history → equation → Prop :=

| (SEqDef: )
    sem_var H x xs →
    sem_caexp H cae xs →
   ------------------------------
    sem_equation G H (EqDef x cae)
```

$$x = (ce)^{ck}$$

...

```
| (SEqFby: )
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
   ------------------------------
    sem_equation G H (EqFby x v0 lae)
```

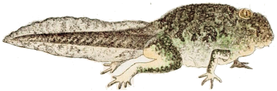
$$x = (v0 \text{ fby } le)^{ck}$$

```
Inductive sem_equation (G: global)
  : history → equation → Prop :=

| (SEqDef:)
    sem_var H x xs →
    sem_caexp H cae xs →
    ────────────────────────────
    sem_equation G H (EqDef x cae)
```

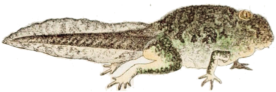
$x = (ce)^{ck}$

```
...

| (SEqFby:)
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ────────────────────────────
    sem_equation G H (EqFby x v0 lae)
```

$x = (v0 \text{ fby } le)^{ck}$



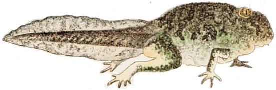
```
Inductive msem_equation G
  : history → memory → equation
    → Prop :=
| SEqDef:
    sem_var H x xs →
    sem_caexp H cae xs →
    ────────────────────────────
    msem_equation G H M (EqDef x cae)
...
| SEqFby:
    mfind_mem x M = Some ms →
    ms 0 = v0 →
    sem_laexp H lae ls →
    sem_var H x xS →
    (∀ n,
      match ls n with
      | absent    ⇒ ms (S n) = ms n
                    ∧ xs n = absent
      | present v ⇒ ms (S n) = v
                    ∧ xs n = present (ms n)
      end) →
    ────────────────────────────
    msem_equation G H M (EqFby x v0 lae)
```



	0	1	2	3	4	5	6		
n	0	1	2	3	4	5	6	...	base
ck	F	F	T	F	T	T	F	...	base
x = n when (ck = T)			3		5	6		...	base on (ck = T)
y = 0 fby x			0		3	5		...	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	...	base

```
...
| SEqFby:
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ─────────────────────────────
    sem_equation G H (EqFby x v0 lae)
```

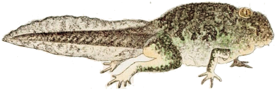
```
...
| SEqFby:
    mfind_mem x M = Some ms →
    ms 0 = v0 →
    sem_laexp H lae ls →
    sem_var H x xS →
    (∀ n,
      match ls n with
      | absent    ⇒ ms (S n) = ms n
                    ∧ xs n = absent
      | present v ⇒ ms (S n) = v
                    ∧ xs n = present (ms n)
      end) →
    ─────────────────────────────
    msem_equation G H M (EqFby x v0 lae)
```



	0	1	2	3	4	5	6		
n	0	1	2	3	4	5	6	…	base
ck	F	F	T	F	T	T	F	…	base
x = n when (ck = T)			3		5	6		…	base on (ck = T)
y = 0 fby x			0		3	5		…	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	…	base

```
...

| SEqFby:
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ─────────────────────────────
    sem_equation G H (EqFby x v0 lae)
```

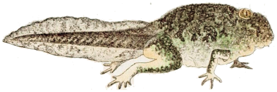
```
...
| SEqFby:
    mfind_mem x M = Some ms →
    ms 0 = v0 →
    sem_laexp H lae ls →
    sem_var H x xS →
    (∀ n,
      match ls n with
      | absent    ⇒ ms (S n) = ms n
                    ∧ xs n = absent
      | present v ⇒ ms (S n) = v
                    ∧ xs n = present (ms n)
      end) →
    ─────────────────────────────
    msem_equation G H M (EqFby x v0 lae)
```



n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

```
...

| SEqFby:
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ─────────────────────────────────
    sem_equation G H (EqFby x v0 lae)
```

```
...
| SEqFby:
    mfind_mem x M = Some ms →
    ms 0 = v0 →
    sem_laexp H lae ls →
    sem_var H x xS →
    (∀ n,
      match ls n with
      | absent    ⇒ ms (S n) = ms n
                    ∧ xs n = absent
      | present v ⇒ ms (S n) = v
                    ∧ xs n = present (ms n)
      end) →
    ─────────────────────────────────────
    msem_equation G H M (EqFby x v0 lae)
```

n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

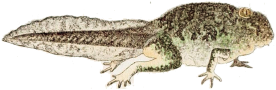
```
...

| SEqFby:
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ─────────────────────────────
    sem_equation G H (EqFby x v0 lae)
```

```
...
| SEqFby:
    mfind_mem x M = Some ms →
    ms 0 = v0 →
    sem_laexp H lae ls →
    sem_var H x xS →
    (∀ n,
      match ls n with
      | absent    ⇒ ms (S n) = ms n
                     ∧ xs n = absent
      | present v ⇒ ms (S n) = v
                     ∧ xs n = present (ms n)
      end) →
    ─────────────────────────────
    msem_equation G H M (EqFby x v0 lae)
```

n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

```
...

| SEqFby:
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ─────────────────────────────
    sem_equation G H (EqFby x v0 lae)
```

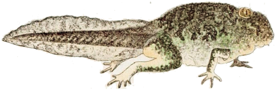
```
...
| SEqFby:
    mfind_mem x M = Some ms →
    ms 0 = v0 →
    sem_laexp H lae ls →
    sem_var H x xS →
    (∀ n,
      match ls n with
      | absent    ⇒ ms (S n) = ms n
                    ∧ xs n = absent
      | present v ⇒ ms (S n) = v
                    ∧ xs n = present (ms n)
      end) →
    ─────────────────────────────
    msem_equation G H M (EqFby x v0 lae)
```



	0	1	2	3	4	5	6		
n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

```
...

| SEqFby:
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ────────────────────────────
    sem_equation G H (EqFby x v0 lae)
```

```
...
| SEqFby:
    mfind_mem x M = Some ms →
    ms 0 = v0 →
    sem_laexp H lae ls →
    sem_var H x xS →
    (∀ n,
      match ls n with
      | absent    ⇒ ms (S n) = ms n
                    ∧ xs n = absent
      | present v ⇒ ms (S n) = v
                    ∧ xs n = present (ms n)
      end) →
    ────────────────────────────
    msem_equation G H M (EqFby x v0 lae)
```

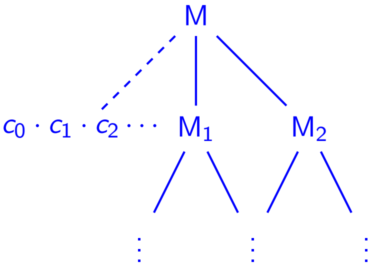


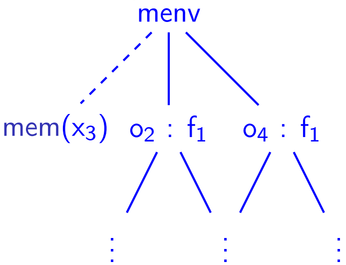
n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

```
...

| SEqFby:
    sem_laexp H lae ls →
    sem_var H x xs →
    xs = fby v0 ls →
    ─────────────────────────────
    sem_equation G H (EqFby x v0 lae)
```

```
...
| SEqFby:
    mfind_mem x M = Some ms →
    ms 0 = v0 →
    sem_laexp H lae ls →
    sem_var H x xS →
    (∀ n,
      match ls n with
      | absent    ⇒ ms (S n) = ms n
                    ∧ xs n = absent
      | present v ⇒ ms (S n) = v
                    ∧ xs n = present (ms n)
      end) →
    ─────────────────────────────
    msem_equation G H M (EqFby x v0 lae)
```

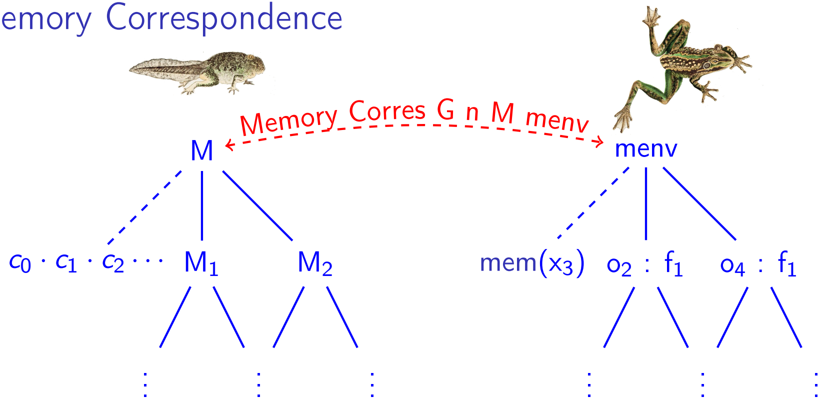
Memory Correspondence





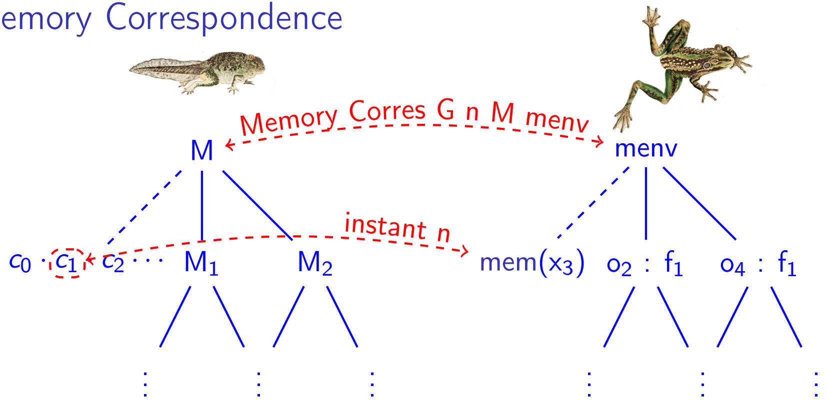

```
Inductive Memory_Corres (G: global) (n: nat) :
      ident → memory → heap → Prop :=
| MemC:
    find_node f G = Some(mk_node f i o eqs) →
    Forall (Memory_Corres_eq G n M menv) eqs →
    Memory_Corres G n f M menv
```

Memory Correspondence



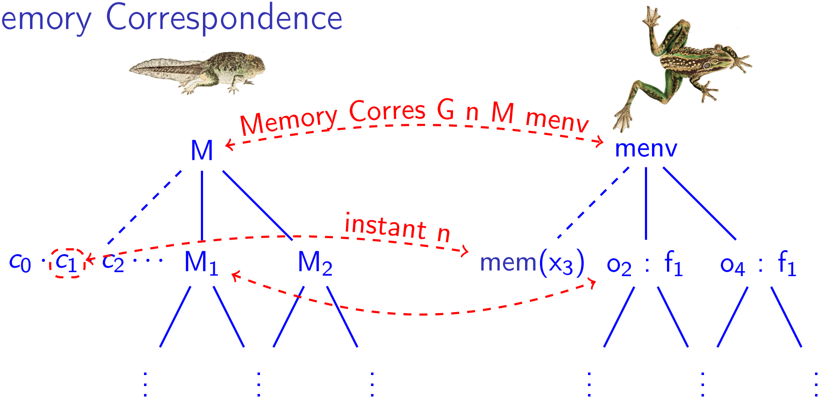

```
Inductive Memory_Corres (G: global) (n: nat) :
      ident → memory → heap → Prop :=
| MemC:
   find_node f G = Some(mk_node f i o eqs) →
   Forall (Memory_Corres_eq G n M menv) eqs →
   Memory_Corres G n f M menv
```

Memory Correspondence



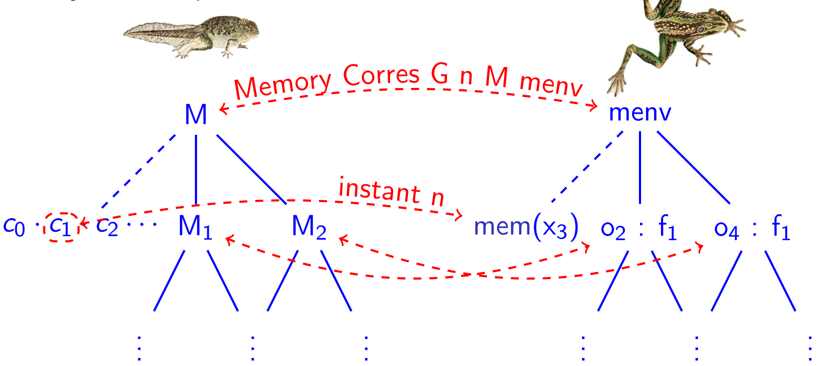

```
Inductive Memory_Corres_eq (G: global) (n: nat) :
      memory → heap → equation → Prop :=
...
| MemC_EqFby:
      (∀ ms, mfind_mem x M = Some ms
                  → mfind_mem x menv = Some (ms n))
      → Memory_Corres_eq G n M menv (EqFby x v0 lae).
```

Memory Correspondence




```
Inductive Memory_Corres_eq (G: global) (n: nat) :
      memory → heap → equation → Prop :=
...
| MemC_EqApp:
    (∀ Mo, mfind_inst x M = Some Mo →
           (∃ omenv, mfind_inst x menv = Some omenv
                     ∧ Memory_Corres G n f Mo omenv))
    → Memory_Corres_eq G n M menv (EqApp x f lae)
```

Memory Correspondence




```
Inductive Memory_Corres_eq (G: global) (n: nat) :
      memory → heap → equation → Prop :=
...
| MemC_EqApp:
    (∀ Mo, mfind_inst x M = Some Mo →
          (∃ omenv, mfind_inst x menv = Some omenv
                  ∧ Memory_Corres G n f Mo omenv))
    → Memory_Corres_eq G n M menv (EqApp x f lae)
```

```
Lemma is_step_correct:
  Forall (msem_equation G H M) alleqs
  (∃ oeqs, alleqs = oeqs ++ eqs)

  Welldef_global G →

  (∀ c, sem_var_instant (restr H n) input (present c)
           ↔ PM.find input env = Some c) →
  ¬ Is_defined_in input eqs →

  Is_well_sch mems input eqs →

  (* hypothesis for earlier nodes... *)

  Forall (Memory_Corres_eq G n M menv) alleqs →

  (∃ menv' env',
     stmt_eval (translate G) menv env
        (translate_eqns mems eqs) (menv', env')
     ∧ (∀ x, Is_variable_in x eqs →
              ∀ c, sem_var_instant (restr H n) x (present c)
                   ↔ PM.find x env' = Some c)
     ∧ Forall (Memory_Corres_eq G (S n) M menv') eqs).
```

- induction n
  - induction G
    - induction eqs
      - case: $x = (ce)^{ck}$
        - case: present
        - case: absent
      - case: $x = (f\ e)^{ck}$
        - case: present
        - case: absent
      - case: $x = (k \text{ fby } e)^{ck}$
        - case: present
        - case: absent

```
Lemma is_step_correct:
  Forall (msem_equation G H M) alleqs
  (∃ oeqs, alleqs = oeqs ++ eqs)

  Welldef_global G →
  (∀ c, sem_var_instant (restr H n) input (present c)
        ↔ PM.find input env = Some c) →
  ¬Is_defined_in input eqs →
  Is_well_sch mems input eqs →

  (* hypothesis for earlier nodes... *)

  Forall (Memory_Corres_eq G n M menv) alleqs →

  (∃ menv' env',
     stmt_eval (translate G) menv env
       (translate_eqns mems eqs) (menv', env')
     ∧ (∀ x, Is_variable_in x eqs →
             ∀ c, sem_var_instant (restr H n) x (present c)
                  ↔ PM.find x env' = Some c)
     ∧ Forall (Memory_Corres_eq G (S n) M menv') eqs).
```

alleqs: oeqs ++ eqs

oeqs: $[\cdots; w = v_0 \text{ fby } e; \cdots]$

$[\cdots; w = v_0 \text{ fby } e; \cdots] \ ++ \ (x = e :: [\cdots; y = e; \cdots])$ input

eq: $x = e$

eqs: $[\cdots; y = e; \cdots]$

induction n
- induction G
  - induction eqs
    - case: $x = (ce)^{ck}$
      - case: present
      - case: absent
    - case: $x = (f\ e)^{ck}$
      - case: present
      - case: absent
    - case: $x = (k \text{ fby } e)^{ck}$
      - case: present
      - case: absent

Outline

A simple program in Lustre

Verifying Lustre compilation in Coq

Dataflow language: syntax and semantics

Imperative language: syntax and semantics

Relating the Dataflow and Imperative models

Optimization of control structures

Conclusion

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 {
    t := count.step o2 (1, 1, false)
  };
  if sec {
    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 {
    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: requires invariant

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

➡ if e {s1; t1} else {s2; t2};

Fusion of control structures: requires invariant

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

➡ `if e {s1; t1} else {s2; t2};`

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

✗

Fusion of control structures: requires invariant

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

➡ `if e {s1; t1} else {s2; t2};`

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

✗

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

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

$\ldots$

Fusion of control structures: implementation

$$\mathrm{fuse}\left(\begin{array}{ccc} & ; & \\ / & & \backslash \\ s & & t \end{array}\right) = \mathrm{fuse}'\,(s, t)$$

$$\mathrm{fuse}\,(s) = s$$

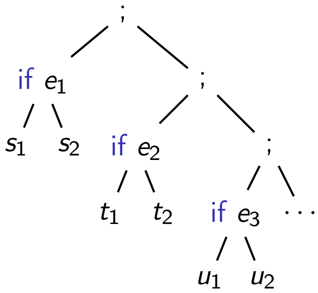
$$\mathrm{fuse}'\left(s,\ \begin{array}{ccc} & ; & \\ / & & \backslash \\ t_1 & & t_2 \end{array}\right) = \mathrm{fuse}'\,(\mathrm{zip}\,(s, t_1)\,, t_2)$$

$$\mathrm{fuse}'\,(s, t) = \mathrm{zip}\,(s, t)$$

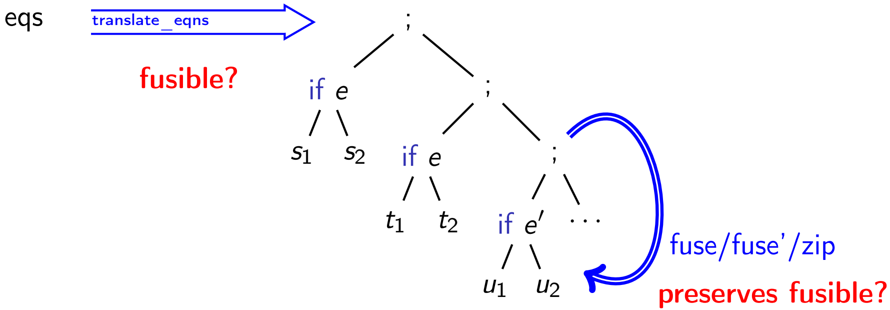
$$\mathrm{zip}\left(\begin{array}{ccc} & \textbf{if } e & \\ / & & \backslash \\ s_1 & & s_2 \end{array},\ \begin{array}{ccc} & \textbf{if } e & \\ / & & \backslash \\ t_1 & & t_2 \end{array}\right) = \begin{array}{ccc} & \textbf{if } e & \\ / & & \backslash \\ \mathrm{zip}\,(s_1, t_1) & & \mathrm{zip}\,(s_2, t_2) \end{array}$$

$$\mathrm{zip}\left(\begin{array}{ccc} & ; & \\ / & & \backslash \\ s_1 & & s_2 \end{array},\ t\right) = \begin{array}{ccc} & ; & \\ / & & \backslash \\ s_1 & & \mathrm{zip}\,(s_2, t) \end{array}$$

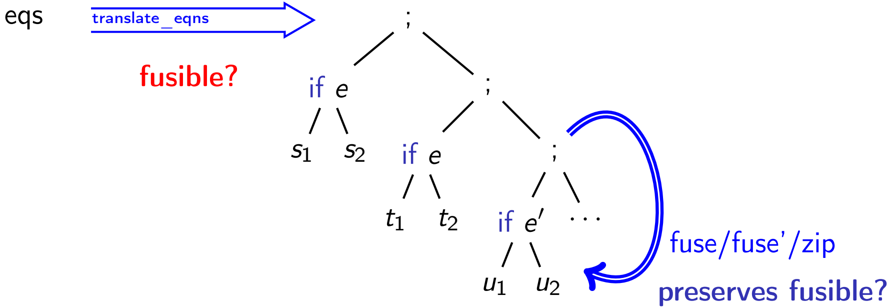
$$\mathrm{zip}\,(s, t) = \begin{array}{ccc} & ; & \\ / & & \backslash \\ s & & t \end{array}$$



Fusion of control structures: correctness



Fusion of control structures: correctness

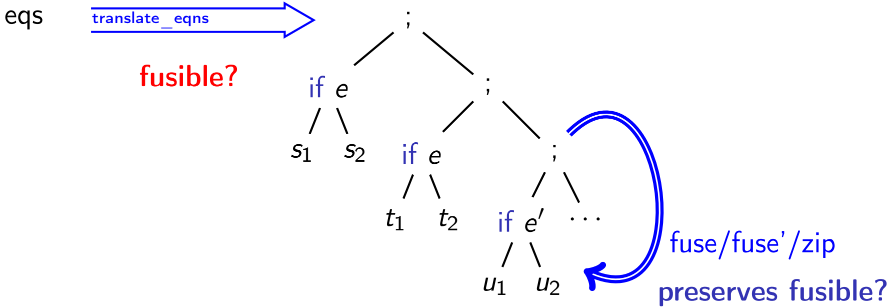


x = (merge b e1 e2)$^{\text{base on ck}}$

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

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

Fusion of control structures: correctness

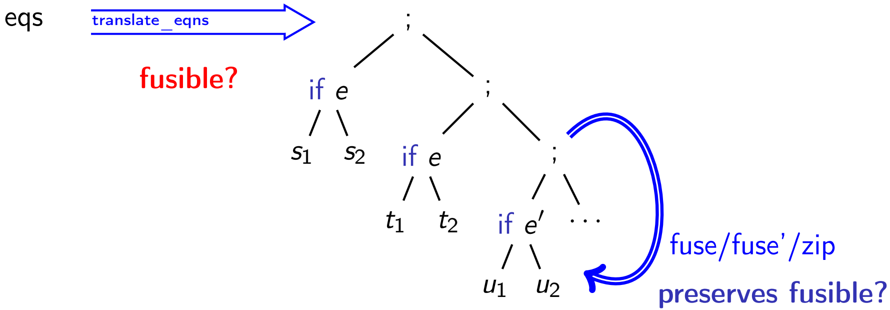


$x = (0 \text{ fby } (x + 1))^{\text{base on ck}}$

```
if ck {
  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



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

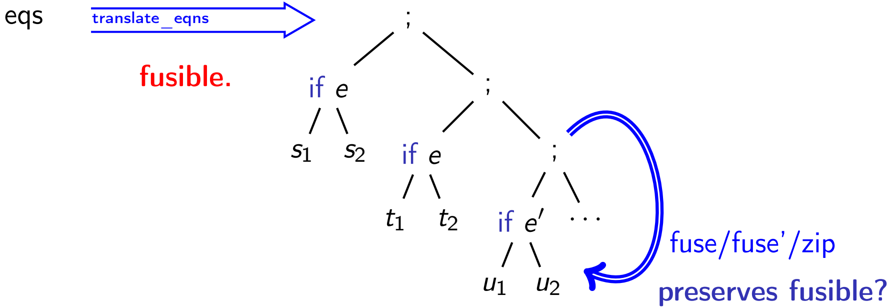
```
if mem(x) {
  y := true
}
if mem(x) {
  mem(x) := y
}
```

- 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

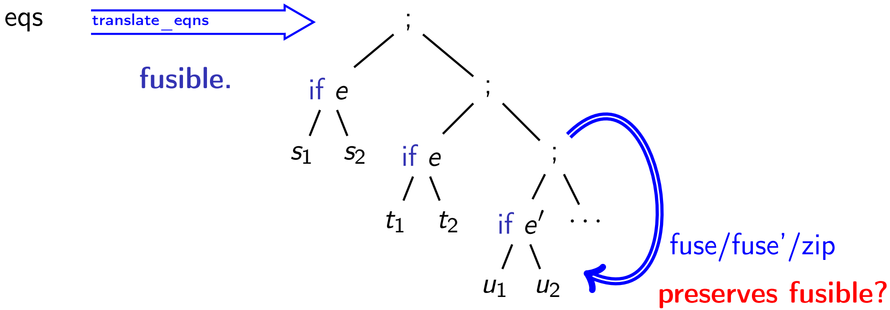


y = (true when x)$^{\text{base on x}}$
x = (true fby y)$^{\text{base on x}}$

```
if mem(x) {
  y := true
}
if mem(x) {
  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:
  $C \not\vdash x :: \mathit{base}$ on $\cdots$ on x on $\cdots$
- Compiling x = (v0 fby le)$^{\text{ck}}$ also gives fusible imperative code.

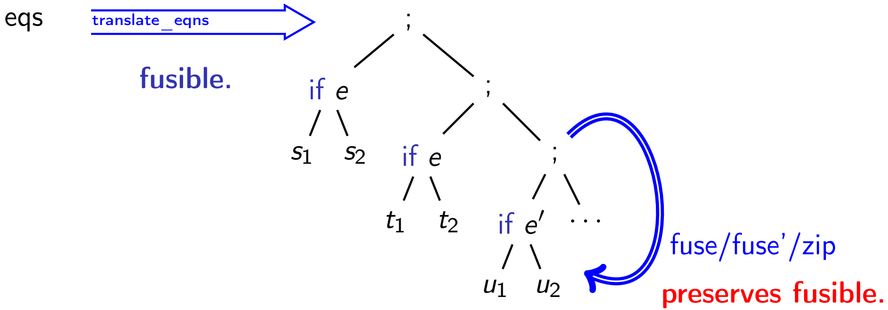
Fusion of control structures: correctness



- 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 ('Proper' instances) for ;/fuse/fuse'/zip.

- Rewrite until $$\frac{\text{fusible}(s)}{\text{fuse}\,(s) \approx_{eval} s}$$

Conclusion

Preliminary results

- Semantics based on (nat → value).
- Showed correctness of imperative code generation in Coq.
- Showed correctness of if/then/else fusion in Coq.

Ongoing work

- Well-typed, Well-clocked, Causal → sem_node G f xs ys.
- Connection to CompCert Clight.
- Working tool-chain:
  - Verified parser generator.
  - Incorporation of scheduling and normalization.

Longer term aims

- Treat more sophisticated language features.
- Verify synchronous models in Coq and generate correct code.

References I

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

Bedin França, R. et al. (2011). "Towards Formally Verified Optimizing Compilation in Flight Control Software". In: *Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*. Ed. by P. Lucas et al. Vol. 18. OpenAccess Series in Informatics (OASIcs). Grenoble, France: Schloss Daghstuhl, pp. 59–68.

Biernacki, D. et al. (2008). "Clock-directed modular code generation for synchronous data-flow languages". In: *Proc. 2008 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.

Boulmé, S. and G. Hamon (2001). *A clocked denotational semantics for Lucid-Synchrone in Coq*. Tech. rep. LIP6.

References II

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.

Colaço, J.-L., B. Pagano, and M. Pouzet (2005). "A Conservative Extension of Synchronous Data-flow with State Machines". In: *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*. Ed. by W. Wolf. Jersey City, USA: ACM Press, pp. 173–182. ISBN: 1-59593-091-4.

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

McCoy, F. (1885). *Natural history of Victoria: Prodromus of the Zoology of Victoria*. Frog images.

Paulin-Mohring, C. (2009). "A constructive denotational semantics for Kahn networks in Coq". In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*. Ed. by Y. Bertot et al. Cambridge University Press, pp. 383–413.