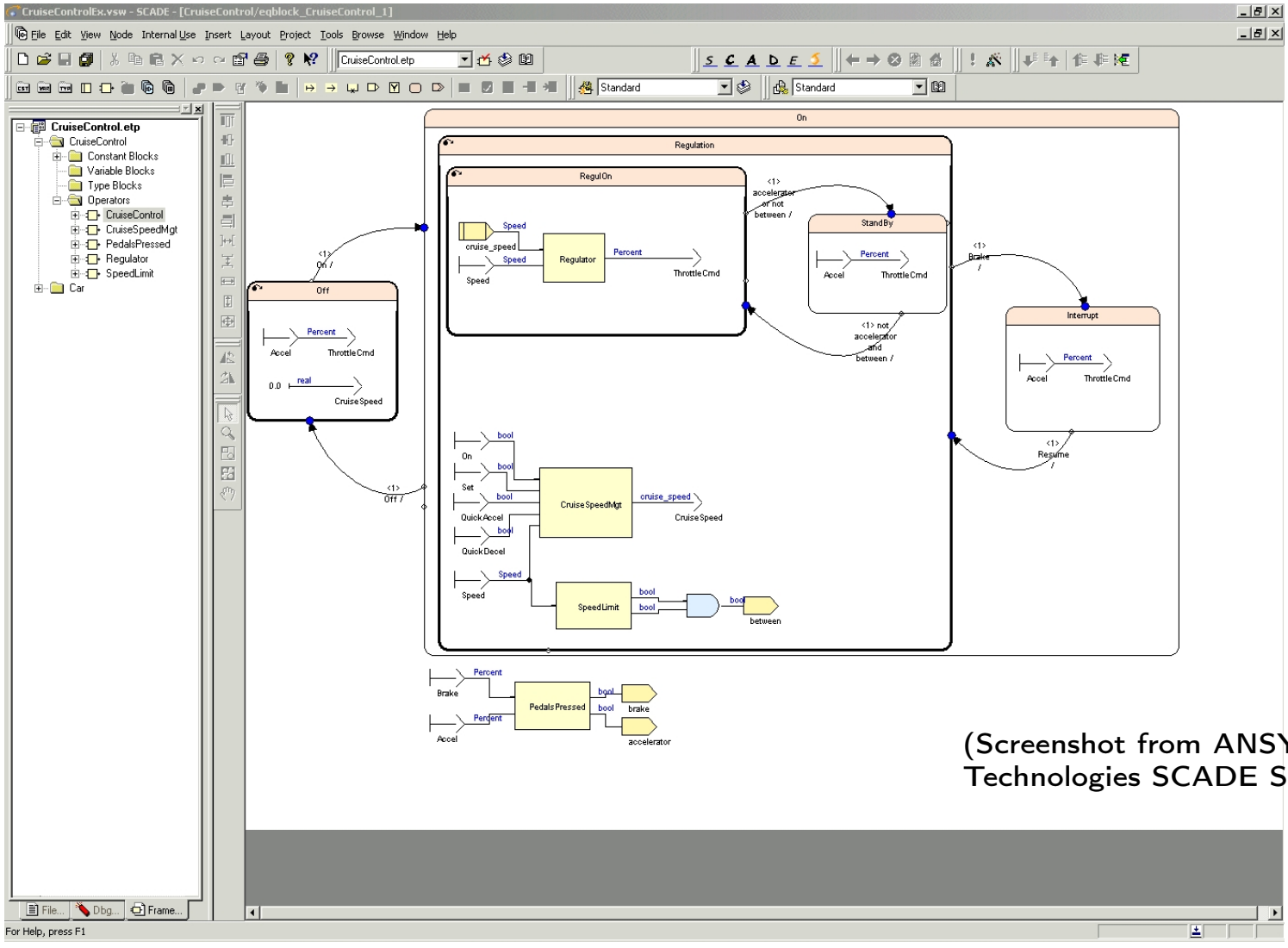
Parallélisme synchrone MPRI 2-23-1
A formally verified compiler for Lustre

Timothy Bourke

This course describes an ongoing research collaboration with Lélio Brun, Pierre-Évariste Dagand, Paul Jeanmaire, Xavier Leroy, Basile Pesin, Marc Pouzet, and Lionel Rieg.

21 September 2021



(Screenshot from ANSYS/Esterel Technologies SCADE Suite)

- Widely used to program safety-critical software:
  - » Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear.
  - » Airbus (A340, A380), Comac, EADS Astrium, Embraer, Eurocopter, PIAGGIO Aerospace, Pratt & Whitney, Sukhoi, Turbomeca, Siemens, ...
- DO-178B level A certified development tool.

Certification for Safety-Critical Systems

- DO-178C: "Software Considerations in Airborne Systems and Equipment Certification"
  - » Defines a software development process.
  - » Manual review of artifacts related to design, code, and test cases.

  [Rushby (2011): New Challenges in Certification for Aircraft Software]

- The Scade KCG code generator is *certified*.
  - » No need for code reviews of generated code.
  - » Can test based on High-Level Requirements.
  - » Evaluate testing coverage on models.

- Different approach: formal specification and *verification* of a code generator.
  - » Not directly applicable to certification processes

    [Wagner, Cofer, Slind, Tinelli, and Mebsout (2017): Formal Methods Tool Qualification]
  - » Can it complement existing approaches in other ways?

We are interested in the scientific questions and the technical challenges.

Overview

Lustre: syntax and semantics

Lustre: normalization

Translation: from dataflow programs to imperative code

Optimization: control structure fusion

Generation: from Obc to Clight

Conclusion

What are we doing?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - » Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2014): A Formalization and Proof of a Modular Lustre Code Generator].
- Prove that the generated code implements the dataflow semantics.

What are we doing?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - » Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2014): A Formalization and Proof of a Modular Lustre Code Generator].
- Prove that the generated code implements the dataflow semantics.
- Coq? [The Coq Development Team (2016): The Coq proof assistant reference manual]
  - » A functional programming language;
  - » 'Extraction' to OCaml programs;
  - » A specification language (a form of higher-order logic);
  - » Tactic-based interactive proof.

What are we doing?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - » Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2014): A Formalization and Proof of a Modular Lustre Code Generator].
- Prove that the generated code implements the dataflow semantics.
- Coq? [The Coq Development Team (2016): The Coq proof assistant reference manual]
  - » A functional programming language;
  - » 'Extraction' to OCaml programs;
  - » A specification language (a form of higher-order logic);
  - » Tactic-based interactive proof.
- Why not use HOL, Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

What are we doing?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - » Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2014): A Formalization and Proof of a Modular Lustre Code Generator].
- Prove that the generated code implements the dataflow semantics.
- Coq? [The Coq Development Team (2016): The Coq proof assistant reference manual]
  - » A functional programming language;
  - » 'Extraction' to OCaml programs;
  - » A specification language (a form of higher-order logic);
  - » Tactic-based interactive proof.
- Why not use HOL, Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

  CompCert: a formal model and compiler for a subset of C
  - » A generic machine-level model of execution and memory
  - » A verified path to assembly code output (PowerPC, ARM, x86)
  - » No need for a garbage-collected runtime.

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

What are we doing?

- Implement a Lustre compiler in the Coq Interactive Theorem Prover.
  - » Following a previous attempt [Auger, Colaço, Hamon, and Pouzet (2014): A Formalization and Proof of a Modular Lustre Code Generator].
- Prove that the generated code implements the dataflow semantics.
- Coq? [The Coq Development Team (2016): The Coq proof assistant reference manual]
  - » A functional programming language;
  - » 'Extraction' to OCaml programs;
  - » A specification language (a form of higher-order logic);
  - » Tactic-based interactive proof.
- Why not use HOL, Isabelle, PVS, ACL2, Agda, or ⟨your favourite tool⟩?

  CompCert: a formal model and compiler for a subset of C
  - » A generic machine-level model of execution and memory
  - » A verified path to assembly code output (PowerPC, ARM, x86)
  - » No need for a garbage-collected runtime.

  [Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End] [Leroy (2009): Formal verification of a realistic compiler]
- Computer assistance is all but essential for such detailed models.

Why do it?

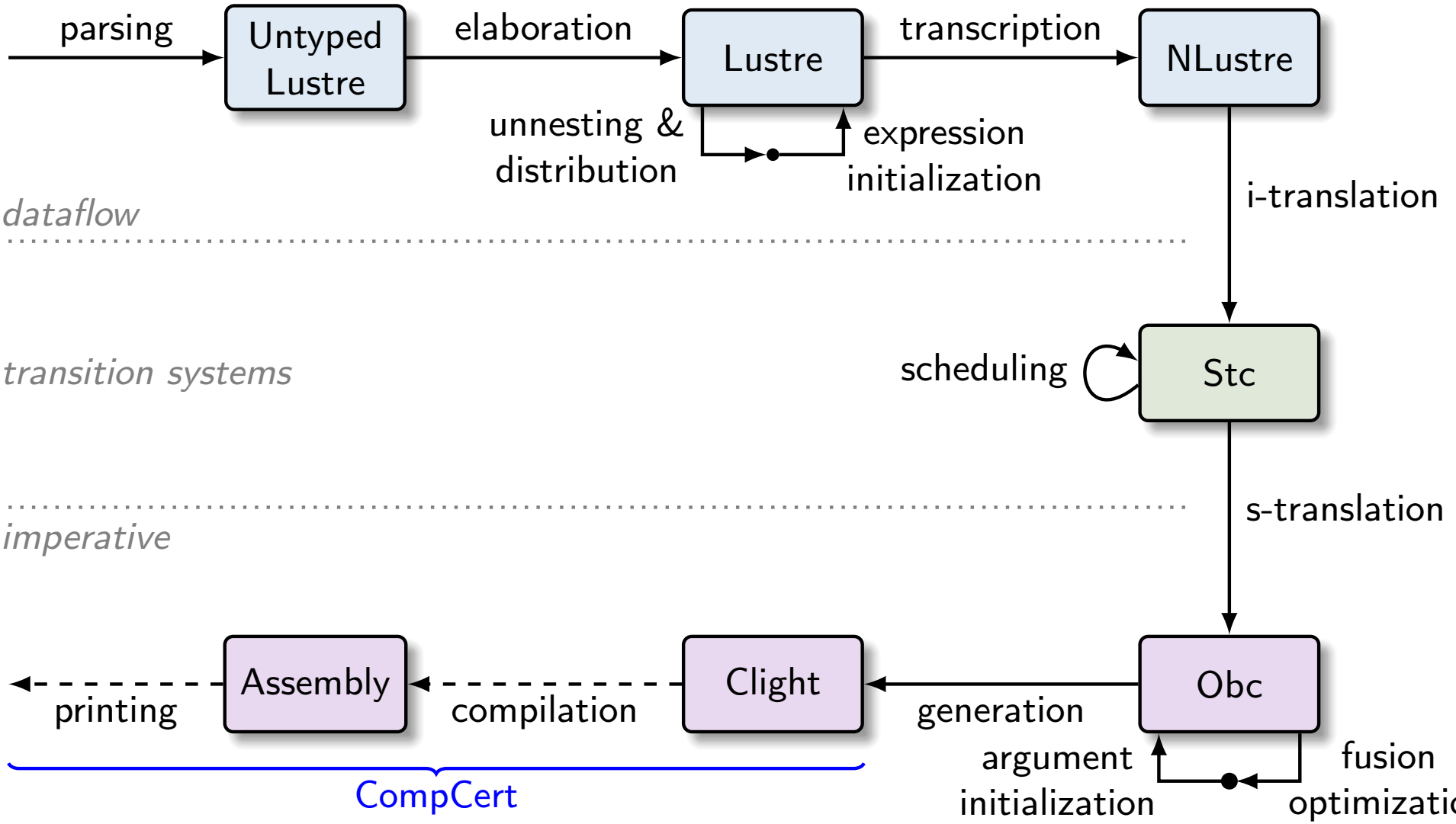
- Challenge: Never done before: how to do it?
  (as methodically and as simply as possible)

- Vision: Verify Lustre programs;
  Get guarantees about assembly code.
  - » Treat the details of real processors
    (memory representations and arithmetic).
  - » Treat external functions called by Lustre programs.
  - » Verify abstract ('almost executable') models of reactive-systems
    (static parameters, real-time properties).

- 'Digitized' formal models of Lustre.
  - » Machine-manipulable specification of language, type systems, compilation.
  - » Starting point for other projects:
    - Semantics of modular reset and state machines.
    - Detailed treatment of side-effects and host language.
    - Analysis of optimization passes. An aid to industrial certification?

The Vélus Lustre Compiler

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



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

Overview

Lustre: syntax and semantics

Lustre: normalization

Translation: from dataflow programs to imperative code

Optimization: control structure fusion

Generation: from Obc to Clight

Conclusion

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 → option 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

- The usual story: associate Abstract Syntax Trees (programs) to models
- Sequential programs:
  - A state associates each variable to a value
  - Expressions are evaluated in the state
  - The state is updated by successive commands
  - Possibly include a sequence of interleaved 'external events'
- Dataflow programs:
  - Associate each variable with a stream of values
  - Track synchronization by making absence explicit
  - Models the 'grid' that associates variables to synchronized streams
  - Each equation constrains the set of possible models

Semantic model: design constraints

- Judging the (informal) 'correctness' of the development.
- Proving properties of programs.
- Proving correctness of compilation.
- Proving properties of the language
  - » Existence of semantics
  - » Determinism of language
  - » Correctness of the clocking system

Pen-and-paper Kahn semantics

[Colaço and Pouzet (2003): Clocks as First Class Abstract Types]

$$
\begin{aligned}
i^{\#} &= i.i^{\#} \\
op^{\#}(s_1, s_2) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(v_1.s_1, v_2.s_2) &= (v_1\ op\ v_2).op^{\#}(s_1, s_2) \\
\\
\texttt{fby}^{\#}(\epsilon, ys) &= \epsilon \\
\texttt{fby}^{\#}(v.xs, ys) &= v.ys \\
\\
\texttt{when}^{\#}(s_1, s_2) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\texttt{when}^{\#}(v_1.s_1, true.s_2) &= v_1.(\texttt{when}^{\#}(s_1, s_2)) \\
\texttt{when}^{\#}(v_1.s_1, false.s_2) &= \texttt{when}^{\#}(s_1, s_2) \\
\\
\texttt{merge}^{\#}(s_1, s_2, s_3) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(true.s_1, v_2.s_2, s_3) &= v_2.\texttt{merge}^{\#}(s_1, s_2, s_3) \\
\texttt{merge}^{\#}(false.s_1, s_2, v_3.s_3) &= v_3.\texttt{merge}^{\#}(s_1, s_2, s_3)
\end{aligned}
$$

- Ignoring synchronization gives a relatively simple model based on fixed points of mutually recursive stream transformers.
- 'Machines' reading and writing through unbounded FIFOs

[Kahn (1974): The Semantics of a Simple Language for Parallel Programming]

Pen-and-paper synchronous semantics

[Colaço and Pouzet (2003): Clocks as First Class Abstract Types]

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

$$
\begin{aligned}
op^{\#}(s_1, s_2) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(abs.s_1, abs.s_2) &= abs.op^{\#}(s_1, s_2) \\
op^{\#}(v_1.s_1, v_2.s_2) &= (v_1\ op\ v_2).op^{\#}(s_1, s_2)
\end{aligned}
$$

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

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

$$
\begin{aligned}
\texttt{merge}^{\#}(s_1, s_2, s_3) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(abs.cs, abs.xs, abs.ys) &= abs.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(true.cs, x.xs, abs.ys) &= x.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(false.cs, abs.xs, y.ys) &= y.\texttt{merge}^{\#}(cs, xs, ys)
\end{aligned}
$$

- Synchronous semantics: explicitly represent absence
- The definitions become more technical
- Defined for fewer programs
- The link to discrete time is explicit
- The model is closer to an implementation
- Unbounded FIFOs are not required

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

```
CoInductive Stream (A : Type) :=
  Cons : A → Stream A → Stream A.
```

- Functions from natural numbers to values:

```
Notation stream A := (nat → A).
```

Lustre: semantics 1/4

```
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.
```

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

```
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)
```

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

Lustre: semantics 2/4

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

| ...

| Smerge:
    sem_var H x s →
    Forall2 (sem_exp H b) ets ts →
    Forall2 (sem_exp H b) efs fs →
    Forall3 (merge s) (concat ts) (concat fs) os →
    sem_exp H b (Emerge x ets efs lann) os
```

$$
\begin{aligned}
\texttt{merge}^{\#}(s_1, s_2, s_3) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(abs.cs, abs.xs, abs.ys) &= abs.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(true.cs, x.xs, abs.ys) &= x.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(false.cs, abs.xs, y.ys) &= y.\texttt{merge}^{\#}(cs, xs, ys)
\end{aligned}
$$

```
CoInductive merge
| MergeA:
    merge xs ts fs rs →
    merge (          absent ::: xs) (   absent ::: ts) (   absent ::: fs) (   absent ::: rs)
| MergeT:
    merge xs ts fs rs →
    merge (present  true_val ::: xs) (present t ::: ts) (   absent ::: fs) (present t ::: rs)
| MergeF:
    merge xs ts fs rs →
    merge (present false_val ::: xs) (   absent ::: ts) (present f ::: fs) (present f ::: rs).
```

$$\frac{s \equiv \text{const } bs\ [\![\mathrm{c}]\!]}{G, H, bs \vdash \mathrm{c} \Downarrow [s]}$$

$$\text{const } (\mathrm{T} \cdot bs)\ c = \langle c \rangle \cdot \text{const } bs\ c$$
$$\text{const } (\mathrm{F} \cdot bs)\ c = \langle\rangle \cdot \text{const } bs\ c$$

(a) Constants

$$\frac{G, H, bs \vdash \boldsymbol{e} \Downarrow H(\boldsymbol{x})}{G, H, bs \vdash \boldsymbol{x} = \boldsymbol{e}}$$

(b) Equations

$$\frac{H(x) = s \qquad G, H, bs \vdash \boldsymbol{e_t} \Downarrow \boldsymbol{ts} \qquad G, H, bs \vdash \boldsymbol{e_f} \Downarrow \boldsymbol{fs} \qquad \text{merge } s\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{G, H, bs \vdash \texttt{merge}(\ x;\ \boldsymbol{e_t};\ \boldsymbol{e_f}) \Downarrow \boldsymbol{vs}}$$

$$\frac{\text{merge } cs\ ts\ fs \doteq vs}{\text{merge } (\langle\rangle \cdot cs)\ (\langle\rangle \cdot ts)\ (\langle\rangle \cdot fs) \doteq \langle\rangle \cdot vs}$$

$$\frac{\text{merge } cs\ ts\ fs \doteq vs}{\text{merge } (\langle \mathrm{T} \rangle \cdot cs)\ (\langle t \rangle \cdot ts)\ (\langle\rangle \cdot fs) \doteq \langle t \rangle \cdot vs}$$

$$\frac{\text{merge } cs\ ts\ fs \doteq vs}{\text{merge } (\langle \mathrm{F} \rangle \cdot cs)\ (\langle\rangle \cdot ts)\ (\langle f \rangle \cdot fs) \doteq \langle f \rangle \cdot vs}$$

(c) Rule and operator for `merge`

$$\frac{\text{node}(G, f) \doteq n \qquad H(n.\mathbf{in}) = \boldsymbol{xs} \qquad H(n.\mathbf{out}) = \boldsymbol{ys} \qquad \forall eq \in n.\mathbf{eqs},\ G, H, (\texttt{base-of}\ \boldsymbol{xs}) \vdash eq}{G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}$$

(d) Nodes

Fig. 8. Lustre: Selected semantic rules and operators

Lustre: semantics 3/4

```
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
```

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

```
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 4/4

```
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).
```

Semantics Summary

- Relational semantic model: a set of constraints on an environment H
- Inductive rules over syntax using coinductive operators on streams.
- Good:
  - No need to worry about fixed points
  - Easy to destruct and construct in correctness proofs
  - Works well in compiler correctness proofs
- Less good:
  - We cannot execute the model to see what it means
  - Are the constraints satisfiable? (Existence of a semantics, > 0)
  - Does a program have one model? (Determinism, $<= 1$)
  - What about proofs about 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*.

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.

Semantics of the Modular Reset

- The modular reset is a relatively new feature compared to the classic Lustre operators
- It introduces imperative characteristics into the dataflow language
- Coq formalization:
  - » Cannot simply just encode existing pen-and-paper definitions
  - » Need to invent new ones

- Early definition propagates 'reset wires' through to fbys

  [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs]
- In fact, the reset can be expressed as a recursive dataflow program

  [Caspi (1994): Towards recursive block diagrams] [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs]
- This inspired the original approach in L. Brun's thesis

  [Brun (2020): Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset] [Bourke, Brun, and Pouzet (2020): Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset]

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F
i	0
$nat(i)$	0
(restart nat every r)(i)	0

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F
i	0	5
$nat(i)$	0	1
(restart nat every r)(i)	0	1

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T
i	0	5	10
$nat(i)$	0	1	2
(restart nat every r)(i)	0	1	10

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F
i	0	5	10	15
$nat(i)$	0	1	2	3
(restart nat every r)(i)	0	1	10	11

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F
i	0	5	10	15	20
$nat(i)$	0	1	2	3	4
(restart nat every r)(i)	0	1	10	11	12

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T
i	0	5	10	15	20	25
$nat(i)$	0	1	2	3	4	5
(restart nat every r)(i)	0	1	10	11	12	25

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F
i	0	5	10	15	20	25	30
$nat(i)$	0	1	2	3	4	5	6
(restart nat every r)(i)	0	1	10	11	12	25	26

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T
i	0	5	10	15	20	25	30	35
$nat(i)$	0	1	2	3	4	5	6	7
(restart *nat* every r)(i)	0	1	10	11	12	25	26	35

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F
i	0	5	10	15	20	25	30	35	40
$nat(i)$	0	1	2	3	4	5	6	7	8
(restart nat every r)(i)	0	1	10	11	12	25	26	35	36

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
$nat(i)$	0	1	2	3	4	5	6	7	8	$\cdots$
(restart nat every r)(i)	0	1	10	11	12	25	26	35	36	$\cdots$

Semantics of modular reset

Recursive definition (not valid in Lustre) [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs]

> f(x) every r acts as f(x) until r is true and after r has been true, [it] acts as f(x') every r' where x' (resp. r') is the sub-stream of x (resp. r) starting when r is true.

Semantics of modular reset

Recursive definition (not valid in Lustre) [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs]

> f(x) every r acts as f(x) until r is true and after r has been true, [it] acts as f(x') every r' where x' (resp. r') is the sub-stream of x (resp. r) starting when r is true.

```
node true_until(r: bool) returns (c: bool)
let
  c = if r then false else (true fby c);
tel

node reset_f(x: int, r: bool) returns (y: int)
  var c: bool;
let
  c = true_until(r);
  y = merge c (f(x when c)) (reset_f((x, r) whenot c));
tel
```

Semantics of modular reset

Recursive definition (not valid in Lustre) [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs]

> f(x) every r acts as f(x) until r is true and after r has been true, [it] acts as f(x') every r' where x' (resp. r') is the sub-stream of x (resp. r) starting when r is true.

```
node true_until(r: bool) returns (c: bool)
let
  c = if r then false else (true fby c);
tel

node reset_f(x: int, r: bool) returns (y: int)
  var c: bool;
let
  c = true_until(r);
  y = merge c (f(x when c)) (reset_f((x, r) whenot c));
tel
```

Use an intricate co-inductive predicate or find a simpler solution?

Semantics of modular reset: unrolling the recursion

r	F	F	T	F	F	T	F	T	F	⋯
i	0	5	10	15	20	25	30	35	40	⋯

Semantics of modular reset: unrolling the recursion

r	F	F	T	F	F	T	F	T	F	⋯
i	0	5	10	15	20	25	30	35	40	⋯
mask 0 r i	0	5	–	–	–	–	–	–	–	⋯
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	⋯

Semantics of modular reset: unrolling the recursion

r	F	F	T	F	F	T	F	T	F	⋯
i	0	5	10	15	20	25	30	35	40	⋯
mask 0 r i	0	5	–	–	–	–	–	–	–	⋯
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	⋯
mask 1 r i	–	–	10	15	20	–	–	–	–	⋯
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	⋯

Semantics of modular reset: unrolling the recursion

r	F	F	T	F	F	T	F	T	F	⋯
i	0	5	10	15	20	25	30	35	40	⋯
mask 0 r i	0	5	–	–	–	–	–	–	–	⋯
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	⋯
mask 1 r i	–	–	10	15	20	–	–	–	–	⋯
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	⋯
mask 2 r i	–	–	–	–	–	25	30	–	–	⋯
nat(mask 2 r i)	–	–	–	–	–	25	26	–	–	⋯

Semantics of modular reset: unrolling the recursion

r	F	F	T	F	F	T	F	T	F	⋯
i	0	5	10	15	20	25	30	35	40	⋯
mask 0 r i	0	5	–	–	–	–	–	–	–	⋯
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	⋯
mask 1 r i	–	–	10	15	20	–	–	–	–	⋯
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	⋯
mask 2 r i	–	–	–	–	–	25	30	–	–	⋯
nat(mask 2 r i)	–	–	–	–	–	25	26	–	–	⋯
mask 3 r i	–	–	–	–	–	–	–	35	40	⋯
nat(mask 3 r i)	–	–	–	–	–	–	–	35	36	⋯

Semantics of modular reset: unrolling the recursion

r	F	F	T	F	F	T	F	T	F	⋯
i	0	5	10	15	20	25	30	35	40	⋯
mask 0 r i	0	5	–	–	–	–	–	–	–	⋯
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	⋯
mask 1 r i	–	–	10	15	20	–	–	–	–	⋯
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	⋯
mask 2 r i	–	–	–	–	–	25	30	–	–	⋯
nat(mask 2 r i)	–	–	–	–	–	25	26	–	–	⋯
mask 3 r i	–	–	–	–	–	–	–	35	40	⋯
nat(mask 3 r i)	–	–	–	–	–	–	–	35	36	⋯
⋮										
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	⋯

Semantics of modular reset in LaTeX

$$\frac{bs = \text{base-of } xs \quad G(f) = n \quad H_*(n.\mathbf{in}) = xs \quad H_*(n.\mathbf{out}) = ys \quad \text{respects-clock } bs\ H\ n.\mathbf{in} \quad \forall eq \in n.\mathbf{eqs},\ G, bs, H \vdash eq}{G \vdash f(xs) \Downarrow ys}$$

$$\frac{bs, H \vdash e :: ck \Downarrow H_*(x)}{G, bs, H \vdash x =_{ck} e}$$

$$\frac{bs, H \vdash \boldsymbol{e} \Downarrow vs \quad bs, H \vdash ck \Downarrow \text{base-of } vs \quad G \vdash f(vs) \Downarrow H_*(\boldsymbol{x})}{G, bs, H \vdash \boldsymbol{x} =_{ck} f\,(\boldsymbol{e})}$$

$$\frac{bs, H \vdash e :: ck \Downarrow vs \quad H_*(x) \approx \mathsf{fby}\,([\![c]\!], vs)}{G, bs, H \vdash x =_{ck} c\ \mathsf{fby}\ e}$$

$$\frac{bs, H \vdash \boldsymbol{e} \Downarrow vs \quad \text{bools-of } H_*(y)\ rs \quad bs, H \vdash ck \Downarrow \text{base-of } vs \quad \forall k,\ G \vdash f\left(\text{mask}^k_{rs}\ vs\right) \Downarrow \text{mask}^k_{rs}\ H_*(\boldsymbol{x})}{G, bs, H \vdash \boldsymbol{x} =_{ck} \left(\mathtt{restart}\ f\ \mathtt{every}\ y^{ck_y}\right)(\boldsymbol{e})}$$

$$\text{bools-of } xs\ rs = \forall n, \text{if } rs_n \text{ then } (xs_n = \langle T \rangle) \text{ else } (xs_n = \langle F \rangle \vee xs_n = \langle\rangle)$$

$$\text{mask}^k_{rs}\ vs = \lambda\, n.\ \text{if } (\text{count } rs\ n = k) \text{ then } vs_n \text{ else } (\text{map } (\lambda x.\ \langle\rangle)\ vs_0)$$

$$\text{count } rs\ n = \text{if } rs_n \text{ then } c + 1 \text{ else } c \quad \text{where } c = (\text{if } n = 0 \text{ then } 0 \text{ else count } rs\ (n-1))$$

Semantics of modular reset in Coq

```
Inductive sem_equation : history → clock → equation → Prop :=
...
| SEqApp:
    Forall2 (sem_lexp bk H) es ls →
    Forall2 (sem_var H) xs xss →
    sem_clock bk H ck (clock_of ls) →
    sem_node f ls xss →
    sem_equation bk H (EqApp xs ck f es None)

| SEqReset:
    Forall2 (sem_lexp bk H) es ls →
    Forall2 (sem_var H) xs xss →
    sem_clock bk H ck (clock_of ls) →
    sem_var H y ys →
    bools_of ys rs →
    (∀ k, sem_node f (mask k rs ls) (mask k rs xss)) →
    sem_equation bk H (EqApp xs ck f es (Some y))
...
```

```
Fixpoint count (rs: cstream) (n: nat) : nat :=
    let c := match n with
             | 0   ⇒ 0
             | S n ⇒ count rs n
             end
    in
    if rs n then S c else c.

Definition all_absent {A} (l: list A) : list value :=
    map (fun _ ⇒ absent) l.

Definition mask (k: nat) (rs: cstream)
    (xss: stream (list value)) : stream (list value) :=
    fun n ⇒ if k =? (count rs n) then xss n
            else all_absent (xss 0).
```

```
with sem_node: ident → stream (list value) → stream (list value) → Prop :=
| SNode:
    bk = clock_of xss →
    find_node f G = Some n →
    Forall2 (sem_var H) (map fst n.(n_in)) xss →
    Forall2 (sem_var H) (map fst n.(n_out)) yss →
    sem_clocked_vars bk H (idck n.(n_in)) →
    Forall (sem_equation bk H) n.(n_eqs) →
    sem_node f xss yss.
```

Overview

Lustre: syntax and semantics

Lustre: normalization

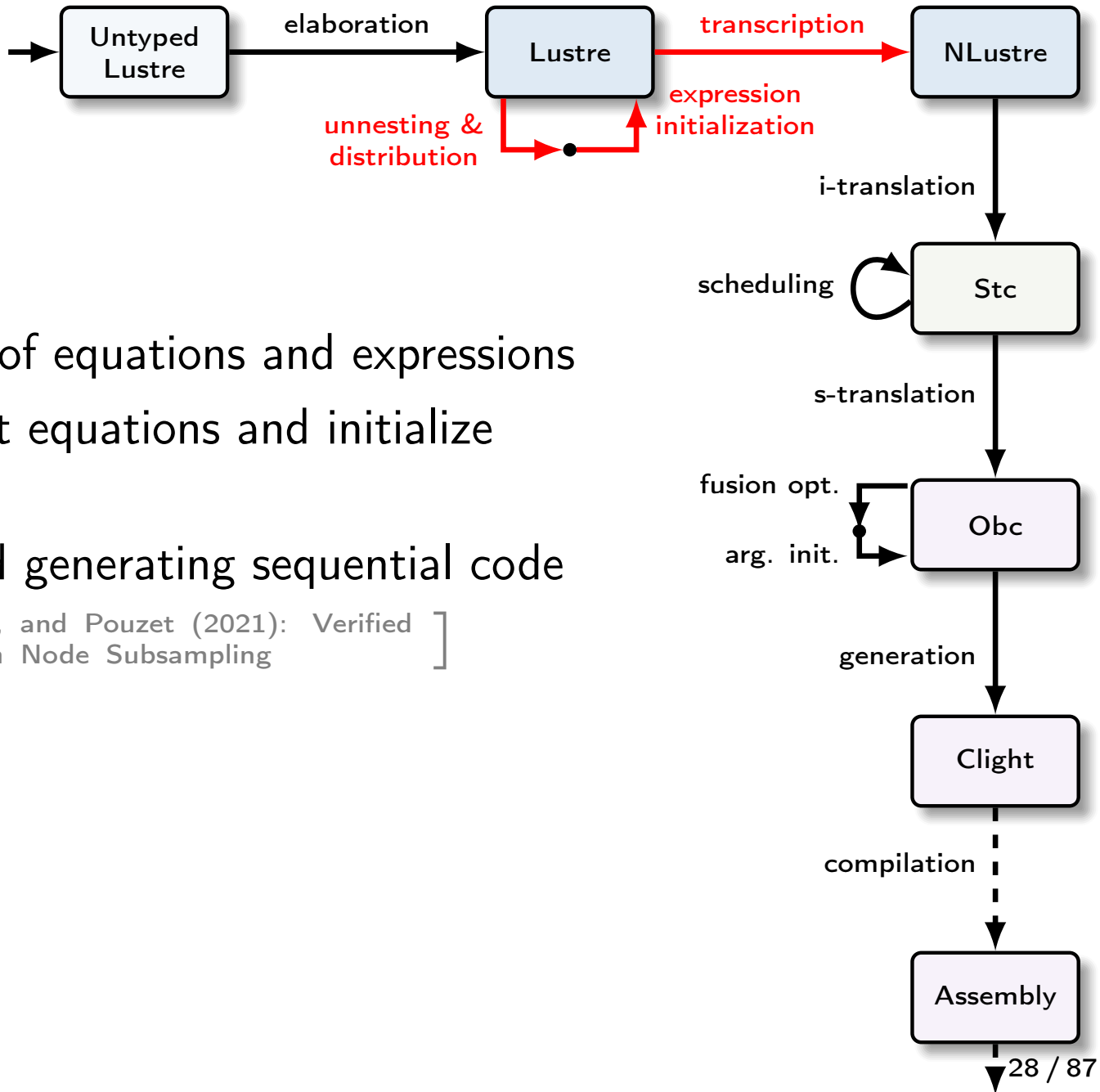
Translation: from dataflow programs to imperative code

Optimization: control structure fusion

Generation: from Obc to Clight

Conclusion

Normalization: Lustre to NLustre



- Simplify the form of equations and expressions
- Put fbys in distinct equations and initialize with a constant
- A first step toward generating sequential code
- [Bourke, Jeanmaire, Pesin, and Pouzet (2021): Verified Lustre Normalization with Node Subsampling]

Ultimate goal: transform Lustre AST into NLustre AST

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \diamond\ e \\
& | & e \oplus e \\
& | & e^+ \ \texttt{fby}\ e^+ \\
& | & e^+ \ \texttt{when}\ x \\
& | & \texttt{merge(}\ x\ \texttt{;}\ e^+\ \texttt{;}\ e^+\ \texttt{)} \\
& | & \texttt{if}\ e\ \texttt{then}\ e^+\ \texttt{else}\ e^+ \\
& | & f\ \texttt{(}\ e^+\ \texttt{)} \\
eq & ::= & x^+ = e^+\ \texttt{;}
\end{array}
$$

Fig. 5. Lustre: equations

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \diamond\ e \\
& | & e \oplus e \\
& | & e\ \texttt{when}\ x \\
ce & ::= & e \\
& | & \texttt{merge(}\ x\ \texttt{;}\ ce\ \texttt{;}\ ce\ \texttt{)} \\
& | & \texttt{if}\ e\ \texttt{then}\ ce\ \texttt{else}\ ce \\
eq & ::= & x = ce \\
& | & x = c\ \texttt{fby}\ e \\
& | & x^+ = f\ \texttt{(}\ e^+\ \texttt{)}
\end{array}
$$

Fig. 6. NLustre: equations

$$
\begin{array}{rcl}
n & ::= & \texttt{node}\ f\ \texttt{(}\ d^+\ \texttt{)}\ \texttt{returns}\ \texttt{(}\ d^+\ \texttt{)} \\
& & (\texttt{var}\ d^+\ \texttt{;})^? \\
& & \texttt{let}\ eq^*\ \texttt{tel} \\
d & ::= & x^{ck}_{ty} \\
G & ::= & n^+
\end{array}
$$

Fig. 7. Nodes and Programs

NLustre

- Distinguish *expressions* (e) from *control expressions* (ce)
- Eliminate most expression lists (still allowed in node instances)
- Each fby in a separate equation and initialized by a constant
- Each node instance in a separate equation

N-Lustre: syntax

```
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.
```

le when x / le when not x

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

merge x ce1 ce2

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

$x = (ce)^{ck}$

$y,\ldots,y = (f(le, \ldots, le))^{ck}$

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

```
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_vout  : ∀ out, In out (map fst n_out) →
              ¬ In out (vars_defined (filter is_fby n_eqs));
    n_nodup : NoDupMembers (n_in ++ n_vars ++ n_out);
    n_good  : Forall NotReserved (n_in ++ n_vars ++ n_out)
  }.
```

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

Example: Rising-Edge Retrigger [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

"Every rising edge on input i, hold the output o true for n cycles."

Example: Rising-Edge Retrigger [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

"Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
let
  ...
tel
```

i	F	T	F	F	F	F	F	T	F	...
n	3	3	3	3	3	3	3	3	3	...

Example: Rising-Edge Retrigger

[Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

"Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge : bool;
let
  edge = i and (false fby (not i));
  ...
tel
```

i	F	T	F	F	F	F	F	T	F	⋯
n	3	3	3	3	3	3	3	3	3	⋯
edge	F	T	F	F	F	F	F	T	F	⋯

Example: Rising-Edge Retrigger [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

"Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge : bool; v : int;
let
  edge = i and (false fby (not i));
  v = /* count down from n whenever edge is true */
  o = v > 0;
tel
```

i	F	T	F	F	F	F	F	T	F	⋯
n	3	3	3	3	3	3	3	3	3	⋯
edge	F	T	F	F	F	F	F	T	F	⋯
v	0	3	2	1	0	0	0	3	2	⋯
o	F	T	T	T	F	F	F	T	T	⋯

Example: Rising-Edge Retrigger [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

"Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge : bool; v : int;
let
  edge = i and (false fby (not i));
  v = /* count down from n whenever edge is true */
  o = v > 0;
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n
        else (n fby (cpt − 1));
tel
```

i	F	T	F	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···
v	0	3	2	1	0	0	0	3	2	···
o	F	T	T	T	F	F	F	T	T	···

Example: Rising-Edge Retrigger

[Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

"Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = dcount((false, n) when ck);
  o = v > 0;
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n
        else (n fby (cpt − 1));
tel
```

i	F	T	F	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···
v	0	3	2	1	0	0	0	3	2	···
ck	F	T	T	T	T	F	F	T	T	···
n when ck		3	3	3	3			3	3	···
dcount(n when ck)		3	2	1	0			-1	-2	···
o	F	T	T	T	F	F	F	T	T	···

Example: Rising-Edge Retrigger [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

"Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck
        (true -> dcount((false, n) when ck))
        (false -> 0);
  o = v > 0;
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n
        else (n fby (cpt − 1));
tel
```

i	F	T	F	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···
v	0	3	2	1	0	0	0	3	2	···
ck	F	T	T	T	T	F	F	T	T	···
n when ck		3	3	3	3			3	3	···
dcount(n when ck)		3	2	1	0			-1	-2	···
o	F	T	T	T	F	F	F	T	T	···

Example: Rising-Edge Retrigger [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

"Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck
        (true -> dcount((edge, n) when ck))
        (false -> 0);
  o = v > 0;
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n
        else (n fby (cpt − 1));
tel
```

i	F	T	F	F	F	F	F	T	F	⋯
n	3	3	3	3	3	3	3	3	3	⋯
edge	F	T	F	F	F	F	F	T	F	⋯
v	0	3	2	1	0	0	0	3	2	⋯
ck	F	T	T	T	T	F	F	T	T	⋯
n when ck		3	3	3	3			3	3	⋯
dcount(n when ck)		3	2	1	0			3	2	⋯
o	F	T	T	T	F	F	F	T	T	⋯

Example: Rising-Edge Retrigger

[Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n else (n fby (cpt − 1));
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
var norm1$1, norm2$2 : int; norm2$1 : bool;
let
  norm2$1 = true fby false;
  norm2$2 = 0 fby (cpt − 1);
  norm1$1 = if norm2$1 then n else norm2$2;
  cpt = if res then n else norm1$1;
tel
```

Example: Rising-Edge Retrigger

[Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n else (n fby (cpt − 1));
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
var norm1$1, norm2$2 : int; norm2$1 : bool;
let
  norm2$1 = true fby false;
  norm2$2 = 0 fby (cpt − 1);
  norm1$1 = if norm2$1 then n else norm2$2;
  cpt = if res then n else norm1$1;
tel
```

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck
        (true −> dcount((edge, n) when ck))
        (false −> 0);
  o = v > 0;
tel
```

```
node rising_edge_retrigger (i : bool; n : int)
returns (o : bool)
var edge, ck, norm1$1, norm1$2 : bool;
    v : int; elab$4 : int when ck;
let
  norm1$2 = false fby (not i);
  edge = i and norm1$2;
  norm1$1 = false fby o;
  ck = edge or norm1$1;
  elab$4 = dcount(edge when ck, n when ck);
  v = merge(ck; elab$4; 0 when not ck);
  o = v > 0;
tel
```

Unnesting and distribution: selected cases

$$
\begin{aligned}
\lfloor \mathsf{c} \rfloor &= ([\mathsf{c}], [\,]) \\
\lfloor \mathsf{x} \rfloor &= ([\mathsf{x}], [\,]) \\
\lfloor e_1 \oplus e_2 \rfloor &= ([e_1'], \boldsymbol{eqs}_1') \leftarrow \lfloor e_1 \rfloor \\
&\quad\ ([e_2'], \boldsymbol{eqs}_2') \leftarrow \lfloor e_2 \rfloor \\
&\quad\ ([e_1' \oplus e_2'], \boldsymbol{eqs}_1' \ \cup\ \boldsymbol{eqs}_2') \\
\lfloor (e_1, \ldots, e_n) \text{ when } b \rfloor &= ([e_1', \ldots, e_m'], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([e_1' \text{ when } b, \ldots, e_m' \text{ when } b], \boldsymbol{eqs}') \\
\lfloor (e_1, \ldots, e_n) \text{ fby } (f_1, \ldots, f_m) \rfloor &= ([e_1', \ldots, e_k'], \boldsymbol{eqs}_1') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([f_1', \ldots, f_k'], \boldsymbol{eqs}_2') \leftarrow \lfloor f_1, \ldots, f_m \rfloor \\
&\quad\ ([x_1, \ldots, x_k], [x_1 = e_1' \text{ fby } f_1', \ldots, x_k = e_k' \text{ fby } f_k'] \ \cup\ \boldsymbol{eqs}_1' \ \cup\ \boldsymbol{eqs}_2') \\
\lfloor f(e_1, \ldots, e_n) \rfloor &= ([e_1', \ldots, e_m'], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([x_1, \ldots, x_k], [(x_1, \ldots x_k) = f(e_1', \ldots, e_m')] \ \cup\ \boldsymbol{eqs}')
\end{aligned}
$$

Expression initialization

$$\left\lfloor \mathrm{x} = (\mathrm{e0}\ \mathbf{fby}\ \mathrm{e})^{ck} \right\rfloor_{\mathrm{fby}} = \begin{cases} \mathrm{x} = \mathbf{if}\ \mathrm{xinit}\ \mathbf{then}\ \mathrm{e0}\ \mathbf{else}\ \mathrm{px}; \\ \mathrm{xinit} = \mathrm{true}^{ck}\ \mathbf{fby}\ \mathrm{false}^{ck}; \\ \mathrm{px} = \mathrm{def}_{ty}^{ck}\ \mathbf{fby}\ \mathrm{e}; \end{cases}$$

Transcription

- Convert between normalized Lustre AST and NLustre AST
- Difficulty: alignment clause in NLustre semantics

Indexed or coinductive: fby

```
CoFixpoint fby (c: val) (xs: Stream value) : Stream value :=
  match xs with
  | absent ::: xs ⇒ absent ::: fby c xs
  | present x ::: xs ⇒ present c ::: fby x xs
  end.
```

Indexed or coinductive: fby

```
CoFixpoint fby (c: val) (xs: Stream value) : Stream value :=
  match xs with
  | absent ::: xs ⇒ absent ::: fby c xs
  | present x ::: xs ⇒ present c ::: fby x xs
  end.


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

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

Indexed or coinductive (NLustre)

Indexed streams (nat ↦ value)

```
fun n ⇒ n
```

Coinductive streams

```
(cofix f n := n ::: f (S n)) 0
```

Indexed or coinductive (NLustre)

Indexed streams (nat ↦ value)

Coinductive streams

```
CoFixpoint idx_to_coind (n: nat) (xs: nat → A) : Stream A :=
  xs n ::: idx_to_coind (S n) xs.
```

```
fun n ⇒ n
```

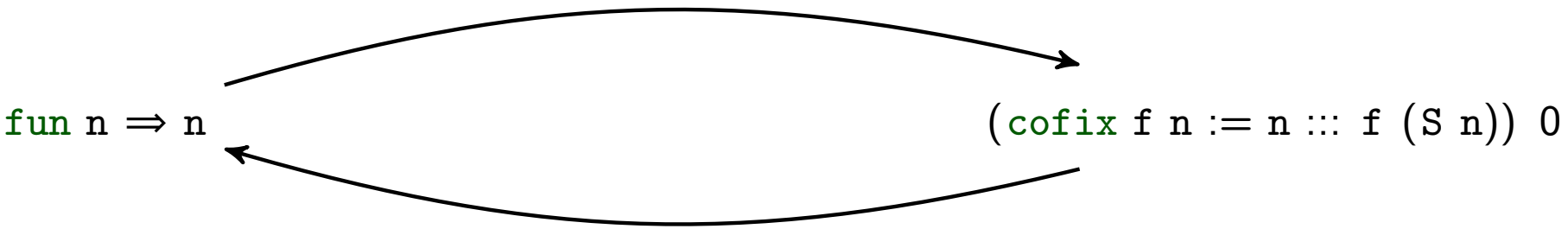
```
(cofix f n := n ::: f (S n)) 0
```

Indexed or coinductive (NLustre)

Indexed streams (nat ↦ value)

Coinductive streams

```
CoFixpoint idx_to_coind (n: nat) (xs: nat → A) : Stream A :=
  xs n ::: idx_to_coind (S n) xs.
```



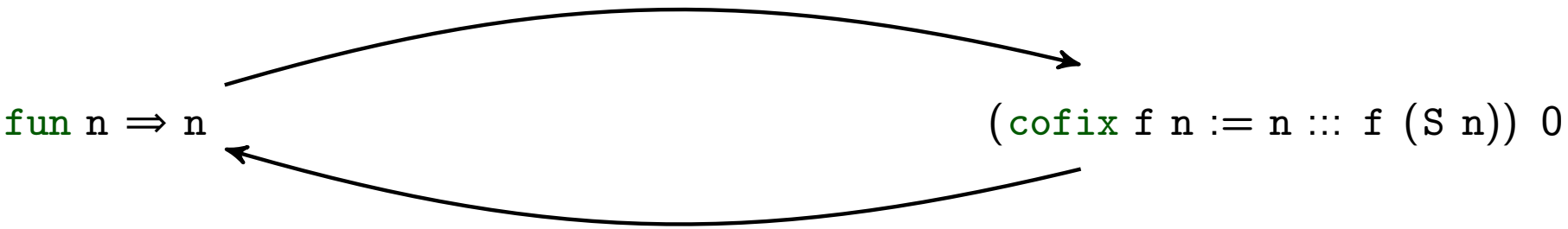

```
Definition coind_to_idx (xs: Stream A) : nat → A :=
  fun n ⇒ hd (Str_nth_tl n xs).
```

Indexed or coinductive (NLustre)

Indexed streams (nat ↦ value)

Coinductive streams

```
CoFixpoint idx_to_coind (n: nat) (xs: nat → A) : Stream A :=
  xs n ::: idx_to_coind (S n) xs.
```




```
Definition coind_to_idx (xs: Stream A) : nat → A :=
  fun n ⇒ hd (Str_nth_tl n xs).
```

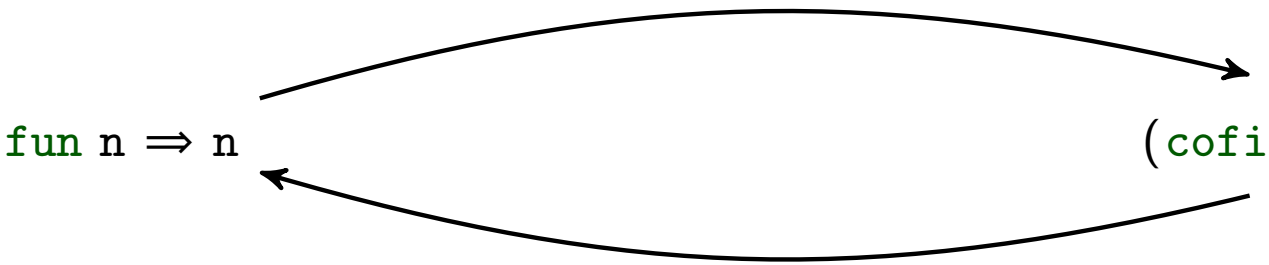
```
idx_to_coind n (Idx.fby k xs)
== CoInd.fby (Idx.hold k xs n) (idx_to_coind n xs).
```

```
coind_to_idx (CoInd.fby k xs) =*= Idx.fby c (coind_to_idx xs).
```

Indexed or coinductive (NLustre)

Indexed streams (nat ↦ value) | Coinductive streams

```
CoFixpoint idx_to_coind (n: nat) (xs: nat → A) : Stream A :=
  xs n ::: idx_to_coind (S n) xs.
```




```
Definition coind_to_idx (xs: Stream A) : nat → A :=
  fun n ⇒ hd (Str_nth_tl n xs).
```

```
idx_to_coind n (Idx.fby k xs)
== CoInd.fby (Idx.hold k xs n) (idx_to_coind n xs).
```

```
coind_to_idx (CoInd.fby k xs) =*= Idx.fby c (coind_to_idx xs).
```

Extends to NLustre semantics (proof: L. Brun)

N-Lustre: semantics 1/3

```
Notation stream A := (nat → A).

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

Section InstantSemantics.

  Variable base : bool.
  Variable R : R.

  Inductive sem_var_instant (x: ident) (v: value): Prop :=
  | Sv:
      PM.find x R = Some v →
      sem_var_instant x v.

End InstantSemantics.

Section LiftSemantics.

  Variable bk : stream bool.

  Definition restr H (n: nat): R :=
    PM.map (fun xs ⇒ xs n) H.

  Definition lift (sem: bool → R → A → B → Prop) H x (ys: stream B): Prop :=
    ∀ n, sem (bk n) (restr H n) x (ys n).

  Definition sem_var H (x: ident)(xs: stream value): Prop :=
    lift (fun base ⇒ sem_var_instant) H x xs.

End LiftSemantics.
```

N-Lustre: semantics 2/3

```
Inductive sem_lexp_instant: lexp → value → Prop :=
| Sconst:
    v = (if base then present (sem_const c) else absent) →
    sem_lexp_instant (Econst c) v

| Svar:
    sem_var_instant x v →
    sem_lexp_instant (Evar x ty) v

| Swhen_abs:
    sem_lexp_instant s absent →
    sem_var_instant x absent →
    sem_lexp_instant (Ewhen s x b) absent

| Swhen_eq:
    sem_lexp_instant s (present sc) →
    sem_var_instant x (present xc) →
    val_to_bool xc = Some b →
    sem_lexp_instant (Ewhen s x b) (present sc)

| Swhen_abs1:
    sem_lexp_instant s (present sc) →
    sem_var_instant x (present xc) →
    val_to_bool xc = Some b →
    sem_lexp_instant (Ewhen s x (negb b)) absent

...

Definition sem_lexp H (e: lexp) (xs: stream value) : Prop :=
  lift sem_lexp_instant H e xs.
```

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

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

[Colaço and Pouzet (2003): Clocks as First Class Abstract Types]

N-Lustre: semantics 2/3

```
Inductive sem_lexp_instant: lexp → value → Prop :=
| (Sconst:)
    v = (if base then present (sem_const c) else absent) →
    sem_lexp_instant (Econst c) v

| (Svar:)
    sem_var_instant x v →
    sem_lexp_instant (Evar x ty) v

| (Swhen_abs:)
    sem_lexp_instant s absent →
    sem_var_instant x absent →
    sem_lexp_instant (Ewhen s x b) absent

| (Swhen_eq:)
    sem_lexp_instant s (present sc) →
    sem_var_instant x (present xc) →
    val_to_bool xc = Some b →
    sem_lexp_instant (Ewhen s x b) (present sc)

| (Swhen_abs1:)
    sem_lexp_instant s (present sc) →
    sem_var_instant x (present xc) →
    val_to_bool xc = Some b →
    sem_lexp_instant (Ewhen s x (negb b)) absent

...

Definition sem_lexp H (e: lexp) (xs: stream value) : Prop :=
  lift sem_lexp_instant H e xs.
```

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

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

[Colaço and Pouzet (2003): Clocks as First Class Abstract Types]

N-Lustre: semantics 3/3

```
Inductive sem_equation G : stream bool → history → equation → Prop :=
| (SEqDef:)
    sem_var bk H x xs →
    sem_caexp bk H ck ce xs →
    ──────────────────────────
    sem_equation G bk H (EqDef x ck ce)

| (SEqApp:)
    sem_lexps bk H arg ls →
    sem_vars bk H x xs →
    sem_clock bk H ck cks →
    clock_of ls cks →
    sem_node G f ls xs →
    ──────────────────────────
    sem_equation G bk H (EqApp x ck f arg)

| (SEqFby:)
    sem_laexp bk H ck le ls →
    sem_var bk H x xs →
    (∀ n, xs n = fby (sem_const c0) ls n) →
    ──────────────────────────
    sem_equation G bk H (EqFby x ck c0 le)

with sem_node G :
  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.
```

$f : \mathrm{stream}(T^+) \rightarrow \mathrm{stream}(T^+)$

```
Inductive sem_laexp_instant
    : clock → lexp → value → Prop:=
| (SLtick:)
    sem_lexp_instant ce (present c) →
    sem_clock_instant ck true →
    ──────────────────────────
    sem_laexp_instant ck ce (present c)
| (SLabs:)
    sem_clock_instant ck false →
    ──────────────────────────
    sem_laexp_instant ck ce absent.
```

$$\mathrm{fby}^{\#}_{v_0}(v.s) = v_0.\mathrm{fby}^{\#}_{v}(s)$$

$$\mathrm{fby}^{\#}_{v_0}(\mathit{abs}.s) = \mathit{abs}.\mathrm{fby}^{\#}_{v_0}(s)$$

$$\mathrm{fby}^{\#}_{v_0}(\epsilon) = \epsilon$$

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

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

Overview

Lustre: syntax and semantics

Lustre: normalization

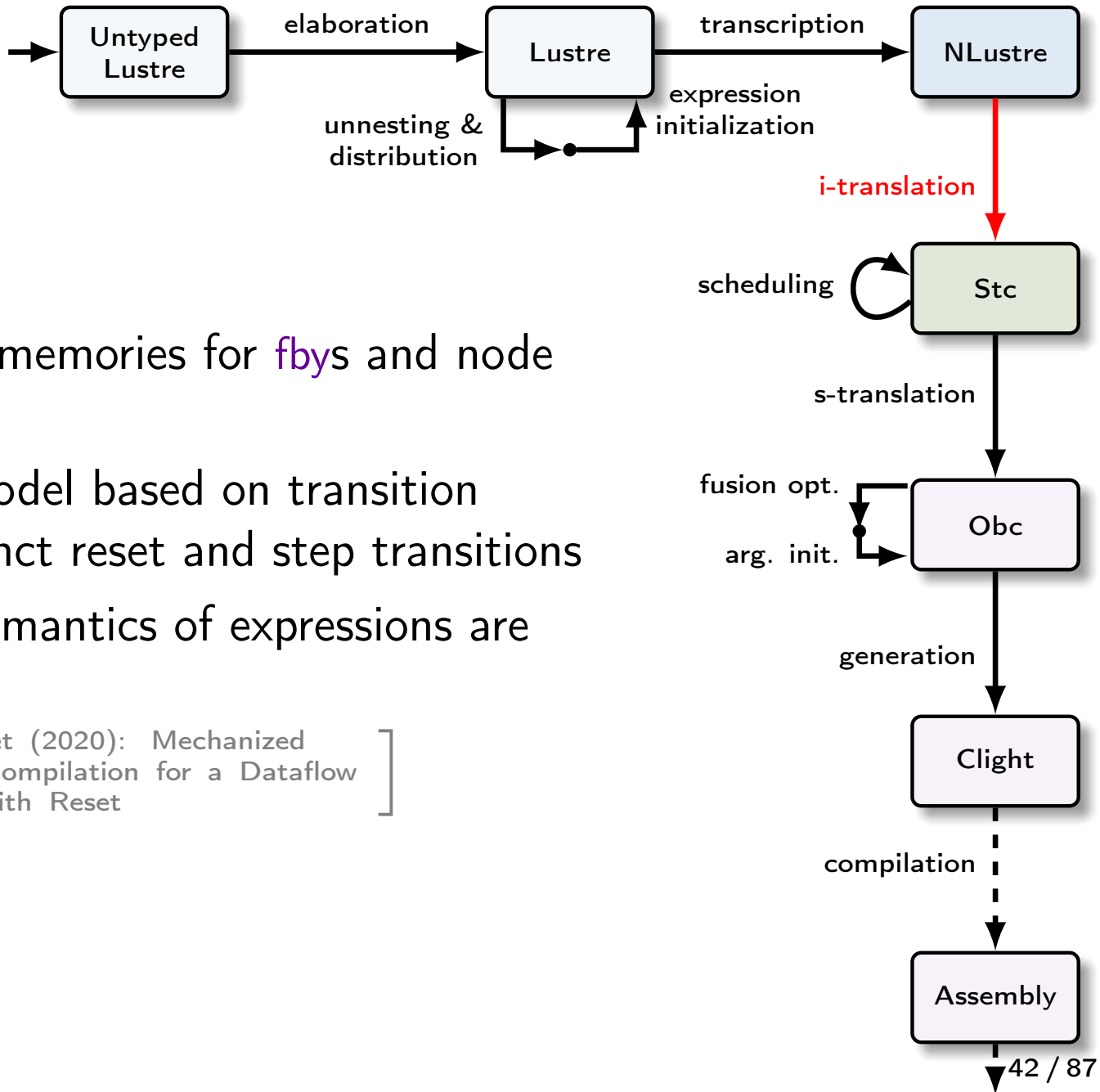
Translation: from dataflow programs to imperative code

Optimization: control structure fusion

Generation: from Obc to Clight

Conclusion

i-translation: NLustre to Stc



- Introduce explicit memories for fbys and node instances
- Use a semantic model based on transition systems with distinct reset and step transitions
- The syntax and semantics of expressions are unchanged
- [Bourke, Brun, and Pouzet (2020): Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset]

Naive compilation of modular reset

```
y = (restart f every r)(x)
```

```
if (ck_r) {
  if (r) { f(y).reset() };
};
y := f(y).step(x)
```

Naive compilation of modular reset

```
y = (restart f every r)(x)
```

```
if (ck_r) {
  if (r) { f(y).reset() };
};
y := f(y).step(x)
```

Problem: interferes with the fusion optimization

```
node main(x0, x': int; ck, r: bool)
  returns (x: int) var v, w: int when ck;
let
  v = filter(x' when ck);
  w = euler((x0, v) when ck) every r;
  x = merge ck w 0;
tel
```

```
step(x0, x': int; ck, r: bool)
returns (x: int);
var v, w : int;
{
  if (ck) { v := filter(v).step(x') };
  if (r) { euler(w).reset() };
  if (ck) { w := euler(w).step(x0, v) };
  if (ck) { x := w } else { x := 0 }
}
```

Naive compilation of modular reset

```
y = (restart f every r)(x)
```

```
if (ck_r) {
  if (r) { f(y).reset() };
};
y := f(y).step(x)
```

Solution: a new intermediate language (Stc)

```
node main(x0, x': int; ck, r: bool)
  returns (x: int) var v, w: int when ck;
let
  v = filter(x' when ck);
  w = euler((x0, v) when ck) every r;
  x = merge ck w 0;
tel
```

```
step(x0, x': int; ck, r: bool)
returns (x: int);
var v, w : int;
{
  if (r) { euler(w).reset() };
  if (ck) {
    v := filter(v).step(x');
    w := euler(w).step(x0, v);
    x := w
  } else { x := 0 }
}
```

Stc: expose node instances, separate reset and step

$$
\begin{array}{rcl}
tc & ::= & x =_{ck} ce \\
& | & next\ x =_{ck} e \\
& | & reset\ f\langle x \rangle\ every\ ck \\
& | & x, \ldots, x =_{ck} f\langle x, k \rangle(e, \ldots, e)
\end{array}
$$

- No `fby`; use `next` to constrains successor state.
- Distinguish `reset` and `step` constraints.
- Associate each with an instance ($\langle x \rangle$).
- For `step` constraints, k is 1 if there is a `reset` on the same instance and 0 otherwise.

Make states explicit, separate reset and step transitions

```
node rising_edge_retrigger (i : bool; n : int32)
returns (o : bool)
var edge, ck, v, norm1$1, norm1$2 : bool;
    elab$3 : int32 when ck;
let
  norm1$2 = 0 fby not i;
  edge = i and norm1$2;
  norm1$1 = 0 fby o;
  ck = edge or norm1$1;
  elab$3 = (restart dcount every edge)(n when ck);
  v = merge ck
        (true -> elab$3)
        (false -> 0 when not ck);
  o = v > 0;
tel
```

Make states explicit, separate reset and step transitions

```
node rising_edge_retrigger (i : bool; n : int32)
returns (o : bool)
var edge, ck, v, norm1$1, norm1$2 : bool;
    elab$3 : int32 when ck;
let
  norm1$2 = 0 fby not i;
  edge = i and norm1$2;
  norm1$1 = 0 fby o;
  ck = edge or norm1$1;
  elab$3 = (restart dcount every edge)(n when ck);
  v = merge ck
        (true -> elab$3)
        (false -> 0 when not ck);
  o = v > 0;
tel
```

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    next norm1$2 = not i;
    edge = i and norm1$2;
    next norm1$1 = o;
    ck = edge or norm1$1;
    elab$3 = dcount<elab$3>(n when ck);
    reset(dcount<elab$3>) every (. on edge);
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck);
    o = v > 0;
  }
}
```

Stc: new intermediate language

- Explicit states
- Distinct *step* and *reset* constraints
- Constraint ordering is unimportant

Make states explicit, separate reset and step transitions

```
node rising_edge_retrigger (i : bool; n : int32)
returns (o : bool)
var edge, ck, v, norm1$1, norm1$2 : bool;
    elab$3 : int32 when ck;
let
  norm1$2 = 0 fby not i;
  edge = i and norm1$2;
  norm1$1 = 0 fby o;
  ck = edge or norm1$1;
  elab$3 = (restart dcount every edge)(n when ck);
  v = merge ck
        (true -> elab$3)
        (false -> 0 when not ck);
  o = v > 0;
tel
```

i-translation

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    next norm1$2 = not i;
    edge = i and norm1$2;
    next norm1$1 = o;
    ck = edge or norm1$1;
    elab$3 = dcount<elab$3>(n when ck);
    reset(dcount<elab$3>) every (. on edge);
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck);
    o = v > 0;
  }
}
```

Stc: new intermediate language

- Explicit states
- Distinct *step* and *reset* constraints
- Constraint ordering is unimportant

Make states explicit, separate reset and step transitions

```
node rising_edge_retrigger (i : bool; n : int32)
returns (o : bool)
var edge, ck, v, norm1$1, norm1$2 : bool;
    elab$3 : int32 when ck;
let
  norm1$2 = 0 fby not i;
  edge = i and norm1$2;
  norm1$1 = 0 fby o;
  ck = edge or norm1$1;
  elab$3 = (restart dcount every edge)(n when ck);
  v = merge ck
        (true -> elab$3)
        (false -> 0 when not ck);
  o = v > 0;
tel
```

i-translation

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

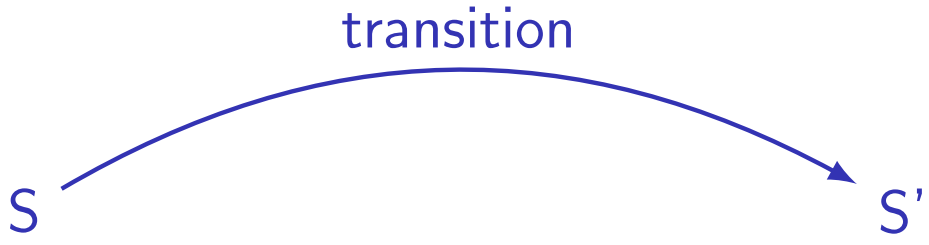
  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    next norm1$2 = not i;
    edge = i and norm1$2;
    next norm1$1 = o;
    ck = edge or norm1$1;
    elab$3 = dcount<elab$3>(n when ck);
    reset(dcount<elab$3>) every (. on edge);
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck);
    o = v > 0;
  }
}
```

Stc: new intermediate language

- Explicit states
- Distinct *step* and *reset* constraints
- Constraint ordering is unimportant

Stc: Synchronous Transition Code

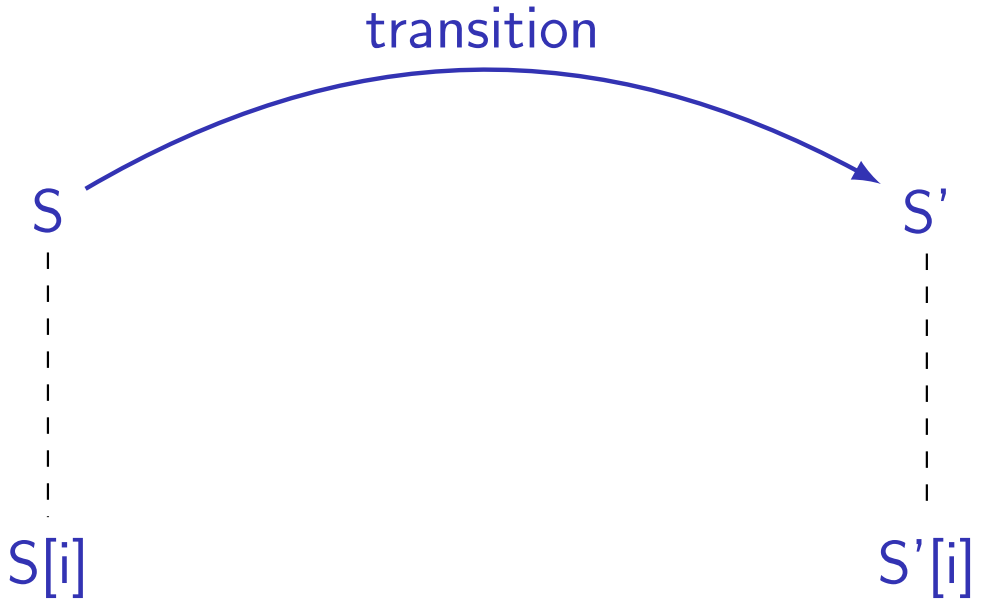
- Transition system: relation between states S and S'.
- next x = y + 1 means $S'[x] = S[y] + 1$.
- reset(dcount<i>) every r and dcount<i> constrain subtrees of S and S'.
- Need to introduce an intermediate state.
- And so on, recursively.
- 'Hiding' the intermediate states is key for inductive proofs.
- Composing transitions gives the columns of the dataflow semantics.



$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \cdots$

Stc: Synchronous Transition Code

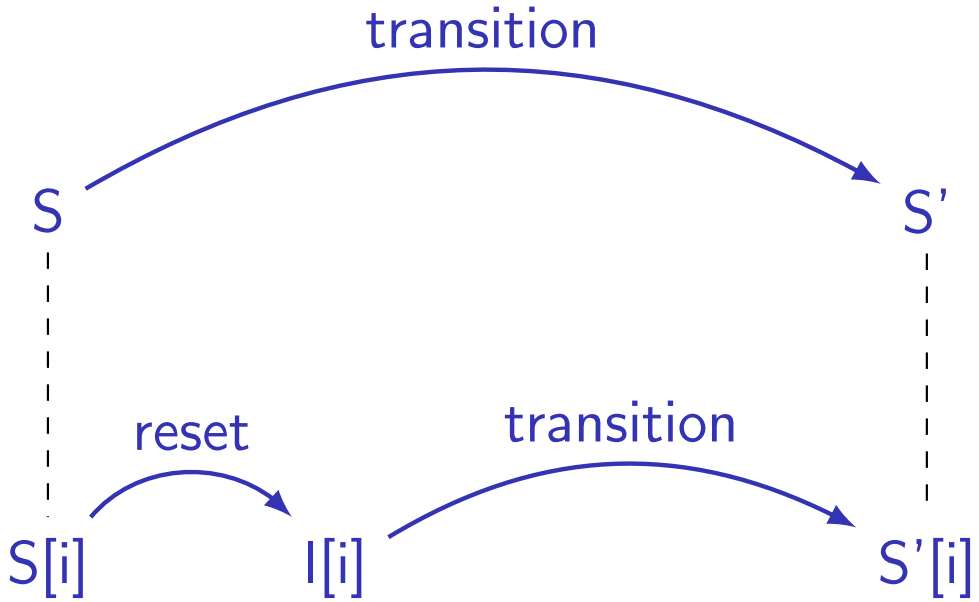
- Transition system: relation between states S and S'.
- next x = y + 1 means $S'[x] = S[y] + 1$.
- reset(dcount<i>) every r and dcount<i> constrain subtrees of S and S'.
- Need to introduce an intermediate state.
- And so on, recursively.
- 'Hiding' the intermediate states is key for inductive proofs.
- Composing transitions gives the columns of the dataflow semantics.



$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \cdots$

Stc: Synchronous Transition Code

- Transition system: relation between states S and S'.
- next x = y + 1 means $S'[x] = S[y] + 1$.
- reset(dcount<i>) every r and dcount<i> constrain subtrees of S and S'.
- Need to introduce an intermediate state.
- And so on, recursively.
- 'Hiding' the intermediate states is key for inductive proofs.
- Composing transitions gives the columns of the dataflow semantics.

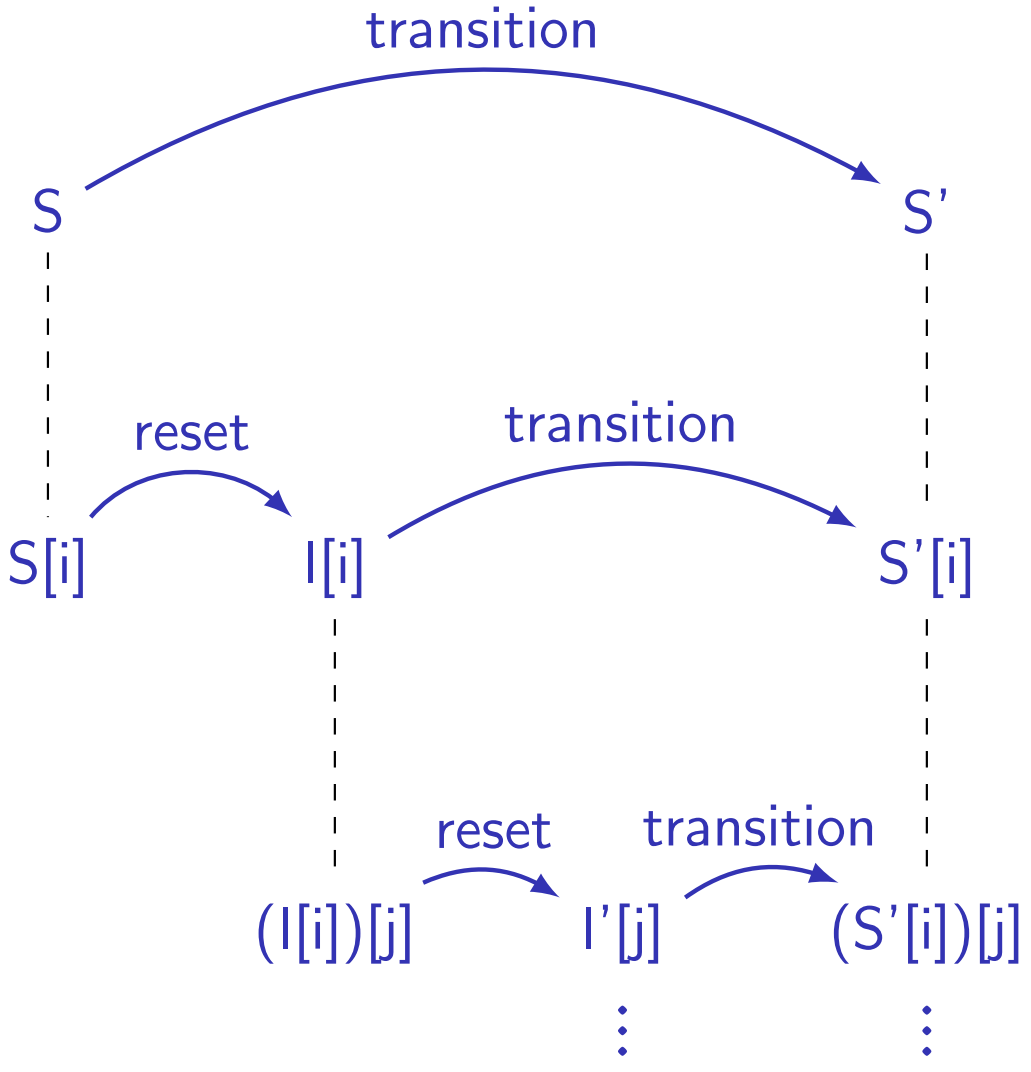


$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \cdots$

Stc: Synchronous Transition Code

- Transition system: relation between states S and S'.
- next x = y + 1 means $S'[x] = S[y] + 1$.
- reset(dcount<i>) every r and dcount<i> constrain subtrees of S and S'.
- Need to introduce an intermediate state.
- And so on, recursively.
- 'Hiding' the intermediate states is key for inductive proofs.
- Composing transitions gives the columns of the dataflow semantics.

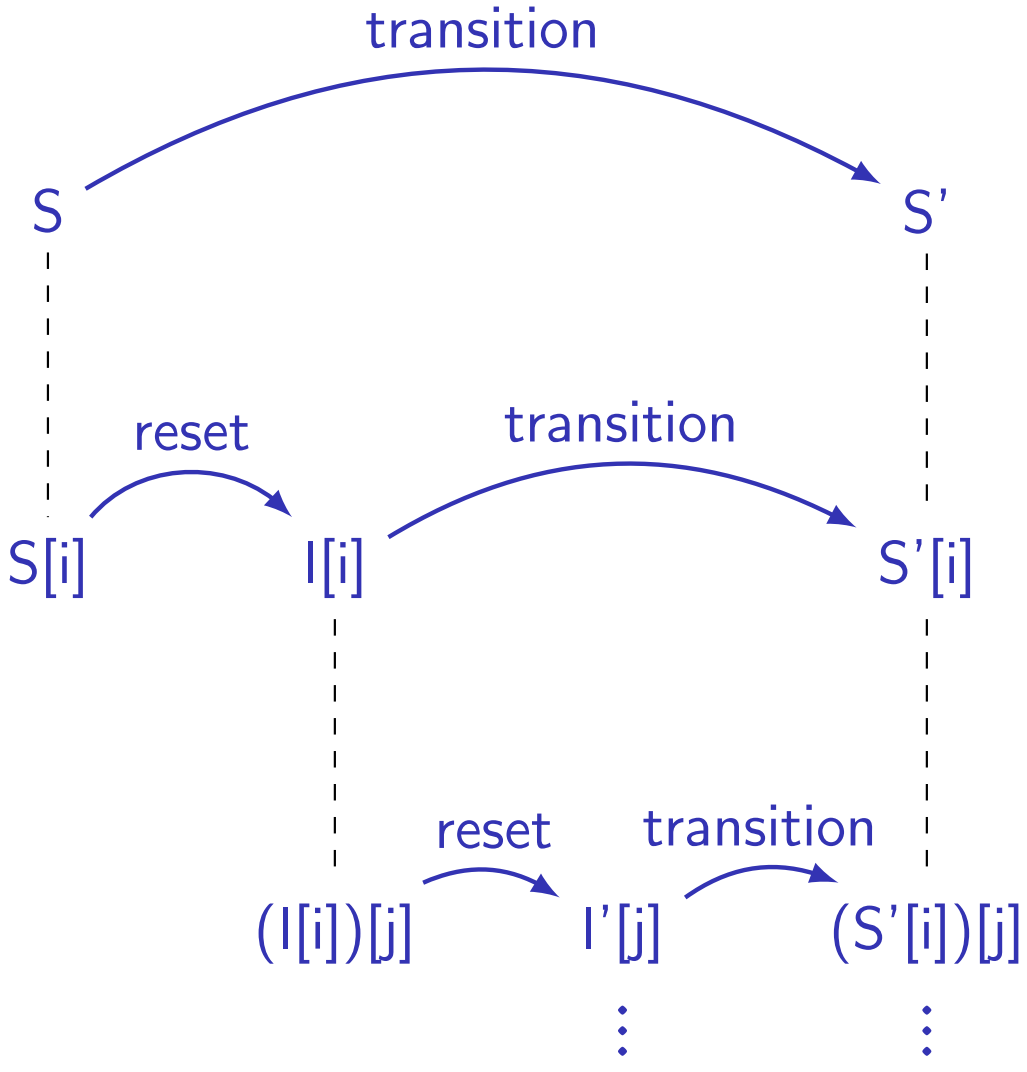
$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \cdots$



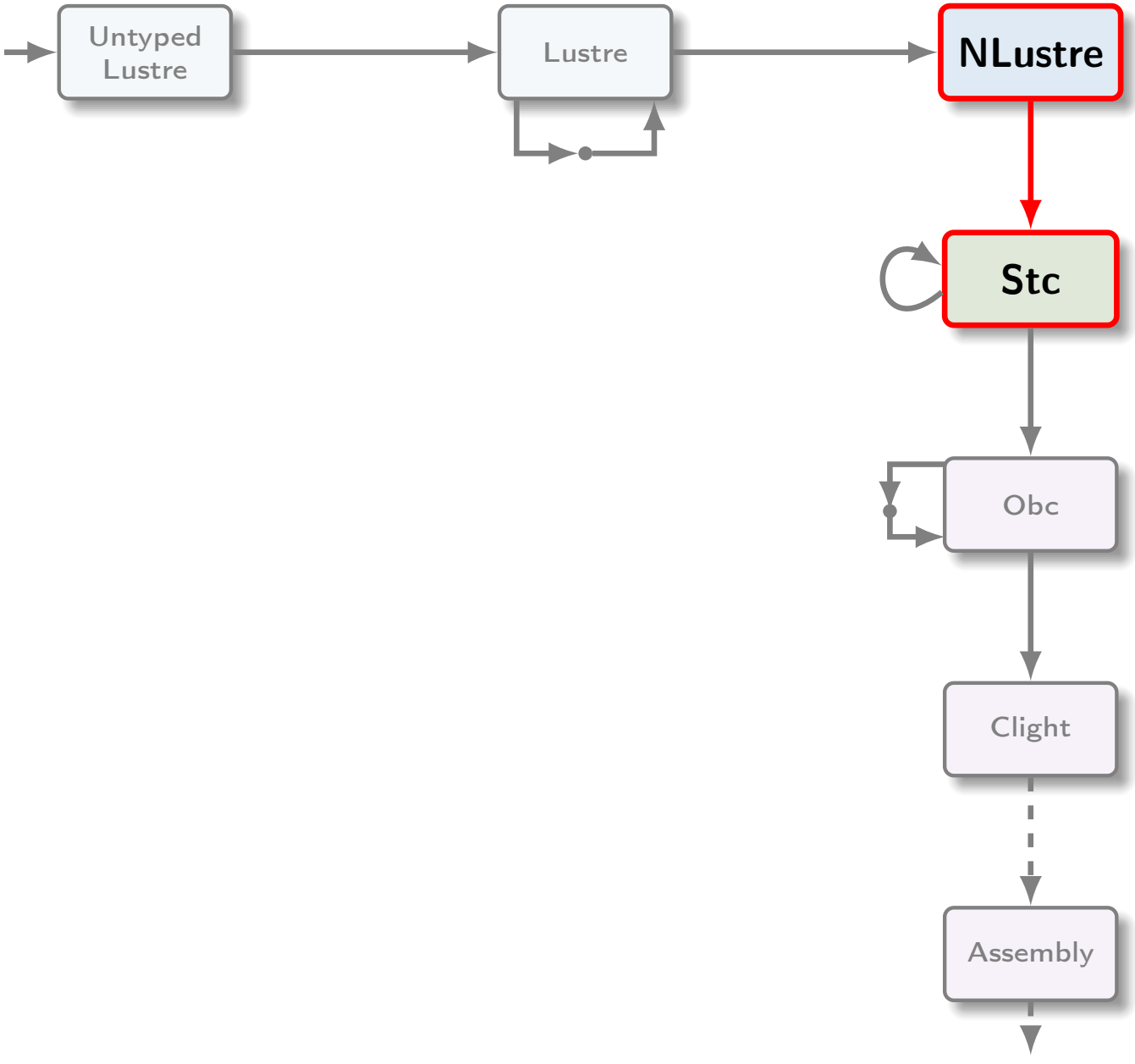
Stc: Synchronous Transition Code

- Transition system: relation between states S and S'.
- next x = y + 1 means $S'[x] = S[y] + 1$.
- reset(dcount<i>) every r and dcount<i> constrain subtrees of S and S'.
- Need to introduce an intermediate state.
- And so on, recursively.
- 'Hiding' the intermediate states is key for inductive proofs.
- Composing transitions gives the columns of the dataflow semantics.

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \cdots$



Correctness of NLustre to Stc translation

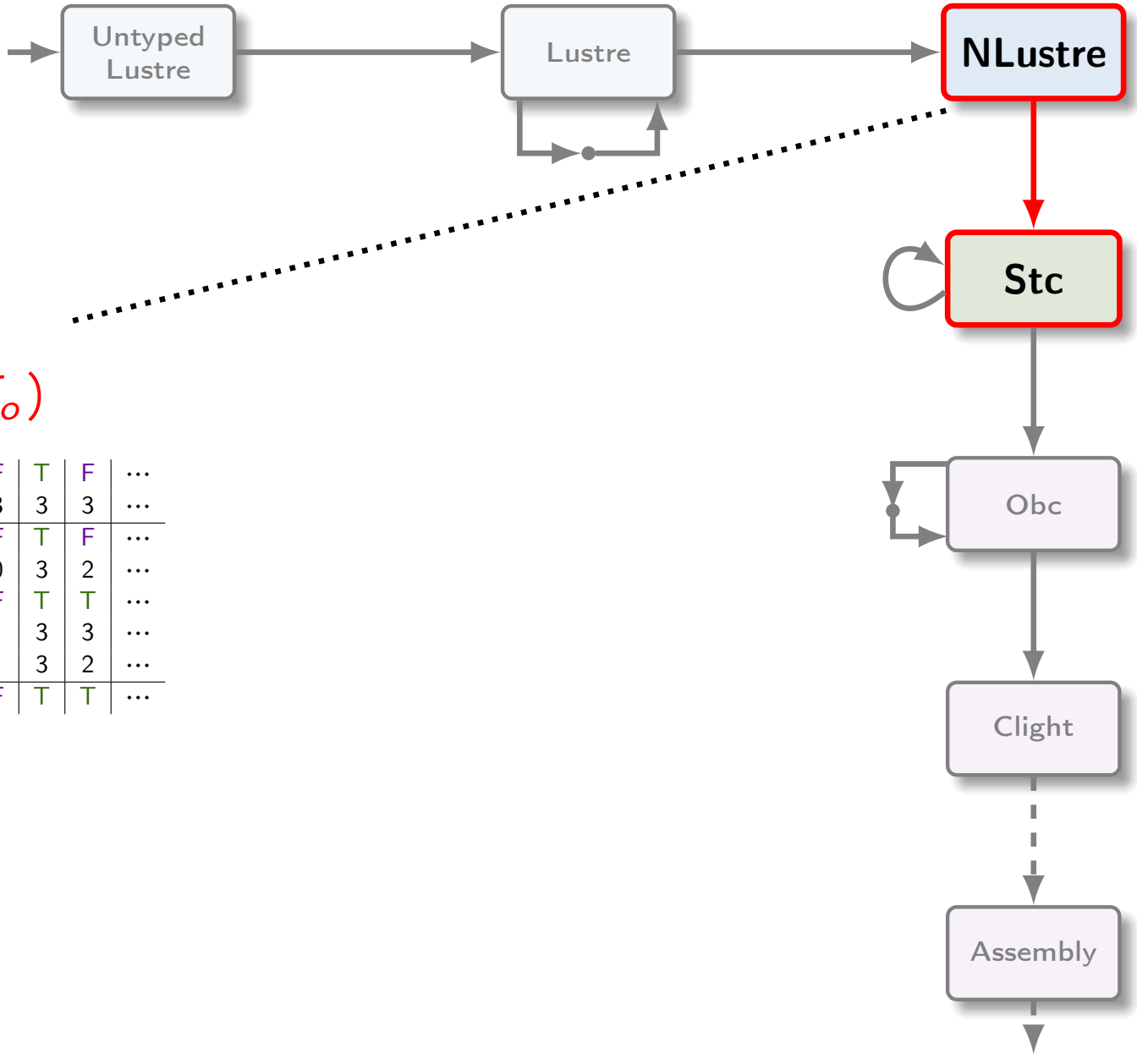


Correctness of NLustre to Stc translation

```
sem_node G f xss yss
```

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

i	F	T	F	F	F	F	F	T	F	⋯
n	3	3	3	3	3	3	3	3	3	⋯
edge	F	T	F	F	F	F	F	T	F	⋯
v	0	3	2	1	0	0	0	3	2	⋯
ck	F	T	T	T	T	F	F	T	T	⋯
n when ck		3	3	3	3			3	3	⋯
dcount(n when ck)		3	2	1	0			3	2	⋯
o	F	T	T	T	F	F	F	T	T	⋯

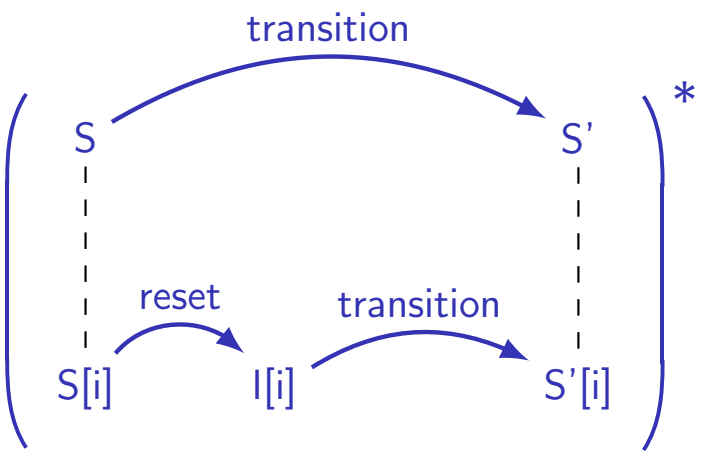


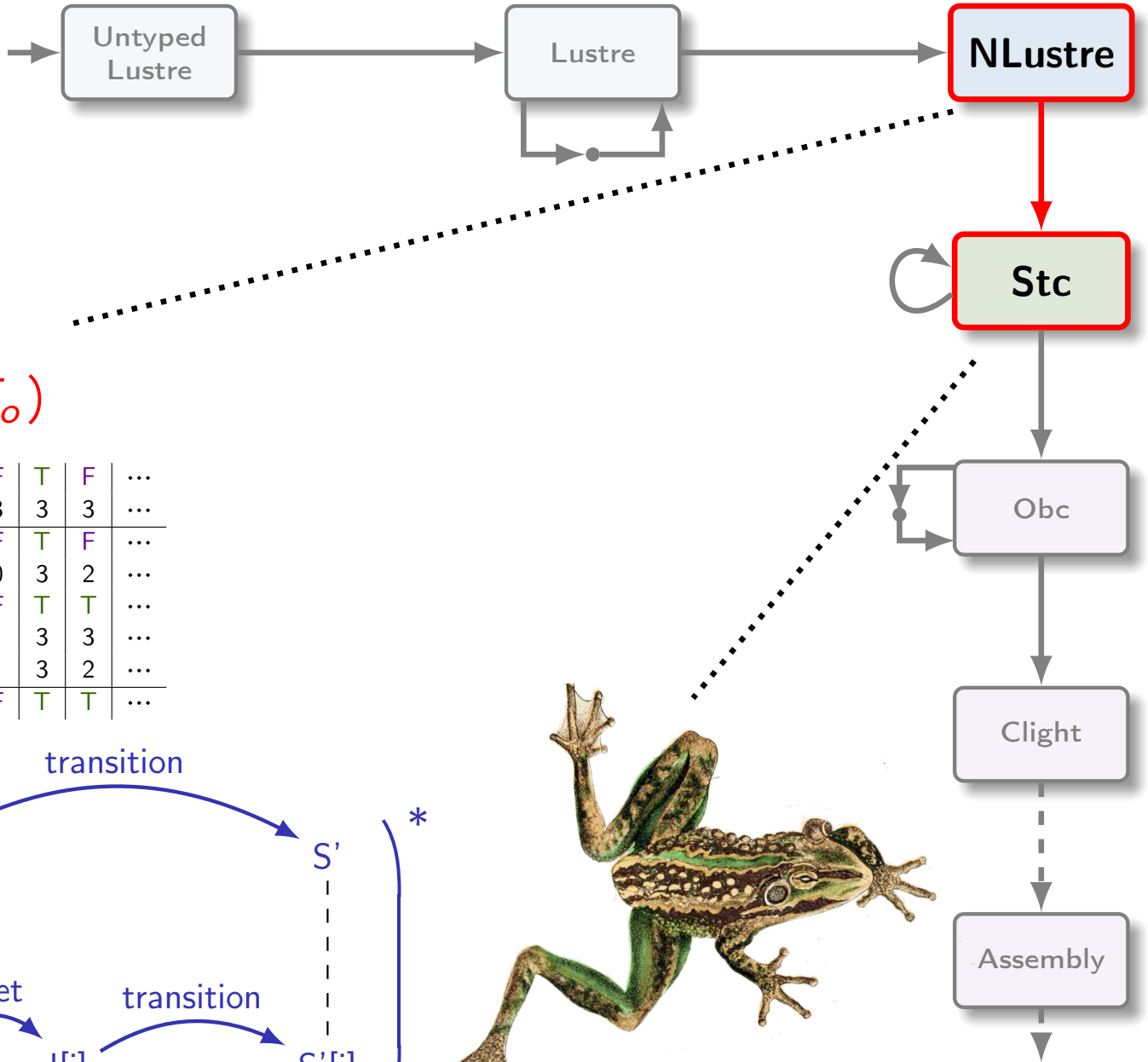
Correctness of NLustre to Stc translation

```
sem_node G f xss yss
```

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

i	F	T	F	F	F	F	F	T	F	⋯
n	3	3	3	3	3	3	3	3	3	⋯
edge	F	T	F	F	F	F	F	T	F	⋯
v	0	3	2	1	0	0	0	3	2	⋯
ck	F	T	T	T	T	F	F	T	T	⋯
n when ck		3	3	3	3			3	3	⋯
dcount(n when ck)		3	2	1	0			3	2	⋯
o	F	T	T	T	F	F	F	T	T	⋯



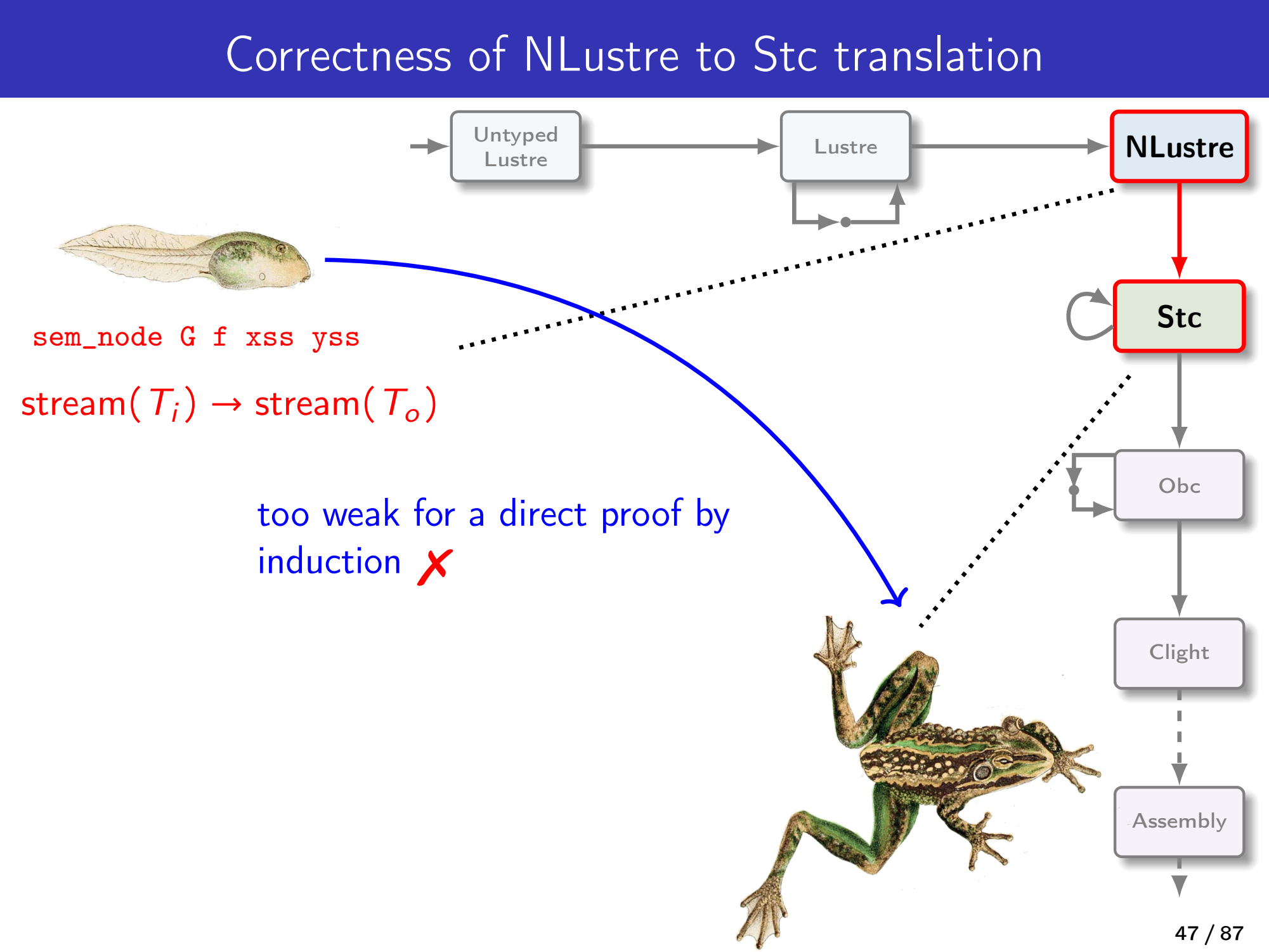


Correctness of NLustre to Stc translation


```
sem_node G f xss yss
```

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

too weak for a direct proof by induction ✗

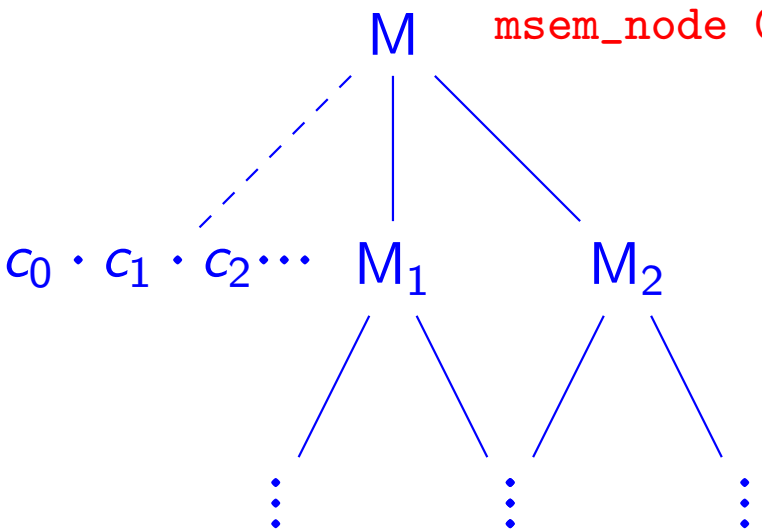


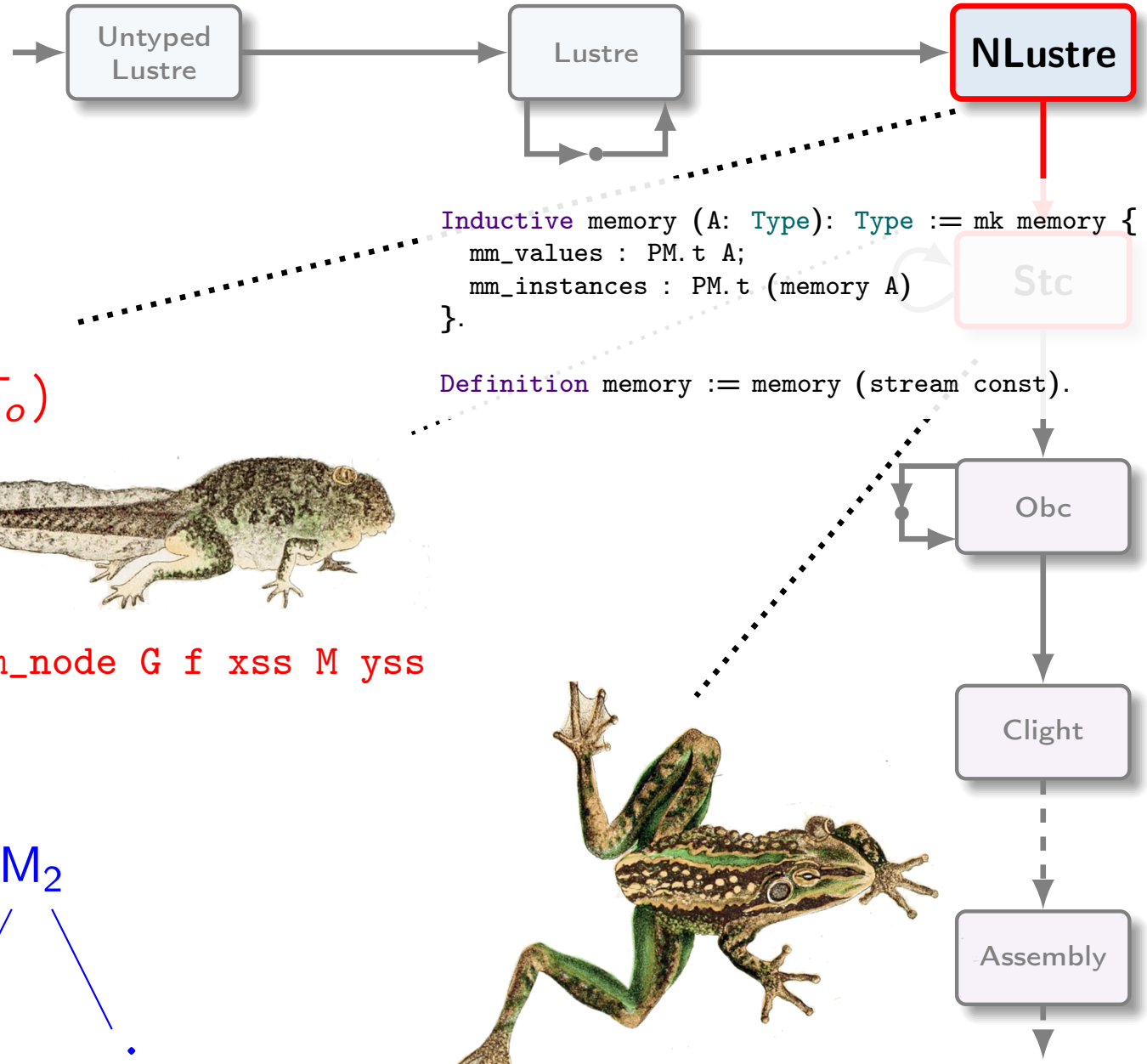
Correctness of NLustre to Stc translation

sem_node G f xss yss

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

msem_node G f xss M yss





Correctness of NLustre to Stc translation

`sem_node G f xss yss`

stream(T_i) → stream(T_o)

`msem_node G f xss M yss`

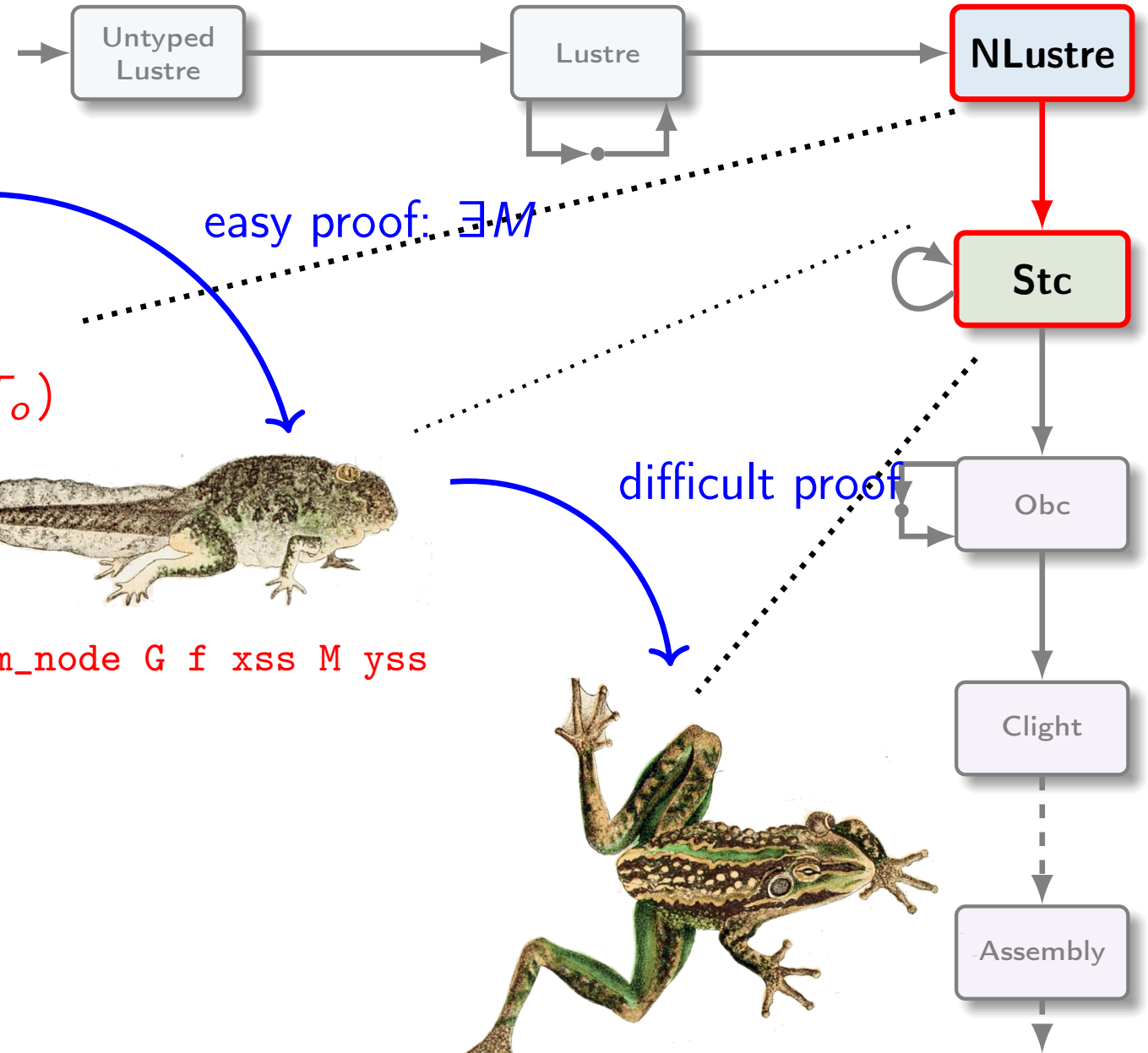
easy proof: $\exists M$

Correctness of NLustre to Stc translation

`sem_node G f xss yss`

stream(T_i) → stream(T_o)

`msem_node G f xss M yss`



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

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

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

...

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

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

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

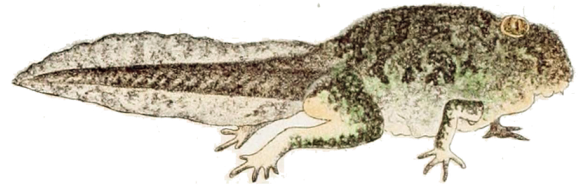
| (SEqDef:)
    sem_var H x xs →
    sem_caexp H cae xs →
    ------------------------
    sem_equation G H (EqDef x cae)
```

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

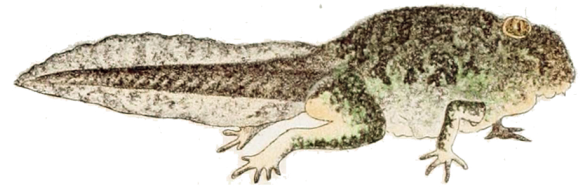
...

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

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



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



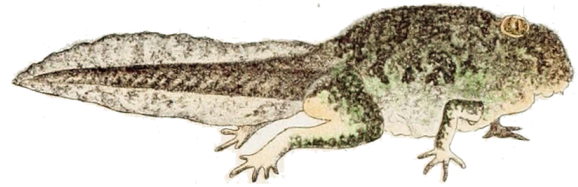
n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

...

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

...

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



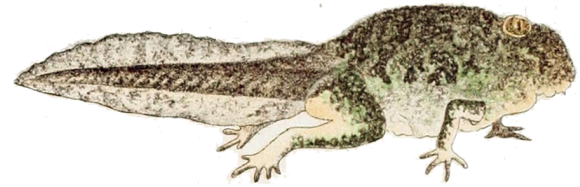
n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

...

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

...

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



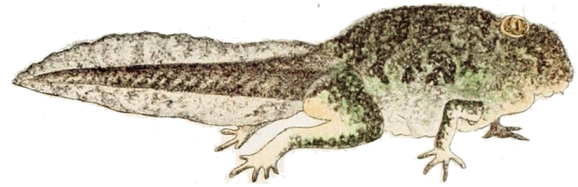
n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

...

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

...

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



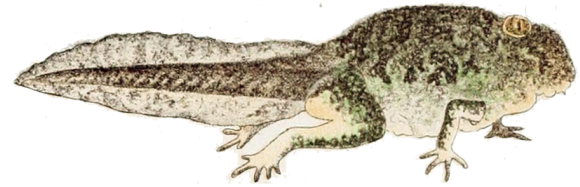
n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

…

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

…

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



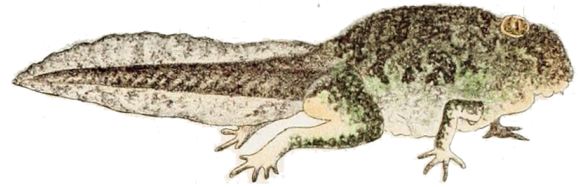
n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

...

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

...

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



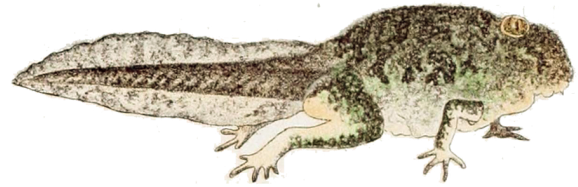
n	0	1	2	3	4	5	6	···	base
ck	F	F	T	F	T	T	F	···	base
x = n when (ck = T)			3		5	6		···	base on (ck = T)
y = 0 fby x			0		3	5		···	base on (ck = T)
y_M = 0 mby x	0	0	0	3	3	5	6	···	base

...

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

...

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



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

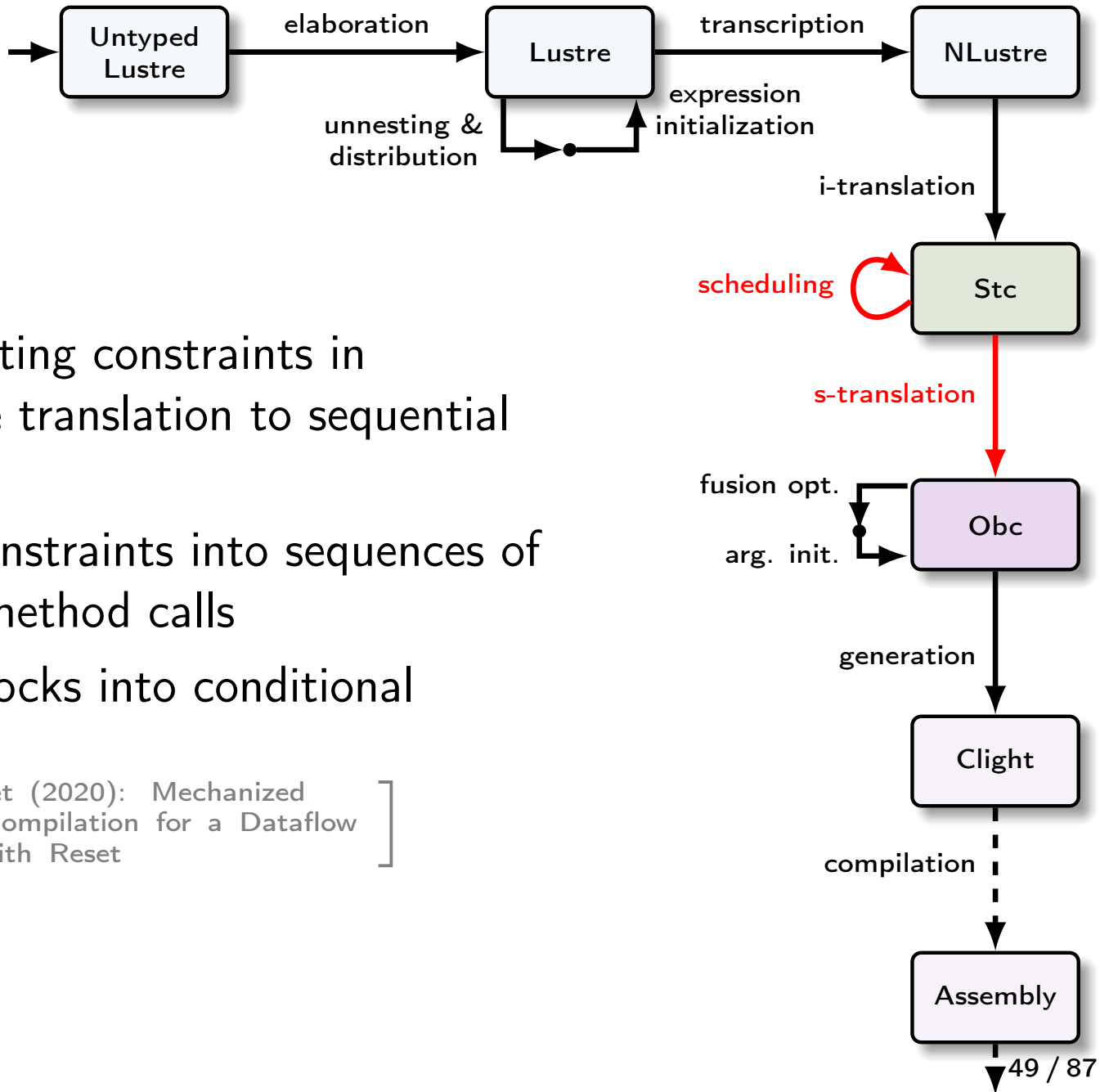
...

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

...

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

s-translation: Stc to Obc



- Schedule the resulting constraints in anticipation of the translation to sequential code
- Convert lists of constraints into sequences of assignments and method calls
- Translate static clocks into conditional statements
- [Bourke, Brun, and Pouzet (2020): Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset]

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: $me, ve \vdash_{prog} s \Downarrow me', ve'$

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

Obc: methods and classes

```
Record method : Type :=
  mk_method {
      m_name : ident;
      m_in   : list (ident * type);
      m_vars : list (ident * type);
      m_out  : list (ident * type);
      m_body : stmt;

      m_nodupvars : NoDupMembers (m_in ++ m_vars ++ m_out);
      m_good      : Forall NotReserved (m_in ++ m_vars ++ m_out)   }.

Record class : Type :=
  mk_class {
      c_name    : ident;
      c_mems    : list (ident * type);
      c_objs    : list (ident * ident);   (* (instance, class) *)
      c_methods : list method;

      c_nodups  : NoDup (map fst c_mems ++ map fst c_objs)
      c_nodupm  : NoDup (map m_name c_methods)
      c_good    : Forall (fun xt ⇒ valid (fst xt)) c_objs ∧ valid c_name  }.

Definition program : Type := list class.
```

- Structural invariants included in dependent records (as in CompCert).
- Typing predicate not shown:
  - » Memories used in methods are declared in class.
  - » Variables used in methods are declared in methods.
  - » Classes and methods used in methods are declared in programs.

Obc: semantic model

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

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

x := e

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

state(x) := e

```
| Istep:
    omenv = match mfind_inst o menv with None ⇒ hempty | Some om ⇒ om end →
    Forall2 (exp_eval menv env) es vos →
    stmt_call_eval prog omenv cls m vos omenv' rvos →
    madd_obj o omenv' menv = menv' →
    updates ys rvos env = env' →
    ------------------------------------------------
    stmt_eval prog menv env (Call ys cls o m es) (menv', env')
```

x,...,x := cls.m o (e,...,e)

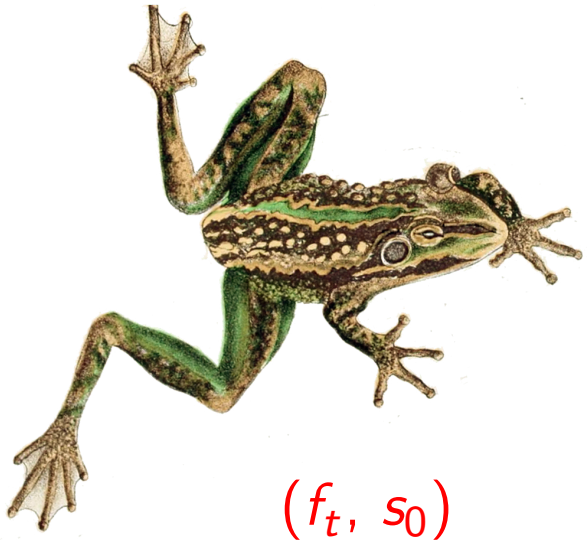
Obc: semantic model

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

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

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

| Istep:
    omenv = match mfind_inst o menv with None ⇒ hempty | Some om ⇒ om end →
    Forall2 (exp_eval menv env) es vos →
    stmt_call_eval prog omenv cls m vos omenv' rvos →
    madd_obj o omenv' menv = menv' →
    updates ys rvos env = env' →
    ------------------------------------------------
    stmt_eval prog menv env (Call ys cls o m es) (menv', env')
⋮
```



x := e

state(x) := e

(f_t, s_0)

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

S

x,...,x := cls.m o (e,...,e)

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    next norm1$2 = not i;
    edge = i and norm1$2;
    next norm1$1 = o;
    ck = edge or norm1$1;
    elab$3 = dcount<elab$3>(n when ck);
    reset(dcount<elab$3>) every (. on edge);
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck);
    o = v > 0;
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    next norm1$2 = not i;
    edge = i and norm1$2;
    next norm1$1 = o;
    ck = edge or norm1$1;
    elab$3 = dcount<elab$3>(n when ck);
    reset(dcount<elab$3>) every (. on edge);
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck);
    o = v > 0;
  }
}
```

Schedule Stc constraints

- Normal variables: write before reads
- Next variables: reads before write
- Semantics is invariant under constraint reordering

- Scheduling in OCaml using a heuristic to group constraints with related guards
- Use result to sort constraints in Coq
- Correctness proof is trivial

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

Schedule Stc constraints

- Normal variables: write before reads
- Next variables: reads before write
- Semantics is invariant under constraint reordering

- Scheduling in OCaml using a heuristic to group constraints with related guards
- Use result to sort constraints in Coq
- Correctness proof is trivial

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n])
    };
    if ck {
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

s-translation

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n])
    };
    if ck {
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

s-translation

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n])
    };
    if ck {
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

s-translation

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n])
    };
    if ck {
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

s-translation

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n])
    };
    if ck {
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

s-translation

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n])
    };
    if ck {
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

s-translation

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n])
    };
    if ck {
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

s-translation

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n])
    };
    if ck {
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

Translation to simple imperative code (Obc)

```
system rising_edge_retrigger {
  init norm1$1 = 0,
       norm1$2 = 0;
  sub <elab$3> : dcount;

  transition(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v : int32;
      elab$3 : int32 when ck;
  {
    edge = i and norm1$2
    next norm1$2 = not i
    ck = edge or norm1$1
    reset(dcount<elab$3>) every (. on edge)
    elab$3 = dcount<elab$3>(n when ck)
    v = merge ck
          (true -> elab$3)
          (false -> 0 when not ck)
    o = v > 0
    next norm1$1 = o
  }
}
```

s-translation

```
class rising_edge_retrigger {
  state norm1$1 : bool;
  state norm1$2 : bool;
  instance elab$3 : dcount;

  step(i : bool; n : int32)
  returns (o : bool)
  var edge, ck : bool; v, elab$3 : int32;
  {
    edge := i and state(norm1$2);
    state(norm1$2) := not i;
    ck := edge or state(norm1$1);
    if edge { dcount(elab$3).reset() };
    if ck {
      elab$3 := dcount(elab$3).step([n]);
      v := elab$3
    } else {
      v := 0
    };
    o := v > 0;
    state(norm1$1) := o
  }

  reset() {
    state(norm1$2) := 0;
    state(norm1$1) := 0;
    dcount(elab$3).reset()
  }
}
```

```
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_exp l)
  end.

Definition translate_tc (tc: trconstr) : stmt :=
  match tc with
  | TcDef x ck ce ⇒ Control ck (translate_cexp x ce)
  | TcNext x ck le ⇒ Control ck (AssignSt x (translate_exp le))
  | TcCall s xs ck rst f es ⇒
      Control ck (Call xs f s step (map (translate_arg ck) es))
  | TcReset s ck f ⇒ Control ck (Call [] f s reset [])
  end.
```

```
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_tc (tc: trconstr) : stmt :=
  match tc with
  | TcDef x ck ce ⇒ Control ck (translate_cexp x ce)
  | TcNext x ck le ⇒ Control ck (AssignSt x (translate_exp le))
  | TcCall s xs ck rst f es ⇒
      Control ck (Call xs f s step (map (translate_arg ck) es))
  | TcReset s ck f ⇒ Control ck (Call [] f s reset [])
  end.

Definition translate_tcs (tcs: list trconstr) : stmt :=
  fold_left (fun i tc ⇒ Comp (translate_tc tc) i) tcs Skip.
```

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

Definition translate_tc (tc: trconstr) : stmt :=
  match tc with
  | TcDef x ck ce ⇒ Control ck (translate_cexp x ce)
  | TcNext x ck le ⇒ Control ck (AssignSt x (translate_exp le))
  | TcCall s xs ck rst f es ⇒
     Control ck (Call xs f s step (map (translate_arg ck) es))
  | TcReset s ck f ⇒ Control ck (Call [] f s reset [])
  end.
```

```
Definition translate_tcs (tcs: list trconstr) : stmt :=
  fold_left (fun i tc ⇒ Comp (translate_tc tc) i) tcs Skip.

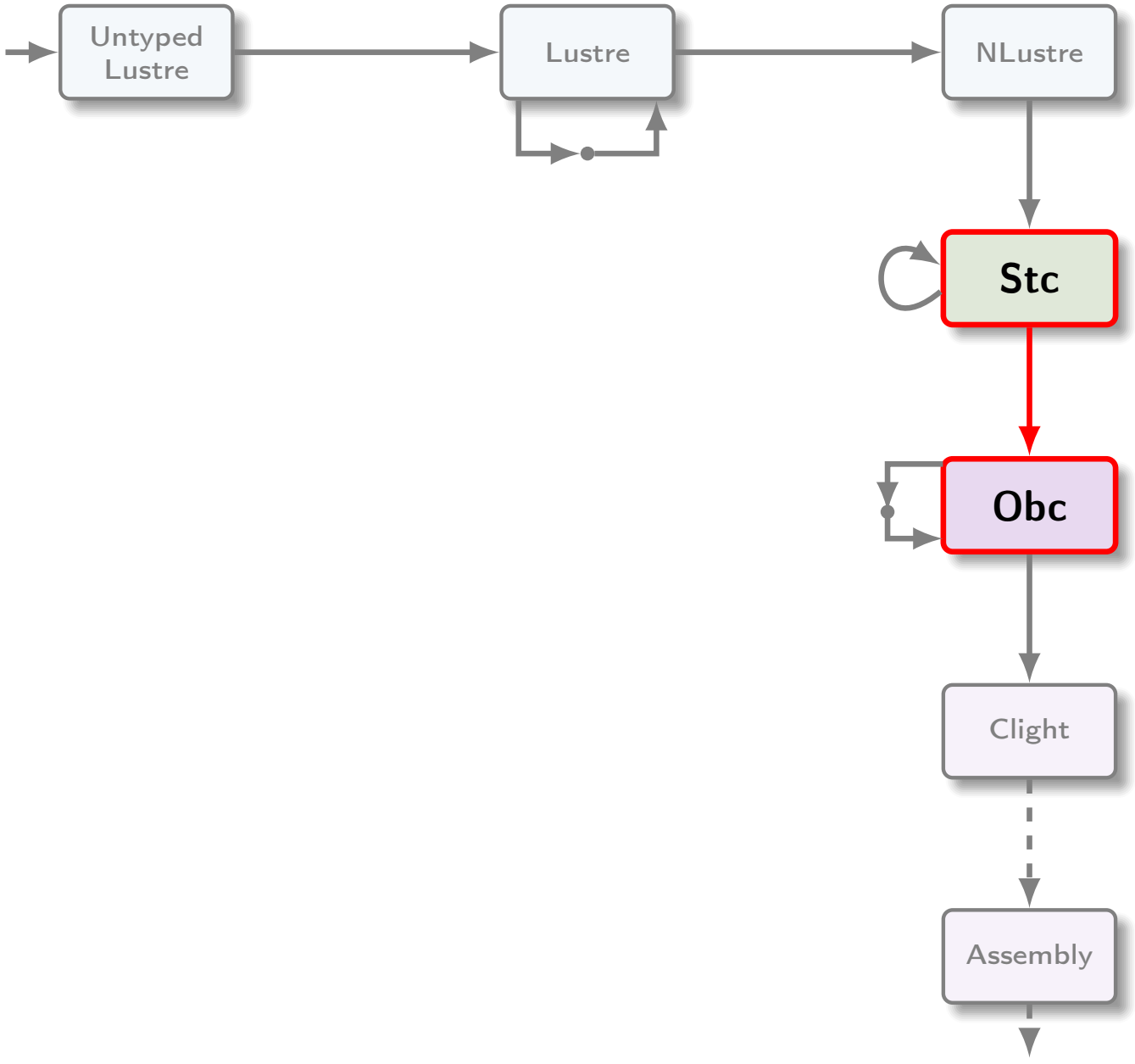
Definition ps_from_list (l: list ident) : PS.t :=
  fold_left (fun s i⇒PS.add i s) l PS.empty.

Program Definition step_method (s: system) : method :=
  let memids := map fst s.(s_lasts) in
  let mems := ps_from_list memids in
  let clkvars := Env.adds_with snd s.(s_out)
                 (Env.adds_with snd s.(s_vars)
                   (Env.from_list_with snd s.(s_in))) in
  {| m_name := step;
     m_in   := idty s.(s_in);
     m_vars := idty s.(s_vars);
     m_out  := idty s.(s_out);
     m_body := translate_tcs mems clkvars s.(s_tcs)
  |}.

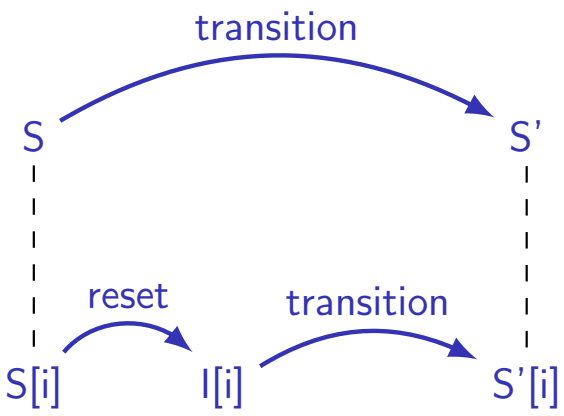
Program Definition translate_system (b: system) : class :=
  {| c_name    := b.(s_name);
     c_mems    :=
      map (fun xc ⇒ (fst xc, type_const (fst (snd xc)))) b.(s_lasts);
     c_objs    := b.(s_subs);
     c_methods := [ step_method b; reset_method b ]
  |}.

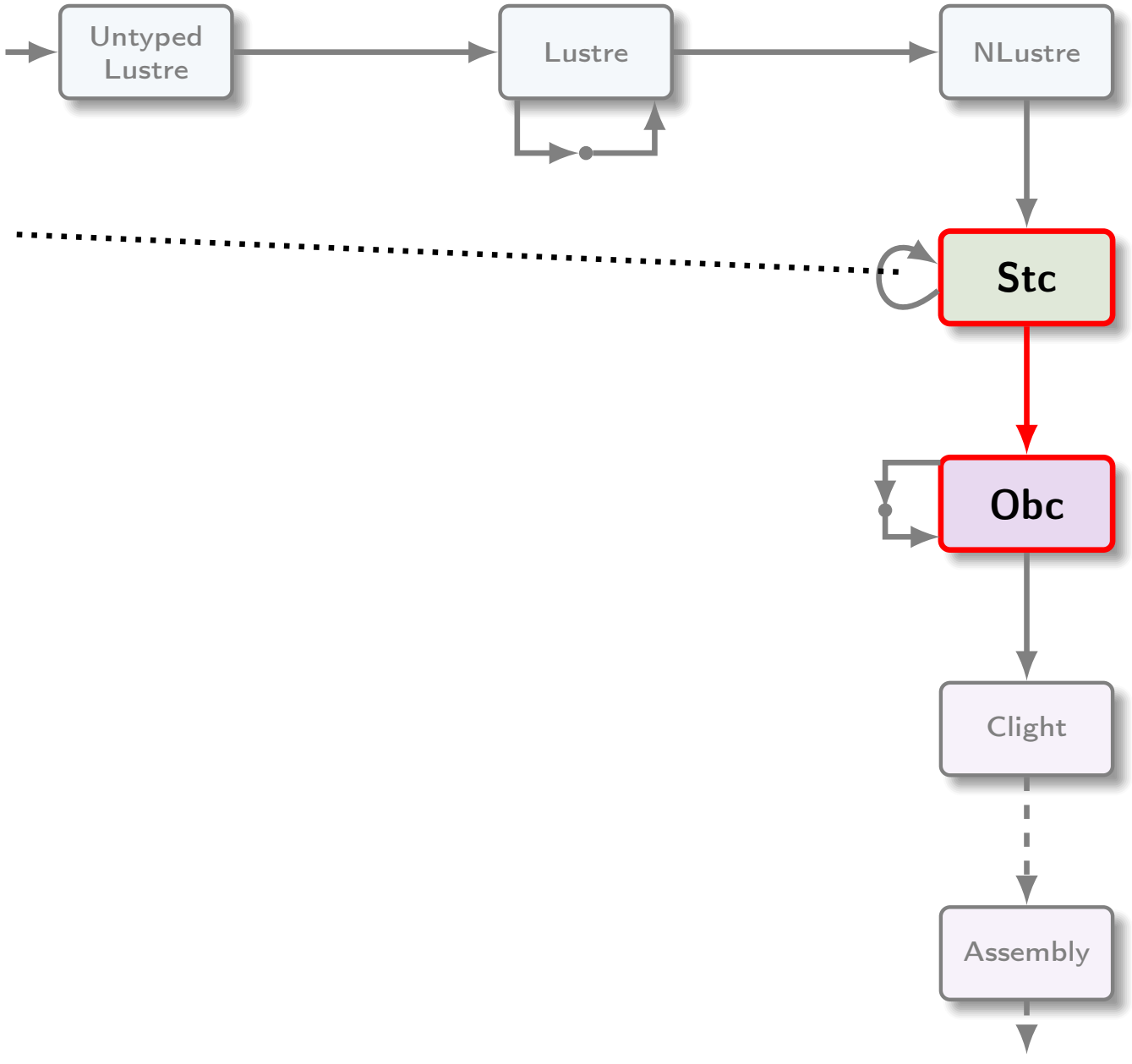
Definition translate (P: SynStc.program) : program :=
  map translate_system P.
```

Correctness of Stc to Obc translation

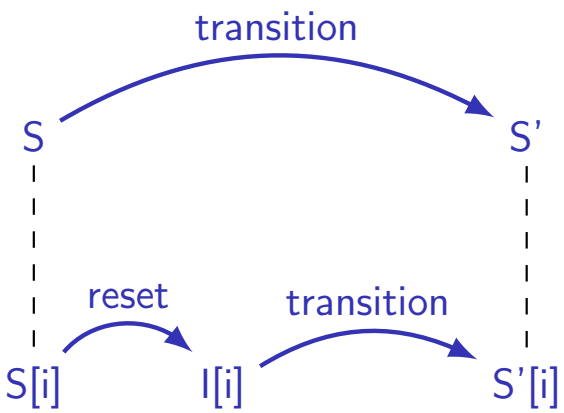


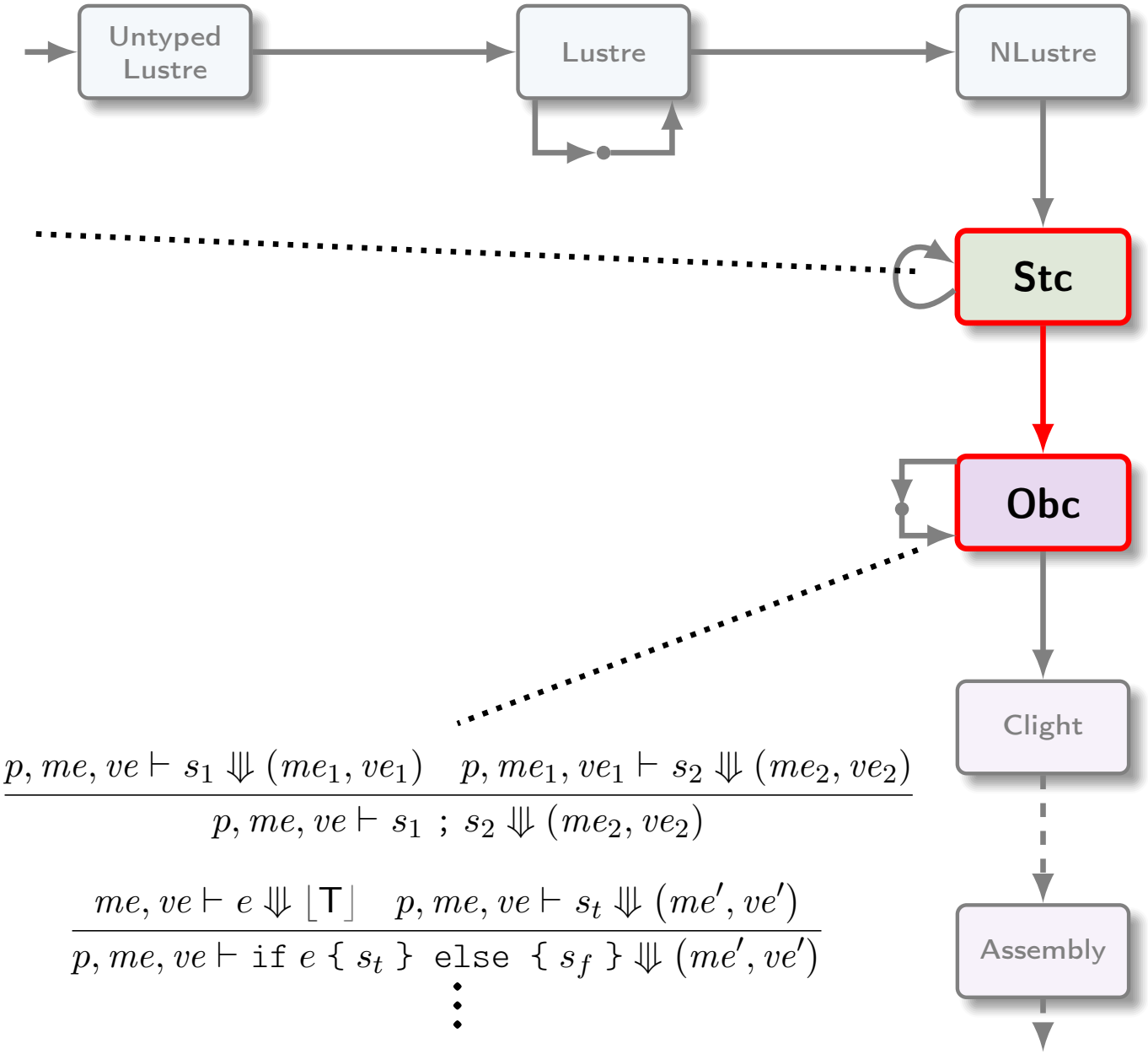
Correctness of Stc to Obc translation





Correctness of Stc to Obc translation



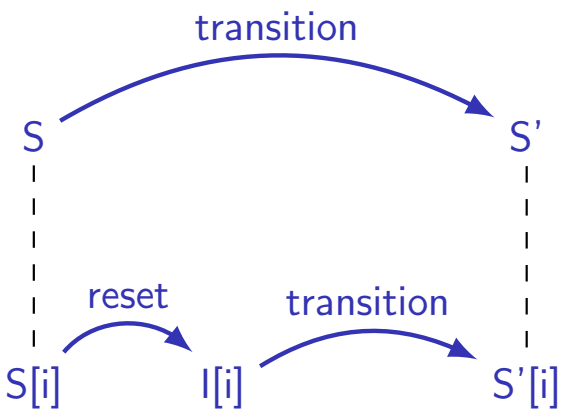


$$\frac{p, me, ve \vdash s_1 \Downarrow (me_1, ve_1) \quad p, me_1, ve_1 \vdash s_2 \Downarrow (me_2, ve_2)}{p, me, ve \vdash s_1 \ ; \ s_2 \Downarrow (me_2, ve_2)}$$

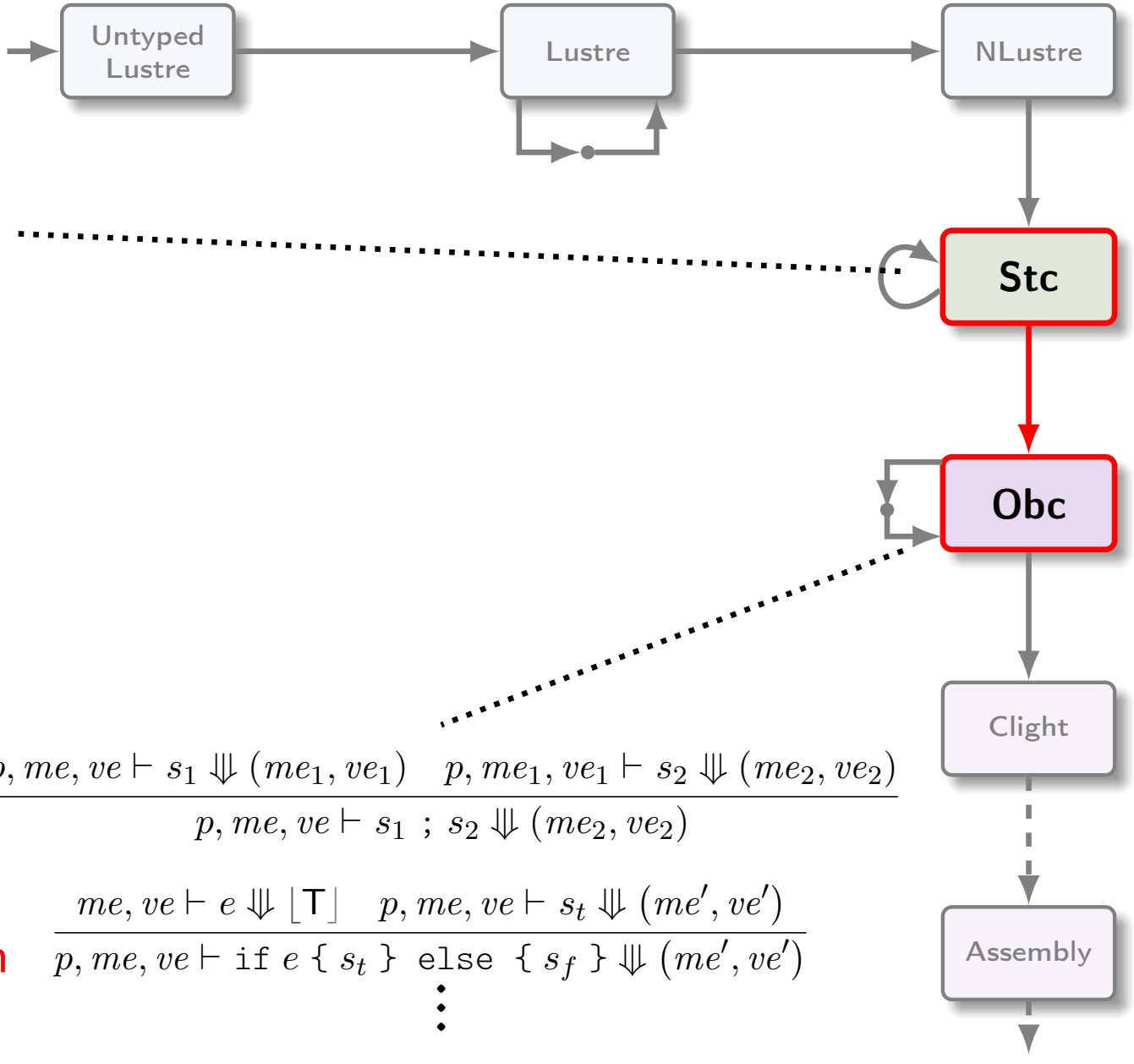
$$\frac{me, ve \vdash e \Downarrow \lfloor \mathsf{T} \rfloor \quad p, me, ve \vdash s_t \Downarrow (me', ve')}{p, me, ve \vdash \texttt{if}\ e\ \{\ s_t\ \}\ \texttt{else}\ \{\ s_f\ \} \Downarrow (me', ve')}$$

$\vdots$

Correctness of Stc to Obc translation



induction using a correspondence between $S/I/S'$ and me



$$\frac{p, me, ve \vdash s_1 \Downarrow (me_1, ve_1) \quad p, me_1, ve_1 \vdash s_2 \Downarrow (me_2, ve_2)}{p, me, ve \vdash s_1 \ ; \ s_2 \Downarrow (me_2, ve_2)}$$

$$\frac{me, ve \vdash e \Downarrow \lfloor \mathsf{T} \rfloor \quad p, me, ve \vdash s_t \Downarrow (me', ve')}{p, me, ve \vdash \texttt{if}\ e\ \{\ s_t\ \}\ \texttt{else}\ \{\ s_f\ \} \Downarrow (me', ve')}$$

$\vdots$

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 states are 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 states are central.

alleqs

oeqs eq eqs

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

Stc to Obc: invariant

```
Definition state := memory val.
Definition menv := memory val.

Definition value_corres (x: ident) (S: state) (me: menv) : Prop :=
  find_val x S = find_val x me.

Definition state_corres (s: ident) (S: state) (me: menv) : Prop :=
  find_inst s S ⌈≡⌉ find_inst s me.

Definition Memory_Corres (tcs: list trconstr) (S I S': state) (me: menv) : Prop
  :=
     (∀ x, (Is_last_in x tcs → value_corres x S' me)
           ∧ (¬ Is_last_in x tcs → value_corres x S me))

   ∧ (∀ s, (¬ Step_in s tcs ∧ ¬ Reset_in s tcs → state_corres s S me)
           ∧ (¬ Step_in s tcs ∧ Reset_in s tcs → state_corres s I me)
           ∧ (Step_in s tcs → state_corres s S' me)).
```

Overview

Lustre: syntax and semantics

Lustre: normalization

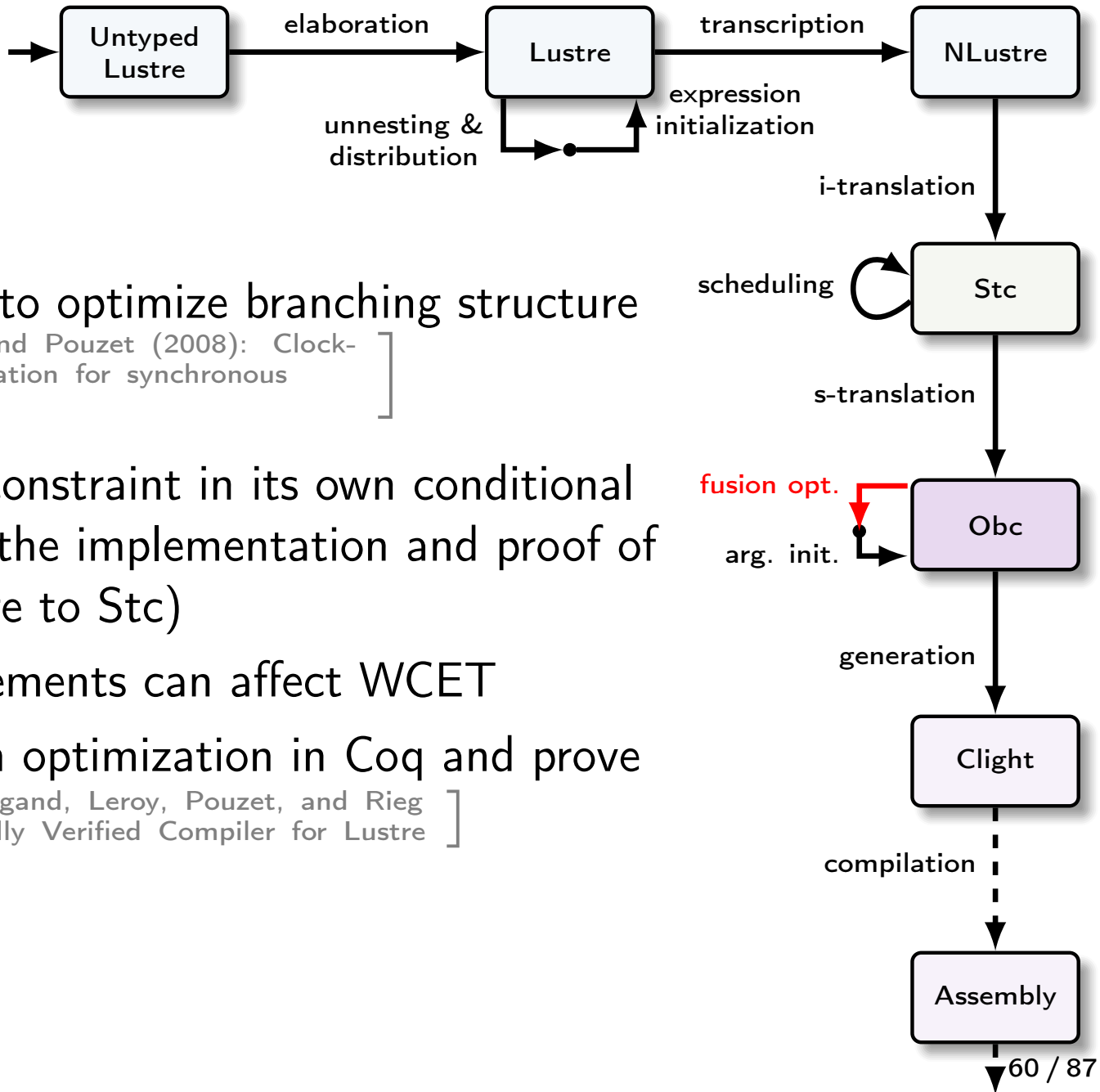
Translation: from dataflow programs to imperative code

Optimization: control structure fusion

Generation: from Obc to Clight

Conclusion

Fusion optimization: Obc to Obc



- Classic optimization to optimize branching structure
  [Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]
- Wrapping each Stc constraint in its own conditional statement simplifies the implementation and proof of s-translation (NLustre to Stc)
- But conditional statements can affect WCET
- So, implement fusion optimization in Coq and prove correct [Bourke, Brun, Dagand, Leroy, Pouzet, and Rieg (2017): A Formally Verified Compiler for Lustre]

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:

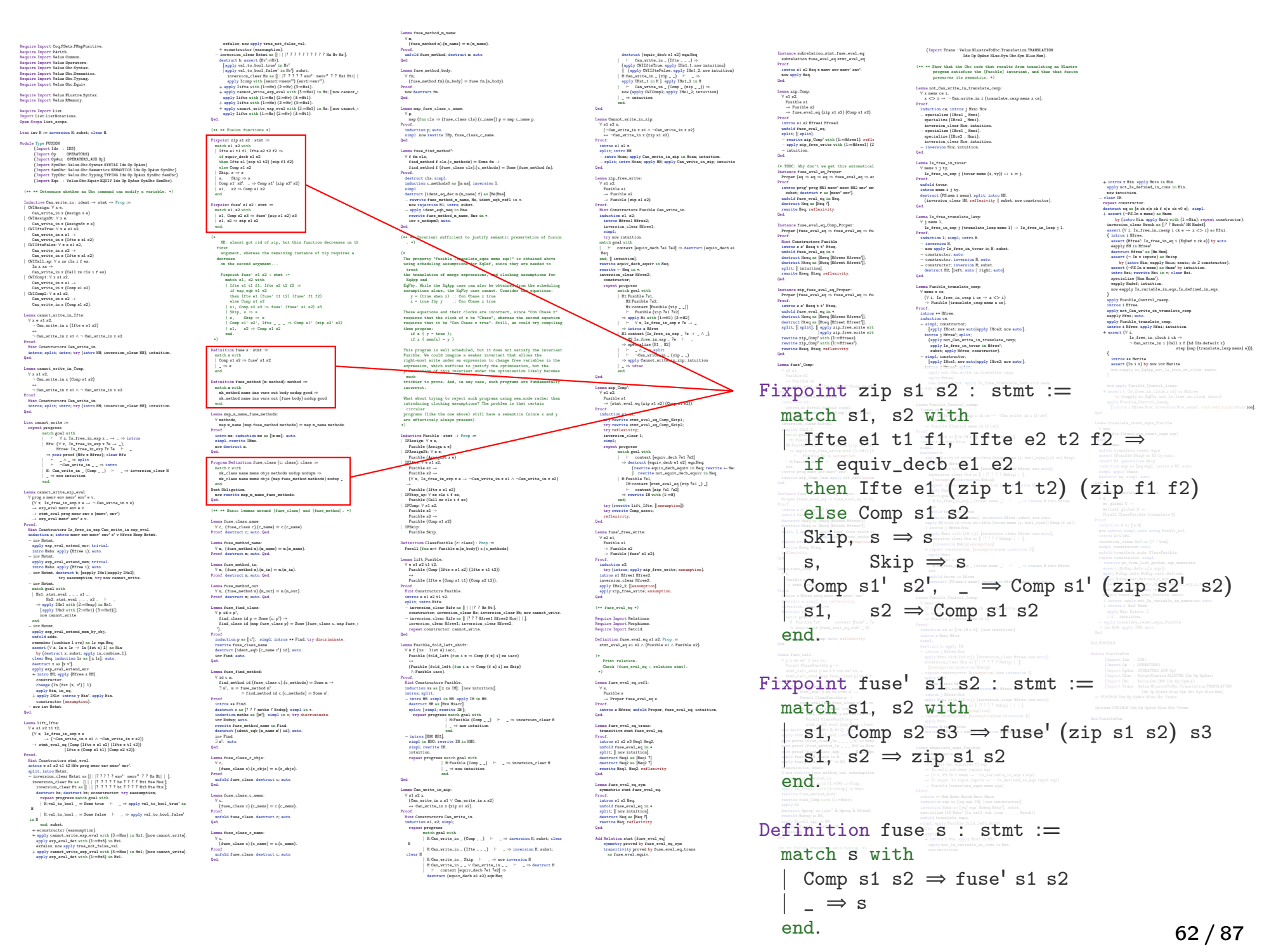
$$
\begin{aligned}
&Join(\ \texttt{case}\ (x)\ \{C_1 : S_1; ...; C_n : S_n\},\\
&\qquad \texttt{case}\ (x)\ \{C_1 : S'_1; ...; C_n : S'_n\})\\
&\quad = \texttt{case}\ (x)\ \{C_1 : Join(S_1, S'_1); ...; C_n : Join(S_n, S'_n)\}\\
&Join(S_1, S_2) = S_1; S_2
\end{aligned}
$$

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

```
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.
```

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

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

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

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

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

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

  Inductive Can_write_in : ident → stmt → Prop :=
  | CWIAssign: ∀ x e,
      Can_write_in x (Assign x e)
  | CWIAssignSt: ∀ x e,
      Can_write_in x (AssignSt x e)
  | CWIIfteTrue: ∀ x e s1 s2,
      Can_write_in x s1 →
      Can_write_in x (Ifte e s1 s2)
  | CWIIfteFalse: ∀ x e s1 s2,
      Can_write_in x s2 →
      Can_write_in x (Ifte e s1 s2)
  | CWICall_ap: ∀ x xs cls i f es,
      In x xs →
      Can_write_in x (Call xs cls i f es)
  | CWIComp1: ∀ x s1 s2,
      Can_write_in x s1 →
      Can_write_in x (Comp s1 s2)
  | CWIComp2: ∀ x s1 s2,
      Can_write_in x s2 →
      Can_write_in x (Comp s1 s2).
  Lemma cannot_write_in_Ifte:
    ∀ x e s1 s2,
      ¬ Can_write_in x (Ifte e s1 s2)
      ↔
      ¬ Can_write_in x s1 ∧ ¬ Can_write_in x s2.
  Proof.
    Hint Constructors Can_write_in.
    intros; split; intro; try (intro HH; inversion_clear HH); intuition.
  Qed.

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

  Ltac cannot_write :=
    repeat progress
           match goal with
           |    ⊢    ∀ x, Is_free_in_exp x _ → _ ⇒ intros
           | Hfw: (∀ x,  Is_free_in_exp x ?e → _),
                Hfree: Is_free_in_exp ?x ?e    ⊢    _
             ⇒ pose proof (Hfw x Hfree); clear Hfw
           |    ⊢    _ ∧ _ ⇒ split
           |    ⊢    ¬Can_write_in _ _ ⇒ intro
           | H:  Can_write_in _ (Comp _ _)    ⊢    _ ⇒ inversion_clear H
           | _ ⇒ now intuition
           end.

  Lemma cannot_write_exp_eval:
    ∀ prog s menv env menv' env' e v,
      (∀ x,  Is_free_in_exp x e → ¬ Can_write_in x s)
      → exp_eval menv env e v
      → stmt_eval prog menv env s (menv', env')
      → exp_eval menv' env' e v.
  Proof.
    Hint Constructors Is_free_in_exp Can_write_in exp_eval.
    induction s; intros menv env menv' env' e' v Hfree Hexp Hstmt.
    - inv Hstmt.
      apply exp_eval_extend_env; trivial.
      intro Habs. apply (Hfree i); auto.
    - inv Hstmt.
      apply exp_eval_extend_mem; trivial.
      intro Habs. apply (Hfree i); auto.
    - inv Hstmt. destruct b; [eapply IHs1|eapply IHs2];
                   try eassumption; try now cannot_write.
    - inv Hstmt.
      match goal with
      | Hs1:  stmt_eval _ _ _ s1 _,
              Hs2:  stmt_eval _ _ _ s2 _    ⊢    _
        ⇒ apply IHs1 with (2:=Hexp) in Hs1;
          [apply IHs2 with (2:=Hs1) (3:=Hs2)|];
          now cannot_write
      end.
    - inv Hstmt.
      apply exp_eval_extend_mem_by_obj.
      unfold adds.
      remember (combine l rvs) as lr eqn:Heq.
      assert (∀ x,  In x lr → In (fst x) l) as Hin
        by (destruct x; subst; apply in_combine_l).
      clear Heq. induction lr as [|x lr]; auto.
      destruct x as [x v'].
      apply exp_eval_extend_env.
      + intro HH; apply (Hfree x HH).
        constructor.
        change (In (fst (x, v')) l).
        apply Hin, in_eq.
      + apply IHlr. intros y Hin'. apply Hin.
        constructor (assumption).
    - now inv Hstmt.
  Qed.

  Lemma lift_Ifte:
    ∀ e s1 s2 t1 t2,
      (∀ x,  Is_free_in_exp x e
             → (¬Can_write_in x s1 ∧ ¬Can_write_in x s2))
      → stmt_eval_eq (Comp (Ifte e s1 s2) (Ifte e t1 t2))
                     (Ifte e (Comp s1 t1) (Comp s2 t2)).
  Proof.
    Hint Constructors stmt_eval.
    intros e s1 s2 t1 t2 Hfw prog menv env menv' env'.
    split; intro Hstmt.
    - inversion_clear Hstmt as [| | |? ? ? ? ? env'' menv'' ? ? Hs Ht| | ].
      inversion_clear Hs as   [| | | |? ? ? ? ? bs ? ? ? ? Hs1 Hse Hss|];
        inversion_clear Ht as [| | | |? ? ? ? ? bt ? ? ? ? Hs3 Hte Hts|];
        destruct bs; destruct bt; econstructor; try eassumption;
          repeat progress match goal with
          | H:val_to_bool _ = Some true    ⊢    _ ⇒ apply val_to_bool_true' in
   H
          | H:val_to_bool _ = Some false    ⊢    _ ⇒ apply val_to_bool_false'
  in H
          end; subst.
      + econstructor (eassumption).
      + apply cannot_write_exp_eval with (3:=Hss) in Hs1; [|now cannot_write].
        apply exp_eval_det with (1:=Hs3) in Hs1.
        exfalso; now apply true_not_false_val.
      + apply cannot_write_exp_eval with (3:=Hss) in Hs1; [|now cannot_write].
        apply exp_eval_det with (1:=Hs3) in Hs1.
```

```
        exfalso; now apply true_not_false_val.
      + econstructor (eassumption).
    - inversion_clear Hstmt as [| | |? ? ? ? ? ? ? ? ? ? Hs Hv Hs|].
      destruct b; assert (Hv':=Hv);
        [apply val_to_bool_true' in Hv'
        |apply val_to_bool_false' in Hv']; subst;
          inversion_clear Hs as [| | |? ? ? ? ? env'' menv'' ? ? Hs1 Ht1| |
          apply Icomp with (menv1:=menv'') (env1:=env'').
      + apply Iifte with (1:=Hx) (2:=Hv) (3:=Hs1).
      + apply cannot_write_exp_eval with (3:=Hs1) in Hx; [|now cannot_w
        apply Iifte with (1:=Hx) (2:=Hv) (3:=Ht1).
      + apply Iifte with (1:=Hx) (2:=Hv) (3:=Hs1).
      + apply cannot_write_exp_eval with (3:=Hs1) in Hx; [|now cannot_w
        apply Iifte with (1:=Hx) (2:=Hv) (3:=Ht1).
  Qed.

  (** ** Fusion functions *)

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

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

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

     Fixpoint fuse' s1 s2 : stmt :=
       match s1, s2 with
       | Ifte e1 t1 f1, Ifte e2 t2 f2 ⇒
         if exp_eqb e1 e2
         then Ifte e1 (fuse' t1 t2) (fuse' f1 f2)
         else Comp s1 s2
       | s1, Comp s2 s3 ⇒ fuse' (fuse' s1 s2) s3
       | Skip, s ⇒ s
       | s,    Skip ⇒ s
       | Comp s1' s2', Ifte _ _ _ ⇒ Comp s1' (zip s2' s2)
       | s1,   s2 ⇒ Comp s1 s2
       end.
  *)

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

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

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

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

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

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

  Lemma fuse_method_name:
    ∀ m,  (fuse_method m).(m_name) = m.(m_name).
  Proof. destruct m; auto. Qed.

  Lemma fuse_method_in:
    ∀ m,  (fuse_method m).(m_in) = m.(m_in).
  Proof. destruct m; auto. Qed.

  Lemma fuse_method_out:
    ∀ m,  (fuse_method m).(m_out) = m.(m_out).
  Proof. destruct m; auto. Qed.

  Lemma fuse_find_class:
    ∀ p id c p',
      find_class id p = Some (c, p') →
      find_class id (map fuse_class p) = Some (fuse_class c, map fuse_c
    ').
  Proof.
    induction p as [|c'];  simpl; intros ** Find; try discriminate.
    rewrite fuse_class_name.
    destruct (ident_eqb (c_name c') id); auto.
    inv Find; auto.
  Qed.

  Lemma fuse_find_method:
    ∀ id c m,
      find_method id (fuse_class c).(c_methods) = Some m →
      ∃ m',  m = fuse_method m'
               ∧ find_method id c.(c_methods) = Some m'.
  Proof.
    intros ** Find.
    destruct c as [? ? ? meths ? Nodup]; simpl in *.
    induction meths as [|m'];  simpl in *; try discriminate.
    inv Nodup; auto.
    rewrite fuse_method_name in Find.
    destruct (ident_eqb (m_name m') id); auto.
    inv Find.
    ∃ m';  auto.
  Qed.

  Lemma fuse_class_c_objs:
    ∀ c,
      (fuse_class c).(c_objs) = c.(c_objs).
  Proof.
    unfold fuse_class. destruct c; auto.
  Qed.

  Lemma fuse_class_c_mems:
    ∀ c,
      (fuse_class c).(c_mems) = c.(c_mems).
  Proof.
    unfold fuse_class. destruct c; auto.
  Qed.

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

```
  Lemma fuse_method_m_name:
    ∀ m,
      (fuse_method m).(m_name) = m.(m_name).
  Proof.
    unfold fuse_method; destruct m; auto.
  Qed.

  Lemma fuse_method_body:
    ∀ fm,
      (fuse_method fm).(m_body) = fuse fm.(m_body).
  Proof.
    now destruct fm.
  Qed.

  Lemma map_fuse_class_c_name:
    ∀ p,
      map (fun cls ⇒ (fuse_class cls).(c_name)) p = map c_name p.
  Proof.
    induction p; auto.
    simpl. now rewrite IHp, fuse_class_c_name.
  Qed.

  Lemma fuse_find_method':
    ∀ f fm cls,
      find_method f cls.(c_methods) = Some fm →
      find_method f (fuse_class cls).(c_methods) = Some (fuse_method fm).
  Proof.
    destruct cls; simpl.
    induction c_methods0 as [m ms]; inversion 1.
    simpl.
    destruct (ident_eq_dec m.(m_name) f) as [He|Hne].
    - rewrite fuse_method_m_name, He, ident_eqb_refl in *.
      now injection H1; intro; subst.
    - apply ident_eqb_neq in Hne.
      rewrite fuse_method_m_name, Hne in *.
      inv c_nodupm0; auto.
  Qed.

  (** ** Invariant sufficient to justify semantic preservation of fusion
    . *)

  (*
   The property "Fusible (translate_eqns mems eqs)" is obtained above
   using scheduling assumptions for EqDef, since they are needed to
    treat
   the translation of merge expressions, and clocking assumptions for
    EqApp and
   EqFby. While the EqApp case can also be obtained from the scheduling
   assumptions alone, the EqFby case cannot. Consider the equations:
      y = (true when x) :: Con Cbase x true
      x = true fby y    :: Con Cbase x true

   These equations and their clocks are incorrect, since "Con Cbase x"
   requires that the clock of x be "Cbase", whereas the second equation
   requires that it be "Con Cbase x true". Still, we could try compiling
   them program:
      if x { y = true };
      if x { mem(x) = y }

   This program is well scheduled, but it does not satisfy the invariant
   Fusible. We could imagine a weaker invariant that allows the
   right-most write under an expression to change free variables in the
   expression, which suffices to justify the optimization, but the
   preservation of this invariant under the optimization likely becomes
    much
   trickier to prove. And, in any case, such programs are fundamentally
   incorrect.

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

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

  Definition ClassFusible (c: class) : Prop :=
    Forall (fun m⇒ Fusible m.(m_body)) c.(c_methods).
  Lemma lift_Fusible:
    ∀ e s1 s2 t1 t2,
      Fusible (Comp (Ifte e s1 s2) (Ifte e t1 t2))
      ↔
      Fusible (Ifte e (Comp s1 t1) (Comp s2 t2)).
  Proof.
    Hint Constructors Fusible.
    intros e s1 s2 t1 t2.
    split; intro Hifw.
    - inversion_clear Hifw as [| | |? ? Hs Ht|].
      constructor; inversion_clear Hs; inversion_clear Ht; now cannot_write.
    - inversion_clear Hifw as [| |? ? ? Hfree1 Hfree2 Hcw| | |].
      inversion_clear Hfree1; inversion_clear Hfree2.
      repeat constructor; cannot_write.
  Qed.

  Lemma Fusible_fold_left_shift:
    ∀ A f (xs :  list A) iacc,
      Fusible (fold_left (fun i x ⇒ Comp (f x) i) xs iacc)
      ↔
      (Fusible (fold_left (fun i x ⇒ Comp (f x) i) xs Skip)
       ∧ Fusible iacc).
  Proof.
    Hint Constructors Fusible.
    induction xs as [|x xs IH];  [now intuition|].
    intros; split.
    - intro HH. simpl in HH. apply IH in HH.
      destruct HH as [Hxs Hiacc].
      split; [simpl; rewrite IH|];
        repeat progress match goal with
                        | H:Fusible (Comp _ _)    ⊢    _ ⇒ inversion_clear H
                        | _ ⇒ now intuition
                        end.
    - intros [HH0 HH1].
      simpl in HH0; rewrite IH in HH0.
      simpl; rewrite IH.
      intuition.
      repeat progress match goal with
                      | H:Fusible (Comp _ _)    ⊢    _ ⇒ inversion_clear H
                      | _ ⇒ now intuition
                      end.
  Qed.

  Lemma Can_write_in_zip:
    ∀ s1 s2 x,
      (Can_write_in x s1 ∨ Can_write_in x s2)
      ↔ Can_write_in x (zip s1 s2).
  Proof.
    Hint Constructors Can_write_in.
    induction s1, s2; simpl;
      repeat progress
             match goal with
             | H:Can_write_in _ (Comp _ _)    ⊢    _ ⇒ inversion H; subst; clear
    H
             | H:Can_write_in _ (Ifte _ _ _)    ⊢    _ ⇒ inversion H; subst;
   clear H
             | H:Can_write_in _ Skip    ⊢    _ ⇒ now inversion H
             | H:Can_write_in _ _ ∨ Can_write_in _ _    ⊢    _ ⇒ destruct H
             |    ⊢    context [equiv_decb ?e1 ?e2] ⇒
               destruct (equiv_decb e1 e2) eqn:Heq
```

```
               destruct (equiv_decb e1 e2) eqn:Heq
             |    ⊢    Can_write_in _ (Ifte _ _ _) ⇒
               (apply CWIIfteTrue; apply IHs1_1; now intuition)
               || (apply CWIIfteFalse; apply IHs1_2; now intuition)
             | H:Can_write_in _ (zip _ _)    ⊢    _ ⇒
               apply IHs1_1 in H || apply IHs1_2 in H
             |    ⊢    Can_write_in _ (Comp _ (zip _ _)) ⇒
               now (apply CWIComp2; apply IHs1_2; intuition)
             | _ ⇒ intuition
             end.
  Qed.

  Lemma Cannot_write_in_zip:
    ∀ s1 s2 x,
      (¬Can_write_in x s1 ∧ ¬Can_write_in x s2)
      ↔ ¬Can_write_in x (zip s1 s2).
  Proof.
    intros s1 s2 x.
    split; intro HH.
    - intro Hcan; apply Can_write_in_zip in Hcan; intuition.
    - split; intro Hcan; apply HH; apply Can_write_in_zip; intuitio
  Qed.

  Lemma zip_free_write:
    ∀ s1 s2,
      Fusible s1
      → Fusible s2
      → Fusible (zip s1 s2).
  Proof.
    Hint Constructors Fusible Can_write_in.
    induction s1, s2;
      intros Hfree1 Hfree2;
      inversion_clear Hfree1;
      simpl;
      try now intuition.
    match goal with
    |    ⊢    context [equiv_decb ?e1 ?e2] ⇒ destruct (equiv_decb e1
     Heq
    end; [|intuition].
    rewrite equiv_decb_equiv in Heq.
    rewrite ← Heq in *.
    inversion_clear Hfree2;
      constructor;
      repeat progress
             match goal with
             | H1:Fusible ?s1,
                 H2:Fusible ?s2,
                 Hi:context [Fusible (zip _ _)]
                 ⊢    Fusible (zip ?s1 ?s2)
               ⇒ apply Hi with (1:=H1) (2:=H2)
             |    ⊢    ∀ x, Is_free_in_exp x ?e → _
               ⇒ intros x Hfree
             | H1:context [Is_free_in_exp _ ?e → _ ∧ _],
                 H2:Is_free_in_exp _ ?e    ⊢    _
               ⇒ specialize (H1 _ H2)
             |    ⊢    _ ∧ _ ⇒ split
             |    ⊢    ¬Can_write_in _ (zip _ _)
               ⇒ apply Cannot_write_in_zip; intuition
             | _ ⇒ idtac
             end.
  Qed.

  Lemma zip_Comp':
    ∀ s1 s2,
      Fusible s1
      → (stmt_eval_eq (zip s1 s2) (Comp s1 s2)).
  Proof.
    induction s1, s2;
      try rewrite stmt_eval_eq_Comp_Skip1;
      try rewrite stmt_eval_eq_Comp_Skip2;
      try reflexivity;
      inversion_clear 1;
      simpl;
      repeat progress
             match goal with
             |    ⊢    context [equiv_decb ?e1 ?e2]
               ⇒ destruct (equiv_decb e1 e2) eqn:Heq;
                 (rewrite equiv_decb_equiv in Heq; rewrite ← Heq
                  || rewrite not_equiv_decb_equiv in Heq
             | H:Fusible ?s1,
                 IH:context [stmt_eval_eq (zip ?s1 _) _]
                 ⊢    context [zip ?s1 ?s2]
               ⇒ rewrite IH with (1:=H)
             end;
      try (rewrite lift_Ifte; [|assumption]);
      try rewrite Comp_assoc;
      reflexivity.
  Qed.

  Lemma fuse'_free_write:
    ∀ s2 s1,
      Fusible s1
      → Fusible s2
      → Fusible (fuse' s1 s2).
  Proof.
    induction s2;
      try (intros; apply zip_free_write; assumption).
    intros s1 Hfree1 Hfree2.
    inversion_clear Hfree2.
    apply IHs2_2; [|assumption].
    apply zip_free_write; assumption.
  Qed.

  (** fuse_eval_eq *)

  Require Import Relations.
  Require Import Morphisms.
  Require Import Setoid.

  Definition fuse_eval_eq s1 s2: Prop :=
    stmt_eval_eq s1 s2 ∧ (Fusible s1 ∧ Fusible s2).

  (*
    Print relation.
    Check (fuse_eval_eq : relation stmt).
   *)

  Lemma fuse_eval_eq_refl:
    ∀ s,
      Fusible s
      → Proper fuse_eval_eq s.
  Proof.
    intros s Hfree; unfold Proper, fuse_eval_eq; intuition.
  Qed.

  Lemma fuse_eval_eq_trans:
    transitive stmt fuse_eval_eq.
  Proof.
    intros s1 s2 s3 Heq1 Heq2.
    unfold fuse_eval_eq in *.
    split; [|now intuition].
    destruct Heq1 as [Heq1 ?].
    destruct Heq2 as [Heq2 ?].
    rewrite Heq1, Heq2; reflexivity.
  Qed.

  Lemma fuse_eval_eq_sym:
    symmetric stmt fuse_eval_eq.
  Proof.
    intros s1 s2 Heq.
    unfold fuse_eval_eq in *.
    split; [|now intuition].
    destruct Heq as [Heq ?].
    rewrite Heq; reflexivity.
  Qed.

  Add Relation stmt (fuse_eval_eq)
      symmetry proved by fuse_eval_eq_sym
      transitivity proved by fuse_eval_eq_trans
        as fuse_eval_equiv.
```

```
  Instance subrelation_stmt_fuse_eval_eq:
    subrelation fuse_eval_eq stmt_eval_eq.
  Proof.
    intros s1 s2 Heq x menv env menv' env'.
    now apply Heq.
  Qed.

  Lemma zip_Comp:
    ∀ s1 s2,
      Fusible s1
      → Fusible s2
      → fuse_eval_eq (zip s1 s2) (Comp s1 s2).
  Proof.
    intros s1 s2 Hfree1 Hfree2.
    unfold fuse_eval_eq.
    split; [|split].
    - rewrite zip_Comp' with (1:=Hfree1); refle
    - apply zip_free_write with (1:=Hfree1) (2
    - intuition.
  Qed.

  (* TODO: Why don't we get this automatical
  Instance fuse_eval_eq_Proper:
    Proper (eq ⇒ eq ⇒ eq ⇒ fuse_eval_eq ⇒ ec
  Proof.
    intros prog' prog HR1 menv' menv HR2 env' en
      subst; destruct r as [menv' env'].
    unfold fuse_eval_eq in Heq.
    destruct Heq as [Heq ?].
    rewrite Heq; reflexivity.
  Qed.

  Instance fuse_eval_eq_Comp_Proper:
    Proper (fuse_eval_eq ⇒ fuse_eval_eq ⇒ fu
  Proof.
    Hint Constructors Fusible.
    intros s s' Hseq t t' Hteq.
    unfold fuse_eval_eq in *.
    destruct Hseq as [Hseq [Hfrees Hfrees']].
    destruct Hteq as [Hteq [Hfreet Hfreet']].
    split; [|intuition].
    rewrite Hseq, Hteq; reflexivity.
  Qed.

  Instance zip_fuse_eval_eq_Proper:
    Proper (fuse_eval_eq ⇒ fuse_eval_eq ⇒ fu
  Proof.
    intros s s' Hseq t t' Hteq.
    unfold fuse_eval_eq in *.
    destruct Hseq as [Hseq [Hfrees Hfrees']].
    destruct Hteq as [Hteq [Hfreet Hfreet']].
    split; [|split]; [|apply zip_free_write wit
                      |apply zip_free_write wit
    rewrite zip_Comp' with (1:=Hfrees).
    rewrite zip_Comp' with (1:=Hfrees').
    rewrite Hseq, Hteq; reflexivity.
  Qed.

  Lemma fuse'_Comp:
    ∀ s2 s1,
      Fusible s1
      → Fusible s2
      → stmt_eval_eq (fuse' s1 s2) (Comp s1 s2)
  Proof.
    Hint Constructors Fusible.
    induction s2;
      intros s1 Hifte1 Hifte2; simpl;
        try now (rewrite zip_Comp'; intuition).
    rewrite Comp_assoc.
    inversion_clear Hifte2.
    rewrite IHs2_2;
      match goal with
      | H1:Fusible ?s1,
      H2:Fusible ?s2    ⊢    Fusible (zip ?s1 ?s2
        ⇒ apply zip_free_write with (1:=H1) (2
      |    ⊢    Fusible ?s ⇒ assumption
      | H:Fusible s2_2    ⊢    _ ⇒ pose proof (fu
      end.
    intros prog menv env menv' env'.
    rewrite zip_Comp; [now apply iff_refl|assum
  Qed.

  Instance fuse'_fuse_eval_eq_Proper:
    Proper (fuse_eval_eq ⇒ fuse_eval_eq ⇒ fu
  Proof.
    intros s s' Hseq t t' Hteq.
    unfold fuse_eval_eq in *.
    destruct Hseq as [Hseq [Hfrees Hfrees']].
    destruct Hteq as [Hteq [Hfreet Hfreet']].
    split; [|split]; [|apply fuse'_free_write w
                      |apply fuse'_free_write w
    repeat rewrite fuse'_Comp; try assumption.
    rewrite Hseq, Hteq.
    reflexivity.
  Qed.

  Lemma fuse_Comp:
    ∀ s,
      Fusible s
      → stmt_eval_eq (fuse s) s.
  Proof.
    intros s Hfree prog menv env menv' env'.
    destruct s; simpl; try reflexivity.
    inversion_clear Hfree.
    destruct s2;
      match goal with
      | H: Fusible ?s2    ⊢    context [fuse' _ ?s
        ⇒ pose proof (fuse_eval_eq_refl _ H)
      end;
      rewrite fuse'_Comp; auto; reflexivity.
  Qed.

  Lemma fuse_call:
    ∀ p n me me' f xss rs,
      Forall ClassFusible p →
      stmt_call_eval p me n f xss me' rs →
      stmt_call_eval (map fuse_class p) me n f :
  Proof.
    cut ((∀ p me ve stmt e',
              stmt_eval p me ve stmt e' →
              Forall ClassFusible p →
              stmt_eval (map fuse_class p) me ve
          ∧ (∀ p me clsid f vs me' rvs,
                stmt_call_eval p me clsid f vs s
                Forall ClassFusible p →
                stmt_call_eval (map fuse_class
    now destruct 1 as (Hf1 & Hf2); intros; apply
    apply stmt_eval_call_ind; intros; eauto us
    pose proof (find_class_In _ _ _ _ H) as Hinp
    pose proof (find_method_In _ _ _ H0) as Hin
    pose proof (find_class_app _ _ _ _ H) as Hp
    apply fuse_find_class in H.
    apply fuse_find_method' in H0.
    econstructor; eauto.
    2:now rewrite fuse_method_out; eassumptio
    rewrite fuse_method_in.
    apply In_Forall with (1:=H4) in Hinp.
    apply In_Forall with (1:=Hinp) in Hinc.
    rewrite fuse_method_body.
    rewrite fuse_Comp with (1:=Hinc).
    apply H2.
    destruct Hprog' as (cls'' & Hprog & Hfind).
    rewrite Hprog in H4.
    apply Forall_app in H4.
    rewrite Forall_cons2 in H4.
    intuition.
  Qed.

  (** ** Fusion preserves well-typing. *)
```

```
         (Import Trans : Velus.NLustreToObc.Translation.TRANSLATION
                         Ids Op OpAux NLus.Syn Obc.Syn NLus.Mem).

  (** ** Show that the Obc code that results from translating an NLustre
         program satisfies the [Fusible] invariant, and thus that fusion
         preserves its semantics. *)

  Lemma not_Can_write_in_translate_cexp:
    ∀ x mems ce i,
      x <> i → ¬ Can_write_in i (translate_cexp mems x ce).
  Proof.
    induction ce; intros j Hxni Hcw.
    - specialize (IHce1 _ Hxni).
      specialize (IHce2 _ Hxni).
      inversion_clear Hcw; intuition.
    - specialize (IHce1 _ Hxni).
      specialize (IHce2 _ Hxni).
      inversion_clear Hcw; intuition.
    - inversion Hcw; intuition.
  Qed.

  Lemma Is_free_in_tovar:
    ∀ mems i j ty,
      Is_free_in_exp j (tovar mems (i, ty)) ↔ i = j.
  Proof.
    unfold tovar.
    intros mems i j ty.
    destruct (PS.mem i mems); split; intro HH;
      (inversion_clear HH; reflexivity || subst; now constructor).
  Qed.

  Lemma Is_free_translate_lexp:
    ∀ j mems l,
      Is_free_in_exp j (translate_lexp mems l) → Is_free_in_lexp j l.
  Proof.
    induction l; simpl; intro H.
    - inversion H.
    - now apply Is_free_in_tovar in H; subst.
    - constructor; auto.
    - constructor; inversion H; auto.
    - constructor; inversion H; subst.
      destruct H2; [left; auto | right; auto].
  Qed.

  Lemma Fusible_translate_cexp:
    ∀ mems x ce,
      (∀ i,  Is_free_in_cexp i ce → x <> i)
      → Fusible (translate_cexp mems x ce).
  Proof.
    intros ** Hfree.
    induction ce.
    - simpl; constructor;
        [apply IHce1; now auto|apply IHce2; now auto|].
      intros j Hfree'; split;
        (apply not_Can_write_in_translate_cexp;
         apply Is_free_in_tovar in Hfree';
         subst; apply Hfree; constructor).
    - simpl; constructor;
        [apply IHce1; now auto|apply IHce2; now auto|].
      intros j Hfree'; split;
        apply not_Can_write_in_translate_cexp;
        apply Hfree;
        now constructor; apply Is_free_translate_lexp with mems.
    - now constructor.
  Qed.

  Lemma Fusible_Control_caexp:
    ∀ mems ck f ce,
      (∀ i,  Is_free_in_caexp i ck ce → ¬Can_write_in i (f ce))
      → Fusible (f ce)
      → Fusible (Control mems ck (f ce)).
  Proof.
    induction ck as [|ck IH i b];  [now intuition|].
    intros f ce Hxni Hfce.
    simpl.
    destruct b.
    - apply IH with (f:=fun ce⇒Ifte (tovar mems (i, bool_type)) (f ce) Skip).
      + intros j Hfree Hcw.
        apply Hxni with (i0:=j); [inversion_clear Hfree; now auto|].
        inversion_clear Hcw as [| | |? ? ? ? Hskip| | |];
          [assumption|inversion Hskip].
      + repeat constructor; [assumption| |now inversion 1].
        apply Hxni.
        match goal with
        | H:Is_free_in_exp _ (tovar mems _)    ⊢    _ ⇒ rename H into Hfree
        end.
        unfold tovar in Hfree.
        destruct (PS.mem i mems); inversion Hfree; subst; now auto.
    - apply IH with (f:=fun ce⇒Ifte (tovar mems (i, bool_type)) Skip (f ce)).
      + intros j Hfree Hcw.
        apply Hxni with (i0:=j); [inversion_clear Hfree; now auto|].
        inversion_clear Hcw as [| |? ? ? ? Hskip| | | |];
          [inversion Hskip|assumption].
      + repeat constructor; [assumption|now inversion 1|].
        apply Hxni.
        match goal with
        | H:Is_free_in_exp _ (tovar mems _)    ⊢    _ ⇒ rename H into Hfree
        end.
        unfold tovar in Hfree.
        destruct (PS.mem i mems); inversion Hfree; subst; now auto.
  Qed.

  Lemma Fusible_Control_laexp:
    ∀ mems ck s,
      (∀ i,  Is_free_in_clock i ck → ¬Can_write_in i s)
      → Fusible s
      → Fusible (Control mems ck s).
  Proof.
    induction ck as [|ck IH i b];  [now intuition|].
    intros s Hxni Hfce.
    simpl.
    destruct b; apply IH.
    - intros j Hfree Hcw.
      apply Hxni with (i0:=j); [inversion_clear Hfree; now auto|].
      inversion_clear Hcw as [| |? ? ? ? Hskip| | |];
        [assumption|inversion Hskip].
    - repeat constructor; [assumption| |now inversion 1].
      apply Hxni.
      rename H into Hfree.
      destruct (PS.mem i mems); inversion Hfree; subst; now auto.
    - intros j Hfree Hcw.
      apply Hxni with (i0:=j); [inversion_clear Hfree; now auto|].
      inversion_clear Hcw as [| |? ? ? ? Hskip| | | |];
        [inversion Hskip|assumption].
    - repeat constructor; [assumption|now inversion 1|].
      apply Hxni.
      rename H into Hfree.
      destruct (PS.mem i mems); inversion Hfree; subst; now auto.
  Qed.

  Lemma translate_eqns_Fusible:
    ∀ C mems inputs eqs,
      wc_env C
      → Forall (wc_equation C) eqs
      → Is_well_sch mems inputs eqs
      → (∀ x,  PS.In x mems → ¬Is_variable_in_eqs x eqs)
      → (∀ input, In input inputs → ¬ Is_defined_in_eqs input eqs)
      → Fusible (translate_eqns mems eqs).
  Proof.
    intros ** Hwk Hwks Hwsch Hnvi Hnin.
    induction eqs as [|eq eqs IH]; [now constructor|].
    inversion Hwks as [|eq' eqs' Hwkeq Hwks'];  subst.
    specialize (IH Hwks' (Is_well_sch_cons _ _ _ _ Hwsch)).
    unfold translate_eqns.
    simpl; apply Fusible_fold_left_shift.
    split.
    - apply IH.
      + intros x Hin; apply Hnvi in Hin.
        apply not_Is_variable_in_cons in Hin.
        now intuition.
```

```
      + intros x Hin. apply Hnin in Hin.
        apply not_Is_defined_in_cons in Hin.
        now intuition.
    - clear IH.
      repeat constructor.
      destruct eq as [x ck e|x ck f e|x ck v0 e];  simpl.
      + assert (¬PS.In x mems) as Hxm
            by (intro Hin; apply Hnvi with (1:=Hin); repeat constructor).
        inversion_clear Hwsch as [|? ? Hwsch' HH Hndef].
        assert (∀ i, Is_free_in_caexp i ck e → x <> i) as Hfni.
        { intros i Hfree.
          assert (Hfree': Is_free_in_eq i (EqDef x ck e)) by auto.
          eapply HH in Hfree'.
          destruct Hfree' as [Hm Hnm].
          assert (¬ In x inputs) as Hnimp
            by (intro Hin; eapply Hnin; eauto; do 2 constructor).
          assert (¬PS.In x mems) as Hxm' by intuition.
          intro Hxi; rewrite Hxi in *; clear Hxi.
          specialize (Hnm Hxm').
          eapply Hndef; intuition.
          now eapply Is_variable_in_eqs_Is_defined_in_eqs.
        }
        apply Fusible_Control_caexp.
        intros i Hfree.
        apply not_Can_write_in_translate_cexp.
        eapply Hfni; auto.
        apply Fusible_translate_cexp.
        intros i Hfree; apply Hfni; intuition.
      + assert (∀ i,
                   Is_free_in_clock i ck →
                   ¬ Can_write_in i (Call x f (hd Ids.default x)
                                          step (map (translate_lexp mems) e))).
        {
          intros ** Hwrite.
          assert (In i x) by now inv Hwrite.
          now eapply wc_EqApp_not_Is_free_in_clock; eauto.
        }

        now apply Fusible_Control_laexp.
      + assert (¬Is_free_in_clock x ck) as Hnfree
            by (eapply wc_EqFby_not_Is_free_in_clock; eauto).
        apply Fusible_Control_laexp;
          [intros i Hfree Hcw; inversion Hcw; subst; contradiction|intuition].
  Qed.

  Lemma translate_reset_eqns_Fusible:
    ∀ eqs,
      Fusible (translate_reset_eqns eqs).
  Proof.
    intro eqs.
    unfold translate_reset_eqns.
    assert (Fusible Skip) as Hf by auto.
    revert Hf. generalize Skip.
    induction eqs as [|eq eqs]; intros s Hf; auto.
    simpl. apply IHeqs.
    destruct eq; simpl; auto.
  Qed.

  Lemma ClassFusible_translate:
    ∀ G,
      wc_global G →
      Welldef_global G →
      Forall ClassFusible (translate G).
  Proof.
    induction G as [|n G].
    now intros; simpl; auto using Forall_nil.
    intros WcG WdG.
    inversion_clear WcG as [|? ? Wcn].
    simpl; constructor; auto.
    unfold translate_node, ClassFusible.
    repeat constructor; simpl.
    - rewrite ps_from_list_gather_eqs_memories.
      assert (NoDup_defs n.(n_eqs)).
      apply NoDup_defs_NoDup_vars_defined.
      apply NoDup_var_defined_n_eqs.
      pose proof (not_Exists_Is_defined_in_eqs_n_in n) as Hin.
      inv Wcn. inv WdG. simpl in *.
      eapply translate_eqns_Fusible; eauto.
      + intros. apply not_Is_variable_in_memories; auto.
      + intros i' Hin' Hdef.
        apply Hin, Exists_∃.
        ∃ i'.  intuition.
    - apply translate_reset_eqns_Fusible.
    - inv WdG. apply IHG; auto.
  Qed.

End FUSIBLE.

Module FusibleFun
       (Import Ids   : IDS)
       (Import Op    : OPERATORS)
       (Import OpAux : OPERATORS_AUX Op)
       (Import NLus  : Velus.NLustre.NLUSTRE Ids Op OpAux)
       (Import Obc   : Velus.Obc.OBC Ids Op OpAux)
       (Import Trans : Velus.NLustreToObc.Translation.TRANSLATION
                       Ids Op OpAux NLus.Syn Obc.Syn NLus.Mem)
  <: FUSIBLE Ids Op OpAux NLus Obc Trans.

  Include FUSIBLE Ids Op OpAux NLus Obc Trans.

End FusibleFun.
```

Fusion of control structures: implementation

$$\text{fuse}\left(\begin{array}{c} ; \\ s \quad t \end{array}\right) = \text{fuse}'(s, t)$$

$$\text{fuse}(s) = s$$

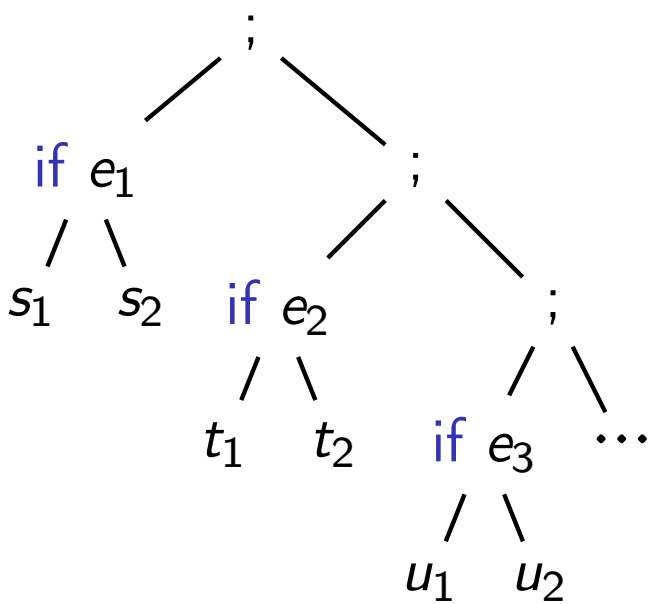
$$\text{fuse}'\left(s, \begin{array}{c} ; \\ t_1 \quad t_2 \end{array}\right) = \text{fuse}'(\text{zip}(s, t_1), t_2)$$

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

$$\text{zip}\left(\begin{array}{c} \text{if } e \\ s_1 \quad s_2 \end{array}, \begin{array}{c} \text{if } e \\ t_1 \quad t_2 \end{array}\right) = \begin{array}{c} \text{if } e \\ \text{zip}(s_1, t_1) \quad \text{zip}(s_2, t_2) \end{array}$$

$$\text{zip}\left(\begin{array}{c} ; \\ s_1 \quad s_2 \end{array}, t\right) = \begin{array}{c} ; \\ s_1 \quad \text{zip}(s_2, t) \end{array}$$

$$\text{zip}(s, t) = \begin{array}{c} ; \\ s \quad t \end{array}$$



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}

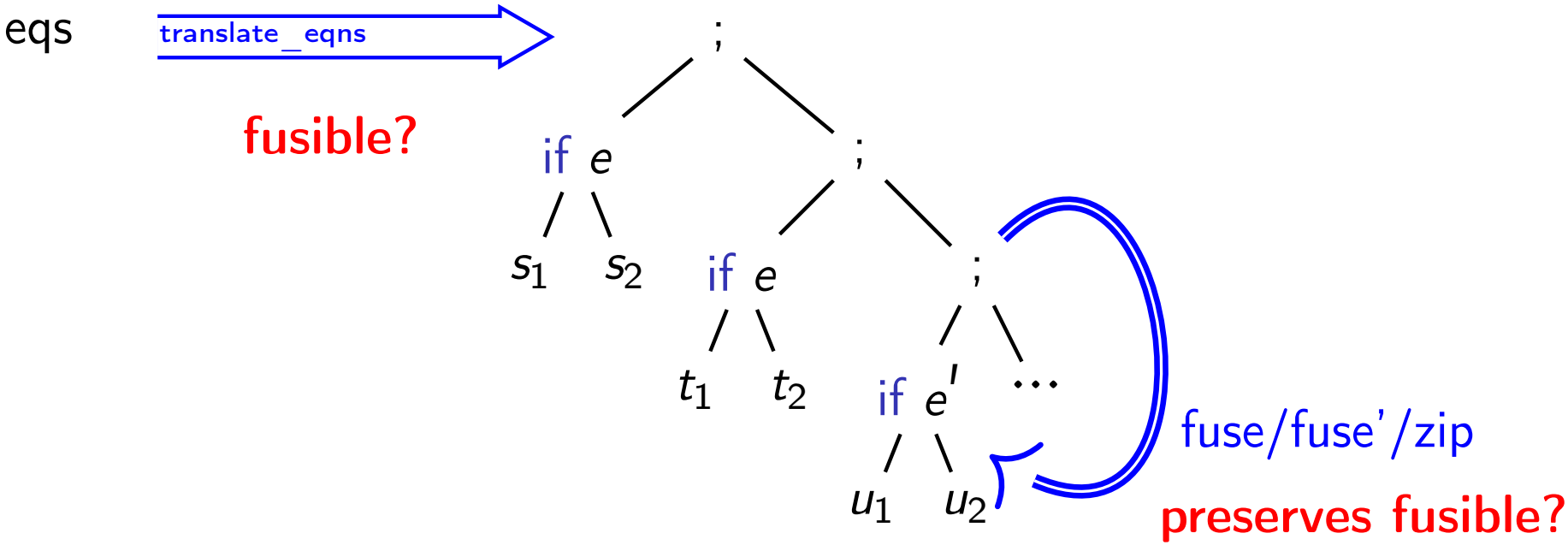
if e then {s1; t1} else {s2; t2};

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

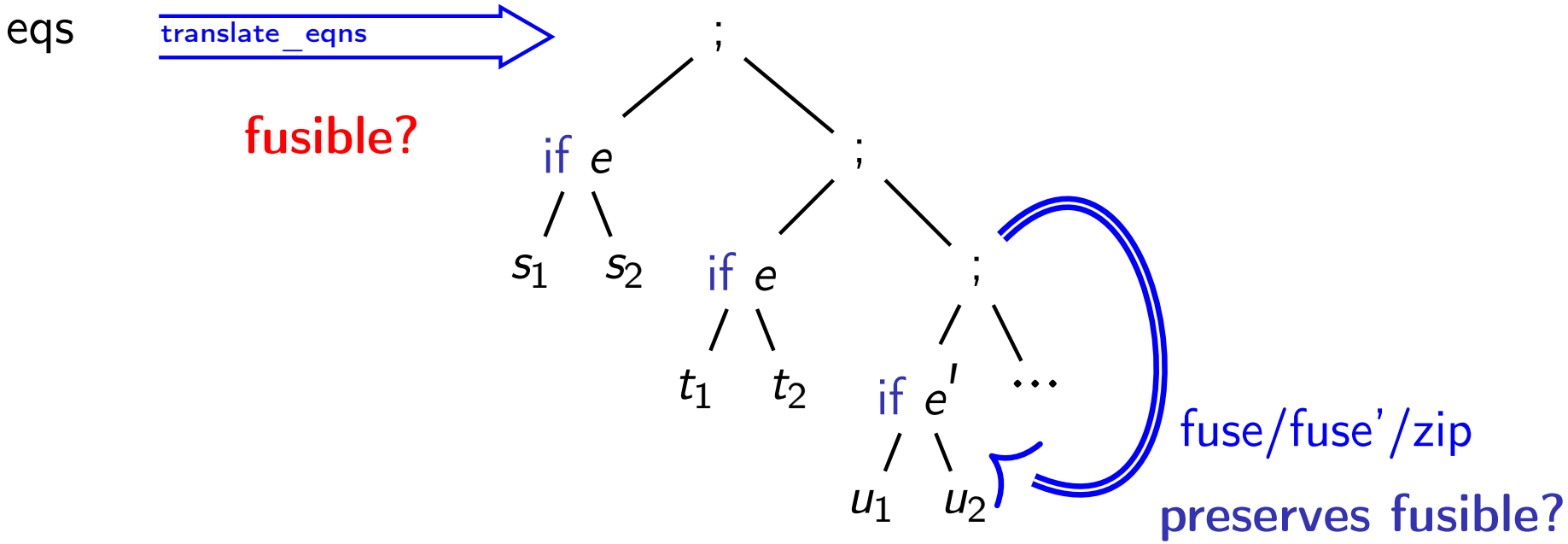
$$\frac{\text{fusionnable}(s_1) \qquad \text{fusionnable}(s_2) \qquad \forall x \in \text{libres}(e), \neg\text{peut-écrire } x\ s_1 \wedge \neg\text{peut-écrire } x\ s_2}{\text{fusionnable}(\text{if } e\ \{s_1\}\ \text{else}\ \{s_2\})}$$

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

Fusion of control structures: correctness



Fusion of control structures: correctness

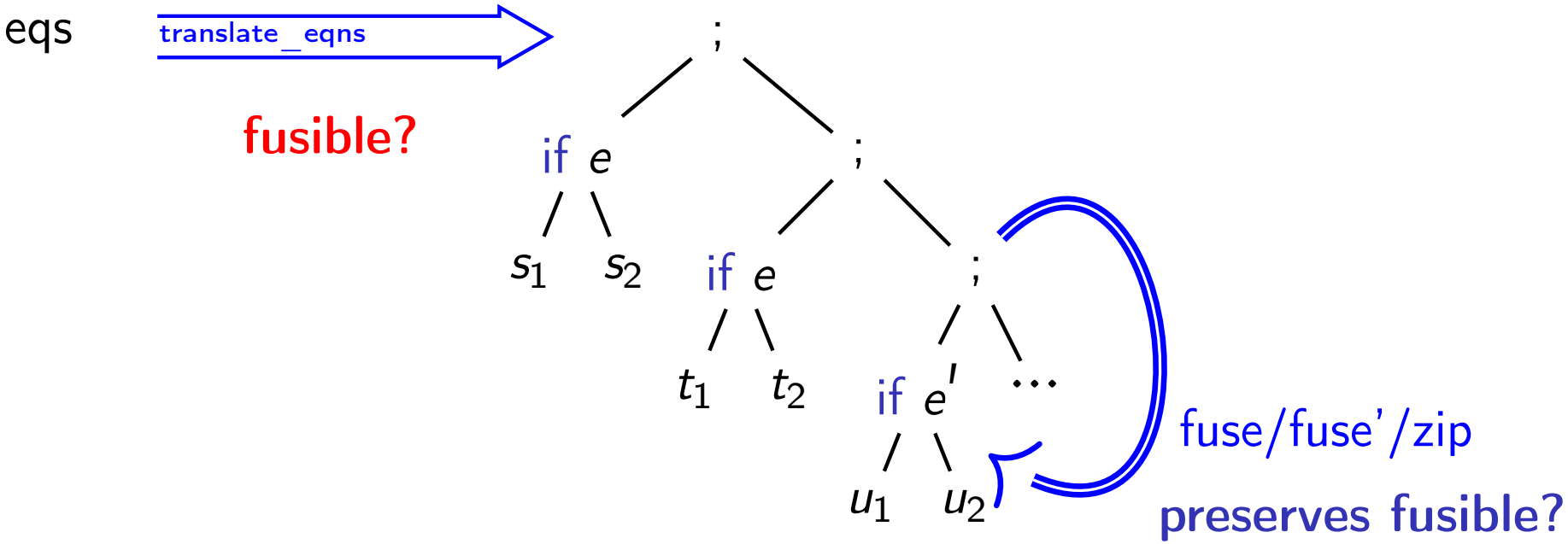


$x = (\text{merge b e1 e2})^{\text{base on ck}}$

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

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

Fusion of control structures: correctness

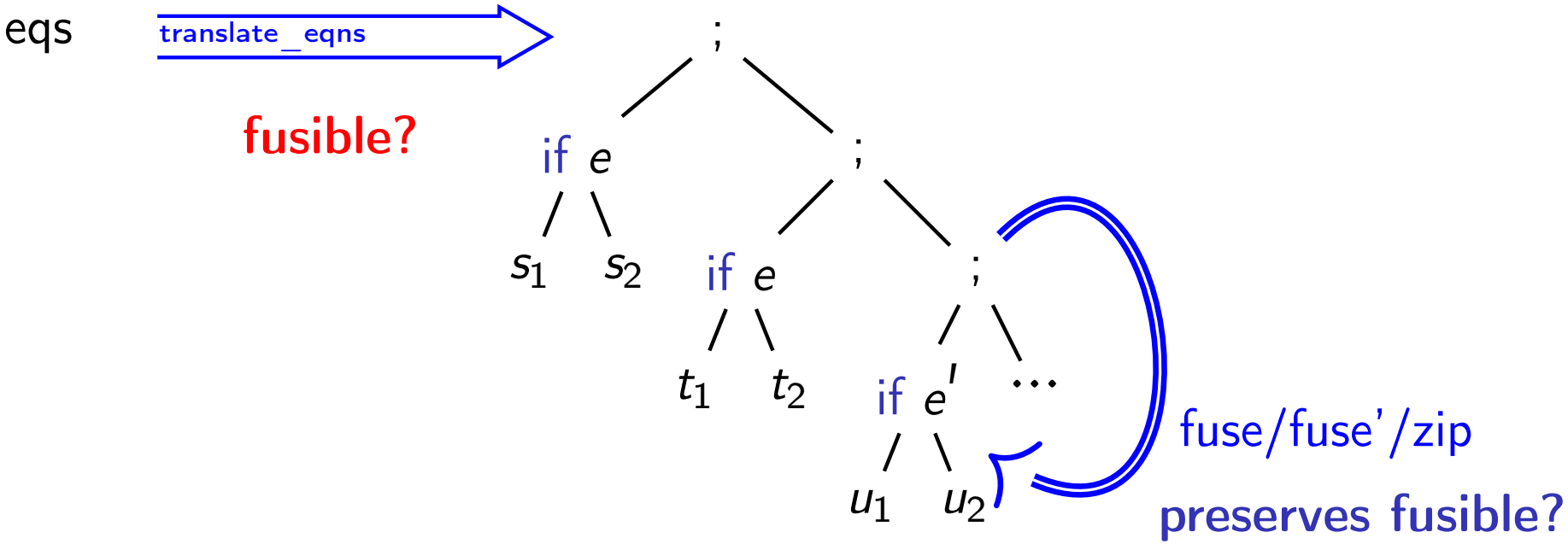


$x = (0 \text{ fby } (x + 1))^{\text{base on ck}}$

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

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

Fusion of control structures: correctness



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

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

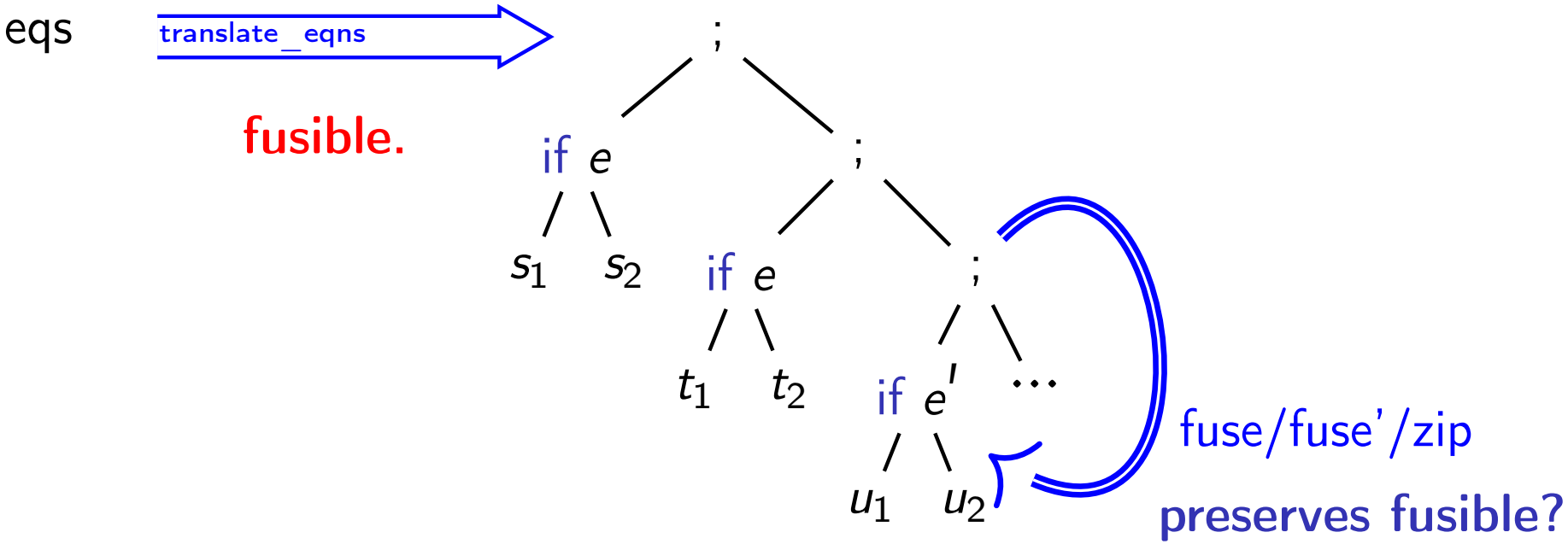
```
if mem(x) {
  y := true
}
if mem(x) {
  mem(x) := y
}
```

- Happily, such programs are not well clocked.

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

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

Fusion of control structures: correctness



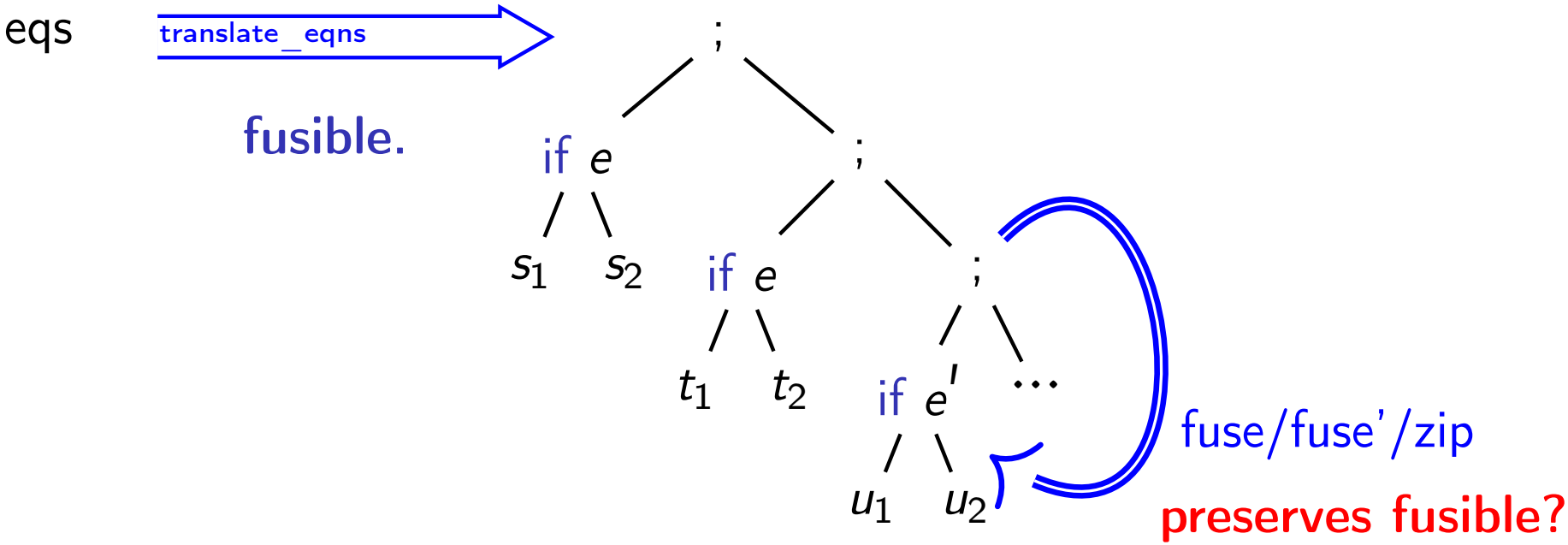
$y = (\text{true when } x)^{\text{base on } x}$

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

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

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

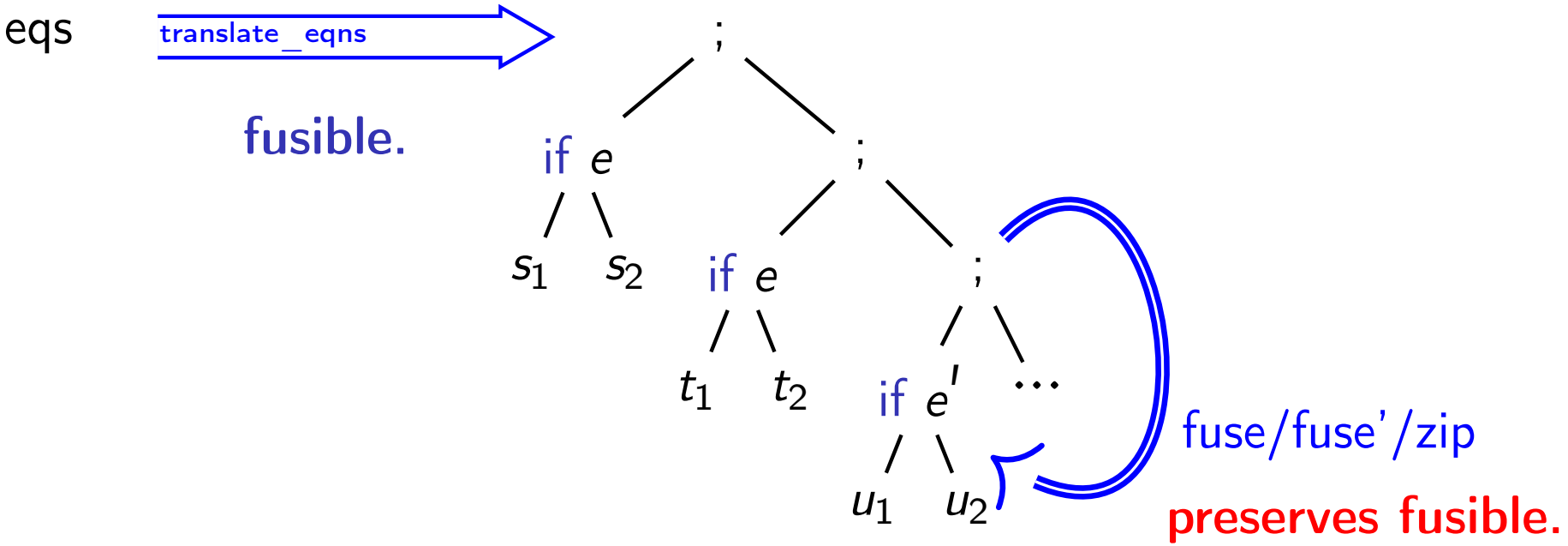
Fusion of control structures: correctness



- Define $s_1 \approx_{eval} s_2$

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

Fusion of control structures: correctness



- Define $s_1 \approx_{eval} s_2$
- Define $s_1 \approx_{fuse} s_2$ as
  $s_1 \approx_{eval} s_2 \wedge \text{fusionnable}(s_1) \wedge \text{fusionnable}(s_2)$
- Show congruence ('Proper' instances) for ;/fuse/fuse'/zip.

- Rewrite until $\dfrac{\text{fusionnable}(s)}{\text{fuse}(s) \approx_{eval} s}$

Overview

Lustre: syntax and semantics

Lustre: normalization

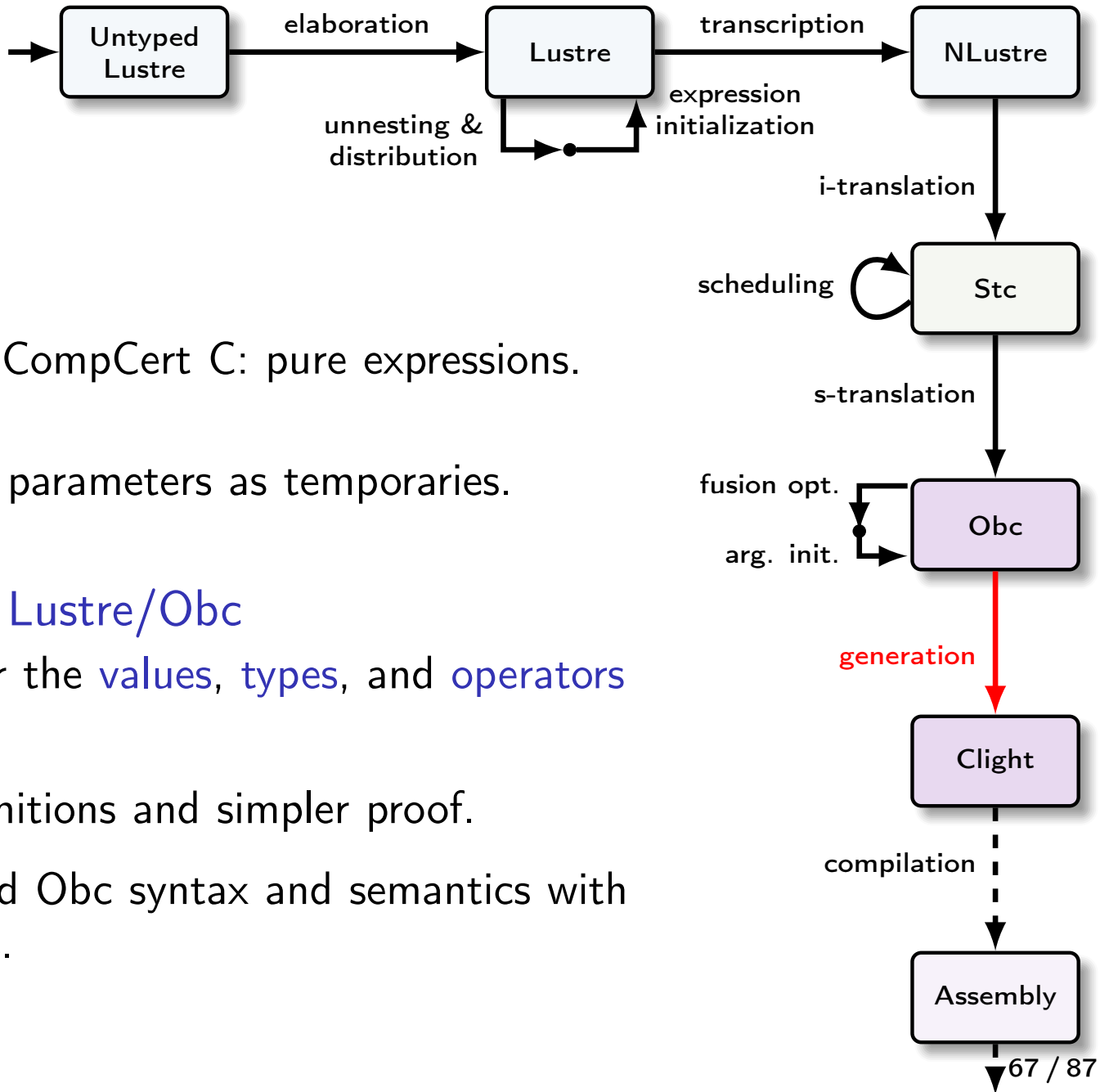
Translation: from dataflow programs to imperative code

Optimization: control structure fusion

Generation: from Obc to Clight

Conclusion

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.
```

Language design: Operator types

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

Language design: Operator types

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

Language design: Operator types

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

Language design: Operator types

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

Language design: Operator types

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

Language design: Operator types

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

Obc

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

x := y
```

Clight

```
unsigned char x;
int y;

x = y;
```

Language design: Operator types

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

Obc

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

x := y
```

Clight

```
unsigned char x;
int y;

x = y;
```

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

Language design: Operator types

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

Obc

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

x := (y : uint8)
```

Clight

```
unsigned char x;
int y;

x = y;
```

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

- Choice: no implicit casting in Obc.
- Explicit casts simplify substitution (referential transparency).
- Alternative: extend `OPERATORS` with an explicit operation for assignment

  `store : type → type → val → val`

Language design: operator types (unfortunate consequences)

```
node pita(x: signed char; y: signed char)
  returns (z: signed char)
  var w : signed char;
let
  w = x + y;
  z = (w / 2) + x;
tel
```

Language design: operator types (unfortunate consequences)

```
node pita(x: signed char; y: signed char)
  returns (z: signed char)
  var w : signed char;
let
  w = x + y;
  z = (w / 2) + x;
tel
node pita(x: signed char; y: signed char)
  returns (z: signed char)
  var w : signed char;
let
  w = (x + y : signed char);
  z = ((w / (2 : signed char)) + x : signed char);
tel
```

Language design: operator types (unfortunate consequences)

```
node pita(x: signed char; y: signed char)
  returns (z: signed char)
  var w : signed char;
let
  w = x + y;
  z = (w / 2) + x;
tel
```

```
node pita(x: signed char; y: signed char)
  returns (z: signed char)
  var w : signed char, t1 : int;
let
  w = (x + y : signed char);
  t1 = w / (2 : signed char);
  z = (t1 + x : signed char);
tel
```

```
node pita(x: signed char; y: signed char)
  returns (z: signed char)
  var w : signed char;
let
  w = (x + y : signed char);
  z = ((w / (2 : signed char)) + x : signed char);
tel
```

Operator types: bool

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

Operator types: bool

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

Operator types: bool

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

Obc

```
var x, y : bool;

x := y
```

explicit cast not mandated

Clight

```
_Bool x, y

x = y;
```

implicit cast: x = (_Bool) y

Operator types: bool

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

Obc

```
var x, y : bool;

x := y
```

explicit cast not mandated

Clight

```
_Bool x, y

x = y;
```

implicit cast: x = (_Bool) y

- The Clight type system is not strong enough for our purposes:
  - » There is no typing invariant on the memory.
  - » A _Bool is stored in 8-bits.
  - » E.g., no way to know that y does not contain 7 (without explicitly adding such an invariant).
  - » Problematic for the correctness proof.

Operator types: bool

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

Obc

var x, y : bool;

x := y

explicit cast not mandated

Clight

_Bool x, y

x = y;

implicit cast: x = (_Bool) y

- The Clight type system is not strong enough for our purposes:
  - » There is no typing invariant on the memory.
  - » A _Bool is stored in 8-bits.
  - » E.g., no way to know that y does not contain 7 (without explicitly adding such an invariant).
  - » Problematic for the correctness proof.
- We refine the types of operators and use a typing invariant.

$$(<) : \text{int} \to \text{int} \to \text{int} \implies (<) : \text{int} \to \text{int} \to \text{bool}$$
$$(\&) : \text{bool} \to \text{bool} \to \text{int} \implies (\&) : \text{bool} \to \text{bool} \to \text{bool}$$

Operator domains

- Partial operators:
  - integer division/modulo x/y, $x \,\%\, y$: $y = 0 \vee (x = \mathrm{INT\_MIN} \wedge y = -1)$
  - shifts $x \ll y$, $x \gg y$: $y < 0 \vee y \geq 32$

- 'Dynamic' precondition in the existence proof.
  $(\forall i,$ `sem_binop_i <> None`$)$

- Alternative `OPERATORS` implementation and translation:
  x / y becomes **if** x != INT_MIN && y != 0 then x / y **else** 0

Unused types

```
Inductive type : Type :=
  | Tvoid: type                                          (**r the [void] type *)
  | Tint: intsize → signedness → attr → type             (**r integer types *)
  | Tlong: signedness → attr → type                      (**r 64-bit integer types *)
  | Tfloat: floatsize → attr → type                      (**r floating-point types *)
  | Tpointer: type → attr → type                         (**r pointer types ([*ty]) *)
  | Tarray: type → Z → attr → type                       (**r array types ([ty[len]]) *)
  | Tfunction: typelist → type → calling_convention → type    (**r function types *)
  | Tstruct: ident → attr → type                         (**r struct types *)
  | Tunion: ident → attr → type                          (**r union types *)
```

- We only use a very limited subset of Clight (Obc is much simpler).
- How to treat variant types in our compiler?
- How to treat arrays in our compiler?

Unknown values

- In Clight (`Ctyping`): `∀ ty, wt_val Vundef ty`
- In Obc: `Vundef` is never well typed, `None` represents uninitialized values.
- Assuming the semantics of an Obc program precludes ever reading `None`.

```
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
  }
}
```

```
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;
}
```

- 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
  }
}
```

```
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;
}
```

- 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
  }
}
```

```
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;
}
```

- 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
  }
}
```

```
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;
}
```

- 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
  }
}
```

```
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;
}
```

- 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;
}
```

CompCert's Clight frontend

Formal model

context:	genv	ident → {type, function, global}
state:	m	block → offset → memval × permission
	e	ident → block × type
	le	ident → val

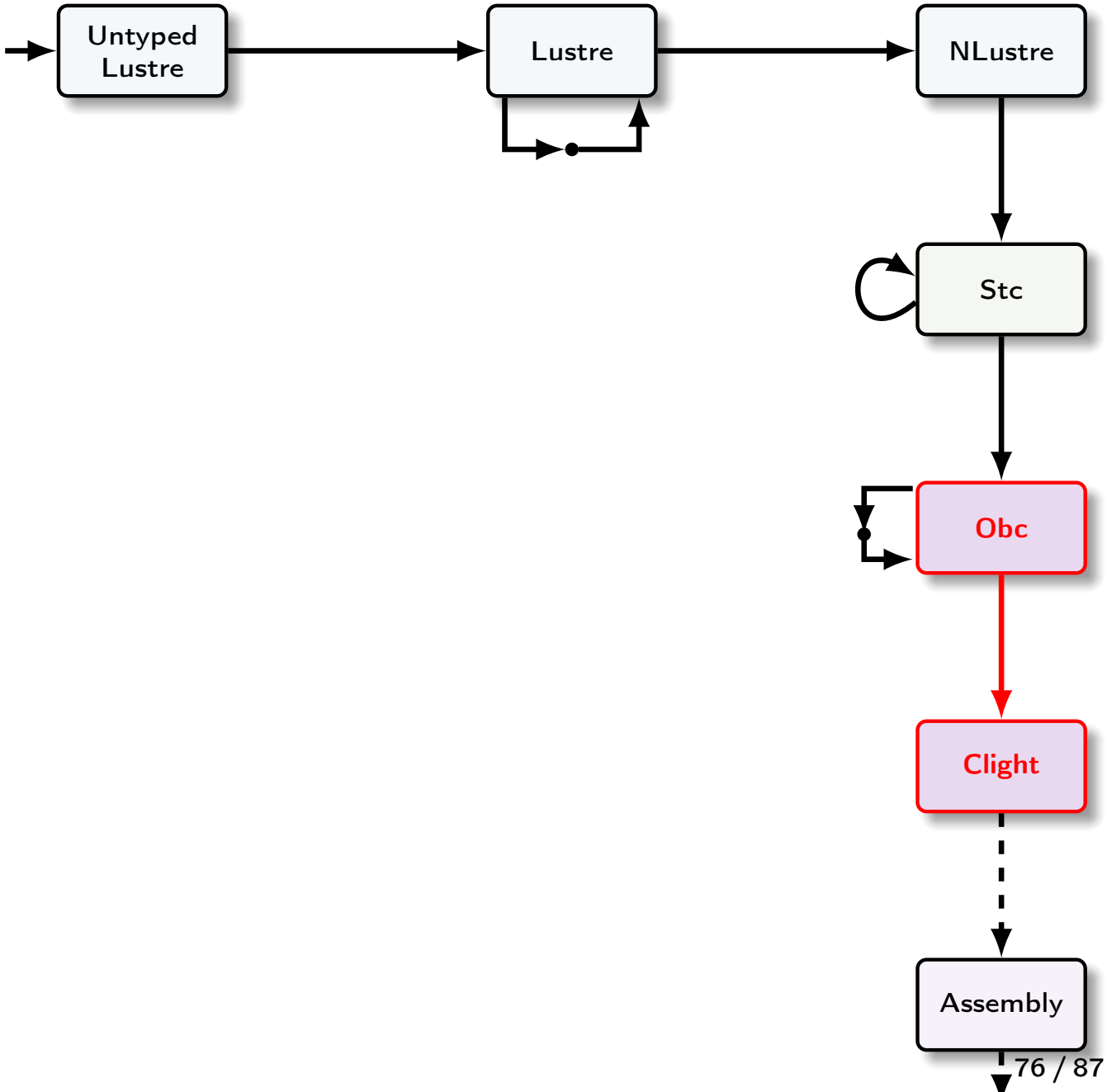
Four semantic models:

	func. params in m	*func. params in le*
$(m, e, le, s) \to (m', e', le', s')$	clight1	clight2
$genv, e \vdash m, le, s \Downarrow m', le'$	bigstep clight1	bigstep clight2

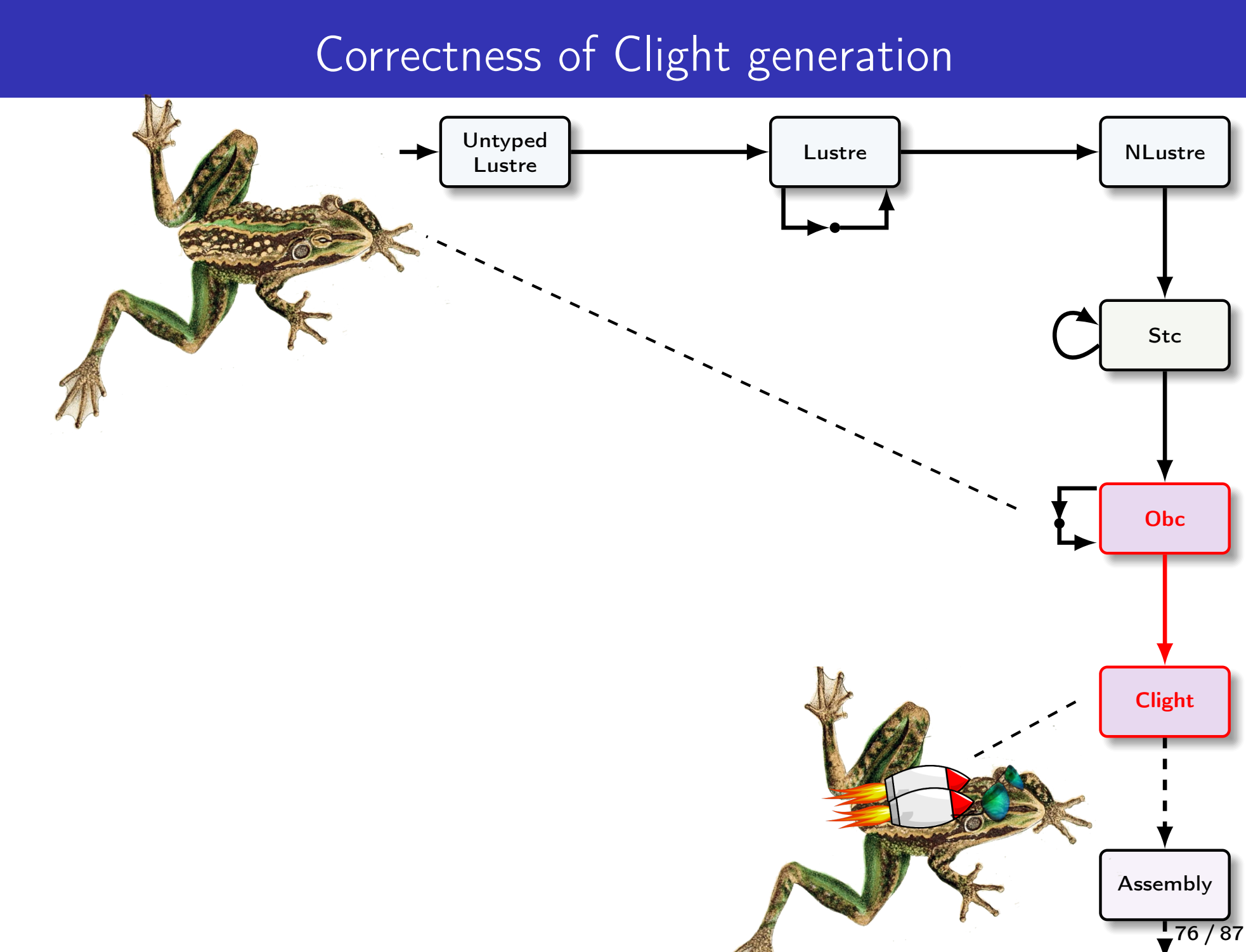
bigstep clight2 $\Longrightarrow$ clight2 $\Longrightarrow$ clight1 and bigstep clight1 $\Longrightarrow$ clight1

- Model of the C 'abstract machine' (standard + implementation).
- Data representations, types, and operators.
- Byte-level memory model: load(), store(), alloc(), and free().

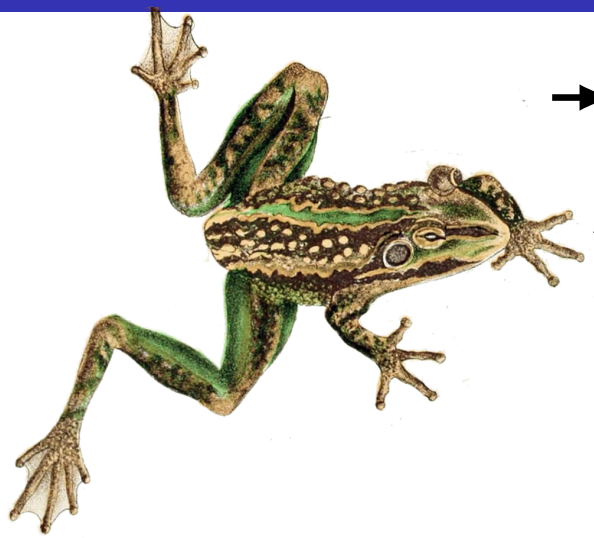
Correctness of Clight generation

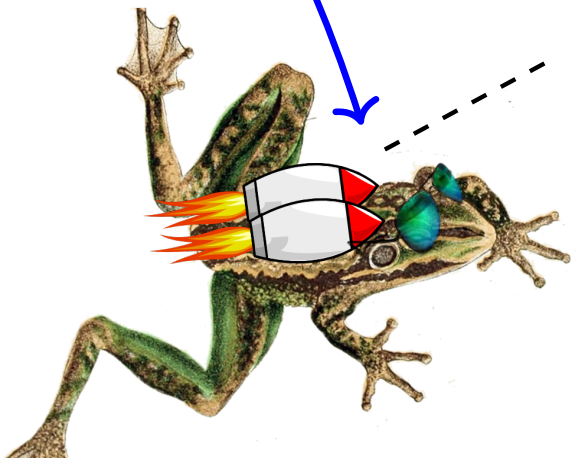


Correctness of Clight generation



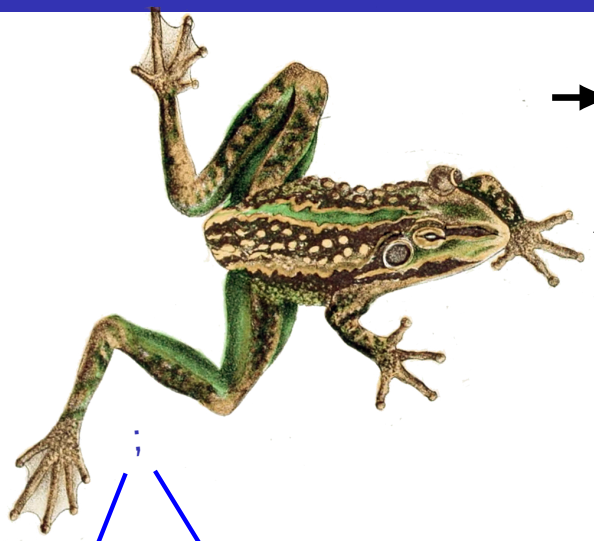
Correctness of Clight generation

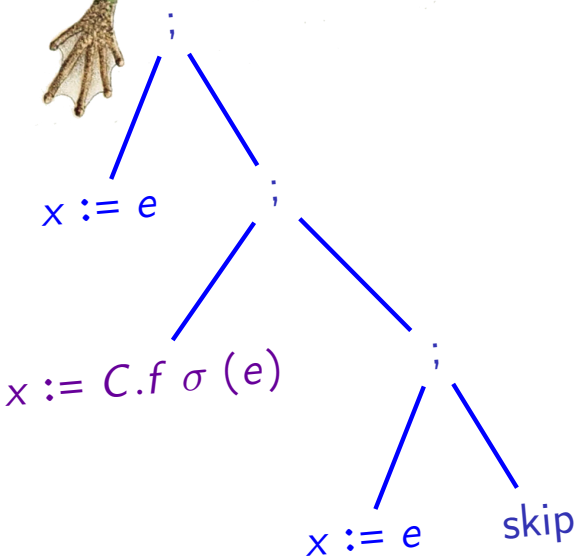


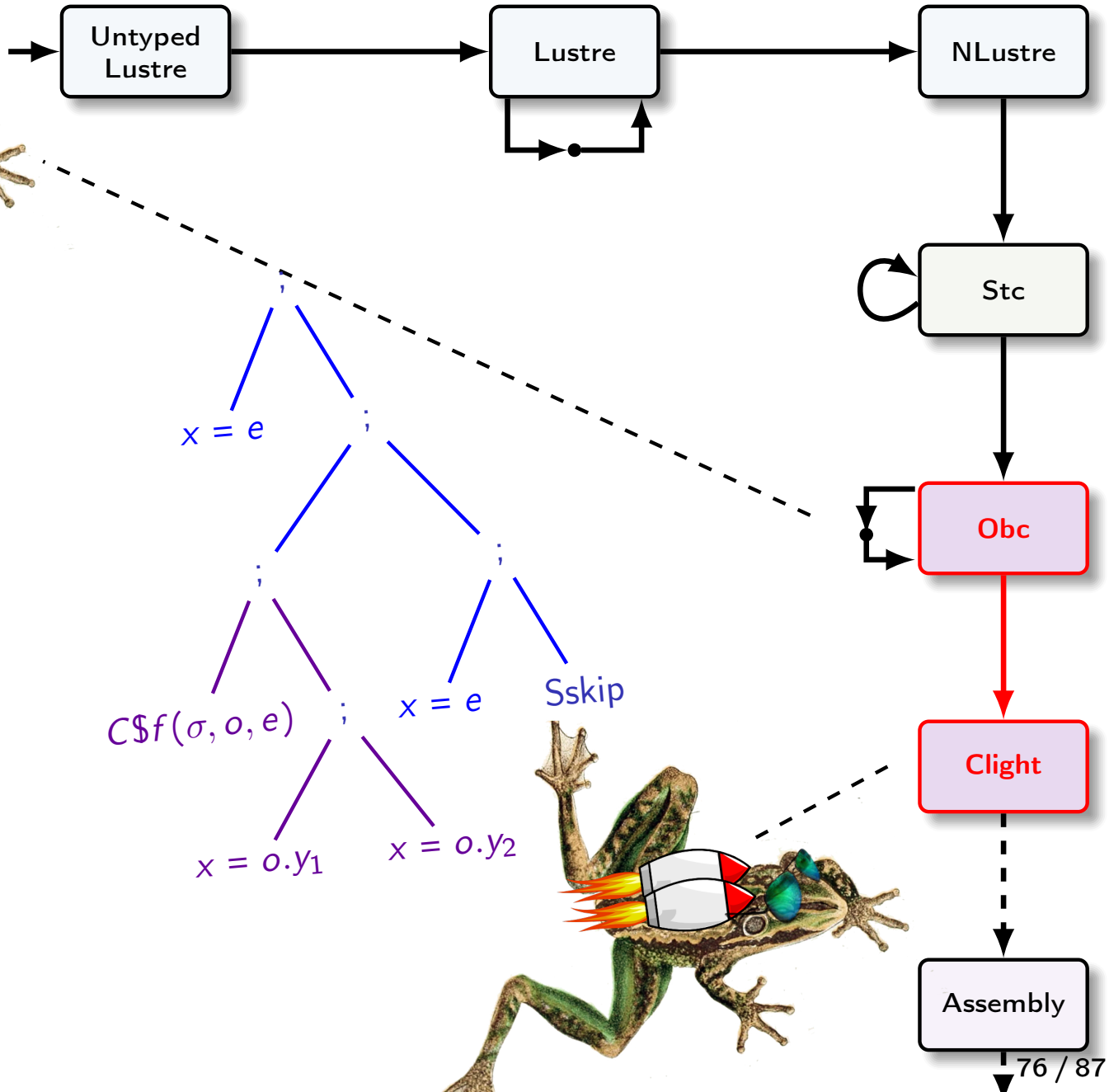


Direct proof by induction
on the big-step semantics

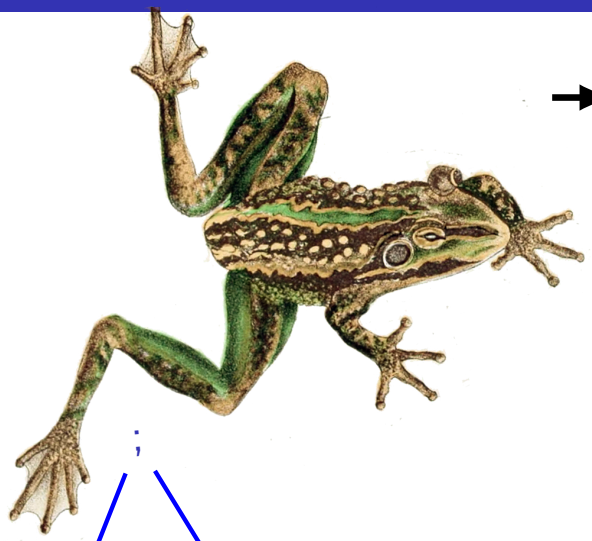
Correctness of Clight generation

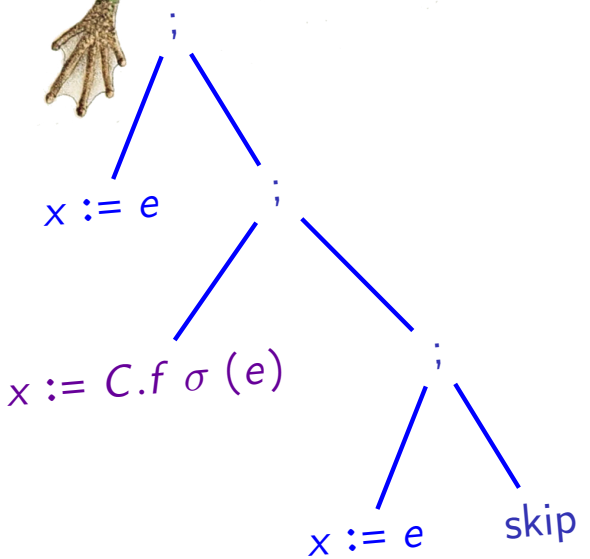


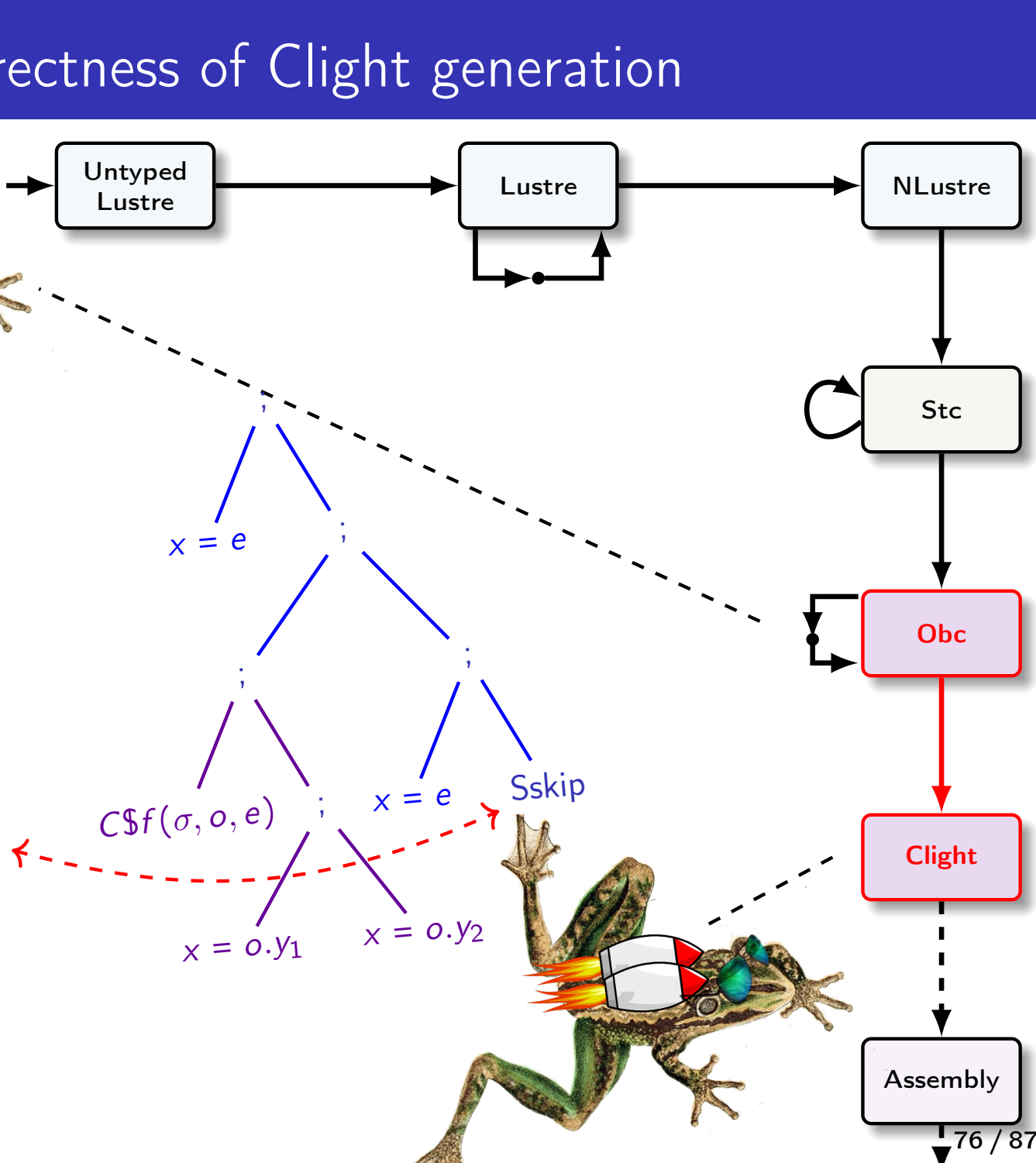




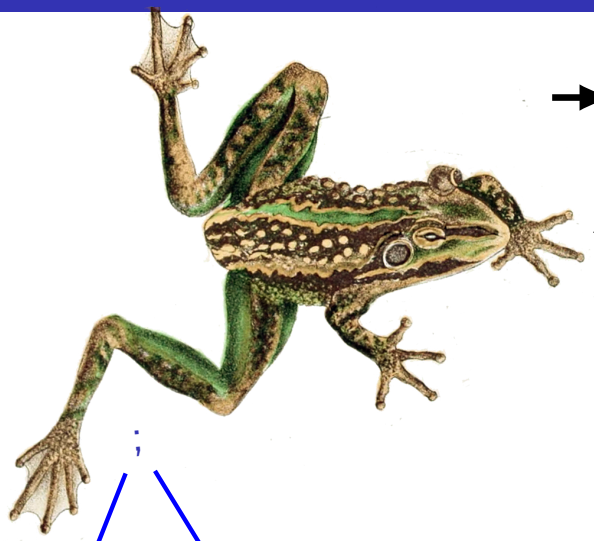
Correctness of Clight generation

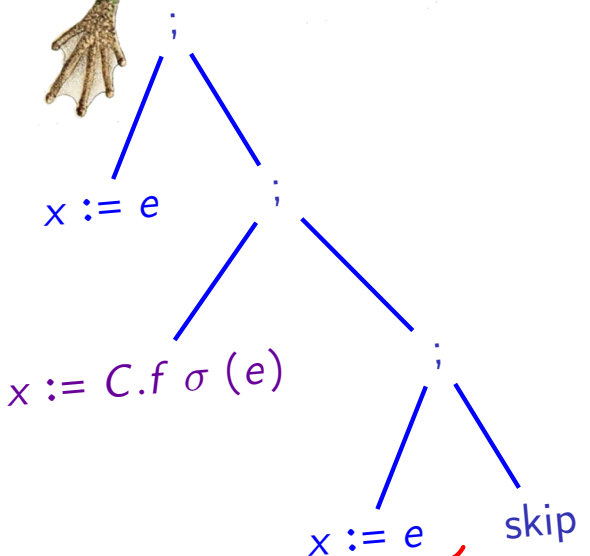


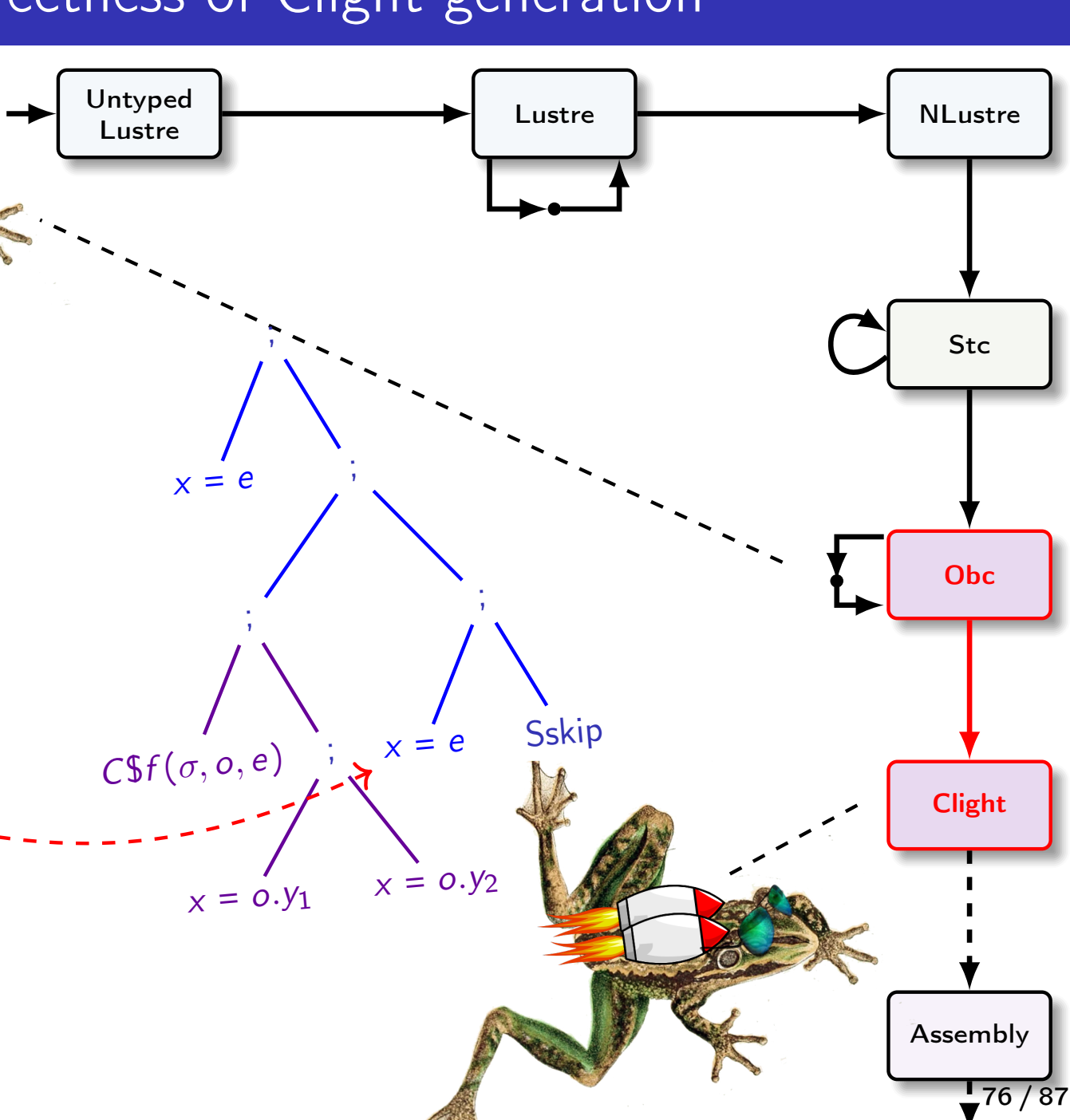




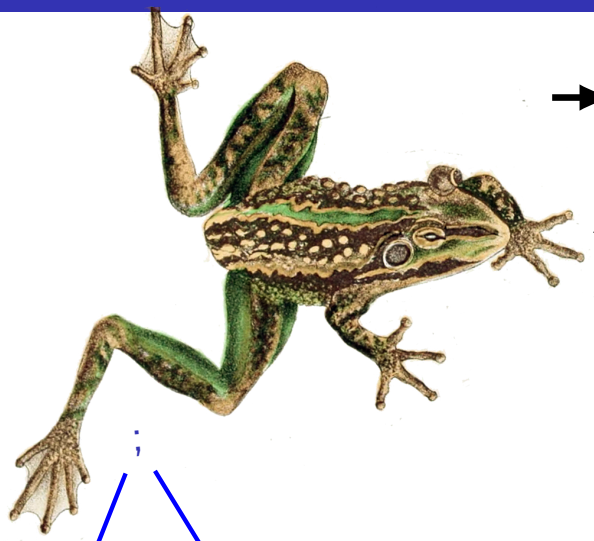
Correctness of Clight generation

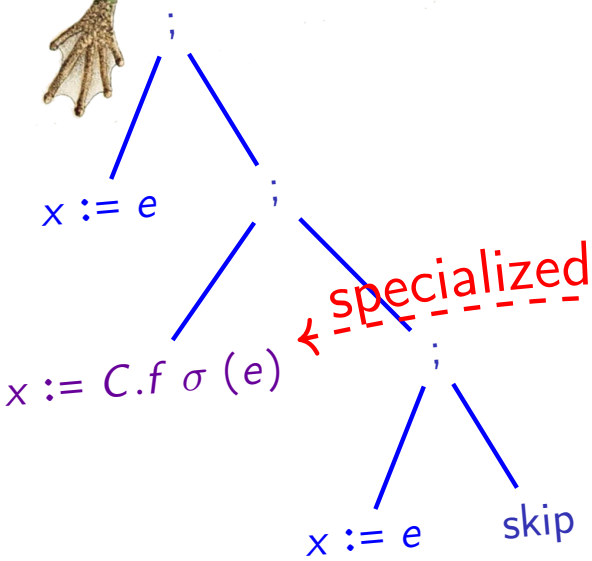




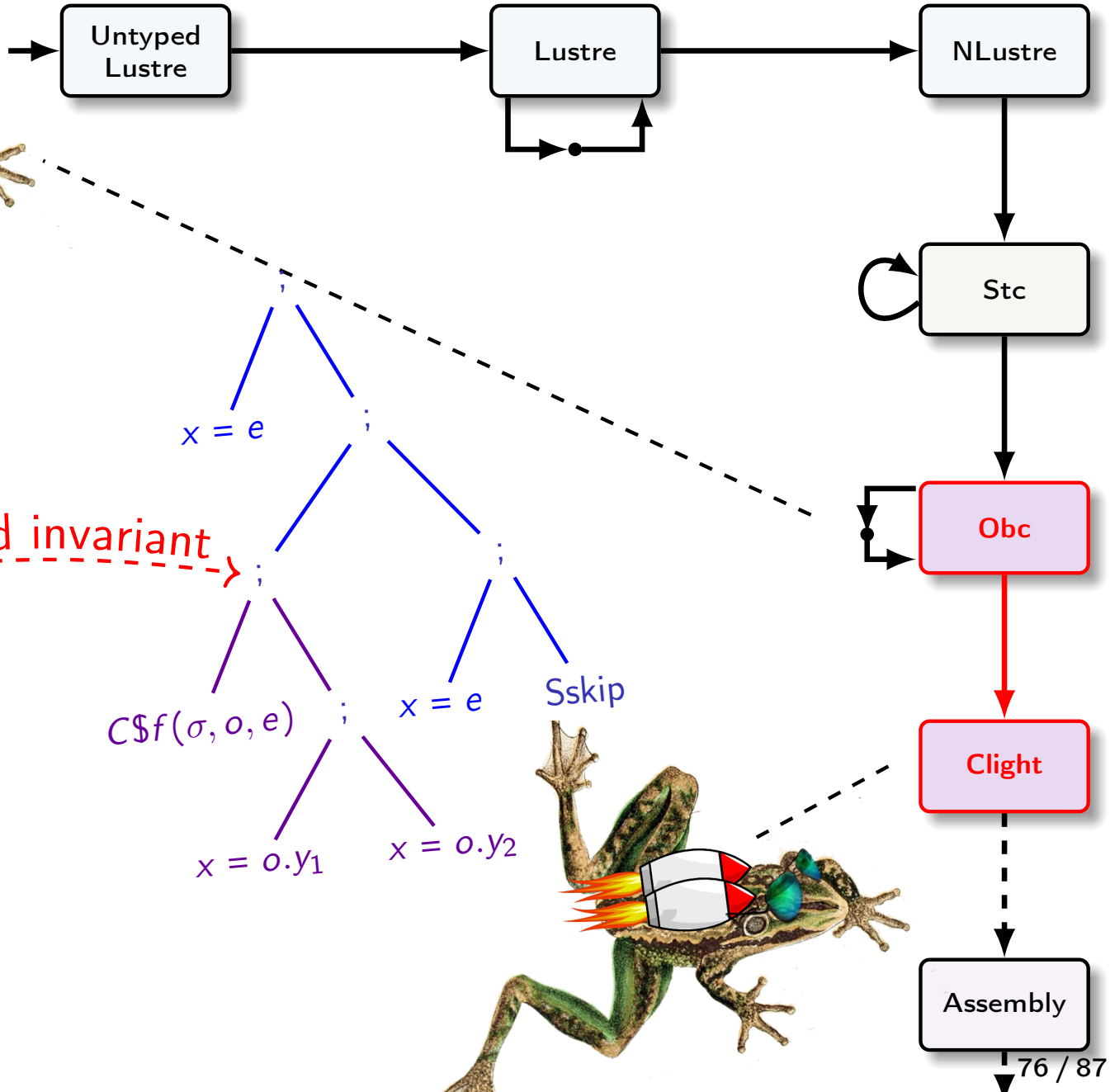


Correctness of Clight generation

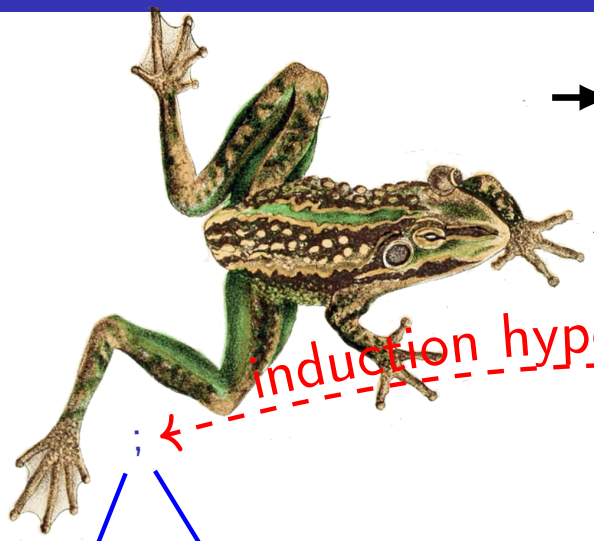




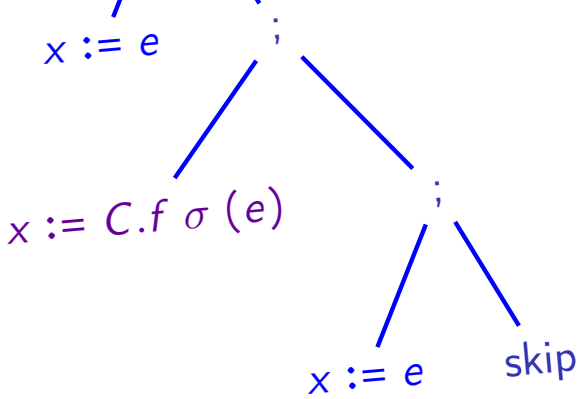
specialized invariant

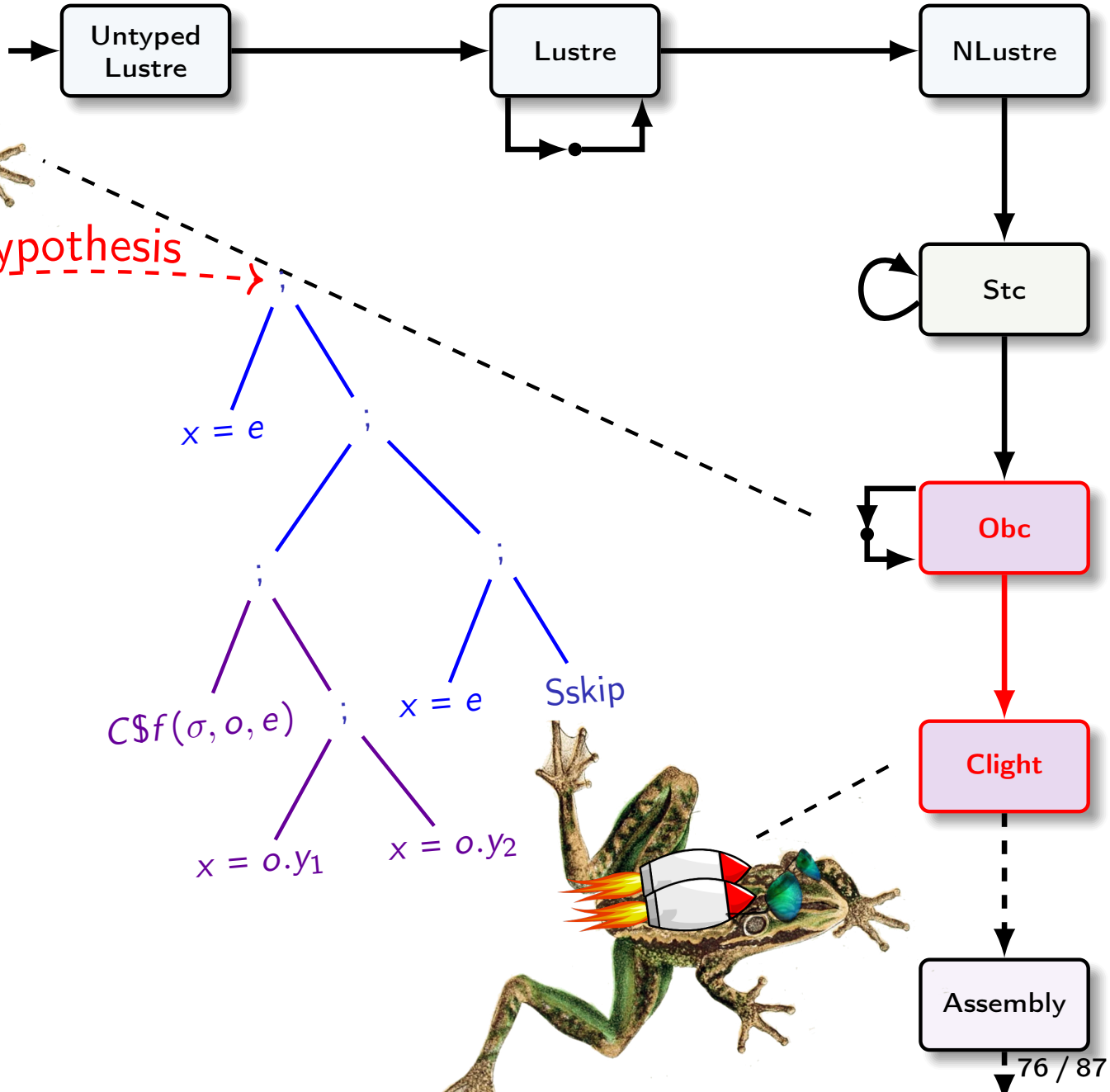


Correctness of Clight generation



induction hypothesis



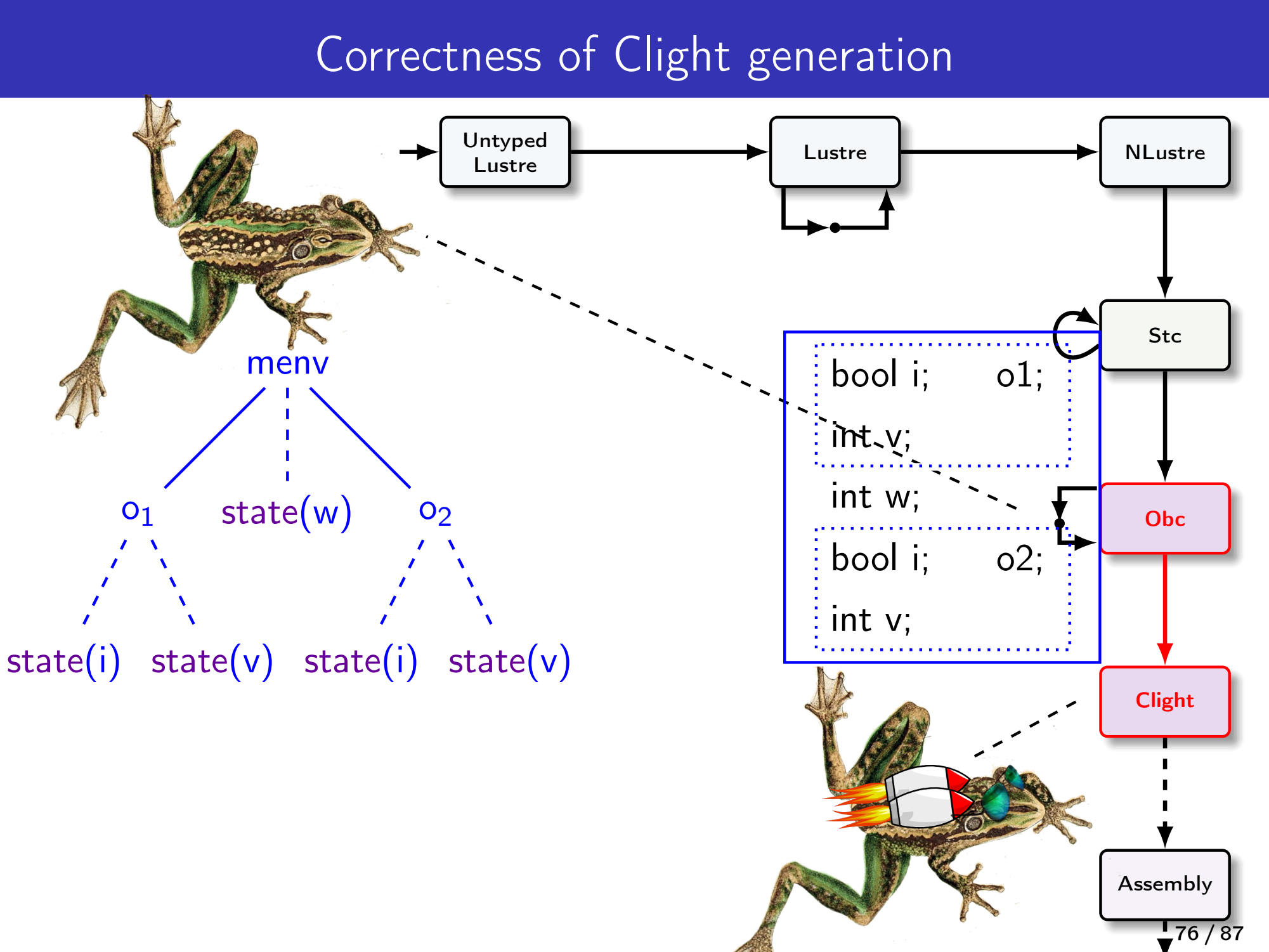


Correctness of Clight generation

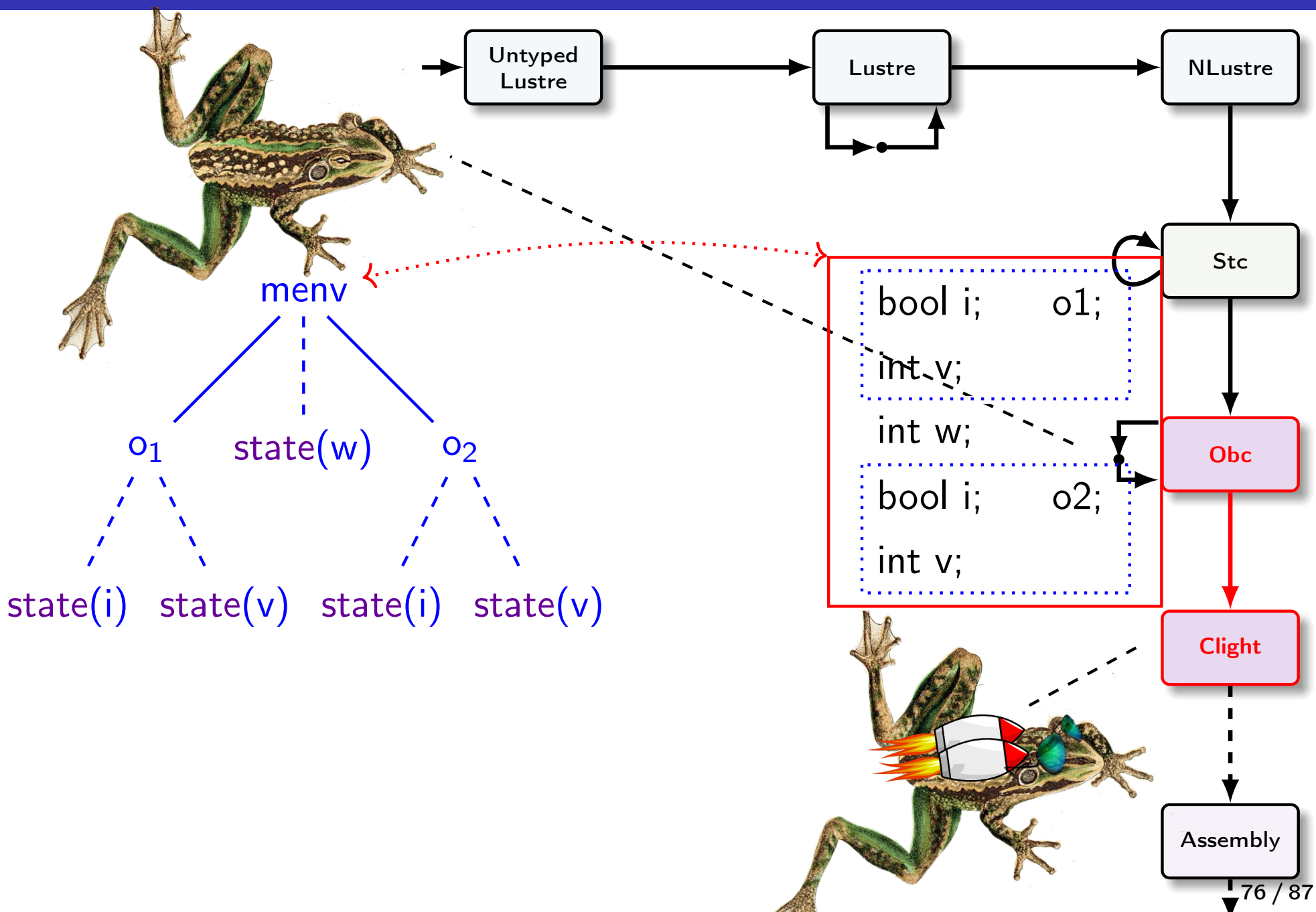
menv

o_1 state(w) o_2

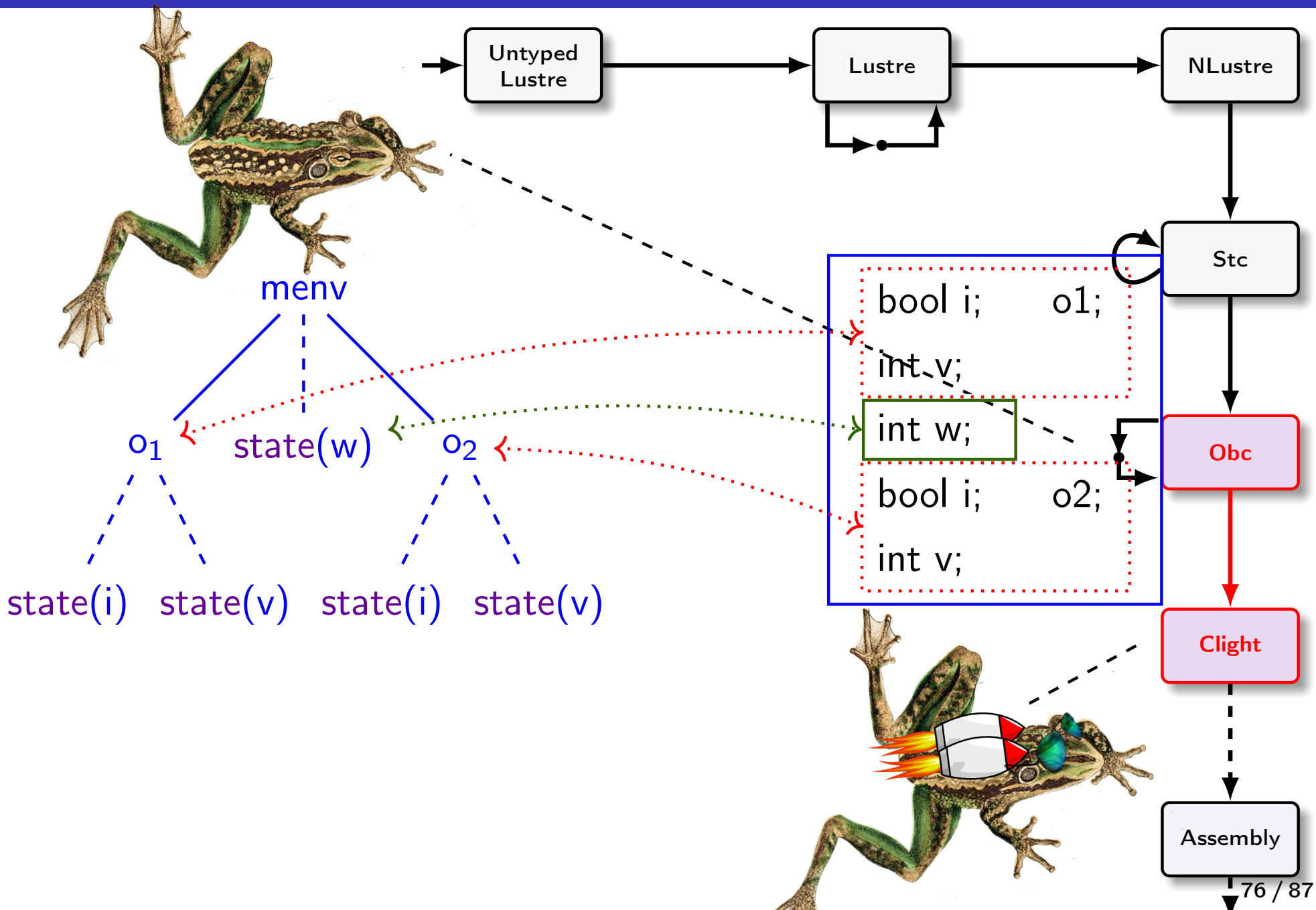
state(i) state(v) state(i) state(v)



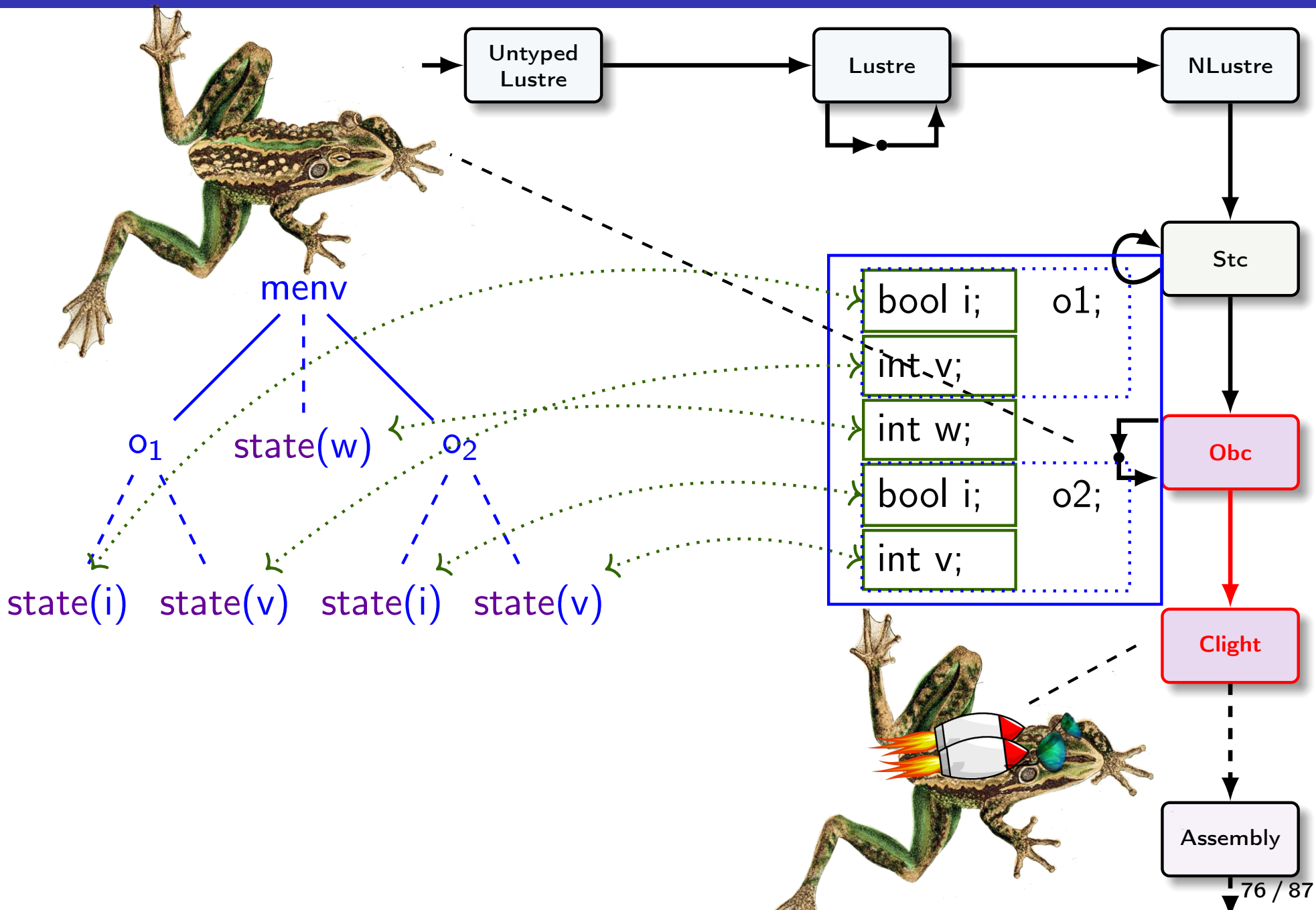
Correctness of Clight generation



Correctness of Clight generation

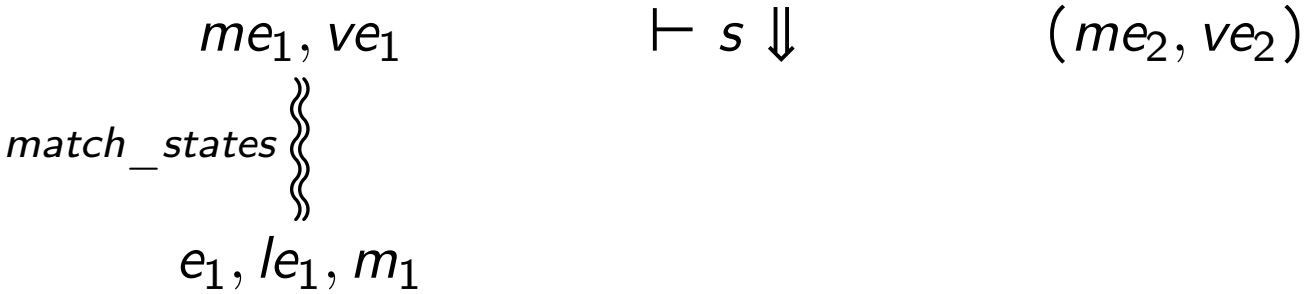


Correctness of Clight generation



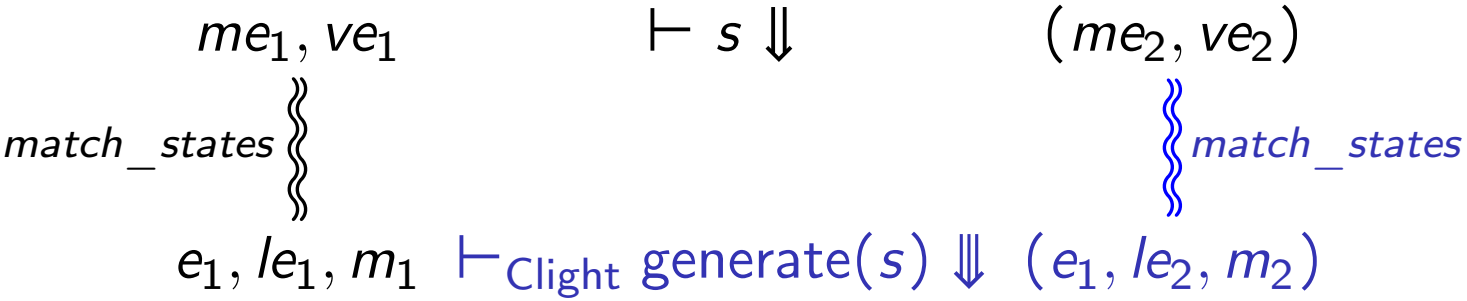
Semantics preservation

Obc : (me, ve) ; Clight : (e, le, m)

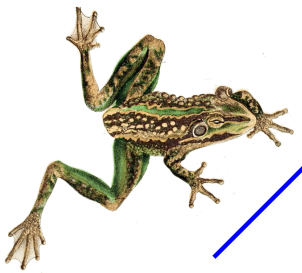


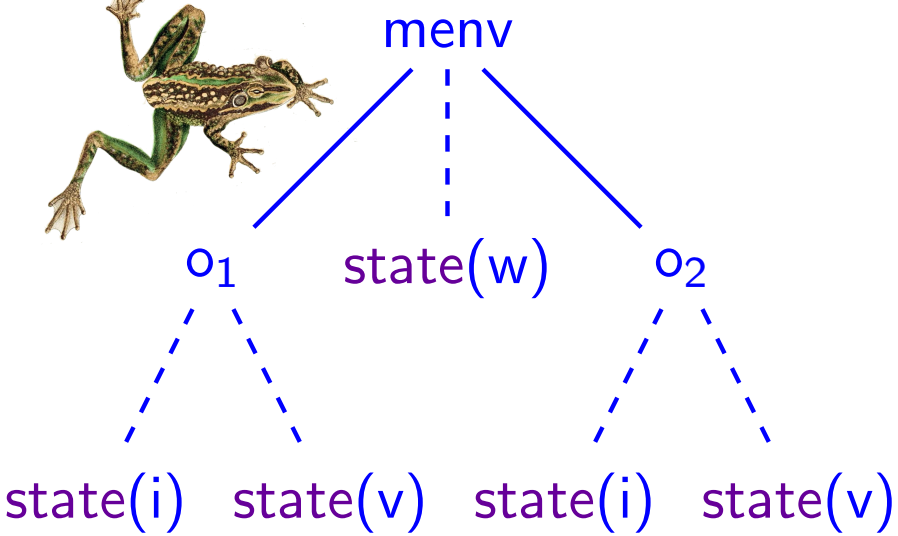
Semantics preservation

Obc : (me, ve) ; Clight : (e, le, m)

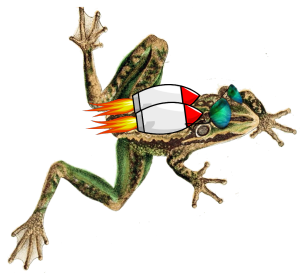


Obc to Clight: memory correspondence






```
bool i;     o1;
int v;
int w;
bool i;     o2;
int v;
```



- This time the semantic models are similar (Clight: very detailed)
- The real challenge is to relate the memory models.
  - » Obc: tree structure, variable separation is manifest.
  - » Clight: block-based, must treat aliasing, alignment, and sizes.

- Extend CompCert's lightweight library of separating assertions: `https://github.com/AbsInt/CompCert/common/Separation.v`.
- Encode simplicity of source model in richer memory model.
- General (and very useful) technique for interfacing with CompCert.

Separation logic in CompCert

predicate

$$massert \triangleq \left\{ \begin{array}{l} \text{pred} : memory \to \mathbb{P} \\ \text{foot} : block \to int \to \mathbb{P} \\ \text{invar} : \forall\, m\, m', \text{pred}\ m \to \\ \qquad\qquad \text{unchanged\_on foot}\ m\ m' \to \\ \qquad\qquad \text{pred}\ m' \end{array} \right\}$$

notation: $m \vDash P \triangleq P.\text{pred}\ m$

conjonction

$$P * Q \triangleq \left\{ \begin{array}{l} \text{pred} = \lambda\, m.\ (m \vDash P) \wedge (m \vDash Q) \\ \qquad\qquad \wedge\ \text{disjoint}\ P.\text{foot}\ Q.\text{foot} \\ \text{foot} = \lambda\, b\, ofs.\ P.\text{foot}\, b\, ofs \vee Q.\text{foot}\, b\, ofs \end{array} \right\}$$

pure formula $m \vDash \text{pure}\,(P) * Q \leftrightarrow P \wedge m \vDash Q$

```
(* Xavier's Separation.v *)

Record massert : Type := { m_pred : mem → Prop;
                           m_footprint : block → Z → Prop; ... }.

Notation "m |= p" := (m_pred p m) : sep_scope.

Definition disjoint_footprint (P Q: massert) : Prop :=
  ∀ b ofs, m_footprint P b ofs → m_footprint Q b ofs → False.

Definition sepconj (P Q: massert) : massert := {|
  m_pred := fun m ⇒ m_pred P m ∧ m_pred Q m ∧ disjoint_footprint P Q;
  m_footprint := fun b ofs ⇒ m_footprint P b ofs ∨ m_footprint Q b ofs |}.

Infix "**" := sepconj : sep_scope.

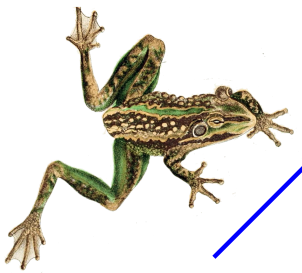
(* Blockrep *)

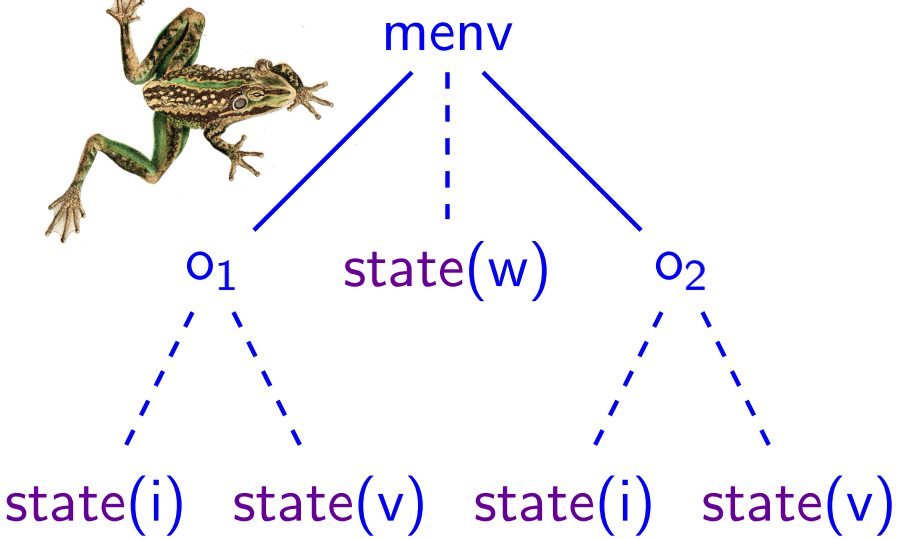
Fixpoint sepall (p: A → massert) (xs: list A) : massert := match xs with
                                                           | nil ⇒ sepemp
                                                           | x:: xs ⇒ p x ** sepall p xs
                                                           end.

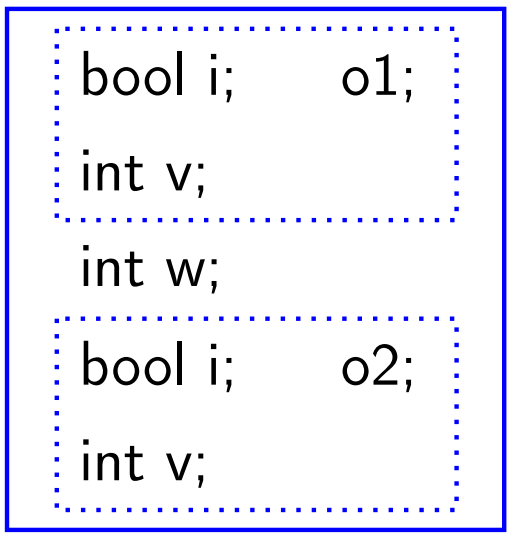
Definition match_value (e: PM.t val) (x: ident) (v': val) : Prop := match PM.find x e with
                                                                    | None ⇒ True
                                                                    | Some v ⇒ v' = v
                                                                    end.

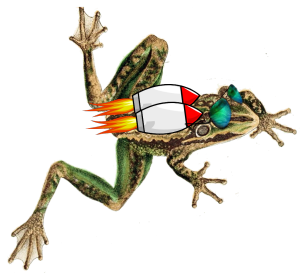

Definition blockrep (ve: venv) (flds: members) (b: block) : massert :=
  sepall (fun (x, ty) : ident * type ⇒
            match field_offset ge x flds, access_mode ty with
            | OK d, By_value chunk ⇒ contains chunk b d (match_value ve x)
            | _, _ ⇒ sepfalse
            end) flds.
```

Obc to Clight: memory correspondence





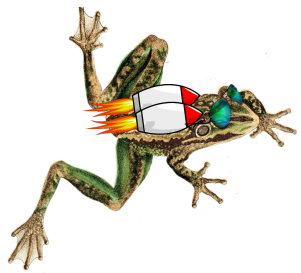


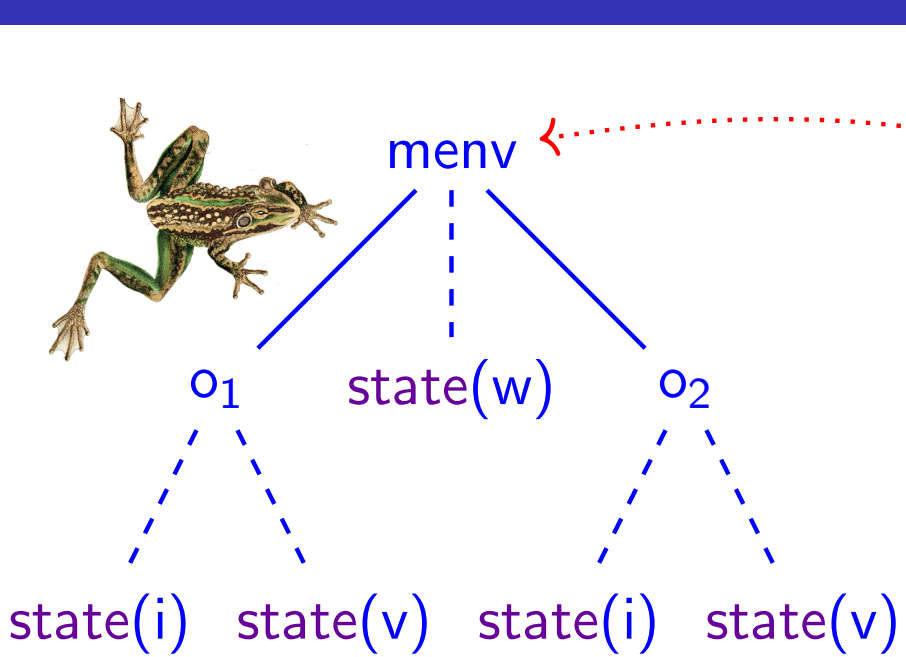


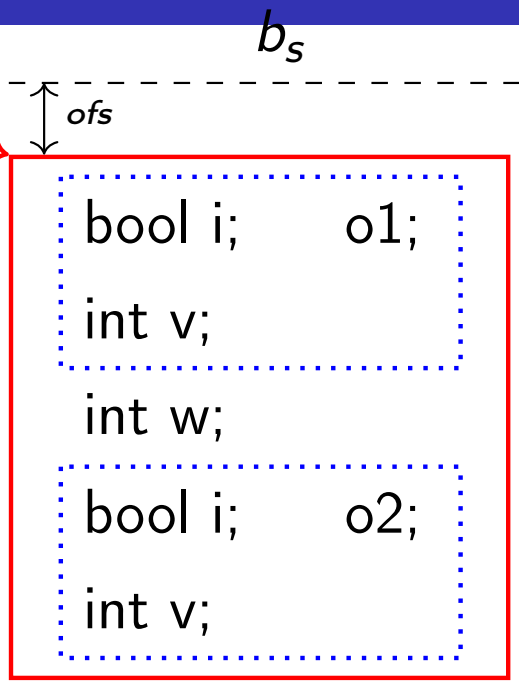
- This time the semantic models are similar (Clight: very detailed)
- The real challenge is to relate the memory models.
  - » Obc: tree structure, variable separation is manifest.
  - » Clight: block-based, must treat aliasing, alignment, and sizes.

- Extend CompCert's lightweight library of separating assertions: `https://github.com/AbsInt/CompCert/common/Separation.v`.
- Encode simplicity of source model in richer memory model.
- General (and very useful) technique for interfacing with CompCert.

Obc to Clight: memory correspondence

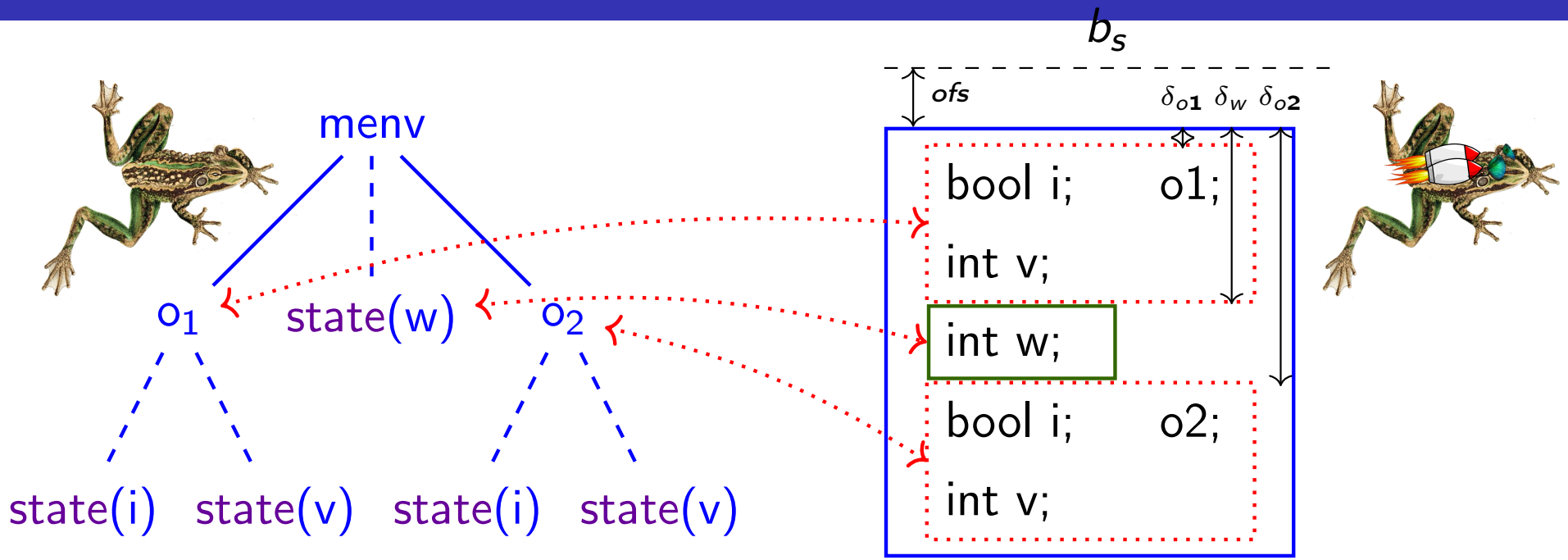






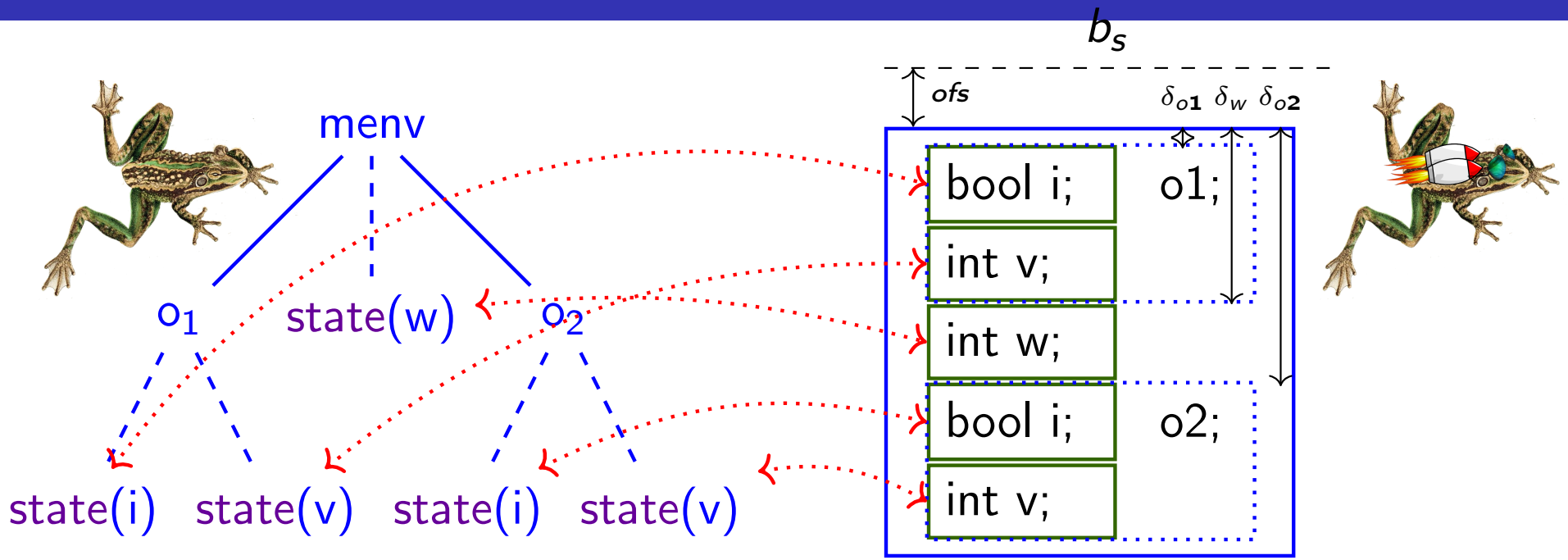
$m \vDash \text{staterep}\ \ \text{avgvelocity}\ \ me\ \ (b_s, ofs)$

Obc to Clight: memory correspondence



$$
\begin{aligned}
m \vDash \quad & \mathit{staterep}\ \ \mathit{count}\ \ \mathit{me}(o1)\ \ (b_s, \mathit{ofs} + \delta_{o1}) \\
& * \ \mathit{contains}\ \ \mathit{tyint32s}\ \ (b_s, \mathit{ofs} + \delta_w)\ \ \lceil \mathit{me}.\mathsf{state}(w) \rceil \\
& * \ \mathit{staterep}\ \ \mathit{count}\ \ \mathit{me}(o2)\ \ (b_s, \mathit{ofs} + \delta_{o2})
\end{aligned}
$$

Obc to Clight: memory correspondence



$$
\begin{array}{lllll}
m \vDash & \text{contains} & ty\,bool & (b_s, ofs + \delta_{o1} + \delta_i) & \lceil me.o_1.\text{state}(i) \rceil \\
& * \text{ contains} & ty\,int32s & (b_s, ofs + \delta_{o1} + \delta_v) & \lceil me.o_1.\text{state}(v) \rceil \\
& * \text{ contains} & ty\,int32s & (b_s, ofs + \delta_w) & \lceil me.\text{state}(w) \rceil \\
& * \text{ contains} & ty\,bool & (b_s, ofs + \delta_{o2} + \delta_i) & \lceil me.o_2.\text{state}(i) \rceil \\
& * \text{ contains} & ty\,int32s & (b_s, ofs + \delta_{o2} + \delta_v) & \lceil me.o_2.\text{state}(v) \rceil
\end{array}
$$

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.
```

Freeing memory/struct padding

- Keep permissions on local memory to 'free' it before return.
- Permissions are 'in' the `contains` predicate.
- But, the padding is not included. How can we free it?

$$\begin{aligned}\text{match\_states} = \forall\, & c\ f\ me\ ve\ e\ le\ sb\ sofs\ outb\ outco,\\ & \dots\\ & * \text{ subrep } f\ e\\ & * \text{ subrep } f\ e \mathrel{-\!\!*} \text{subrep\_range } e\end{aligned}$$

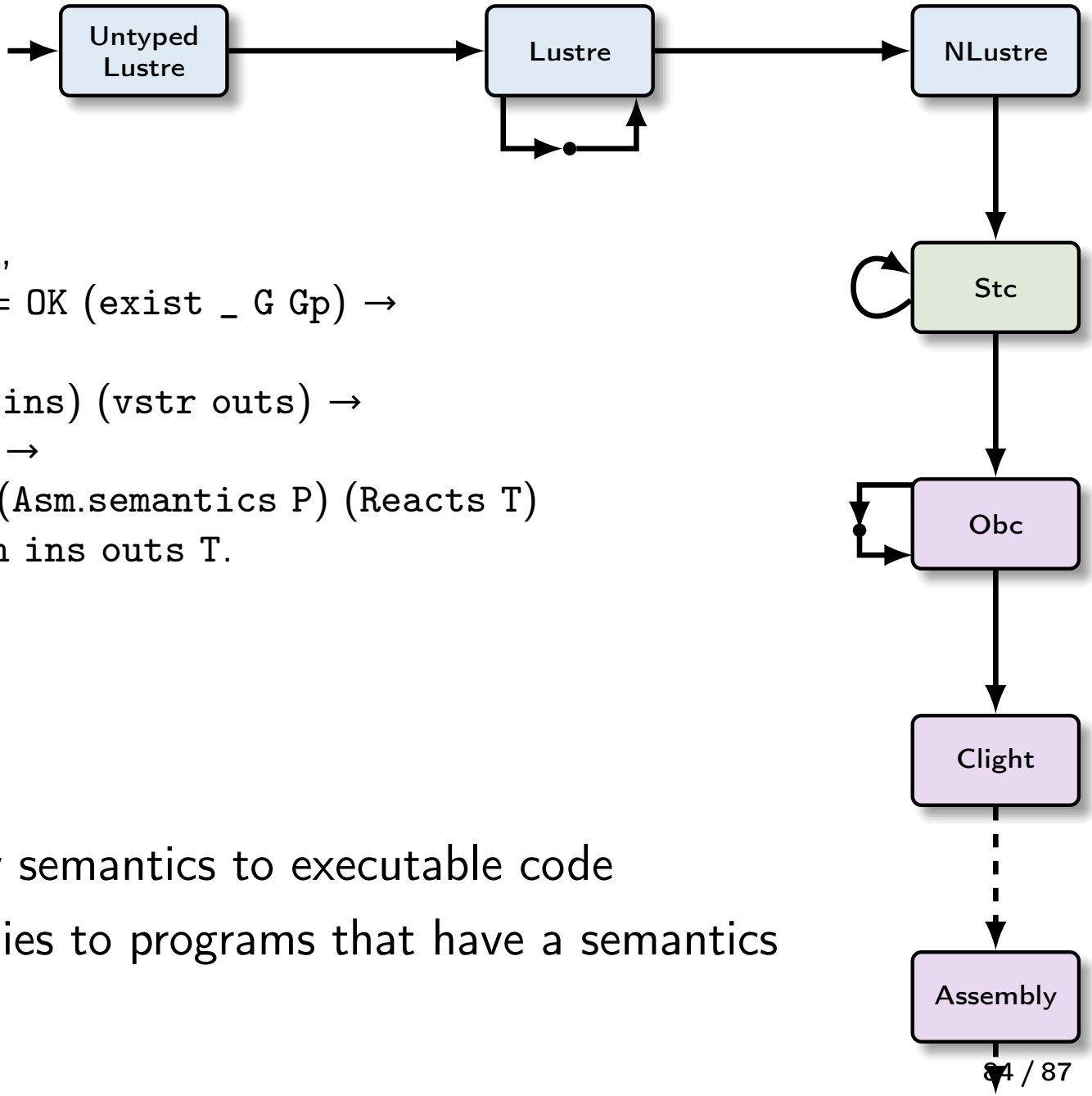
Separating wand

- Key property:

$$P * (P \mathrel{-\!\!*} Q) \Longrightarrow Q$$

- Works together with the frame condition on other lemmas.

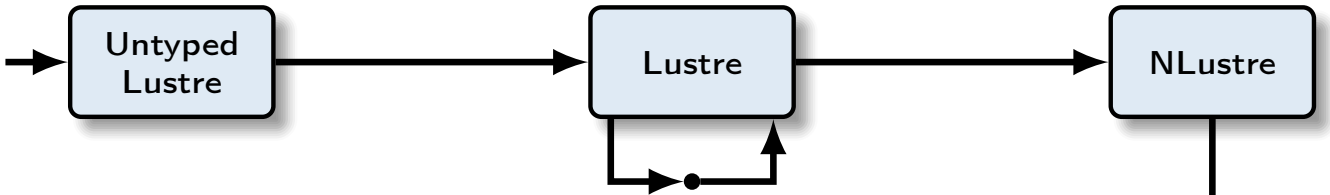
Main theorem




```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

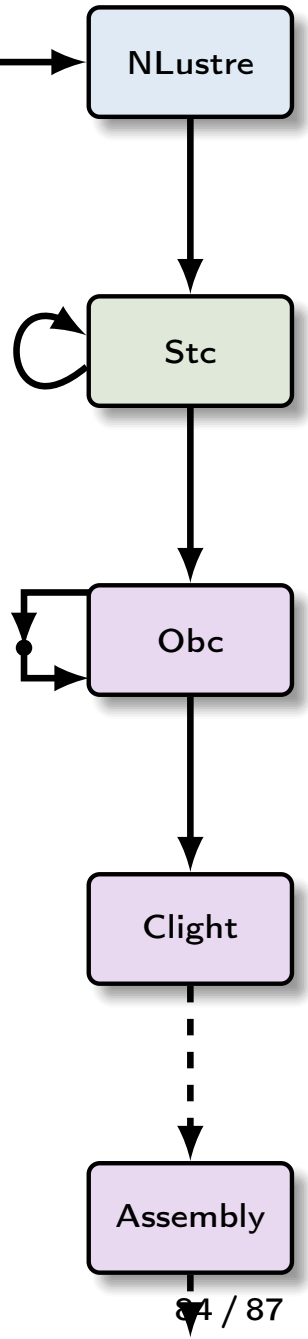
- Good: links dataflow semantics to executable code
- Less good: only applies to programs that have a semantics

Main theorem



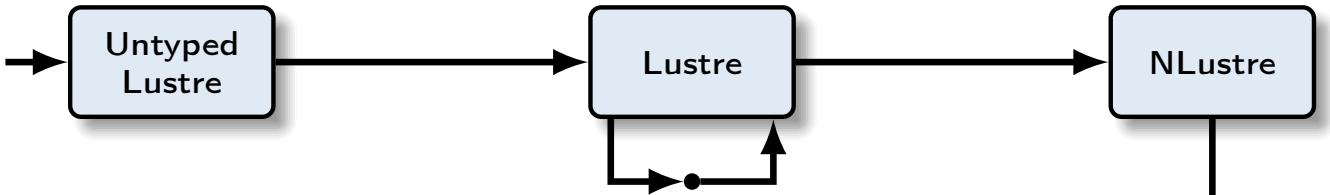
typing/elaboration succeeds,

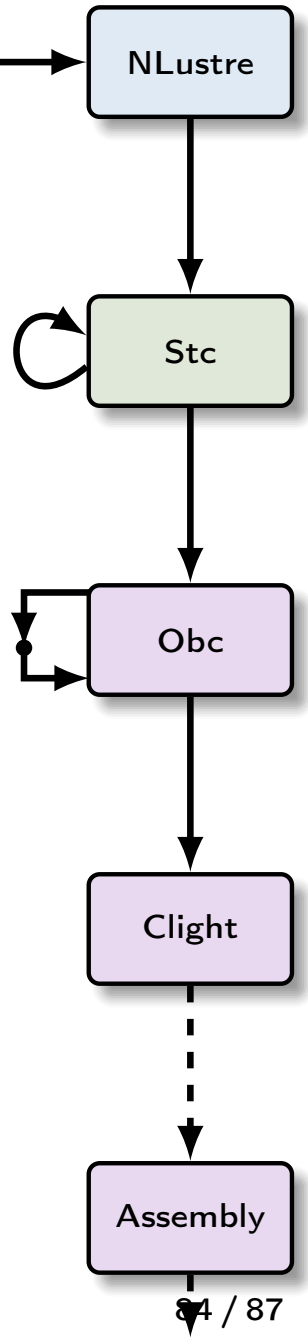
```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```



- Good: links dataflow semantics to executable code
- Less good: only applies to programs that have a semantics

Main theorem





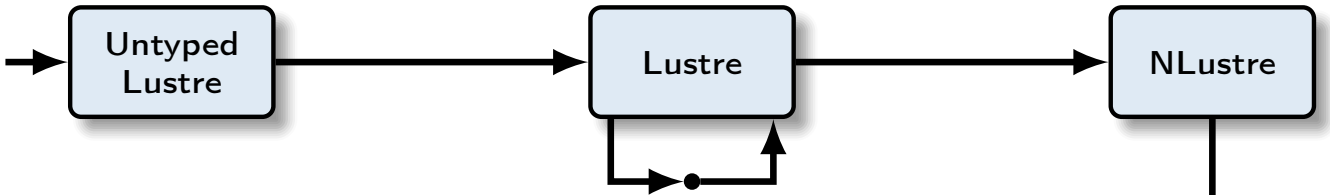

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

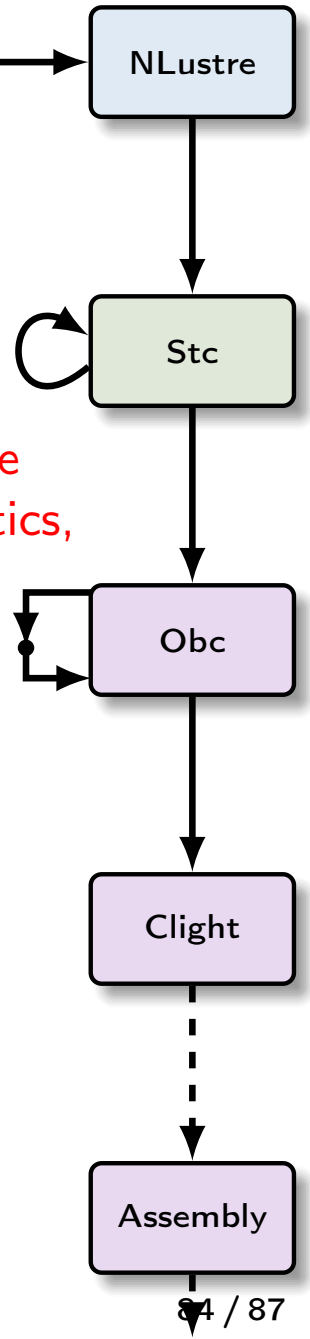
typing/elaboration succeeds,

– ∀ well-typed input streams. . .

- Good: links dataflow semantics to executable code
- Less good: only applies to programs that have a semantics

Main theorem






```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

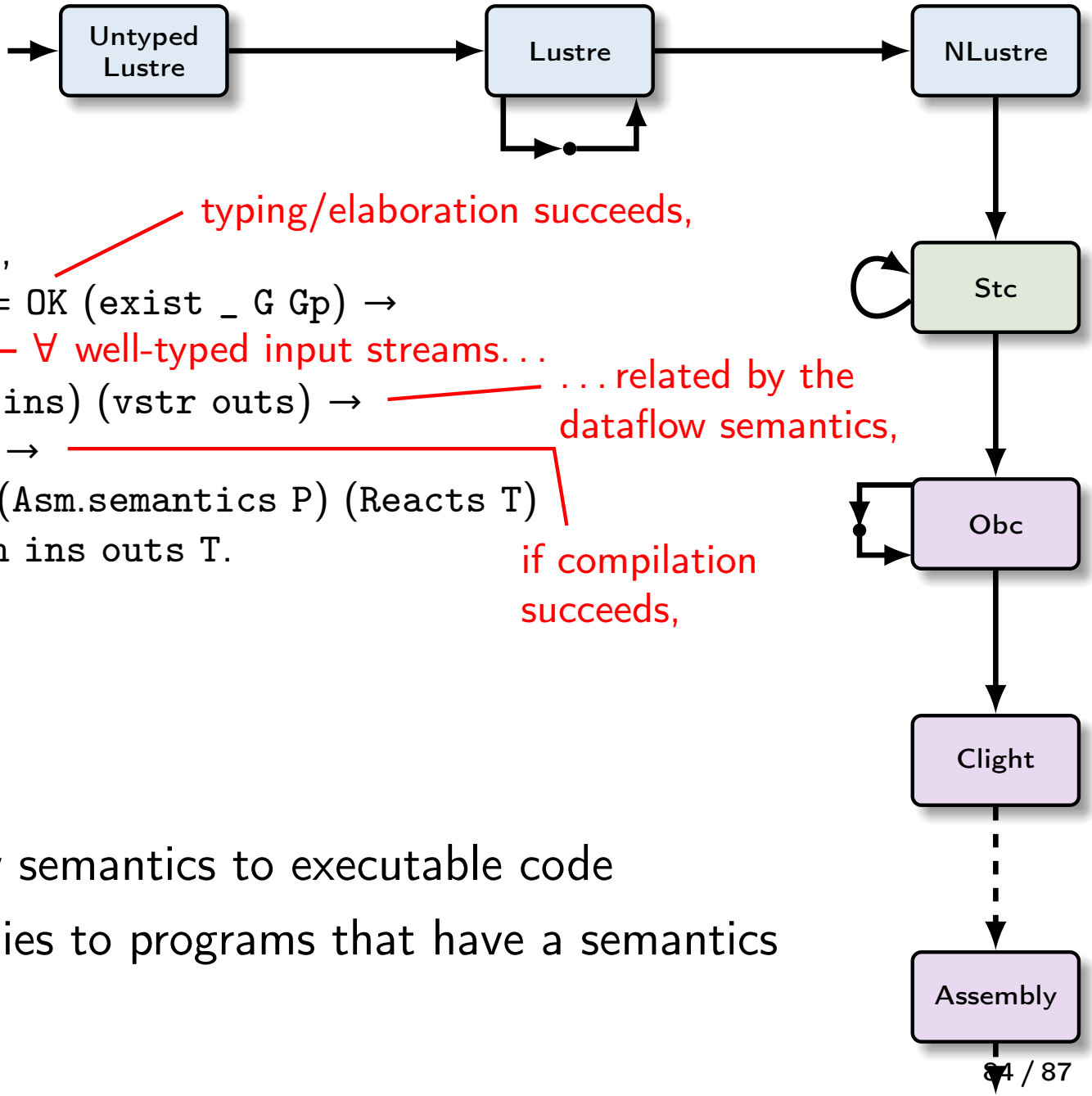
typing/elaboration succeeds,

– ∀ well-typed input streams...

...related by the dataflow semantics,

- Good: links dataflow semantics to executable code
- Less good: only applies to programs that have a semantics

Main theorem




```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

typing/elaboration succeeds,

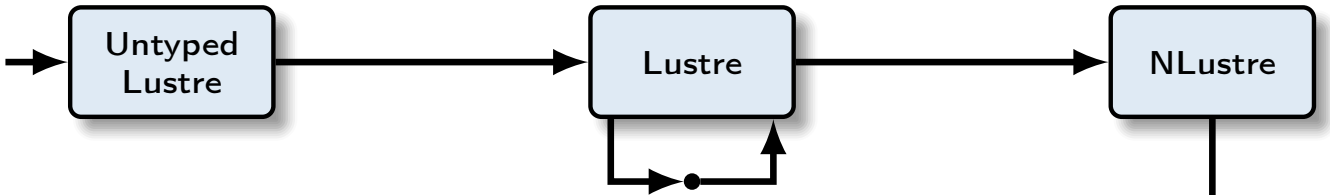
– ∀ well-typed input streams...

...related by the dataflow semantics,

if compilation succeeds,

- Good: links dataflow semantics to executable code
- Less good: only applies to programs that have a semantics

Main theorem




```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

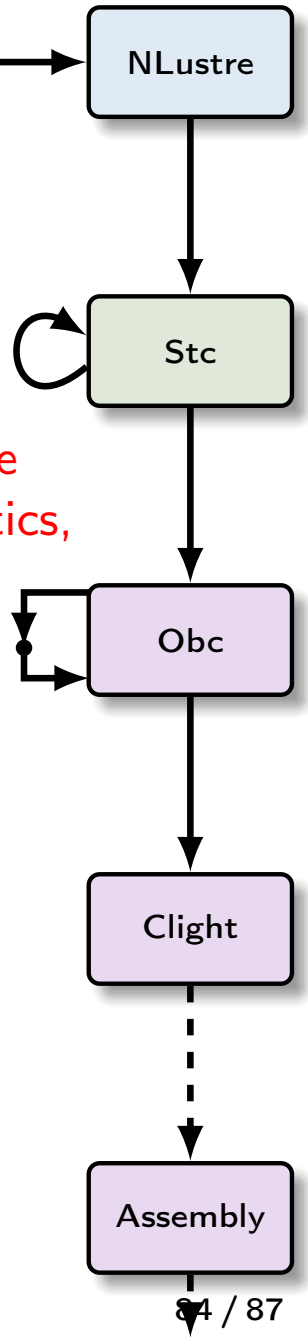
typing/elaboration succeeds,

– ∀ well-typed input streams...

...related by the dataflow semantics,

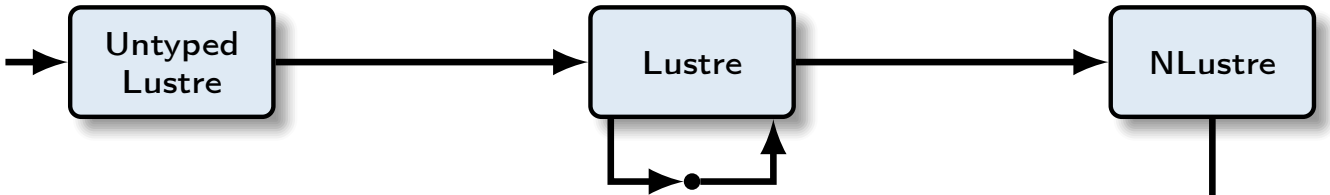
if compilation succeeds,

then, the generated assembly produces an infinite trace...



- Good: links dataflow semantics to executable code
- Less good: only applies to programs that have a semantics

Main theorem




```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

typing/elaboration succeeds,

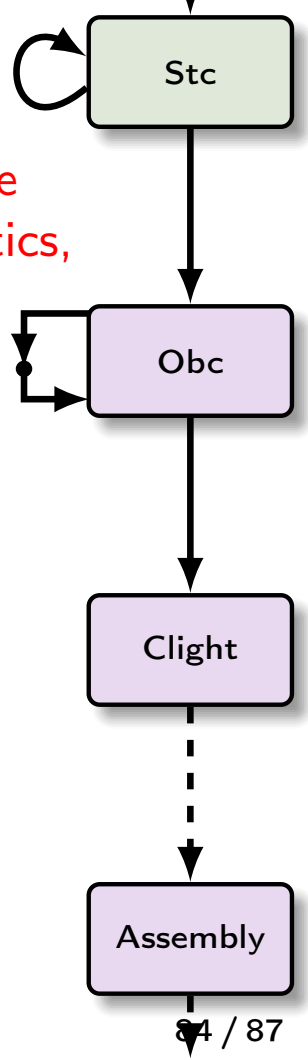
– ∀ well-typed input streams...

...related by the dataflow semantics,

if compilation succeeds,

then, the generated assembly produces an infinite trace...

...that corresponds to the dataflow model.



- Good: links dataflow semantics to executable code
- Less good: only applies to programs that have a semantics

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 `scanfs` inputs and `printfs` 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.

Overview

Lustre: syntax and semantics

Lustre: normalization

Translation: from dataflow programs to imperative code

Optimization: control structure fusion

Generation: from Obc to Clight

Conclusion

Conclusion

Solid progress

- Working compiler from Lustre (with reset) 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.
- Including normalization and fusion passes ...
- Generate Clight for CompCert; change to richer memory model.
- Intermediate language and separation predicates were decisive.

Ongoing work

- Add advanced control structures (state machines)
- 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?

References I

- Auger, C. (Apr. 2013). "Compilation certifiée de SCADE/LUSTRE". PhD thesis. Orsay, France: Univ. Paris Sud 11.
- Auger, C., J.-L. Colaço, G. Hamon, and M. Pouzet (2014). "A Formalization and Proof of a Modular Lustre Code Generator". In preparation.
- 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.
- Blazy, S., Z. Dargaye, and X. Leroy (Aug. 2006). "Formal Verification of a C Compiler Front-End". In: *Proc. 14th Int. Symp. Formal Methods (FM 2006)*. Vol. 4085. LNCS. Hamilton, Canada: Springer, pp. 460–475.

References II

- 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.
- Bourke, T., L. Brun, and M. Pouzet (Jan. 2020). "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset". In: *Proc. of the ACM on Programming Languages* 4.POPL, pp. 1–29.
- Bourke, T., P.-É. Dagand, M. Pouzet, and L. Rieg (Jan. 2017). "Vérification de la génération modulaire du code impératif pour Lustre". In: *28^{jèmes} 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.

References III

- Bourke, T., P. Jeanmaire, B. Pesin, and M. Pouzet (Oct. 2021). "Verified Lustre Normalization with Node Subsampling". In: *ACM Trans. Embedded Computing Systems* 1.1. ESWEEK special issue with presentations at 21st Int. Conf. on Embedded Software (EMSOFT 2021), article 1.
- Brun, L. (June 2020). "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset". PhD thesis. PSL Research University.
- Caspi, P. (1994). "Towards recursive block diagrams". In: *Annual Review in Automatic Programming* 18, pp. 81–85.
- Colaço, J.-L. and M. Pouzet (Oct. 2003). "Clocks as First Class Abstract Types". In: *Proc. 3rd Int. Conf. on Embedded Software (EMSOFT 2003)*. Ed. by R. Alur and I. Lee. Vol. 2855. LNCS. Philadelphia, PA, USA: Springer, pp. 134–155.
- 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. Montreal, Canada: ACM, pp. 289–300.

References IV

- Jourdan, J.-H., F. Pottier, and X. Leroy (Mar. 2012). "Validating LR(1) parsers". In: *21st European Symposium on Programming (ESOP 2012), 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.
- Kahn, G. (Aug. 1974). "The Semantics of a Simple Language for Parallel Programming". In: *Proc. Int. Federation for Information Processing (IFIP) Congress 1974*. Ed. by J. L. Rosenfeld. Stockholm, Sweden: North-Holland, pp. 471–475.
- Leroy, X. (2009). "Formal verification of a realistic compiler". In: *Comms. ACM* 52.7, pp. 107–115.
- McCoy, F. (1885). *Natural history of Victoria: Prodromus of the Zoology of Victoria*. Frog images.
- Paulin-Mohring, C. (2009). "A constructive denotational semantics for Kahn networks in Coq". In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*. Ed. by Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin. Cambridge, UK: CUP, pp. 383–413.

References V

- Pouzet, M. (Apr. 2006). *Lucid Synchrone, v. 3. Tutorial and reference manual*. Université Paris-Sud.
- Rushby, J. (Oct. 2011). "New Challenges in Certification for Aircraft Software". In: *Proc. 11th ACM Int. Conf. on Embedded Software (EMSOFT 2011)*. Rev. 2012. Taipei, Taiwan: ACM Press, pp. 211–218.
- The Coq Development Team (2016). *The Coq proof assistant reference manual*. v. 8.5. Inria.
- Wagner, L. G., D. Cofer, K. Slind, C. Tinelli, and A. Mebsout (Feb. 2017). *Formal Methods Tool Qualification*. Tech. rep. NASA/CR-2017-219371. Langley Research Center, Hampton, VA, USA: National Aeronautics and Space Administration.