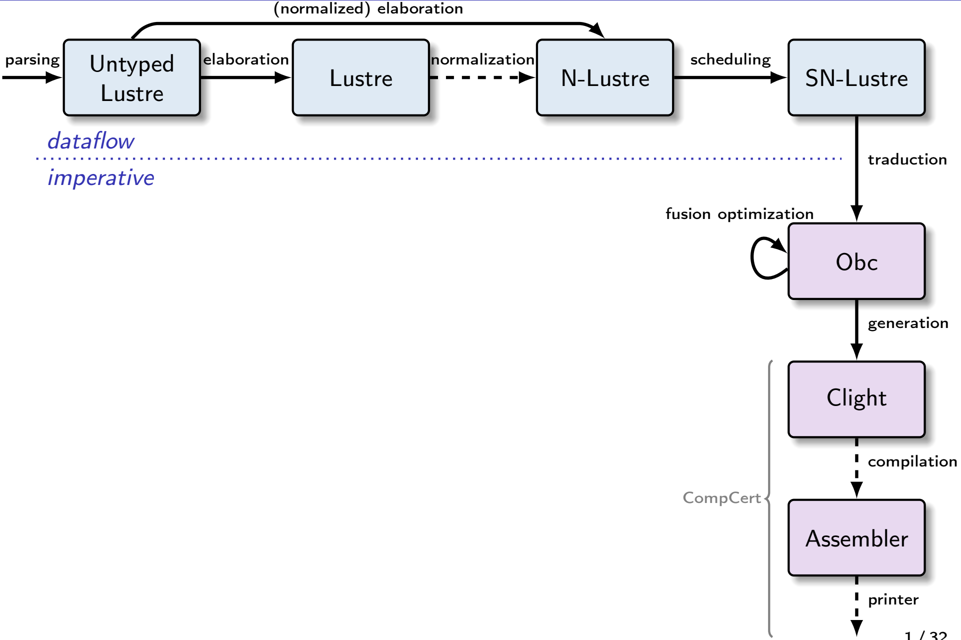
Vélus: A formally verified compiler for Lustre

Timothy Bourke

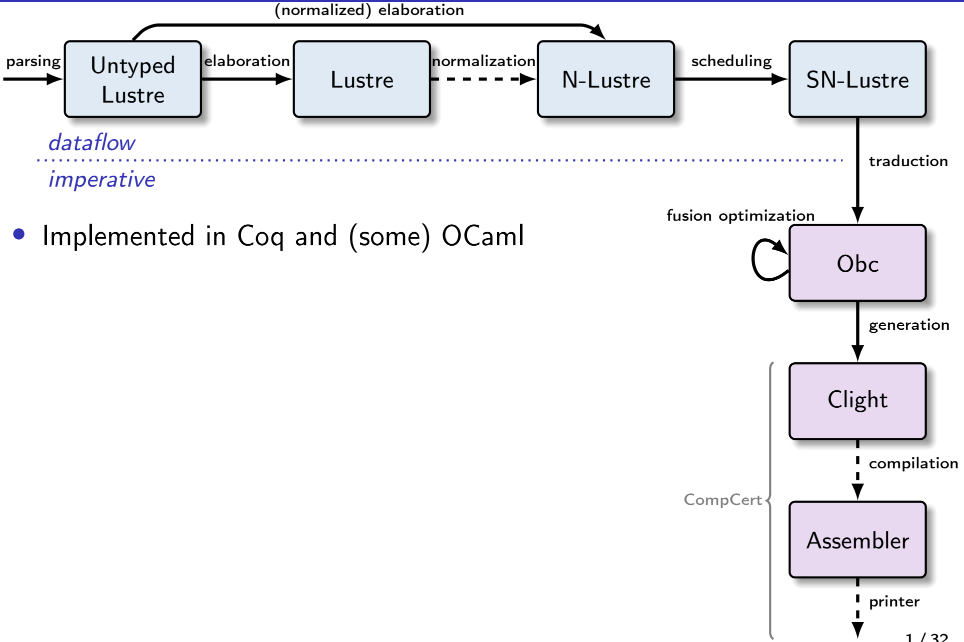
These slides show the results of a collaboration with Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg.

25 October 2018
NII Shonan Meeting Seminar 136
Functional Stream Libraries and Fusion

The Vélus Lustre Compiler

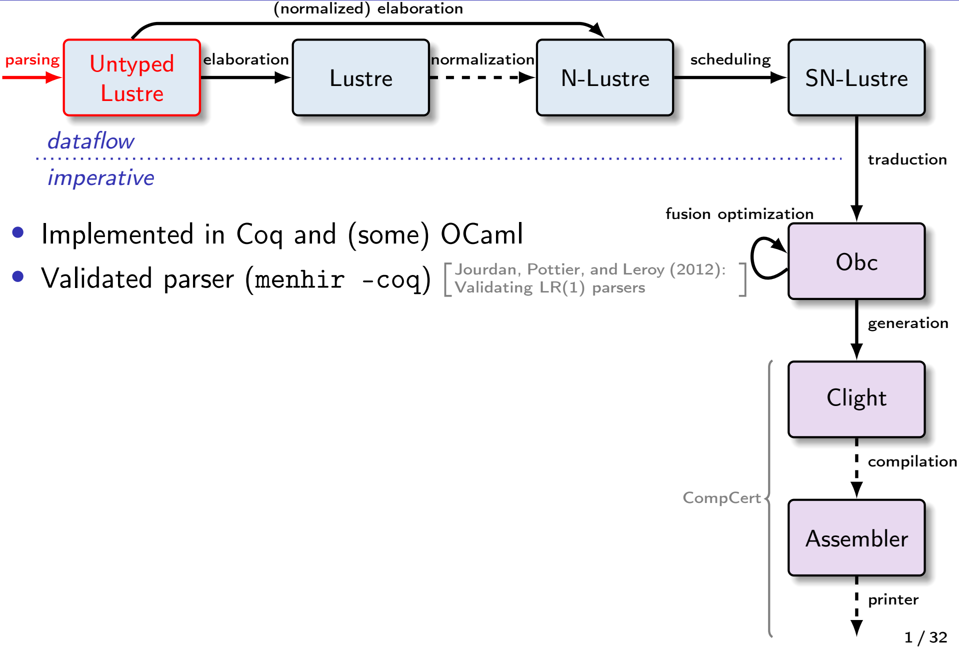


The Vélus Lustre Compiler



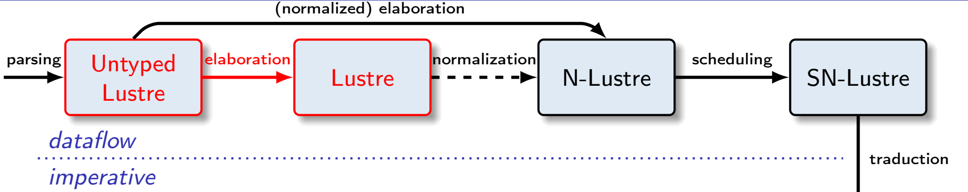
- Implemented in Coq and (some) OCaml

The Vélus Lustre Compiler

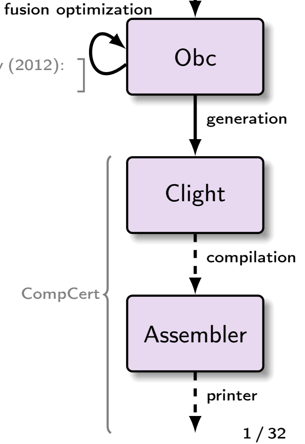


- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]

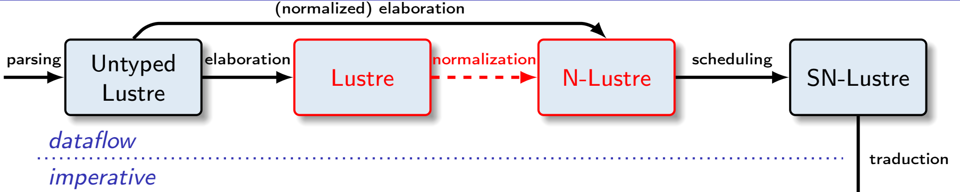
The Vélus Lustre Compiler



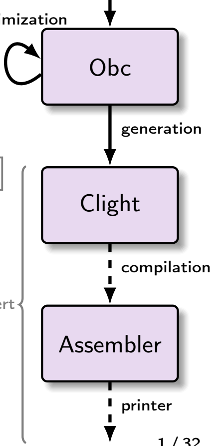
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]
- Check/Add type and clock annotations



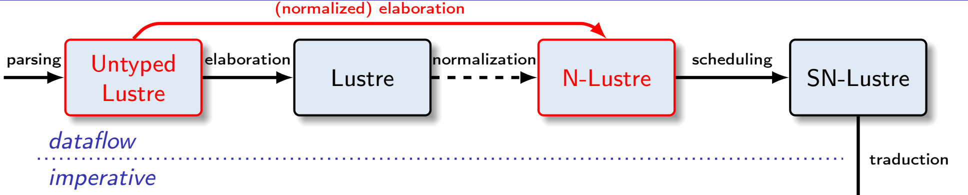
The Vélus Lustre Compiler



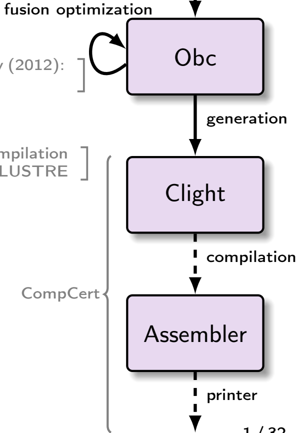
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]
- Check/Add type and clock annotations
- Not yet implemented: normalization [Auger (2013): Compilation certifiée de SCADE/LUSTRE]



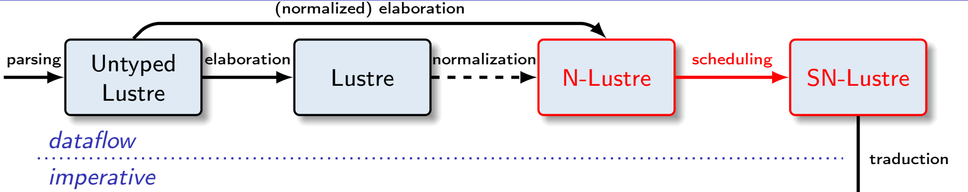
The Vélus Lustre Compiler



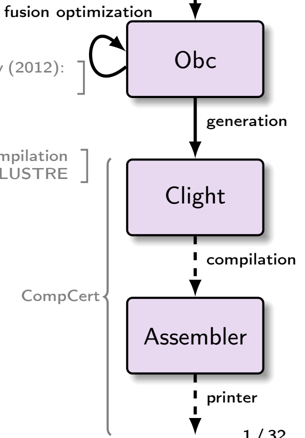
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]
- Check/Add type and clock annotations
- Not yet implemented: normalization [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
- Elaboration to Normalized Lustre.



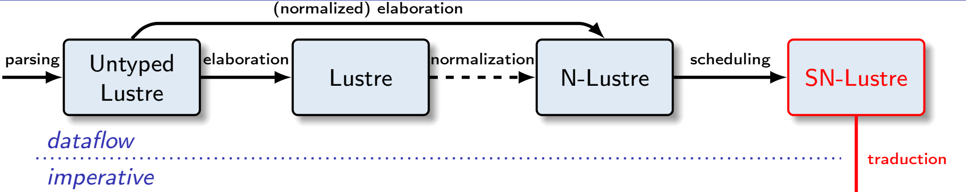
The Vélus Lustre Compiler



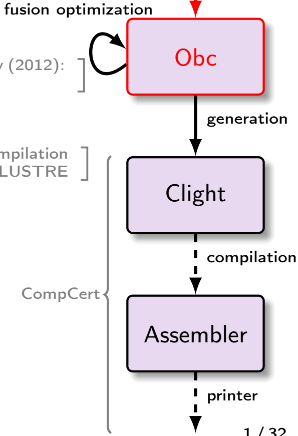
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]
- Check/Add type and clock annotations
- Not yet implemented: normalization [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.



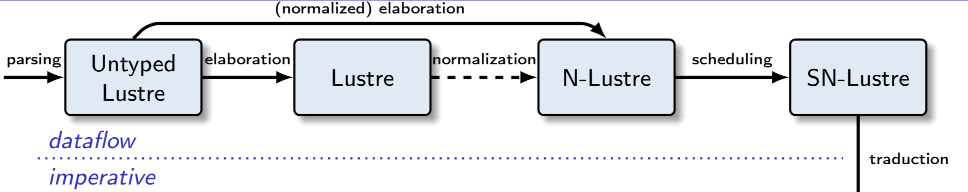
The Vélus Lustre Compiler



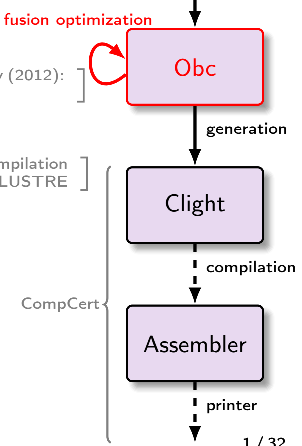
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]
- Check/Add type and clock annotations
- Not yet implemented: normalization [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
- Elaboration to Normalized Lustre.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



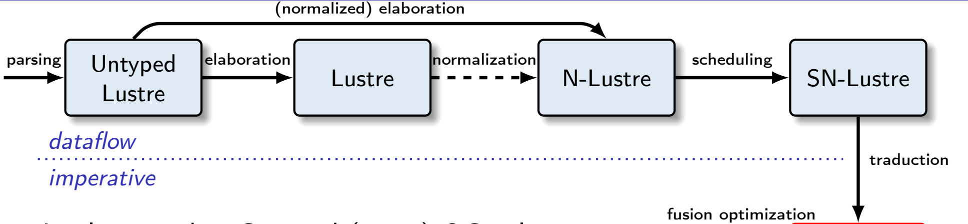
The Vélus Lustre Compiler



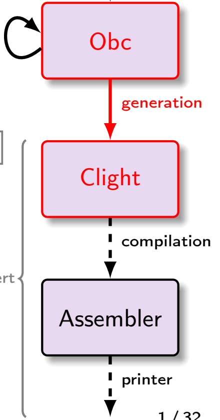
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]
- Check/Add type and clock annotations
- Not yet implemented: normalization [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
- Elaboration to Normalized Lustre.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



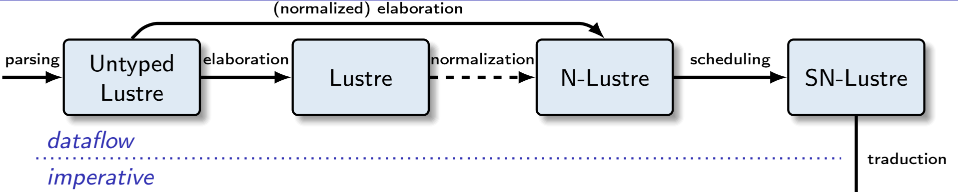
The Vélus Lustre Compiler



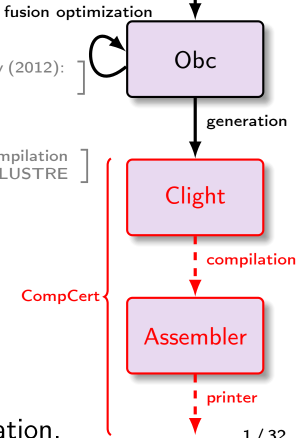
- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]
- Check/Add type and clock annotations
- Not yet implemented: normalization [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.
- Translation to intermediate Obc code.
- Optimization of intermediate Obc code.
- Generation of CompCert Clight code.

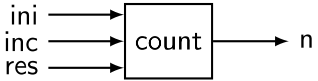


The Vélus Lustre Compiler

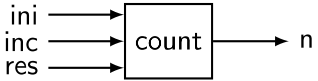


- Implemented in Coq and (some) OCaml
- Validated parser (`menhir -coq`) [Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]
- Check/Add type and clock annotations
- Not yet implemented: normalization [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
- Elaboration to Normalized Lustre.
- Scheduling of dataflow equations.
- Translation to intermediate Obc code.
- Optimization of intermediate Obc code.
- Generation of CompCert Clight code.
- CompCert: operator semantics and assembly generation.

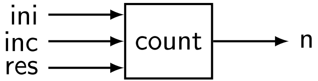



ini
inc
res
count
n

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



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



ini	0	0	0	0	0	0	0	⋯
inc	0	1	2	1	2	3	0	⋯
res	F	F	F	F	T	F	F	⋯
true fby false	T	F	F	F	F	F	F	⋯
0 fby n	0	0	1	3	4	0	3	⋯
n	0	1	3	4	0	3	3	⋯

- Node: set of causal equations (variables at left).
- Semantic model: synchronized streams of values.
- A node defines a function between input and output streams.

N-Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

N-Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯

N-Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

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	⋯

N-Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

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		···

N-Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

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		⋯

N-Lustre: instantiation and sampling

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

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) when not sec	0	0	0		4			3	⋯

N-Lustre: instantiation and sampling

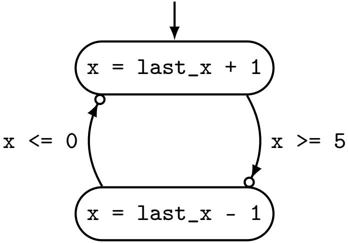
```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

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) when not sec	0	0	0		4			3	···
v	0	0	0	4	4	4	3	3	···

Sampling and merging: what for?

- Provides a means of conditional activation,
- and a target for sophisticated structures [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines].

```
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?

- Provides a means of conditional activation,
- and a target for sophisticated structures [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines].

```
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 = ...
```

N-Lustre: dataflow language

Expressions

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

Equations

$$
\begin{array}{lcl}
eq ::= & & x = (ce)^{ck} \\
 & \mid & x = (k_0 \text{ fby } e)^{ck} \\
 & \mid & x = (f(e, \ldots, e))^{ck}
\end{array}
$$

(Scheduled) Nodes

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

Clocks

ck ::= base | ck on $(x = k)$

Static clocks

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r: int; t: int :: base on r;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

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) when not sec	base on (sec = F)	0	0	0		4		···
v	base	0	0	0	4	4	4	···

Static clocks

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r: int; t: int :: base on r;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

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) when not sec	base on (sec = F)	0	0	0		4		···
v	base	0	0	0	4	4	4	···

- Static checking of ‘clocking’ using a dedicated type system.
- Synchronous Kahn networks that execute in bounded memory.
- “Clocks in the source language are transformed into control structures in the target language.” [Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

Static clocks

```
node avgvelocity(delta: int; sec: bool) returns (v: int)
  var r: int; t: int :: base on r;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

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) when not sec	base on (sec = F)	0	0	0		4		···
v	base	0	0	0	4	4	4	···

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

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

Lustre: syntax and semantics

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

ini	0	0	0	0	0	0	0	⋯
inc	0	1	2	1	2	3	0	⋯
res	F	F	F	F	T	F	F	⋯
true fby false	T	F	F	F	F	F	F	⋯
0 fby n	0	0	1	3	4	0	3	⋯
n	0	1	3	4	0	3	3	⋯

Lustre: syntax and semantics

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

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

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

Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

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

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;

  n_defd  : Permutation (vars_defined n_eqs)
                        (map fst (n_vars ++ n_out));
  n_nodup : NoDupMembers (n_in ++ n_vars ++ n_out);
  ...  }.
```

Lustre: syntax and semantics

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

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

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

Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

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

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;

  n_defd  : Permutation (vars_defined n_eqs)
                        (map fst (n_vars ++ n_out));
  n_nodup : NoDupMembers (n_in ++ n_vars ++ n_out);
  ...  }.
```

```
Inductive sem_node (G: global) :
  ident → stream (list value) → stream (list value) → Prop :=
| SNode:
    find_node f G = Some n →
    clock_of xss bk →
    sem_vars bk H (map fst n.(n_in)) xss →
    sem_vars bk H (map fst n.(n_out)) yss →
    sem_clocked_vars bk H (idck n.(n_in)) →
    Forall (sem_equation G bk H) n.(n_eqs) →
    sem_node G f xss yss.
```

Lustre: syntax and semantics

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

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

Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

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

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;

  n_defd  : Permutation (vars_defined n_eqs)
                        (map fst (n_vars ++ n_out));
  n_nodup : NoDupMembers (n_in ++ n_vars ++ n_out);
  ...  }.
```

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

```
Inductive sem_node (G: global) :
  ident → stream (list value) → stream (list value) → Prop :=
| SNode:
    find_node f G = Some n →
    clock_of xss bk →
    sem_vars bk H (map fst n.(n_in)) xss →
    sem_vars bk H (map fst n.(n_out)) yss →
    sem_clocked_vars bk H (idck n.(n_in)) →
    Forall (sem_equation G bk H) n.(n_eqs) →
    sem_node G f xss yss.
```

sem_node G f xss yss

$f : \text{stream}(T^+) \to \text{stream}(T^+)$

Dataflow semantic model

sec	F	F	F	T	F	T	T	⋯	base
r	0	1	3	4	6	9	9	⋯	base
r when sec				4		9	9	⋯	base on (sec = T)
t				1		2	3	⋯	base on (sec = T)
(0 fby v) when not sec	0	0	0		4			⋯	base on (sec = F)

- History environment maps identifiers to streams.

```
Definition history := PM.t (stream value)
```

- Model absence

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

- 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).
```

Lustre Compilation: normalization and scheduling

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

Lustre Compilation: normalization and scheduling

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

normalization ⟹

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

Normalization

- Rewrite to put each fby in its own equation.
- Introduce fresh variables using the substitution principle.

Lustre Compilation: normalization and scheduling

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

normalization →

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

↓ scheduling

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

Scheduling

- The semantics is independent of equation ordering; but not the correctness of imperative code translation.
- Reorder so that
  - 'Normals' variables are written before being read, . . . and
  - 'fby' variables are read before being written.

Simple Imperative Language: Obc

c	$::=$	x^{ty}	variable
	$\mid$	$\mathsf{state}(x)^{ty}$	memory
	$\mid$	k	constant
	$\mid$	$\diamond^{ty}\ e$	unary operator
	$\mid$	$e \oplus^{ty} e$	binary operator

s	$::=$	$x := c$	variable assignment
	$\mid$	$\mathsf{state}(x) := c$	memory assignment
	$\mid$	$\mathsf{if}\ c\ \{s\}\ \mathsf{else}\ \{s\}$	conditional branching
	$\mid$	$s;\ s$	sequential composition
	$\mid$	$x, \ldots, x := cl.\, m\ o\ (e, \ldots, e)$	class method call
	$\mid$	skip	nop

Big-step semantics: $me, ve \vdash_{prog} s \Downarrow me', ve'$

$me : \cdots$ $\qquad$ $ve : ident \to option\ val$

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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

```
class count {
 memory f : bool;
 memory c : int;

 reset() {
  state(f) := true;
  state(c) := 0
 }

 step(ini: int, inc: int, res: bool)
 returns (n: int) {
  if (state(f) | restart)
   then n := ini
   else  n := state(c) + inc;
  state(f) := false;
  state(c) := n
 }
}
```

Lustre compilation: translation to imperative code

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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

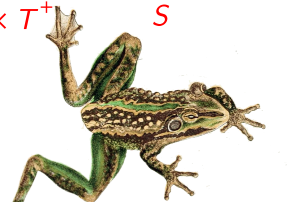
```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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

(f_t, s_0)

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



```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

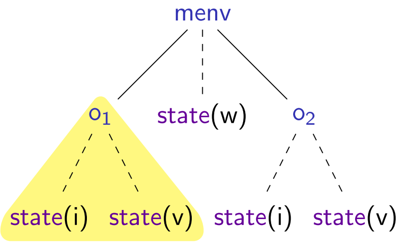
Lustre compilation: translation to clocked imperative code

```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```

```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

Lustre compilation: translation to clocked imperative code

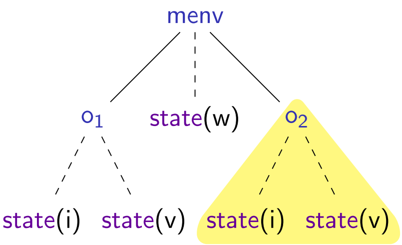
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

Lustre compilation: translation to clocked imperative code

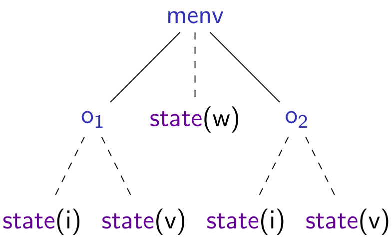
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

Lustre compilation: translation to clocked imperative code

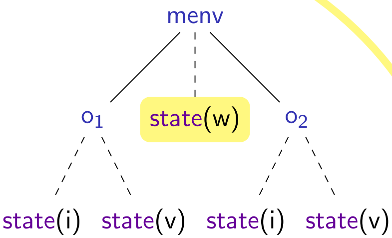
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

Lustre compilation: translation to clocked imperative code

```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
 memory w : int;
 class count o1, o2;

 reset() {
   count.reset o1;
   count.reset o2;
   state(w) := 0
 }

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

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

Implementation of translation

- Translation pass: small set of functions on abstract syntax.
- Challenge: going from one semantic model to another.

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

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

Fixpoint translate_lexp (e : lexp) : exp :=
  match e with
  | Econst c ⇒ Const c
  | Evar x ⇒ tovar x
  | Ewhen e c x ⇒ translate_lexp e
  | Eop op es ⇒ Op op (map translate_lexp es)
  end.

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

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

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

Definition translate_reset_eqn (s: stmt) (eqn: equation) : stmt :=
  match eqn with
  | EqDef _ _ _ ⇒ s
  | EqFby x _ v0 _ ⇒ Comp (AssignSt x (Const v0)) s
  | EqApp x _ f _ ⇒ Comp (Reset_ap f x) s
  end.

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

Definition ps_from_list (l: list ident) : PS.t :=
  fold_left (fun s i⇒PS.add i s) l PS.empty.

Definition translate_node (n: node): class :=
  let names := gather_eqs n.(n_eqs) in
  let mems := ps_from_list (fst names) in
  mk_class n.(n_name) n.(n_input) n.(n_output)
           (fst names) (snd names)
           (translate_eqns mems n.(n_eqs))
           (translate_reset_eqns n.(n_eqs)).

Definition translate (G: global) : program := map translate_node G.
```

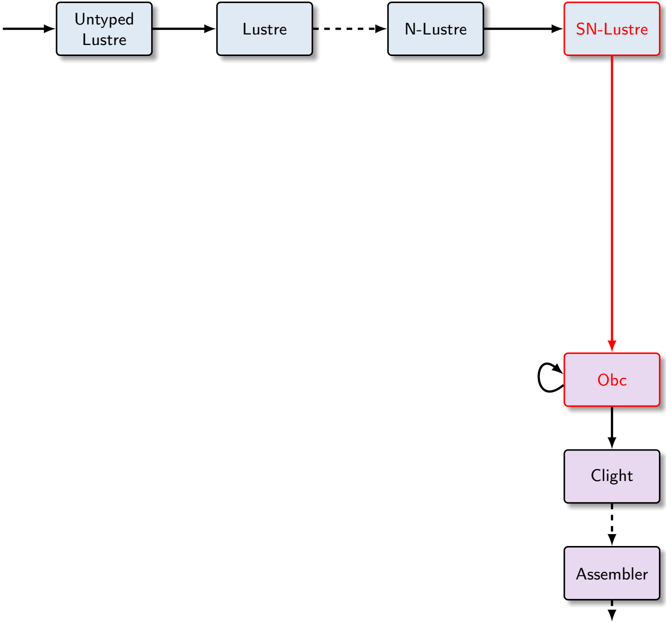
Translation: definition

```
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.
```

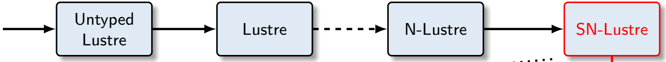
Translation: definition

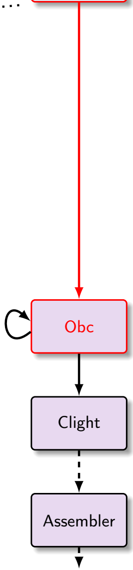
```
Variable mems : PS.t.
Definition tovar (x: ident) : exp := if PS.mem x mems then State x else Var x.

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 of translation to Obc



Correctness of translation to Obc



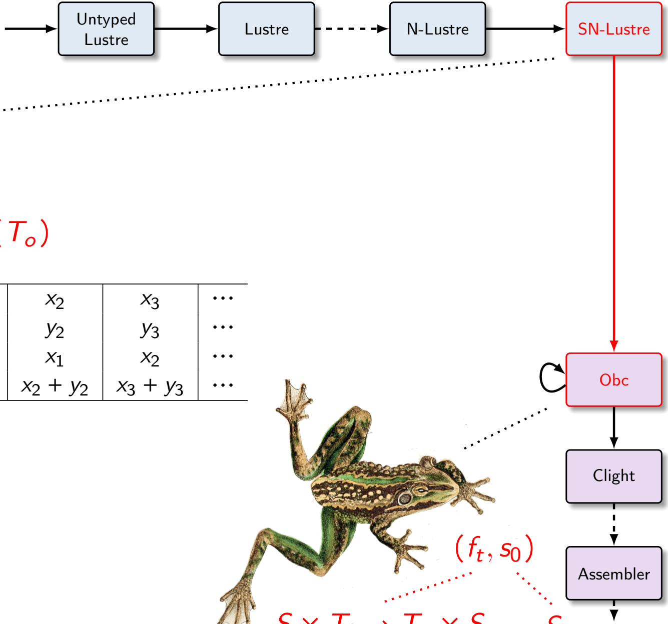


`sem_node G f xss yss`

$\mathrm{stream}(T_i) \rightarrow \mathrm{stream}(T_o)$

x	x_0	x_1	x_2	x_3	⋯
y	y_0	y_1	y_2	y_3	⋯
pre x	nil	x_0	x_1	x_2	⋯
x + y	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	⋯

Correctness of translation to Obc



sem_node G f xss yss

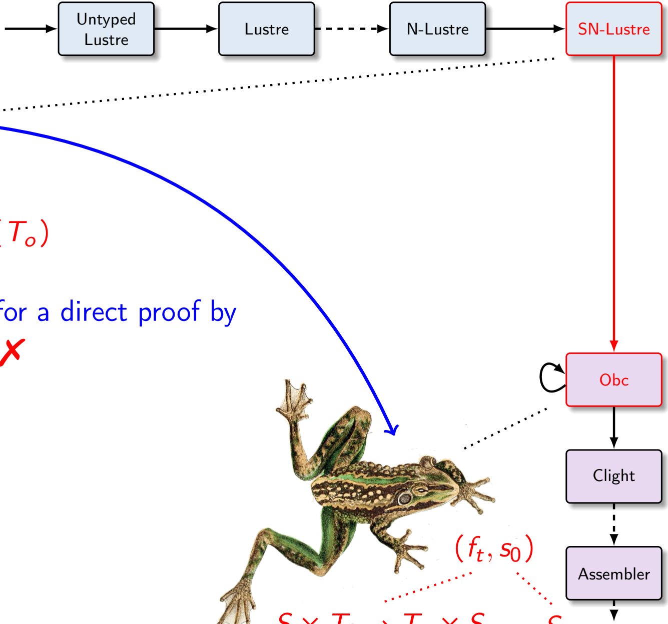
stream(T_i) → stream(T_o)

x	x_0	x_1	x_2	x_3	⋯
y	y_0	y_1	y_2	y_3	⋯
pre x	nil	x_0	x_1	x_2	⋯
x + y	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	⋯

(f_t, s_0)

$S \times T_i \rightarrow T_o \times S$ $\quad S$

Correctness of translation to Obc



`sem_node G f xss yss`

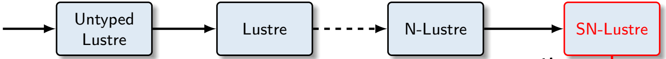
stream(T_i) → stream(T_o)

too weak for a direct proof by induction ✗

(f_t, s_0)

$S \times T_i \rightarrow T_o \times S$ $\quad S$

Correctness of translation to Obc

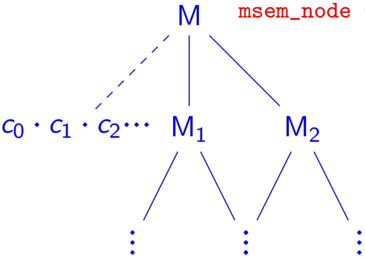


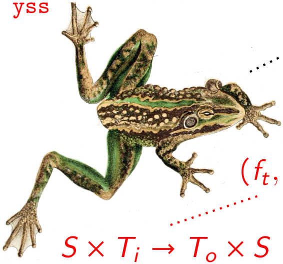
`sem_node G f xss yss`

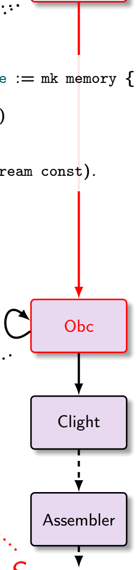
$\text{stream}(T_i) \rightarrow \text{stream}(T_o)$

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

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

`msem_node G f xss M yss`



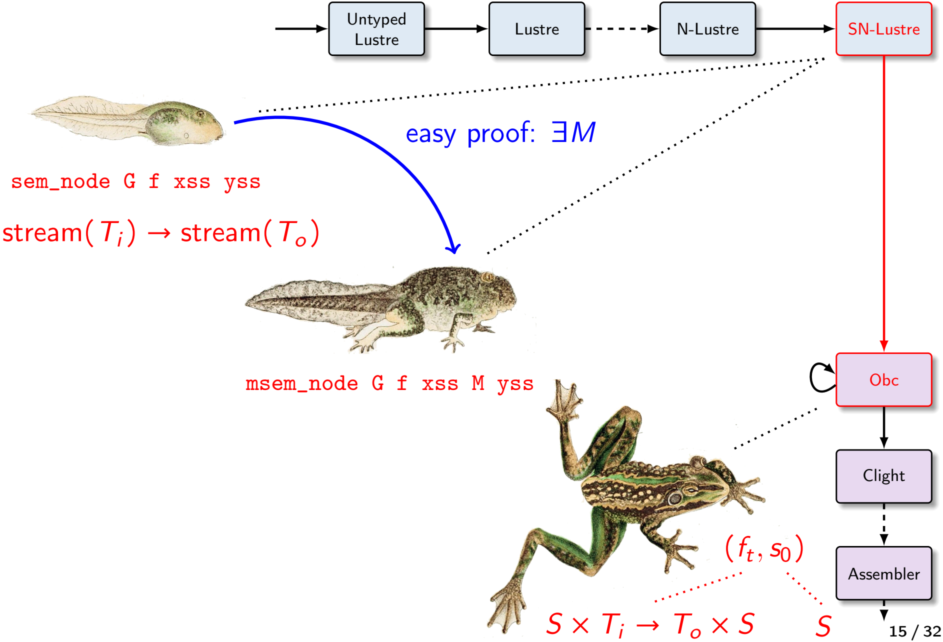




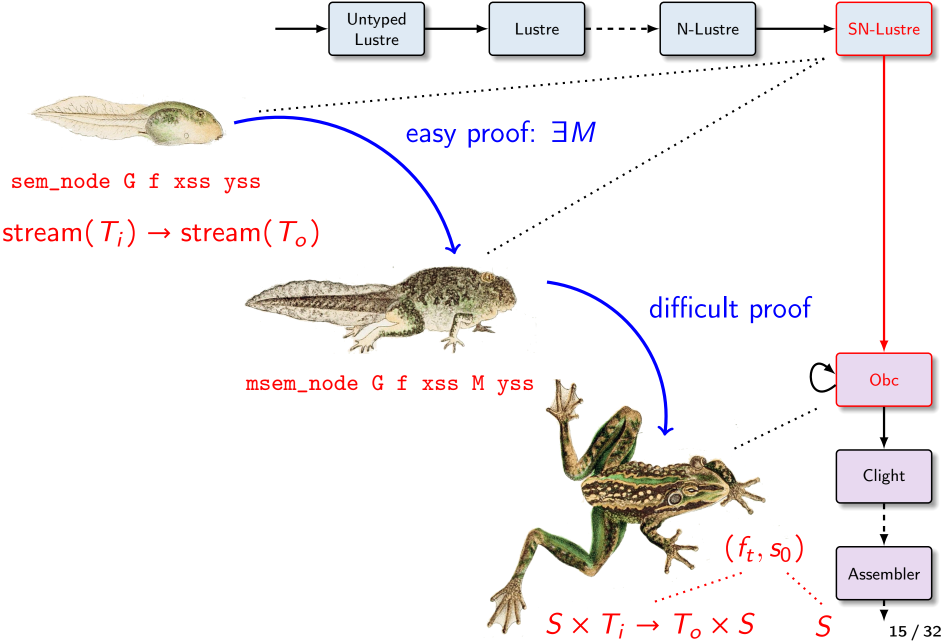
(f_t, s_0)

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

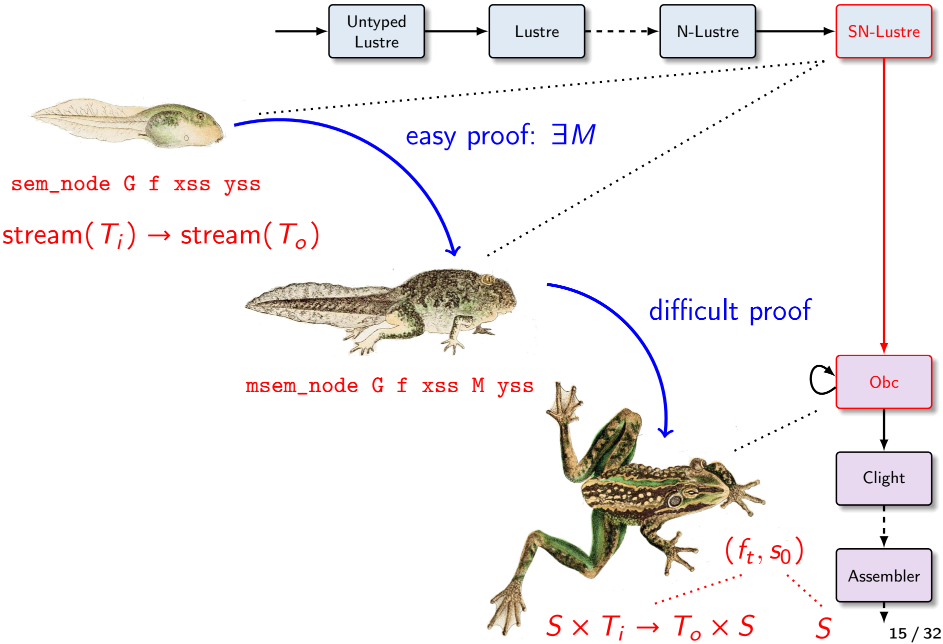
Correctness of translation to Obc



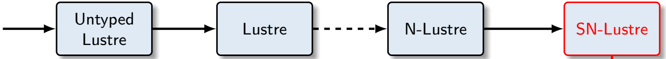
Correctness of translation to Obc



Correctness of translation to Obc

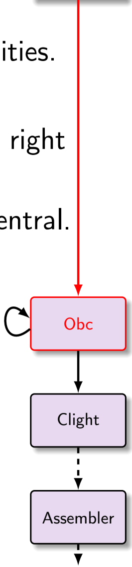


Correctness of translation to Obc

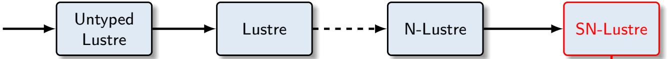


- induction n
  - induction G
    - induction eqs
      - case: $x = (ce)^{ck}$
        - case: present
        - case: absent
      - case: $x = (f\ e)^{ck}$
        - case: present
        - case: absent
      - case: $x = (k\ \texttt{fby}\ e)^{ck}$
        - case: present
        - case: absent

- Tricky proof, many technicalities.
- ≈100 lemmas
- Several iterations to find the right definitions.
- The intermediate model is central.



Correctness of translation to Obc

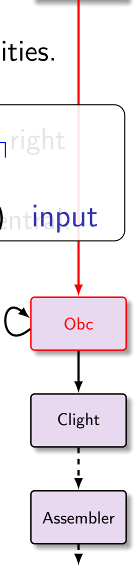


induction n
- induction G
  - induction eqs
    - case: $x = (ce)^{ck}$
      - case: present
      - case: absent
    - case: $x = (f\ e)^{ck}$
      - case: present
      - case: absent
    - case: $x = (k \texttt{ fby } e)^{ck}$
      - case: present
      - case: absent

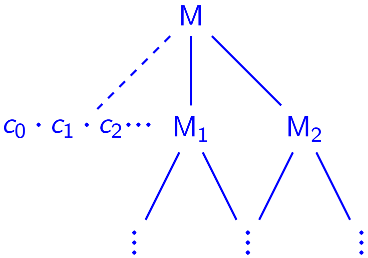
- Tricky proof, many technicalities.
- ≈100 lemmas
- Several iterations to find the right definitions.
- The intermediate model is central.

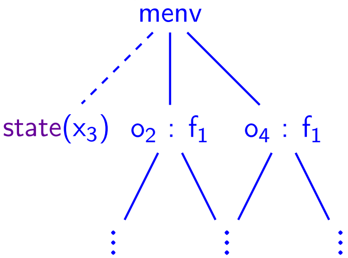
alleqs: oeqs ++ (eq :: eqs) input

$$\underbrace{\underbrace{[\cdots;\ w = v_0 \texttt{ fby } e\ ;\cdots]}_{\text{oeqs}} \ +\!\!+ \ (\underbrace{x = e}_{\text{eq}} :: \underbrace{[\cdots;\ y = e;\ \cdots]}_{\text{eqs}})}_{\text{alleqs}} \quad \text{input}$$



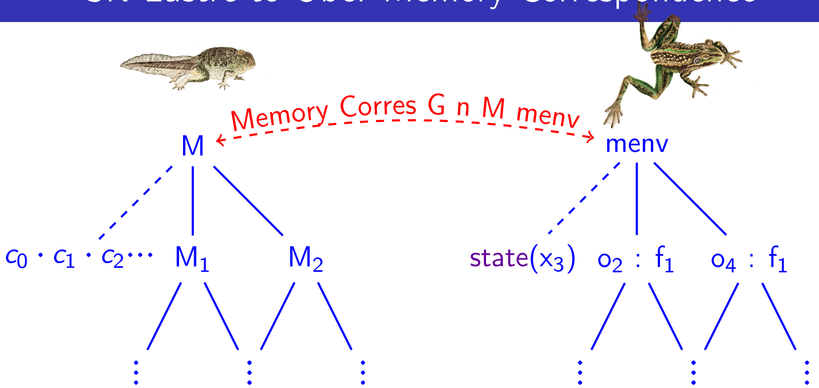
SN-Lustre to Obc: Memory Correspondence



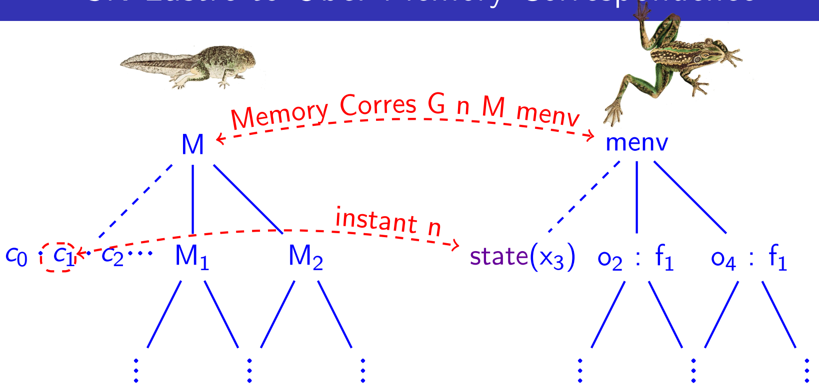



```
Inductive Memory_Corres (G: global) (n: nat) :
      ident → memory → heap → Prop :=
| MemC:
    find_node f G = Some(mk_node f i o eqs) →
    Forall (Memory_Corres_eq G n M menv) eqs →
    Memory_Corres G n f M menv
```

SN-Lustre to Obc: Memory Correspondence

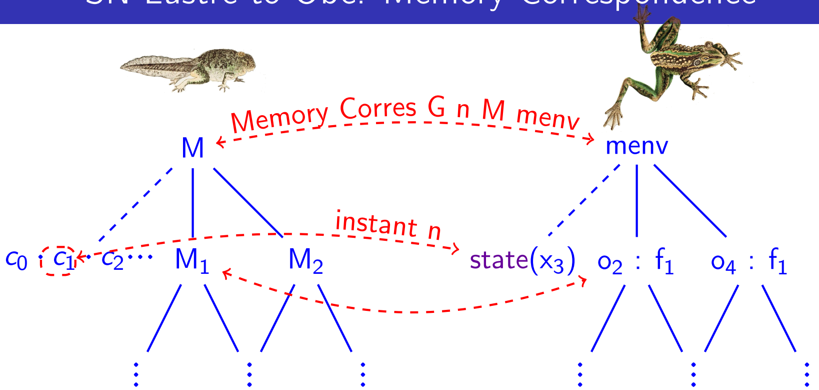



```
Inductive Memory_Corres (G: global) (n: nat) :
      ident → memory → heap → Prop :=
| MemC:
    find_node f G = Some(mk_node f i o eqs) →
    Forall (Memory_Corres_eq G n M menv) eqs →
    Memory_Corres G n f M menv
```



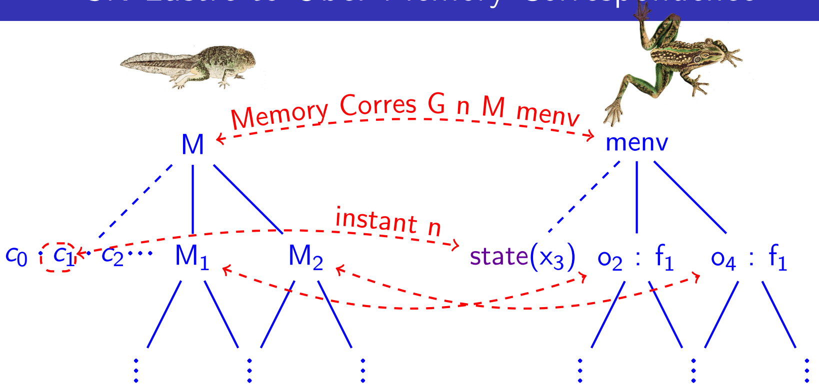

```
Inductive Memory_Corres_eq (G: global) (n: nat) :
      memory → heap → equation → Prop :=
...
| MemC_EqFby:
    (∀ ms, mfind_mem x M = Some ms
                 → mfind_mem x menv = Some (ms n))
    → Memory_Corres_eq G n M menv (EqFby x v0 lae).
```

SN-Lustre to Obc: Memory Correspondence



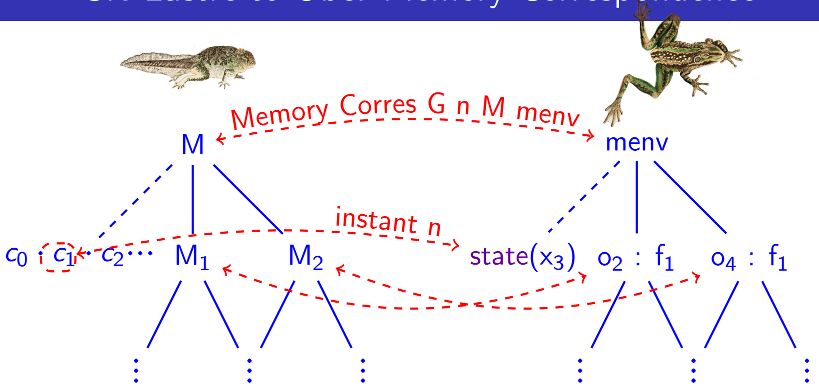

```
Inductive Memory_Corres_eq (G: global) (n: nat) :
      memory → heap → equation → Prop :=
...
| MemC_EqApp:
    (∀ Mo, mfind_inst x M = Some Mo →
          (∃ omenv, mfind_inst x menv = Some omenv
                   ∧ Memory_Corres G n f Mo omenv))
  → Memory_Corres_eq G n M menv (EqApp x f lae)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

SN-Lustre to Obc: Memory Correspondence



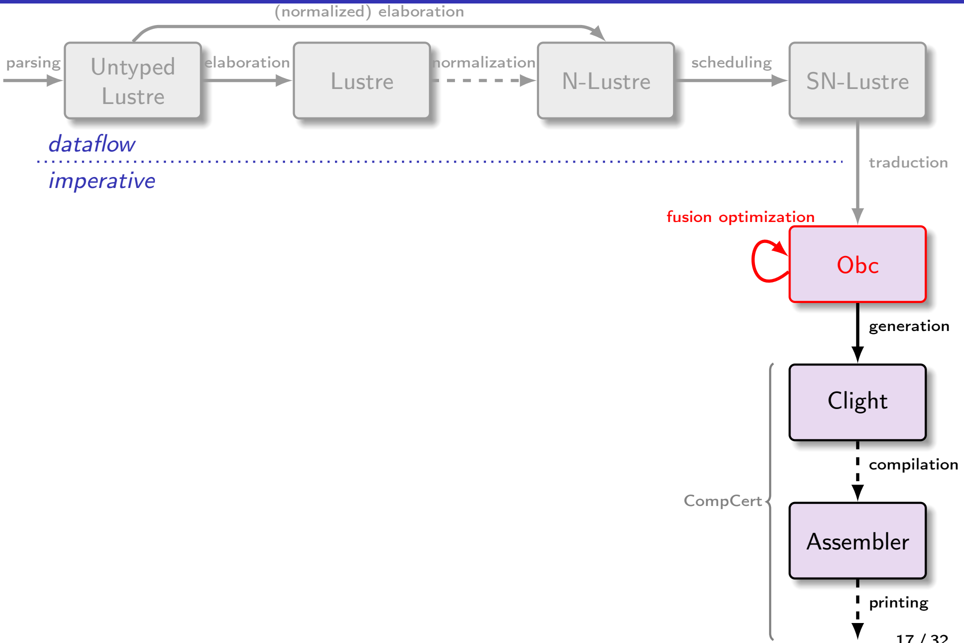

```
Inductive Memory_Corres_eq (G: global) (n: nat) :
      memory → heap → equation → Prop :=
...
| MemC_EqApp:
    (∀ Mo, mfind_inst x M = Some Mo →
          (∃ omenv, mfind_inst x menv = Some omenv
                    ∧ Memory_Corres G n f Mo omenv))
    → Memory_Corres_eq G n M menv (EqApp x f lae)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

SN-Lustre to Obc: Memory Correspondence



- Memory 'model' does not change between N-Lustre and Obc.
  - Corresponds at each 'snapshot'.
- The real challenge is in the change of semantic model: from dataflow streams to sequenced assignments

Fusion optimization: Obc to Obc



Control structure fusion

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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

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

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

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

  state(w) := v
}
```

- Generate control for each equation; splits proof obligation in two.
- Fuse afterward: scheduler places similarly clocked equations together.
- Use whole framework to justify required invariant.
- Easier to reason in intermediate language than in Clight.

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

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

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

[Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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

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

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

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

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

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

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

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

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

Fusion of control structures: requires invariant

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

➠

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

Fusion of control structures: requires invariant

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

➠ `if e then {s1; t1} else {s2; t2};`

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

✗

Fusion of control structures: requires invariant

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

⇒ `if e then {s1; t1} else {s2; t2};`

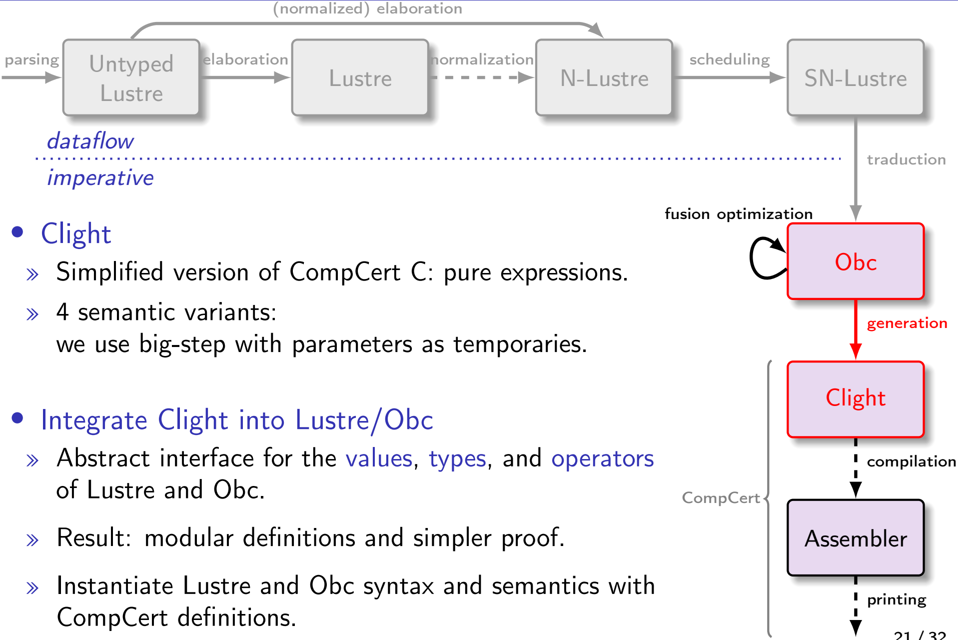
```
if x then {x := false} else {x := true};
if x then {t1} else {t2}
```

✗

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

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

Generation: Obc to Clight



- Clight
  - » Simplified version of CompCert C: pure expressions.
  - » 4 semantic variants: we use big-step with parameters as temporaries.
- Integrate Clight into Lustre/Obc
  - » Abstract interface for the values, types, and operators of Lustre and Obc.
  - » Result: modular definitions and simpler proof.
  - » Instantiate Lustre and Obc syntax and semantics with CompCert definitions.

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

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

```
Module Type OPERATORS.

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

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

```
Module Type OPERATORS.

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

 (* Boolean values *)
 Parameter bool_type : type.

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

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

```
Module Type OPERATORS.

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

 (* Boolean values *)
 Parameter bool_type : type.

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

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

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

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

```
Module Type OPERATORS.

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

 (* Boolean values *)
 Parameter bool_type : type.

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

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

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

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

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

 Parameter type_unop  :
   unop → type → option type.

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

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

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

```
Module Type OPERATORS.

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

 (* Boolean values *)
 Parameter bool_type : type.

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

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

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

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

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

 Parameter type_unop  :
   unop → type → option type.

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

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

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

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

```
Module Type OPERATORS.

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

 (* Boolean values *)
 Parameter bool_type : type.

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

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

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

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

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

 Parameter type_unop  :
   unop → type → option type.

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

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

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

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

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

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

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

```
Module Type OPERATORS.

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

 (* Boolean values *)
 Parameter bool_type : type.

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

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

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

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

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

 Parameter type_unop  :
   unop → type → option type.

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

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

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

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

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

```
Module Type OPERATORS.

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

 (* Boolean values *)
 Parameter bool_type : type.

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

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

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

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

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

 Parameter type_unop  :
   unop → type → option type.

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

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

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

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

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

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

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

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

```
Module Type OPERATORS.

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

 (* Boolean values *)
 Parameter bool_type : type.

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

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

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

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

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

 Parameter type_unop  :
   unop → type → option type.

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

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

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

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

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

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

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

 Definition bool_type : type := Tint IBool Signed.

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

 Definition binop := Cop.binary_operation.

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

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

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }
```

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }
```

```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

```
struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

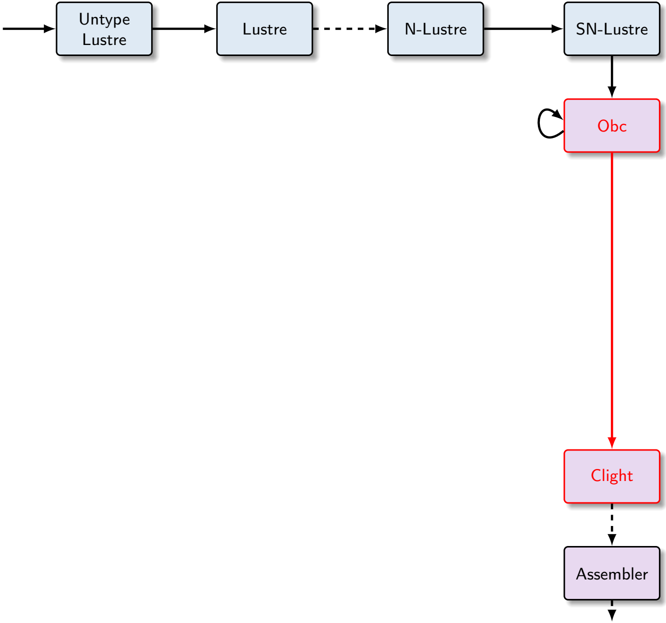
```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

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

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

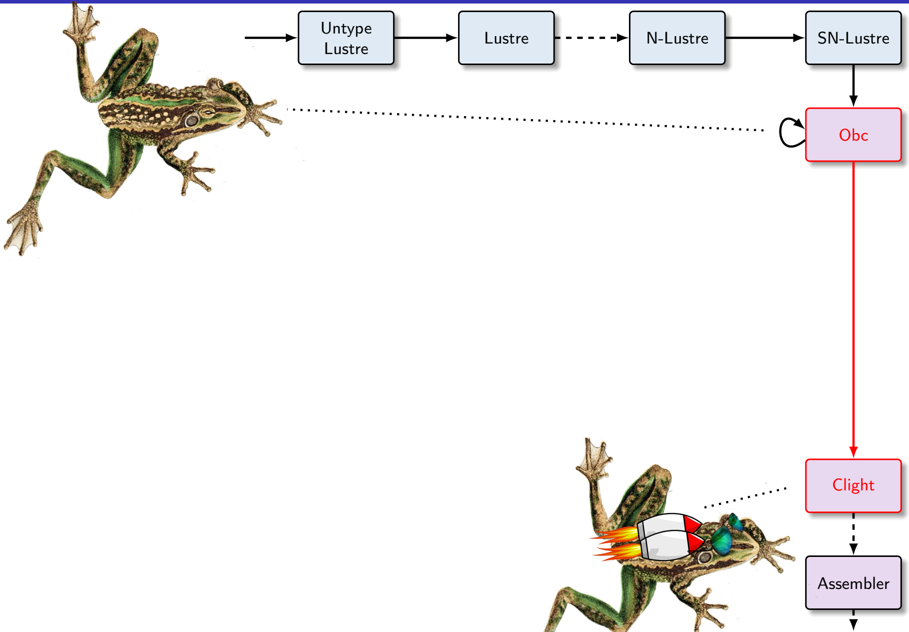
- Standard technique for encapsulating state.
- Each detail entails complications in the proof.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
   int w;
   struct count o1;
   struct count o2;
};

struct avgvelocity$step {
   int r;
   int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
   count$reset(&(self→o1));
   count$reset(&(self→o2));
   self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
   register int t, step$n;

   step$n = count$step(&(self→o1), 0, delta, 0);
   out→r = step$n;
   if (sec) {
      step$n = count$step(&(self→o2), 1, 1, 0);
      t = step$n;
      out→v = out→r / t;
   } else {
      out→v = self→w;
   }
   self→w = out→v;
}
```

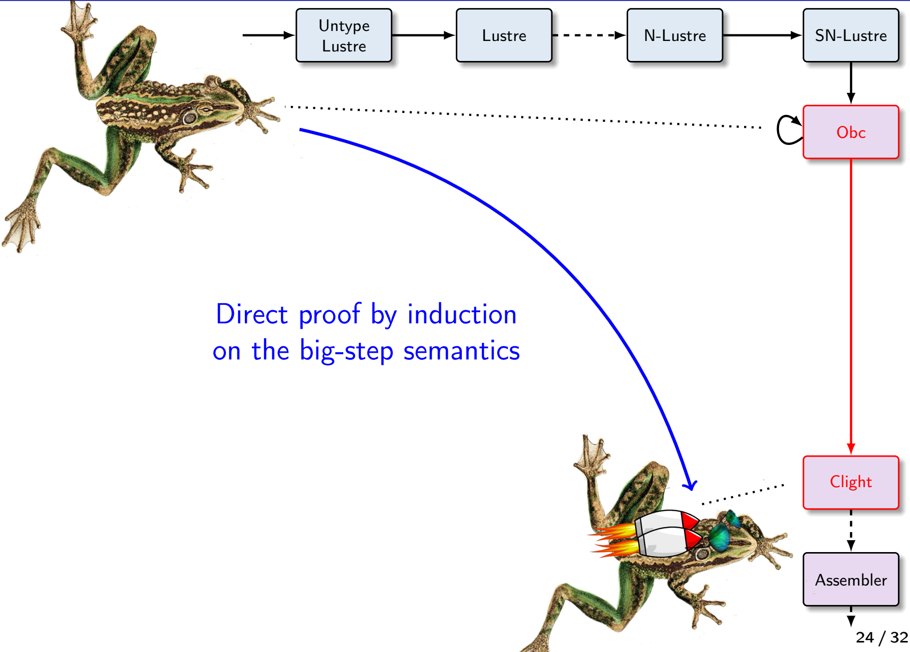
Correctness of Clight generation



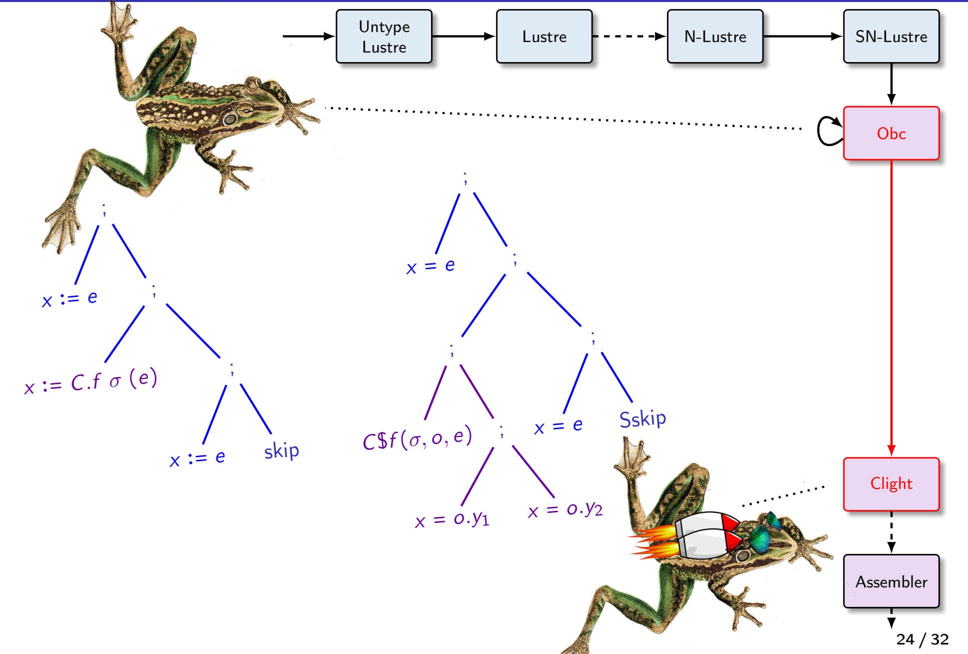
Correctness of Clight generation



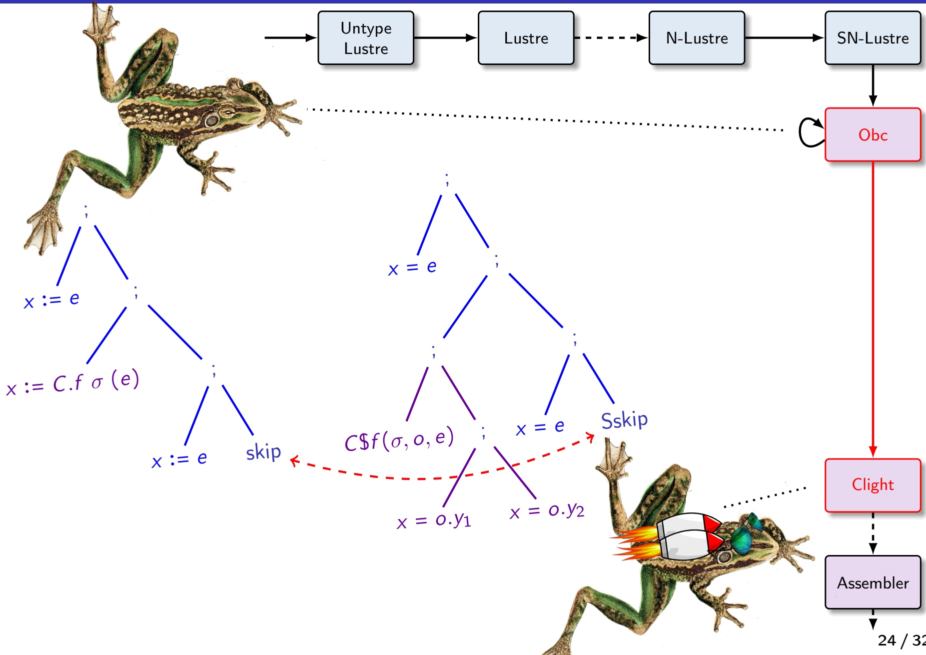
Correctness of Clight generation



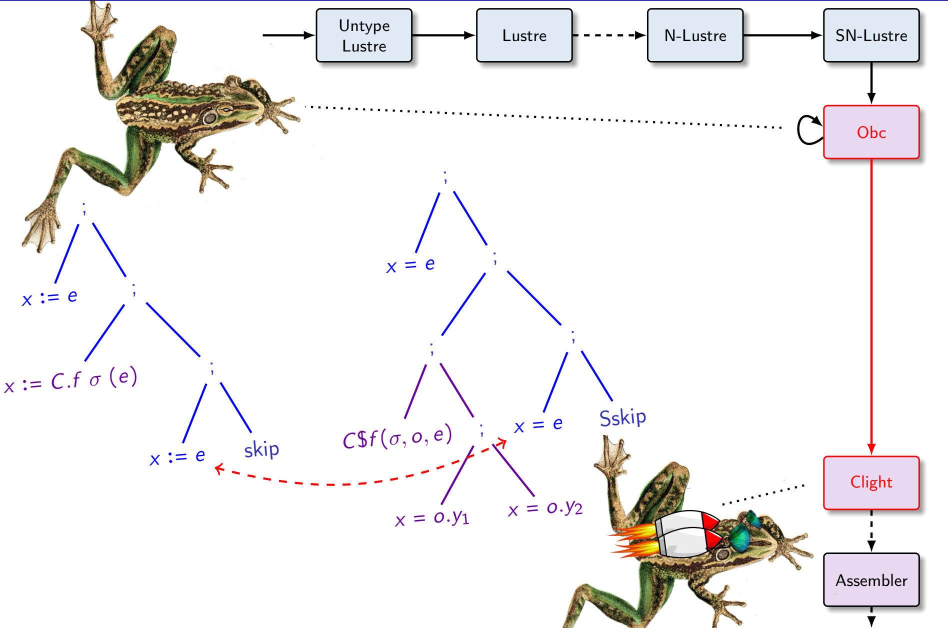
Correctness of Clight generation



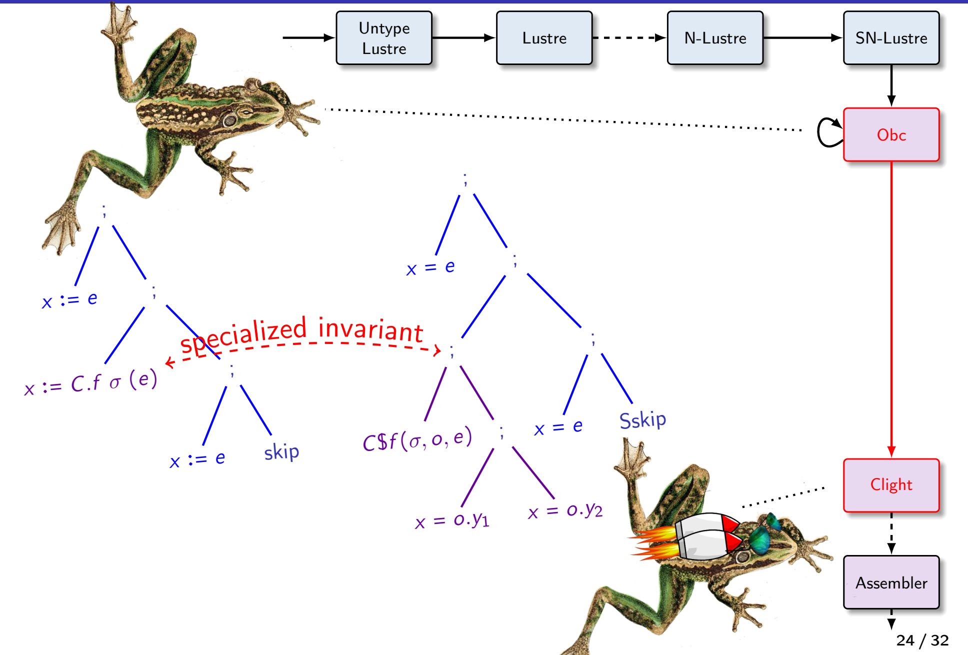
Correctness of Clight generation



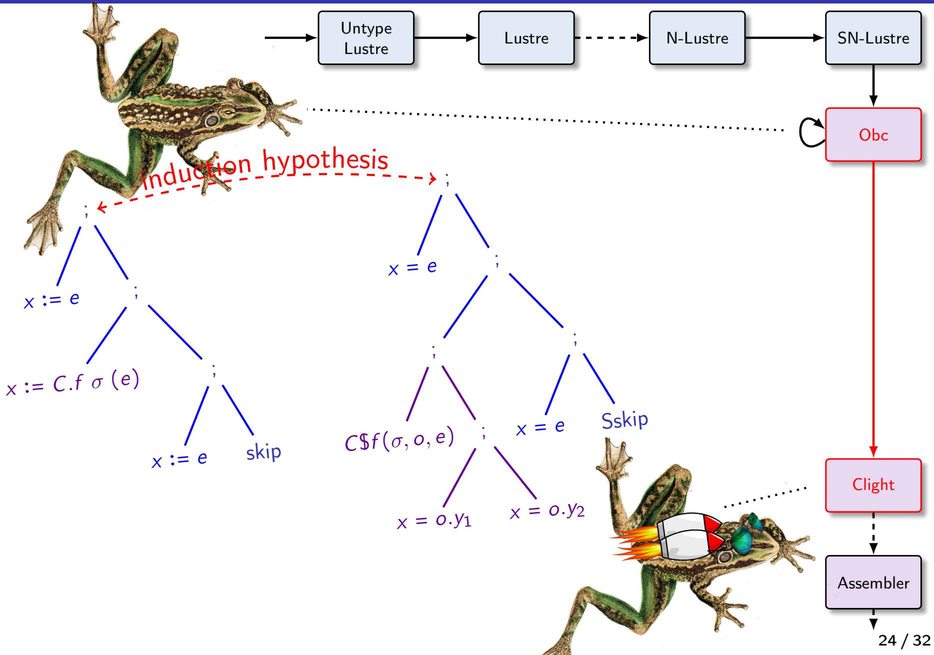
Correctness of Clight generation



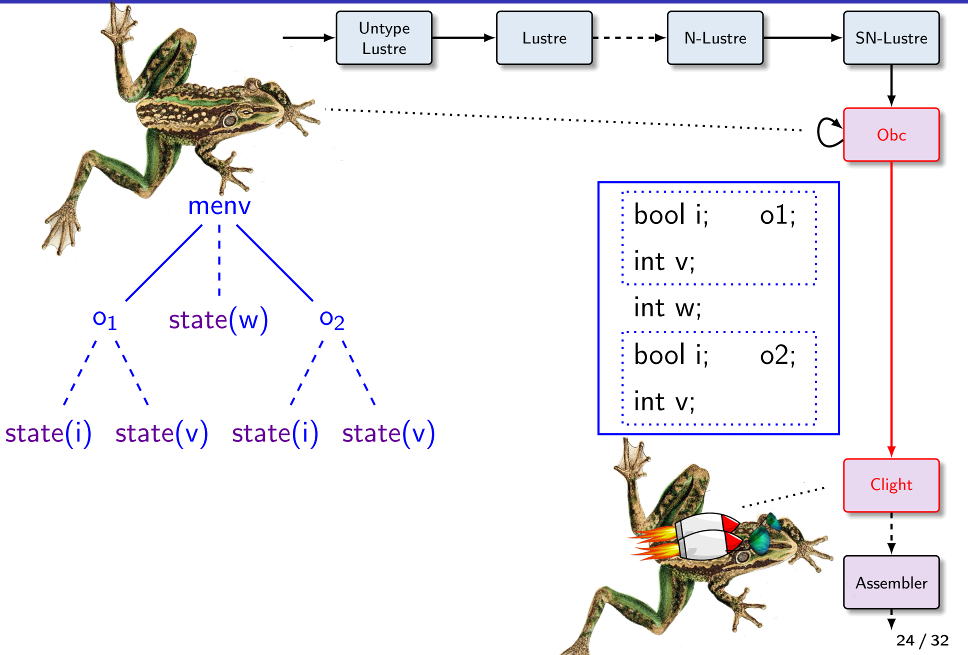
Correctness of Clight generation



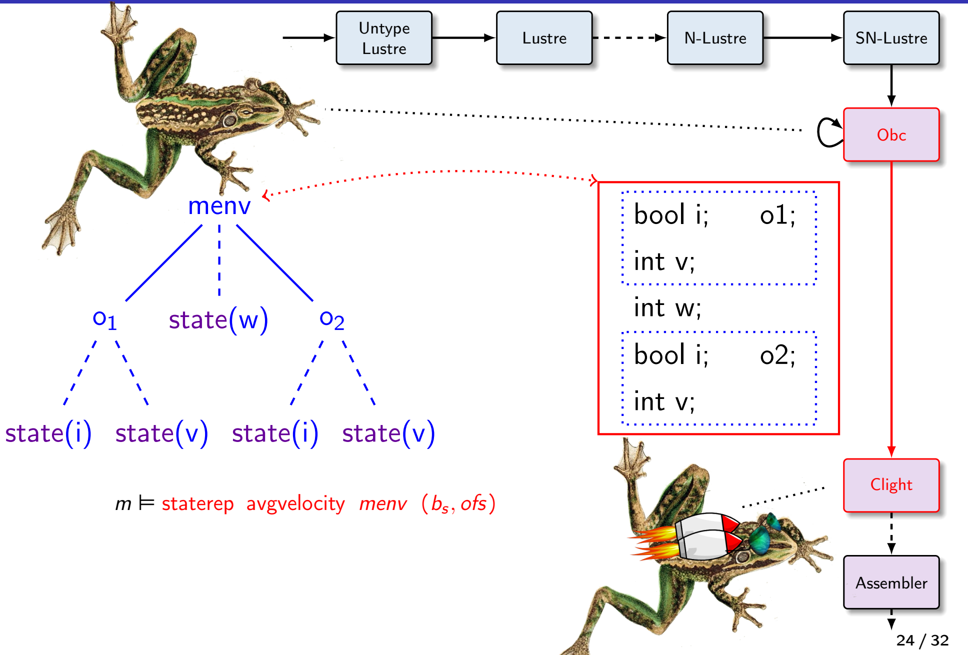
Correctness of Clight generation



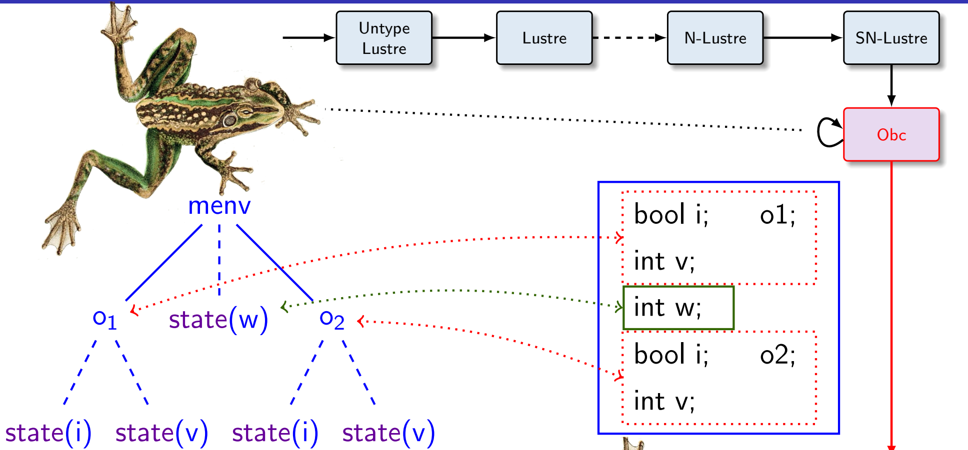
Correctness of Clight generation

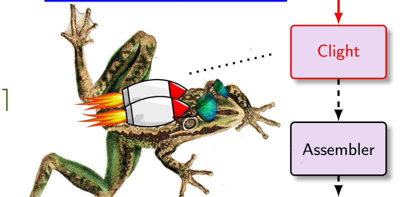


Correctness of Clight generation



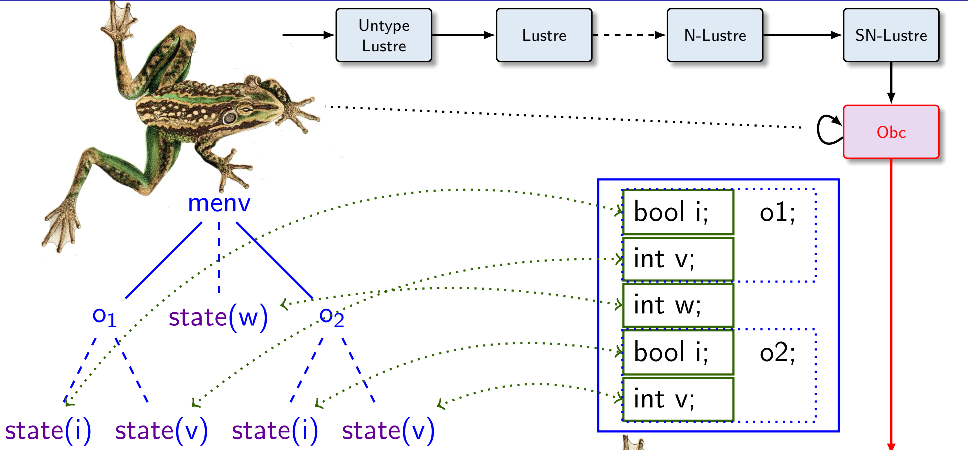
Correctness of Clight generation



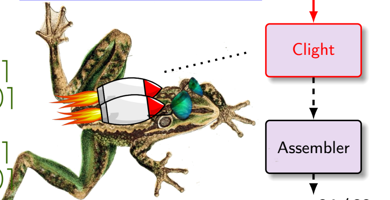


$$
\begin{aligned}
m \vDash\ & \text{staterep count } menv(o1)\ (b_s, ofs + \delta_{o1}) \\
* &\ \text{contains int32s } (b_s, ofs + \delta_w)\ \lceil menv.\text{state}(w) \rceil \\
* &\ \text{staterep count } menv(o2)\ (b_s, ofs + \delta_{o2})
\end{aligned}
$$

Correctness of Clight generation



$$
\begin{aligned}
m \vDash \quad & \text{contains bool} \quad (b_s, ofs + \delta_{o1} + \delta_i) \quad [menv.o_1.\text{state}(i)] \\
* \quad & \text{contains int32s} \quad (b_s, ofs + \delta_{o1} + \delta_v) \quad [menv.o_1.\text{state}(v)] \\
* \quad & \text{contains int32s} \quad (b_s, ofs + \delta_w) \quad [menv.\text{state}(w)] \\
* \quad & \text{contains bool} \quad (b_s, ofs + \delta_{o2} + \delta_i) \quad [menv.o_2.\text{state}(i)] \\
* \quad & \text{contains int32s} \quad (b_s, ofs + \delta_{o2} + \delta_v) \quad [menv.o_2.\text{state}(v)]
\end{aligned}
$$



Invariant: staterep

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


Definition staterep_mems (cls: class) (me: menv) (b: block) (ofs: Z) ((x, ty) : ident * typ) :=
  match field_offset ge x (make_members cls) with
  | OK d ⇒ contains (chunk_of_type ty) b (ofs + d) (match_value me.( mm_values) x)
  | Error _ ⇒ sepfalse
  end.

Fixpoint staterep (p: program) (clsnm: ident) (me: menv) (b: block) (ofs: Z): massert :=
  match p with
  | nil ⇒ sepfalse
  | cls :: p' ⇒ if ident_eqb clsnm cls.( c_name) then
      sepall (staterep_mems cls me b ofs) cls.( c_mems)
      ** sepall (fun ((i, c) : ident * ident) ⇒ match field_offset ge i (make_members cls) with
                                                  | OK d ⇒ staterep p' c (instance_match me i) b (ofs + d)
                                                  | Error _ ⇒ sepfalse
                                                  end) cls.( c_objs)
    else staterep p' clsnm me b ofs
  end.
```

Experimental results

Industrial application

- ≈6 000 nodes
- ≈162 000 equations
- ≈12 MB source file
  (minus comments)
- Modifications:
  - Remove constant lookup tables.
  - Replace calls to assembly code.
- Vélus compilation: ≈1 min 40 s

Experimental results

Industrial application

- ≈6 000 nodes
- ≈162 000 equations
- ≈12 MB source file (minus comments)
- Modifications:
  - » Remove constant lookup tables.
  - » Replace calls to assembly code.
- Vélus compilation: ≈1 min 40 s

	Vélus	*Hept+CC*	*Hept+gcc*	*Hept+gcci*	*Lus6+CC*	*Lus6+gcc*	*Lus6+gcci*
avgvelocity	315	385 (22%)	265 (-15%)	70 (-77%)	1 150 (265%)	625 (98%)	350 (11%)
count	55	55 (0%)	25 (-54%)	25 (-54%)	300 (445%)	160 (190%)	50 (-9%)
tracker	680	790 (16%)	530 (-22%)	500 (-26%)	2 610 (283%)	1 515 (122%)	735 (8%)
pip_ex	4 415	4 065 (-7%)	2 565 (-41%)	2 040 (-53%)	10 845 (145%)	6 245 (41%)	2 905 (-34%)
mp_longitudinal [16]	5 525	6 465 (17%)	3 465 (-37%)	2 835 (-48%)	11 675 (111%)	6 785 (22%)	3 135 (-43%)
cruise [54]	1 760	1 875 (6%)	1 230 (-30%)	1 230 (-30%)	5 855 (232%)	3 595 (104%)	1 965 (11%)
risingedgeretrigger [19]	285	300 (5%)	190 (-33%)	190 (-33%)	1 440 (405%)	820 (187%)	335 (17%)
chrono [20]	410	425 (3%)	305 (-25%)	305 (-25%)	2 490 (507%)	1 500 (265%)	670 (63%)
watchdog3 [26]	610	575 (-5%)	355 (-41%)	310 (-49%)	2 015 (230%)	1 135 (86%)	530 (-13%)
functionalchain [17]	11 550	13 535 (17%)	8 545 (-26%)	7 525 (-34%)	23 085 (99%)	14 280 (23%)	8 240 (-28%)
landing_gear [11]	9 660	8 475 (-12%)	5 880 (-39%)	5 810 (-39%)	25 470 (163%)	15 055 (55%)	8 025 (-16%)
minus [57]	890	900 (1%)	580 (-34%)	580 (-34%)	2 825 (217%)	1 620 (82%)	800 (-10%)
prodcell [32]	1 020	990 (-2%)	620 (-39%)	410 (-59%)	3 615 (254%)	2 050 (100%)	1 070 (4%)
ums_verif [57]	2 590	2 285 (-11%)	1 380 (-46%)	920 (-64%)	11 725 (352%)	6 730 (159%)	3 420 (32%)

Figure 12. WCET estimates in cycles [4] for step functions compiled for an armv7-a/vfpv3-d16 target with CompCert 2.6 (CC) and GCC 4.4.8 `-O1` without inlining (gcc) and with inlining (gcci). Percentages indicate the difference relative to the first column.

It performs loads and stores of volatile variables to model, respectively, input consumption and output production. The coinductive predicate presented in Section 1 is introduced to relate the trace of these events to input and output streams.

Finally, we exploit an existing CompCert lemma to transfer our results from the big-step model to the small-step one, from whence they can be extended to the generated assembly code to give the property stated at the beginning of the paper. The transfer lemma requires showing that a program does not diverge. This is possible because the body of the main loop always produces observable events.

5. Experimental Results

Our prototype compiler, Vélus, generates code for the platforms supported by CompCert (PowerPC, ARM, and x86). The code can be executed in a 'test mode' that `scanf`s inputs and `printf`s outputs using an alternative (unverified) entry point. The *verified* integration of generated code into a complete system where it would be triggered by interrupts and interact with hardware is the subject of ongoing work.

As there is no standard benchmark suite for Lustre, we adapted examples from the literature and the Lustre v4 distribution [57]. The resulting test suite comprises 14 programs, totaling about 160 nodes and 960 equations. We compared the code generated by Vélus with that produced by the Heptagon 1.03 [23] and Lustre v6 [35, 57] academic compilers. For the example with the deepest nesting of clocks (3 levels), both Heptagon and our prototype found the same optimal schedule. Otherwise, we follow the approach of [23, §6.2] and estimate the Worst-Case Execution Time (WCET) of the generated code using the open-source OTAWA v5 framework [4] with the 'trivial' script and default parameters.[10] For the targeted domain, an over-approximation to the WCET is usually more valuable than raw performance numbers. We compiled with CompCert 2.6 and GCC 4.8.4 (`-O1`) for the `arm-none-eabi` target (armv7-a) with a hardware floating-point unit (vfpv3-d16).

The results of our experiments are presented in Figure 12. The first column shows the worst-case estimates in cycles for the step functions produced by Vélus. These estimates compare favorably with those for generation with either Heptagon or Lustre v6 and then compilation with CompCert. Both Heptagon and Lustre (automatically) re-normalize the code to have one operator per equation, which can be costly for nested conditional statements, whereas our prototype simply maintains the (manually) normalized form. This re-normalization is unsurprising: both compilers must treat a richer input language, including arrays and automata, and both expect the generated code to be post-optimized by a C compiler. Compiling the generated code with GCC but still without any inlining greatly reduces the estimated WCETs, and the Heptagon code then outperforms the Vélus code. GCC applies 'if-conversions' to exploit predicated ARM instructions which avoids branching and thereby improves WCET estimates. The estimated WCETs for the Lustre v6 generated code only become competitive when inlining is enabled because Lustre v6 implements operators, like **pre** and −>, using separate functions. CompCert can perform inlining, but the default heuristic has not yet been adapted for this particular case. We note also that we use the modular compilation scheme of Lustre v6, while the code generator also provides more aggressive schemes like clock enumeration and automaton minimization [29, 56].

Finally, we tested our prototype on a large industrial application (≈6 000 nodes, ≈162 000 equations, ≈12 MB source file without comments). The source code was already normalized since it was generated with a graphical interface,

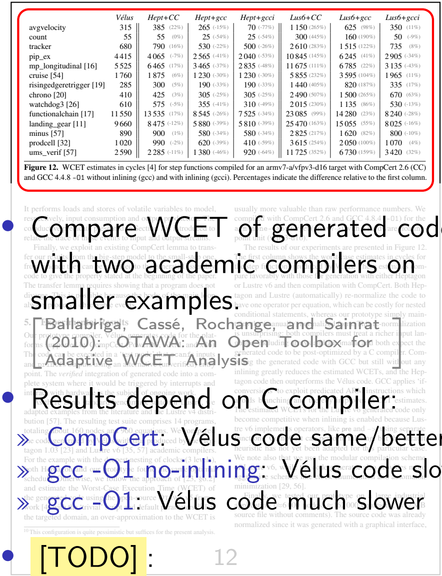
[10] This configuration is quite pessimistic but suffices for the present analysis.

12

Experimental results

Industrial application

- ≈6 000 nodes
- ≈162 000 equations
- ≈12 MB source file (minus comments)
- Modifications:
  - » Remove constant lookup tables.
  - » Replace calls to assembly code.
- Vélus compilation: ≈1 min 40 s



	Vélus	*Hept+CC*	*Hept+gcc*	*Hept+gcci*	*Lus6+CC*	*Lus6+gcc*	*Lus6+gcci*
avgvelocity	315	385 (22%)	265 (-15%)	70 (-77%)	1 150 (265%)	625 (98%)	350 (11%)
count	55	55 (0%)	25 (-54%)	25 (-54%)	300 (445%)	160 (190%)	50 (-9%)
tracker	680	790 (16%)	530 (-22%)	500 (-26%)	2 610 (283%)	1 515 (122%)	735 (8%)
pip_ex	4 415	4 065 (-7%)	2 565 (-41%)	2 040 (-53%)	10 845 (145%)	6 245 (41%)	2 905 (-34%)
mp_longitudinal [16]	5 525	6 465 (17%)	3 465 (-37%)	2 835 (-48%)	11 675 (111%)	6 785 (22%)	3 135 (-43%)
cruise [54]	1 760	1 875 (6%)	1 230 (-30%)	1 230 (-30%)	5 855 (232%)	3 595 (104%)	1 965 (11%)
risingedgeretrigger [19]	285	300 (5%)	190 (-33%)	190 (-33%)	1 440 (405%)	820 (187%)	335 (17%)
chrono [20]	410	425 (3%)	305 (-25%)	305 (-25%)	2 490 (507%)	1 500 (265%)	670 (63%)
watchdog3 [26]	610	575 (-5%)	355 (-41%)	310 (-49%)	2 015 (230%)	1 135 (86%)	530 (-13%)
functionalchain [17]	11 550	13 535 (17%)	8 545 (-26%)	7 525 (-34%)	23 085 (99%)	14 280 (23%)	8 240 (-28%)
landing_gear [11]	9 660	8 475 (-12%)	5 880 (-39%)	5 810 (-39%)	25 470 (163%)	15 055 (55%)	8 025 (-16%)
minus [57]	890	900 (1%)	580 (-34%)	580 (-34%)	2 825 (217%)	1 620 (82%)	800 (-10%)
prodcell [32]	1 020	990 (-2%)	620 (-39%)	410 (-59%)	3 615 (254%)	2 050 (100%)	1 070 (4%)
ums_verif [57]	2 590	2 285 (-11%)	1 380 (-46%)	920 (-64%)	11 725 (352%)	6 730 (159%)	3 420 (32%)

Figure 12. WCET estimates in cycles [4] for step functions compiled for an armv7-a/vfpv3-d16 target with CompCert 2.6 (CC) and GCC 4.4.8 -O1 without inlining (gcc) and with inlining (gcci). Percentages indicate the difference relative to the first column.

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

  [Ballabriga, Cassé, Rochange, and Sainrat (2010): OTAWA: An Open Toolbox for Adaptive WCET Analysis]

- Results depend on C compiler:
  - » CompCert: Vélus code same/better
  - » gcc -O1 no-inlining: Vélus code slower
  - » gcc -O1: Vélus code much slower
- [TODO]: adjust CompCert inlining heuristic.

Lustre: syntax

```
Definition ann : Type := (type * nclock).
Definition lann : Type := (list type * nclock).

Inductive exp : Type :=
| Econst : const → exp
| Evar   : ident → ann → exp
| Eunop  : unop → exp → ann → exp
| Ebinop : binop → exp → exp → ann → exp

| Ewhen  : list exp → ident → bool → lann → exp
| Emerge : ident → list exp → list exp → lann → exp
| Eite   : exp → list exp → list exp → lann → exp
| Efby   : list exp → list exp → list ann → exp

| Eapp   : ident → list exp → list ann → exp.

Definition equation : Type := (list ident * list exp).

Record node : Type :=
  mk_node {
    n_name     : ident;
    n_in       : list (ident * (type * clock));
    n_out      : list (ident * (type * clock));
    n_vars     : list (ident * (type * clock));
    n_eqs      : list equation;

    n_ingt0    : 0 < length n_in;
    n_outgt0   : 0 < length n_out;
    n_defd     : Permutation (vars_defined n_eqs)
                             (map fst (n_vars ++ n_out));
    n_nodup    : NoDupMembers (n_in ++ n_vars ++ n_out);
    n_good     : Forall NotReserved (n_in ++ n_vars ++ n_out)
  }.
```

```
(* No tuples. `Lists of flows' are flattened: *)
node shuffle (a, b, c, d : bool)
returns (w, x, y, z : bool);

(w, x, y, z) = shuffle(((a, (b, (c)), d)));
```

(e,...,e) when x / (e,...,e) when not x

merge x (e,...,e) (e,...,e)

(e,...,e) fby (e,...,e)

f(e, ..., e)

x,...,x = e,...,e

node f(x,...,x) returns (y,...,y);
var w, ..., w;
let x = ...; ... tel

Lustre: semantics 1/3

```
Definition history := PM.t (Stream value).

Notation sem_var H := (fun (x: ident) (s: Stream value) ⇒ PM.MapsTo x s H).

Inductive sem_exp : history → Stream bool → exp → list (Stream value) → Prop :=
| Sconst:
    sem_exp H b (Econst c) [const c b]

| Svar:
    sem_var H x s →
    sem_exp H b (Evar x ann) [s]

| Swhen:
    Forall2 (sem_exp H b) es ss →
    sem_var H x s →
    Forall2 (when k s) (concat ss) os →
    sem_exp H b (Ewhen es x k lann) os

| …
```

```
CoFixpoint const (c: const) (b: Stream bool) : Stream value :=
  match b with
  | true  ::: b' ⇒ present (sem_const c) ::: const c b'
  | false ::: b' ⇒ absent ::: const c b'
  end.
```

$$i^{\#}[\epsilon] = \epsilon$$
$$i^{\#}[true.cl] = v.i^{\#}[cl]$$
$$i^{\#}[false.cl] = abs.i^{\#}[cl]$$

```
CoInductive when (k: bool)
   : Stream value → Stream value → Stream value → Prop :=
| WhenA:
     when k xs cs rs →
     when k (absent ::: cs) (absent ::: xs) (absent ::: rs)

| WhenPA:
     when k xs cs rs →
     val_to_bool c = Some (negb k) →
     when k (present c ::: cs) (present x ::: xs) (absent ::: rs)

| WhenPP:
     when k xs cs rs →
     val_to_bool c = Some k →
     when k (present c ::: cs) (present x ::: xs) (present x ::: rs)
```

$$\texttt{when}^{\#}(s_1, s_2) = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon$$
$$\texttt{when}^{\#}(abs.xs, abs.cs) = abs.\texttt{when}^{\#}(xs, cs)$$
$$\texttt{when}^{\#}(x.xs, true.cs) = x.\texttt{when}^{\#}(xs, cs)$$
$$\texttt{when}^{\#}(x.xs, false.cs) = abs.\texttt{when}^{\#}(xs, cs)$$

Lustre: semantics 2/3

```
Inductive sem_exp : history → Stream bool → exp → list (Stream value) → Prop :=
| Sconst:
    sem_exp H b (Econst c) [const c b]

| ...

| Sfby:
    Forall2 (sem_exp H b) e0s s0ss →
    Forall2 (sem_exp H b) es sss →
    Forall3 fby (concat s0ss) (concat sss) os →
    sem_exp H b (Efby e0s es anns) os
```

$$
\begin{aligned}
\mathtt{fby}^{\#}(\epsilon, ys) &= \epsilon \\
\mathtt{fby}^{\#}(abs.xs, abs.ys) &= abs.\mathtt{fby}^{\#}(xs, ys) \\
\mathtt{fby}^{\#}(x.xs, y.ys) &= x.\mathtt{fby1}^{\#}(y, xs, ys) \\
\mathtt{fby1}^{\#}(v, \epsilon, ys) &= \epsilon \\
\mathtt{fby1}^{\#}(v, abs.xs, abs.ys) &= abs.\mathtt{fby1}^{\#}(v, xs, ys) \\
\mathtt{fby1}^{\#}(v, w.xs, s.ys) &= v.\mathtt{fby1}^{\#}(s, xs, ys)
\end{aligned}
$$

```
CoInductive fby
| FbyA:
      fby xs ys rs →
      fby (absent ::: xs) (absent ::: ys) (absent ::: rs)

| FbyP:
      fby1 y xs ys rs →
      fby (present x ::: xs) (present y ::: ys) (present x ::: rs).

CoInductive fby1
| Fby1A:
      fby1 v xs ys rs →
      fby1 v (absent ::: xs) (absent ::: ys) (absent ::: rs)

| Fby1P:
      fby1 s xs ys rs →
      fby1 v (present w ::: xs) (present s ::: ys) (present v ::: rs).
```

Lustre: semantics 3/3

```
Inductive sem_exp : history → Stream bool → exp → list (Stream value) → Prop :=
| Sconst:
    sem_exp H b (Econst c) [const c b]

| ...

| Sapp:
    Forall2 (sem_exp H b) es ss →
    sem_node f (concat ss) os →
    sem_exp H b (Eapp f es lann) os

| ...

with sem_equation: history → Stream bool → equation → Prop :=
| Seq:
    Forall2 (sem_exp H b) es ss →
    Forall2 (sem_var H) xs (concat ss) →
    sem_equation H b (xs, es)

with sem_node: ident → list (Stream value) → list (Stream value) → Prop :=
| Snode:
    find_node f G = Some n →
    Forall2 (sem_var H) (idents n.(n_in)) ss →
    Forall2 (sem_var H) (idents n.(n_out)) os →
    Forall (sem_equation H b) n.(n_eqs) →
    b = sclocksof ss →
    sem_node f ss os.
```

```
CoFixpoint sclocksof (ss: list (Stream value)) : Stream bool :=
  ∃b (fun s⇒ hd s <>b absent) ss ::: sclocksof (List.map tl ss).
```

Prior work on Lustre semantics in Coq

- [Boulmé and Hamon (2001): A clocked denotational semantics for Lucid-Synchrone in Coq]
  - Shallow embedding of Lucid Synchrone into Coq.
  - Embed the clocking rules into the Coq type system.
  - Use clocks (boolean streams) to control rhythms.
  - Denotational semantics using co-fixpoints.
  - Values: present, absent, and *fail*.

Prior work on Lustre semantics in Coq

- [Boulmé and Hamon (2001): A clocked denotational semantics for Lucid-Synchrone in Coq]
  - Shallow embedding of Lucid Synchrone into Coq.
  - Embed the clocking rules into the Coq type system.
  - Use clocks (boolean streams) to control rhythms.
  - Denotational semantics using co-fixpoints.
  - Values: present, absent, and *fail*.

- [Paulin-Mohring (2009): A constructive denotational semantics for Kahn networks in Coq]
  - Denotational semantics of Kahn process networks.
  - ```
    CoInductive Str (A:Type) : Type :=
    | Eps: StrA → Str A
    | cons: A → StrA → Str A
    ```
  - Least element: $\texttt{Eps}^{\infty}$
  - Shallow embedding of programs.
    ```

Prior work on Lustre semantics in Coq

- [Boulmé and Hamon (2001): A clocked denotational semantics for Lucid-Synchrone in Coq]
  - Shallow embedding of Lucid Synchrone into Coq.
  - Embed the clocking rules into the Coq type system.
  - Use clocks (boolean streams) to control rhythms.
  - Denotational semantics using co-fixpoints.
  - Values: present, absent, and *fail*.

- [Paulin-Mohring (2009): A constructive denotational semantics for Kahn networks in Coq]
  - Denotational semantics of Kahn process networks.
  - ```
    CoInductive Str (A:Type) : Type :=
    | Eps: StrA → Str A
    | cons: A → StrA → Str A
    ```
  - Least element: $\texttt{Eps}^{\infty}$
  - Shallow embedding of programs.

- [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
  - Streams as (backward) finite lists.
  - Deep embedding of programs.
  - Relational semantics linking programs to lists.
    ```

Conclusion

First results

- Working compiler from Lustre to assembler in Coq.
  [Bourke, Dagand, Pouzet, and Rieg (2017): Vérification de la génération modulaire du code impératif pour Lustre] [Bourke, Brun, Dagand, Leroy, Pouzet, and Rieg (2017): A Formally Verified Compiler for Lustre]
- Formally relate dataflow model to imperative code.
- Generate Clight for CompCert; change to richer memory model.
- Intermediate language and separation predicates were decisive.

Ongoing work

- Finish normalization pass, add resets, add automata…
- Prove that a well-typed program has a semantics.
- Combine interactive and automatic proof to verify Lustre programs.
  - Can verify reactive models in Isabelle. [Bourke, Glabbeek, and Höfner (2016): Mechanizing a Process Algebra for Network Protocols]
  - Can compile reactive programs in Coq.
  - What's the best way to do both at the same time?
- Treat side-effects in dataflow model and integrate C code.

References I

- Auger, C. (Apr. 2013). "Compilation certifiée de SCADE/LUSTRE". PhD thesis. Orsay, France: Univ. Paris Sud 11.
- Ballabriga, C., H. Cassé, C. Rochange, and P. Sainrat (Oct. 2010). "OTAWA: An Open Toolbox for Adaptive WCET Analysis". In: *8th IFIP WG 10.2 Int. Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*. Vol. 6399. LNCS. Waidhofen an der Ybbs, Austria: Springer, pp. 35–46.
- Biernacki, D., J.-L. Colaço, G. Hamon, and M. Pouzet (June 2008). "Clock-directed modular code generation for synchronous data-flow languages". In: *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. Tucson, AZ, USA: ACM Press, pp. 121–130.
- Boulmé, S. and G. Hamon (Nov. 2001). *A clocked denotational semantics for Lucid-Synchrone in Coq*. Tech. rep. LIP6.
- Bourke, T., L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg (June 2017). "A Formally Verified Compiler for Lustre". In: *Proc. 2017 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain: ACM Press, pp. 586–601.

References II

- Bourke, T., P.-É. Dagand, M. Pouzet, and L. Rieg (Jan. 2017). "Vérification de la génération modulaire du code impératif pour Lustre". In: *28ièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*. Ed. by J. Signoles and S. Boldo. Gourette, Pyrénées, France, pp. 165–179.
- Bourke, T., R. J. van Glabbeek, and P. Höfner (Mar. 2016). "Mechanizing a Process Algebra for Network Protocols". In: *J. Automated Reasoning* 56.3, pp. 309–341.
- Caspi, P., D. Pilaud, N. Halbwachs, and J. Plaice (Jan. 1987). "LUSTRE: A declarative language for programming synchronous systems". In: *Proc. 14th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1987)*. Munich, Germany: ACM Press, pp. 178–188.
- Colaço, J.-L., B. Pagano, and M. Pouzet (Sept. 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.

References III

- Hamon, G. and M. Pouzet (Sept. 2000). "Modular Resetting of Synchronous Data-Flow Programs". In: *Proc. 2nd ACM SIGPLAN Int. Conf. on Principles and Practice of Declarative Programming (PPDP 2000)*. Ed. by F. Pfenning. ACM. Montreal, Canada, pp. 289–300.
- Jourdan, J.-H., F. Pottier, and X. Leroy (Mar. 2012). "Validating LR(1) parsers". In: *21st European Symposium on Programming (ESOP 2012), held as part of European Joint Conferences on Theory and Practice of Software (ETAPS 2012)*. Ed. by H. Seidl. Vol. 7211. LNCS. Tallinn, Estonia: Springer, pp. 397–416.
- 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, G. Huet, J.-J. Lévy, and G. Plotkin. CUP, pp. 383–413.