Verified Compilation of a Synchronous Dataflow Language with State Machines

Basile Pesin

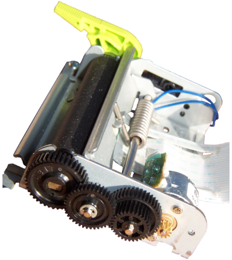
Inria Paris

École normale supérieure, CNRS, PSL University

Friday, October 13

Programming embedded systems

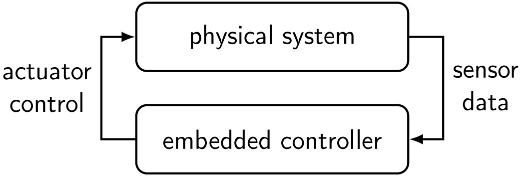
Cjp24

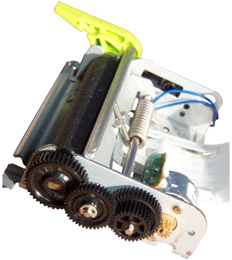


Akiry

Programming embedded systems

(cc) Cjp24





(cc) Akiry

Programming embedded systems

(cc) Cjp24

actuator control

physical system

embedded controller

sensor data

```
every trigger {
  read inputs;
  calculate;
  write outputs;
}
```

(cc) Akiry

Programming embedded systems

Cjp24

Akiry

safety-critical

actuator control

physical system

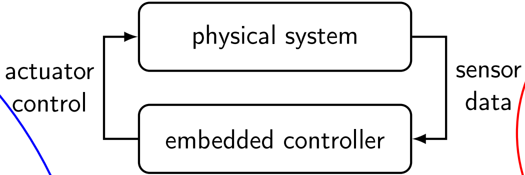
sensor data

embedded controller

```
every trigger {
  read inputs;
  calculate;
  write outputs;
}
```

Programming embedded systems

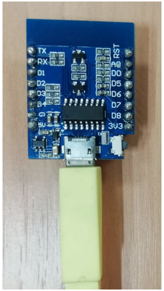
safety-critical




```
every trigger {
  read inputs;
  calculate;
  write outputs;
}
```

Low-level languages and high-level specifications

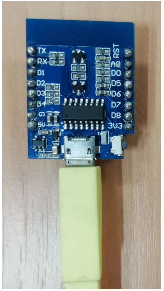
- Engineers write high-level specifications of the system

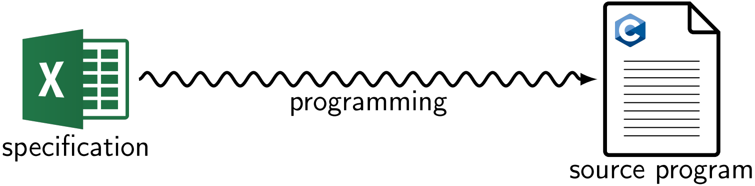


specification

Low-level languages and high-level specifications

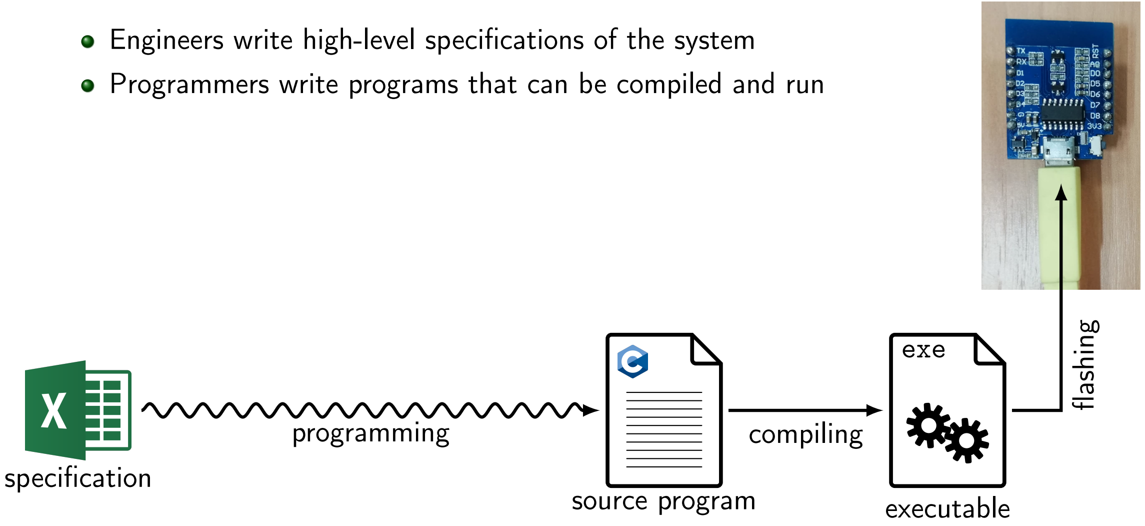
- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run





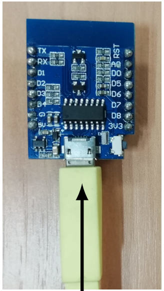
Low-level languages and high-level specifications

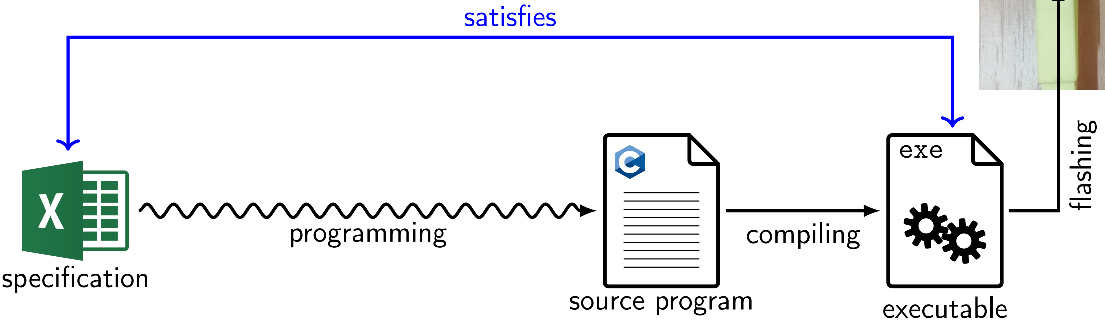
- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run



Low-level languages and high-level specifications

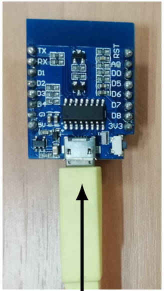
- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run
- Does the program really implement the spec?

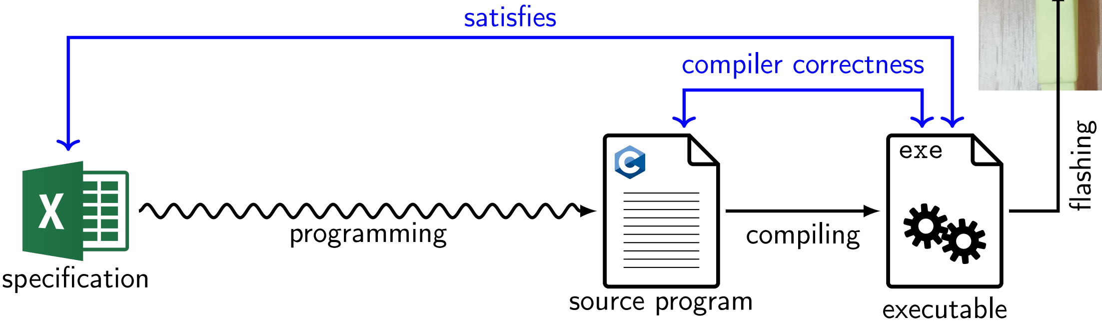




Low-level languages and high-level specifications

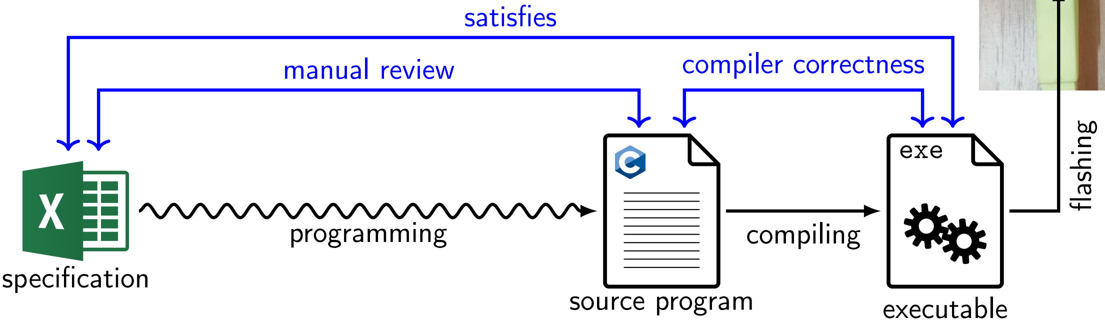
- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run
- Does the program really implement the spec?

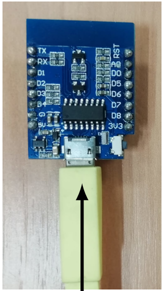




Low-level languages and high-level specifications

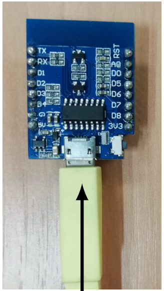
- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run
- Does the program really implement the spec?

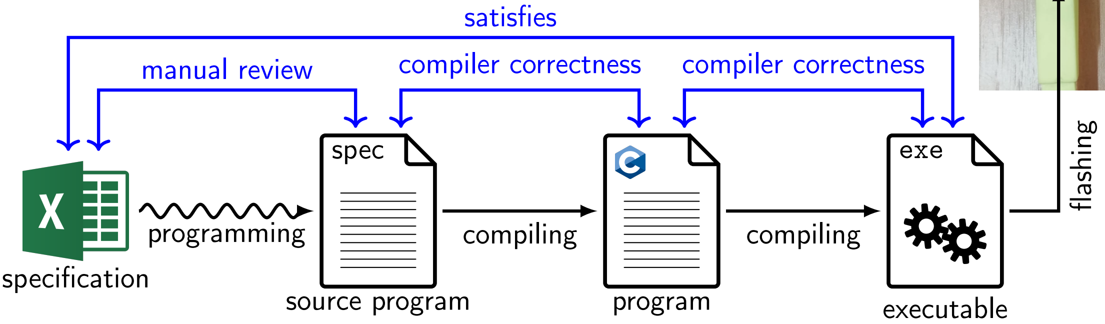




Low-level languages and high-level specifications

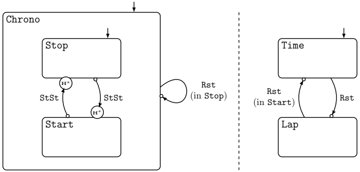
- Engineers write high-level specifications of the system
- Programmers write programs that can be compiled and run
- Does the program really implement the spec?
- Reduce the gap by programming in a language closer to the spec





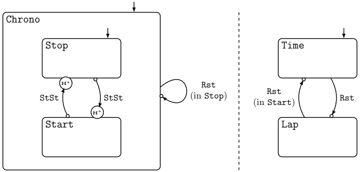
Programming Embedded Systems with State Machines

- Statecharts [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]



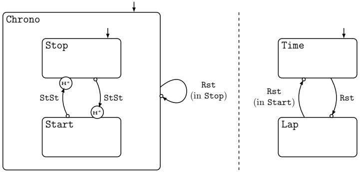
Programming Embedded Systems with State Machines

- Statecharts [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]
- SyncCharts [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- Mode-Automata [Maraninchi and Rémond (1998): Mode-Automata: About Modes and States for Reactive Systems]



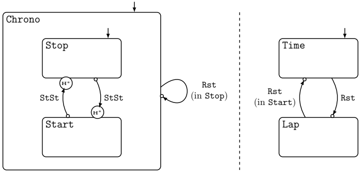
Programming Embedded Systems with State Machines

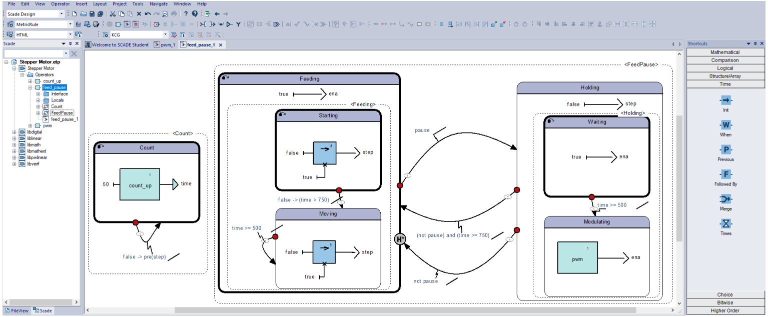
- Statecharts [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]
- SyncCharts [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- Mode-Automata [Maraninchi and Rémond (1998): Mode-Automata: About Modes and States for Reactive Systems]
- Lucid Synchrone [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]



Programming Embedded Systems with State Machines

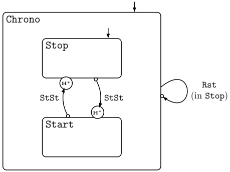
- Statecharts [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]
- SyncCharts [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- Mode-Automata [Maraninchi and Rémond (1998): Mode-Automata: About Modes and States for Reactive Systems]
- Lucid Synchrone [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]
- Scade 6 [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]

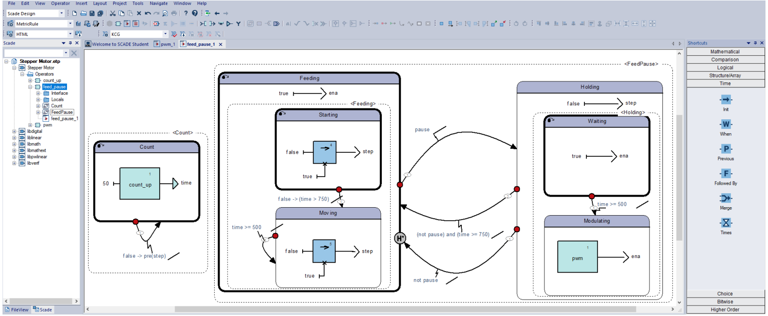




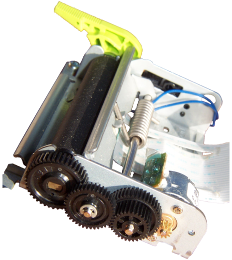
Programming Embedded Systems with State Machines

- Statecharts [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]
- SyncCharts [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- Mode-Automata [Maraninchi and Rémond (1998): Mode-Automata: About Modes and States for Reactive Systems]
- Lucid Synchrone [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]
- Scade 6 [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]
- Vélus: A subset of Scade 6

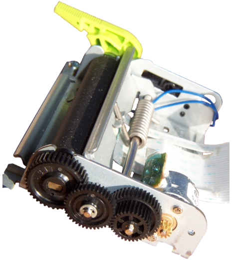


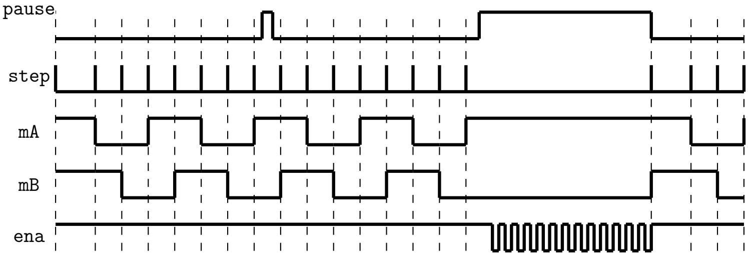


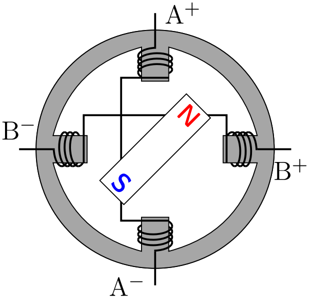
An embedded example: stepper motor for a small printer



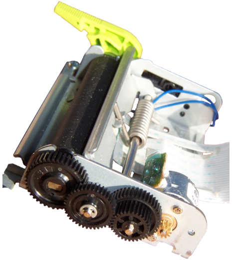
An embedded example: stepper motor for a small printer

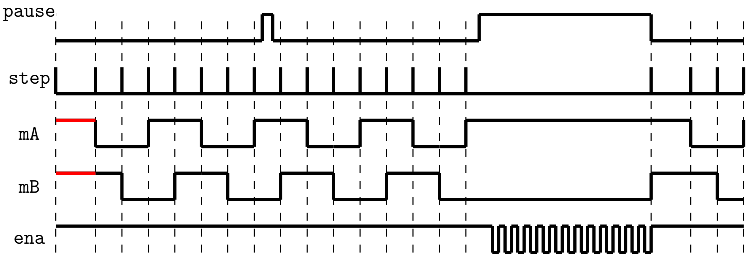


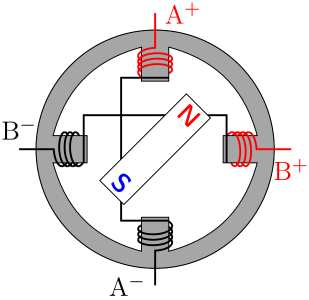




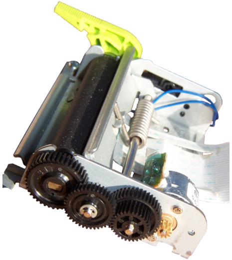
An embedded example: stepper motor for a small printer

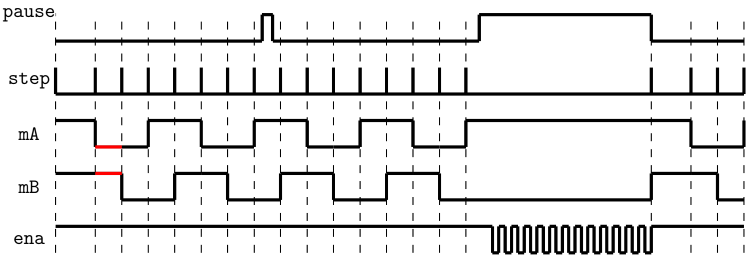


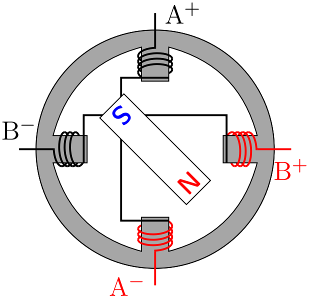




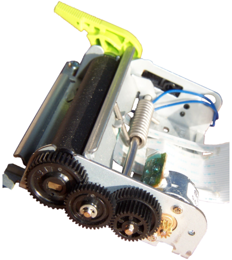
An embedded example: stepper motor for a small printer

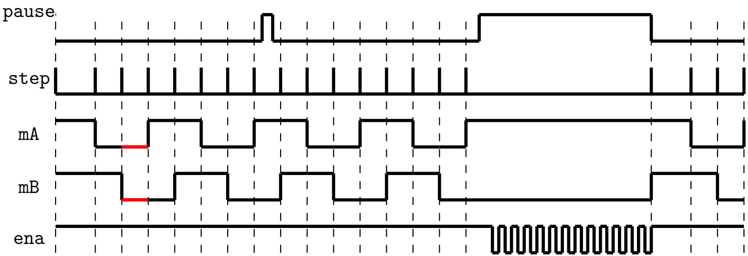


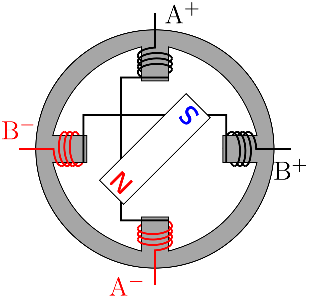




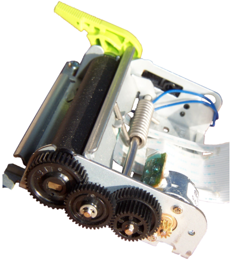
An embedded example: stepper motor for a small printer

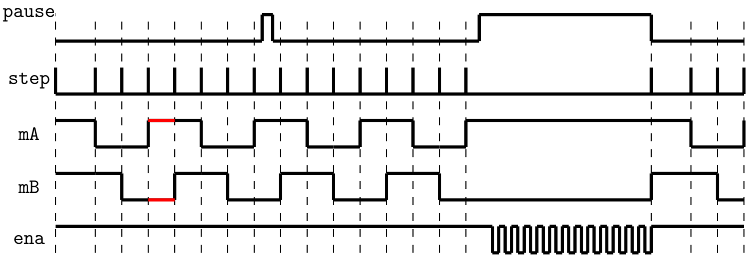


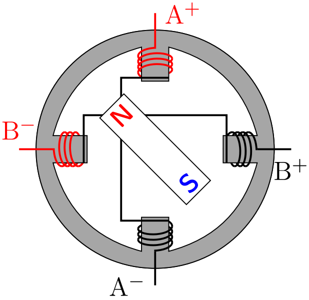




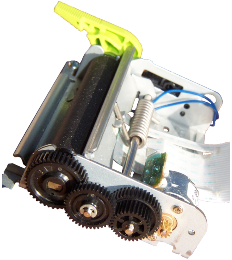
An embedded example: stepper motor for a small printer

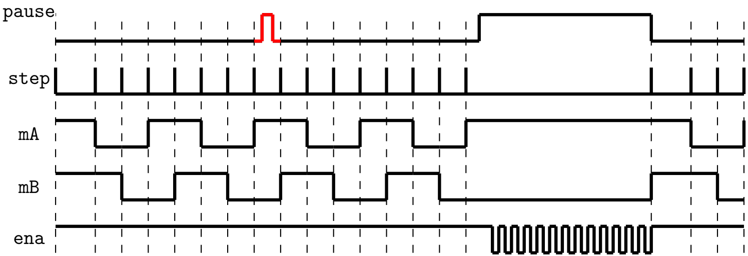


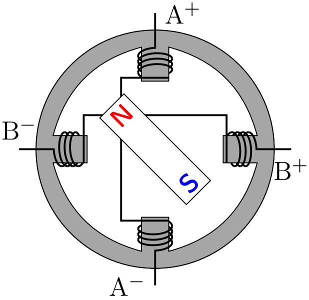




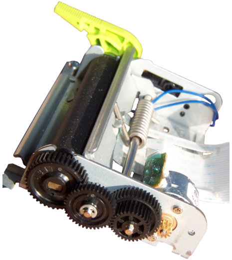
An embedded example: stepper motor for a small printer

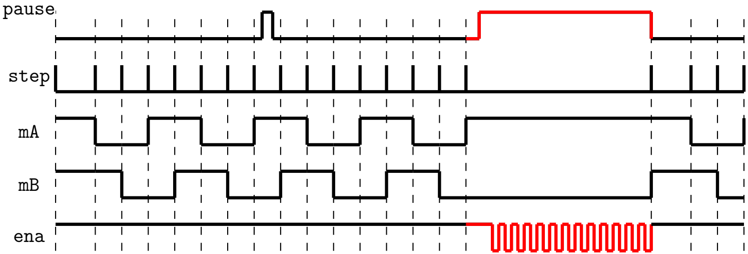


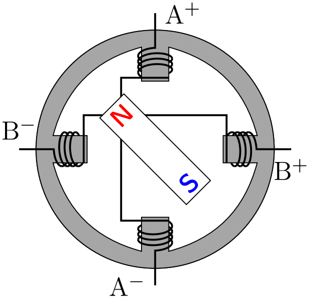




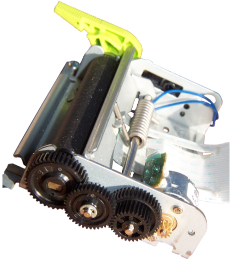
An embedded example: stepper motor for a small printer

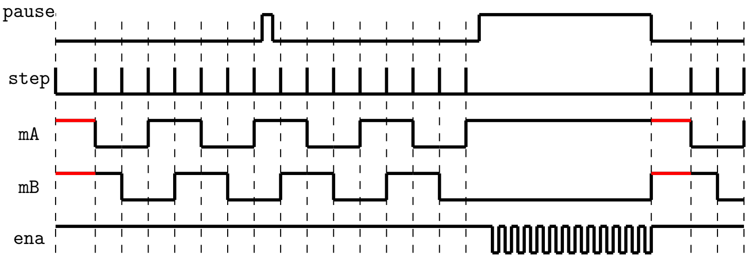


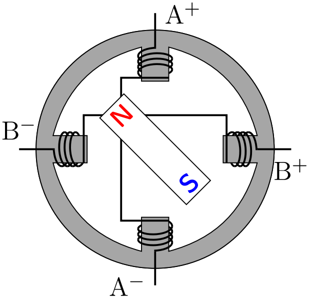




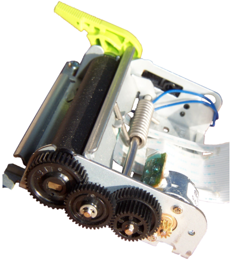
An embedded example: stepper motor for a small printer

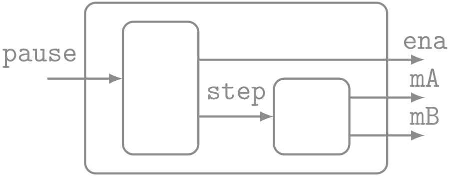


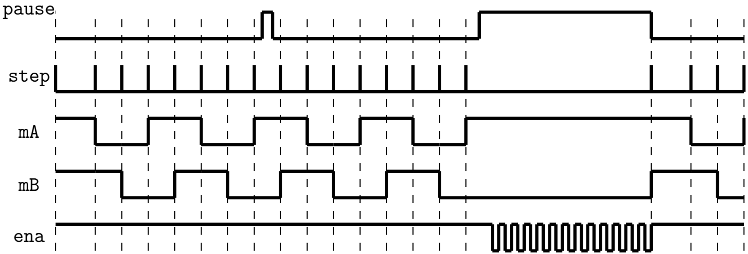


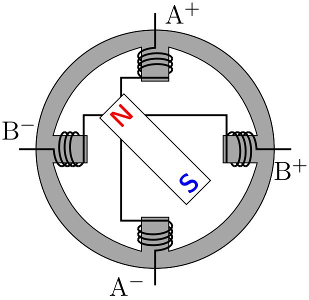


An embedded example: stepper motor for a small printer

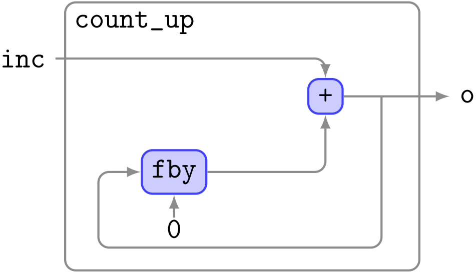


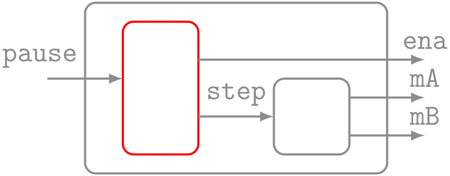






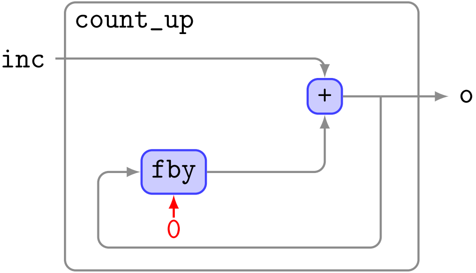
A simple dataflow program

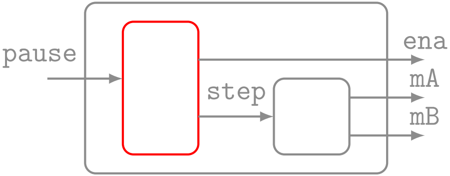




inc	5	4	1	3	2	8	3	...
0 fby o								
o								

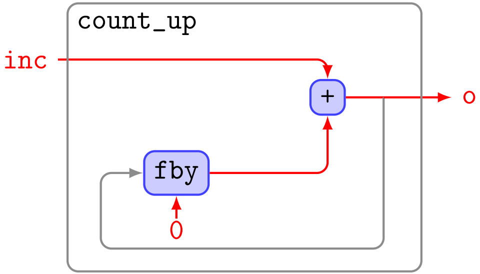
A simple dataflow program

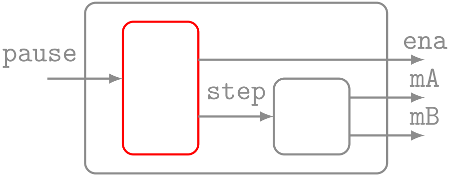




inc	5	4	1	3	2	8	3	...
0 fby o	0							
o								

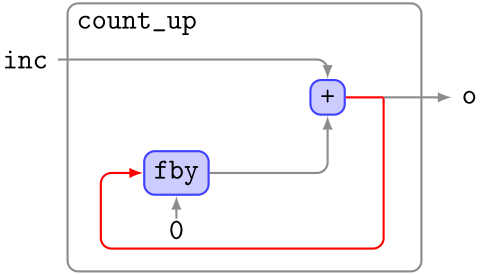
A simple dataflow program

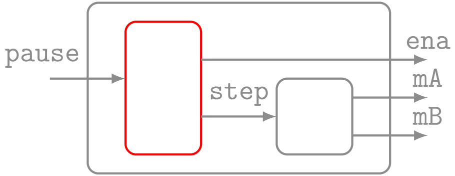




inc	5	4	1	3	2	8	3	...
0 fby o	0							
o	5							

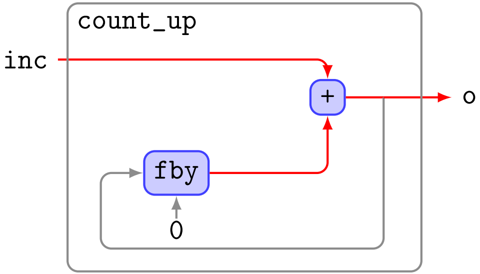
A simple dataflow program

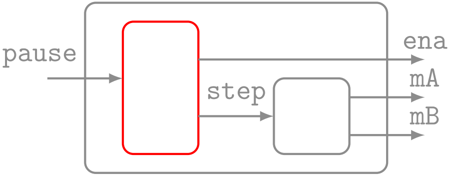




inc	5	4	1	3	2	8	3	...
0 fby o	0	5						
o	5							

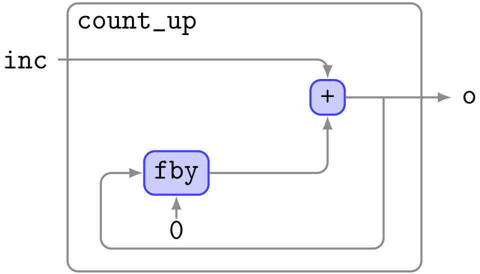
A simple dataflow program

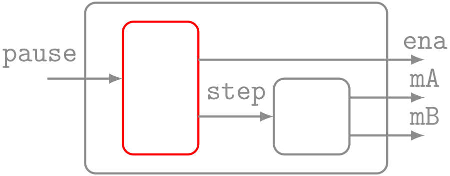




inc	5	4	1	3	2	8	3	...
0 fby o	0	5						
o	5	9						

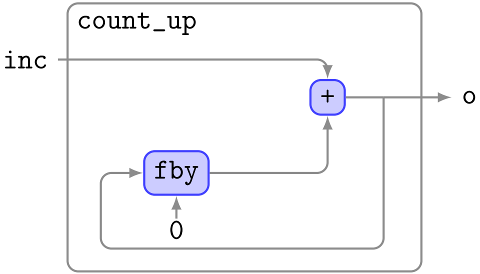
A simple dataflow program

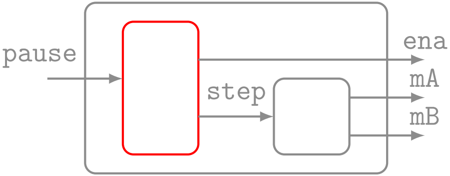




inc	5	4	1	3	2	8	3	...
0 fby o	0	5	9	10	13	15	23	...
o	5	9	10	13	15	23	26	...

A simple dataflow program

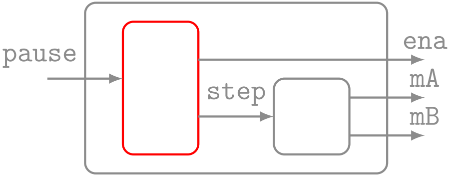





```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

inc	5	4	1	3	2	8	3	...
0 fby o	0	5	9	10	13	15	23	...
o	5	9	10	13	15	23	26	...

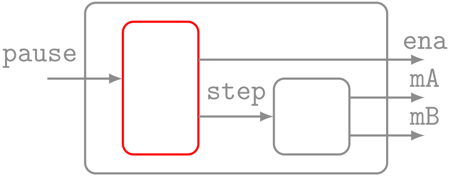
Modular resetting of equations




```
reset
  time = count_up(50)
every (false fby step)
```

step	...
time	...

Modular resetting of equations



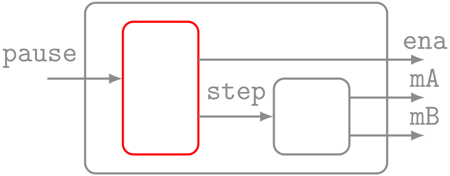

```
reset
  time = count_up(50)
every (false fby step)
```

⇕ equivalent

```
reset
  time = (0 fby time) + 50
every (false fby step)
```

step	...
time	...

Modular resetting of equations



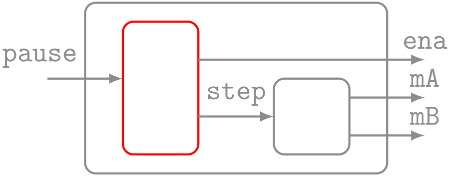

```
reset
  time = count_up(50)
every (false fby step)
```

⟺ equivalent

```
reset
  time = (0 fby time) + 50
every (false fby step)
```

step	F	F	T	...
time	50	100	150	...

Modular resetting of equations



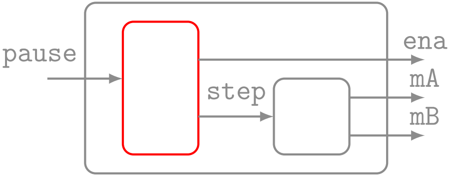

```
reset
  time = count_up(50)
every (false fby step)
```

$\Updownarrow$ equivalent

```
reset
  time = (0 fby time) + 50
every (false fby step)
```

step	F	F	T	F	F	F	T	...
time				50	100	150	200	...

Modular resetting of equations



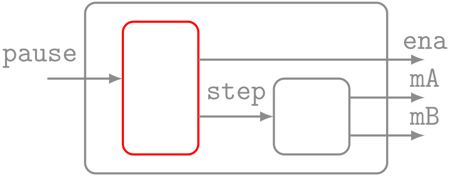

```
reset
  time = count_up(50)
every (false fby step)
```

⇕ equivalent

```
reset
  time = (0 fby time) + 50
every (false fby step)
```

step	F	F	T	F	F	F	T	F	...
time								50	...

Modular resetting of equations



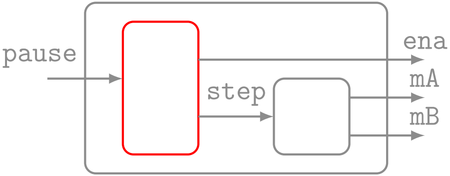

```
reset
  time = count_up(50)
every (false fby step)
```

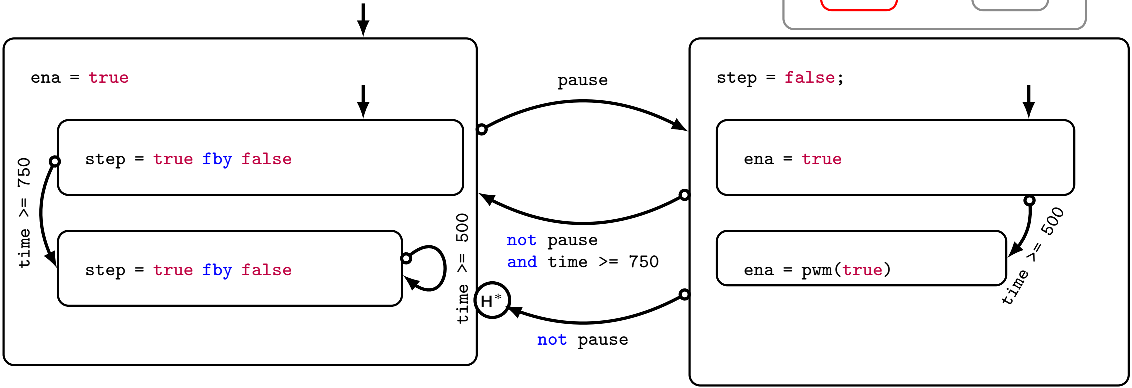
$\Updownarrow$ equivalent

```
reset
  time = (0 fby time) + 50
every (false fby step)
```

step	F	F	T	F	F	F	T	F	...
time	50	100	150	50	100	150	200	50	...

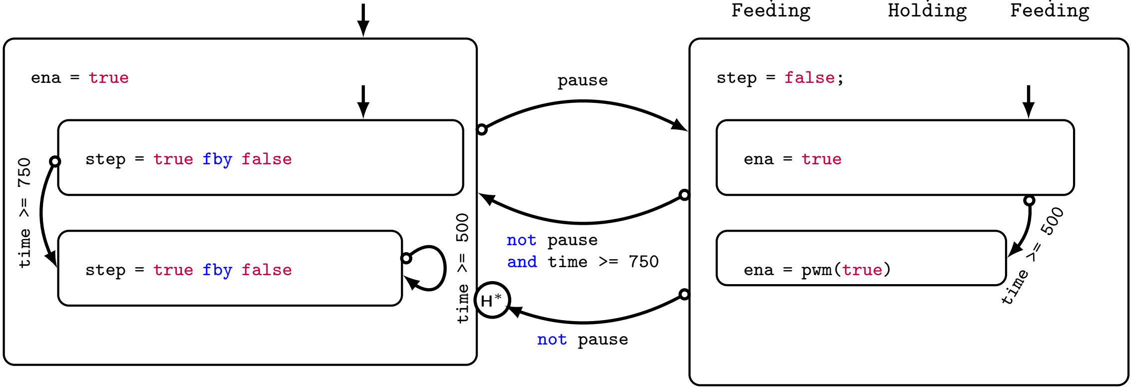
Hierarchical State Machines





Hierarchical State Machines

pause	F	F	F	...	F	F	...	T	...	F	...	F	...
time	0	0	50	...	750	0	...	150	...	350	...	500	...
step	T	F	F	...	T	F	...	F	...	F	...	T	...
ena	T	T	T	...	T	T	...	T	...	T	...	T	...
				Feeding					Holding			Feeding	

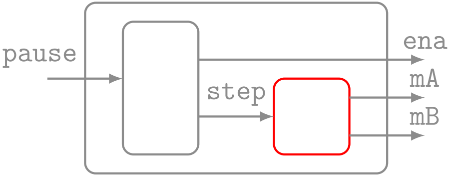


Hierarchical State Machines

```
node feed_pause(pause : bool) returns (ena, step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);

  automaton initially Feeding
  state Feeding do
    ena = true;
    automaton initially Starting
    state Starting do
      step = true fby false
    unless time >= 750 then Moving

    state Moving do
      step = true fby false
    unless time >= 500 then Moving

    end;
  unless pause then Holding

  state Holding do
    step = false;
    automaton initially Waiting
    state Waiting do
      ena = true
    unless time >= 500 then Modulating

    state Modulating do
      ena = pwm(true)

    end;
  unless
    | not pause and time >= 750 then Feeding
    | not pause continue Feeding
  end
tel
```

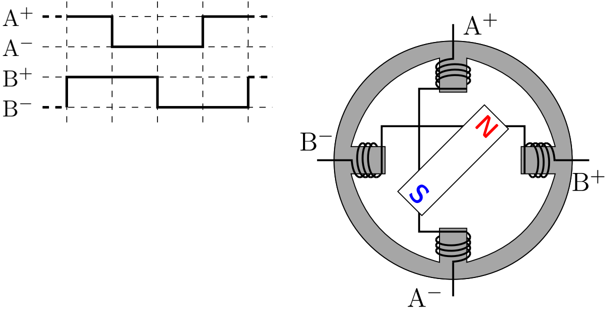
pause	F	F	F	...	F	F	...	T	...	F	...	F	...
time	0	0	50	...	750	0	...	150	...	350	...	500	...
step	T	F	F	...	T	F	...	F	...	F	...	T	...
ena	T	T	T	...	T	T	...	T	...	T	...	T	...
	Feeding							Holding		Feeding			

Switch blocks



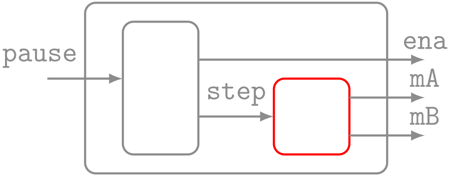

```
  mA = not (last mB);
  mB = last mA;
```

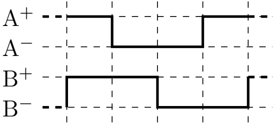
```
last mA = true;
last mB = false;
```

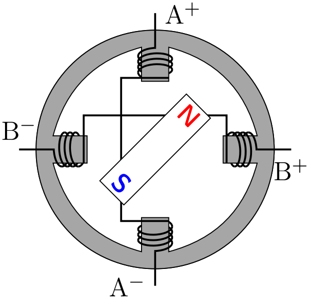


Switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```



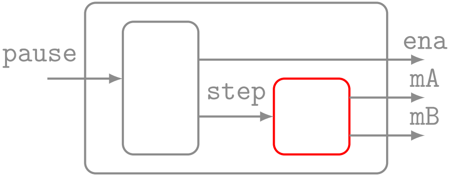


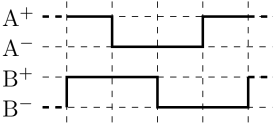


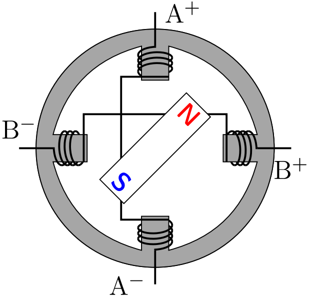
step	F	T	T	F	F	T	F	T	F	T	F	...
last mA	T											...
last mB	F											...
mA	T											...
mB	F											...

Switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```



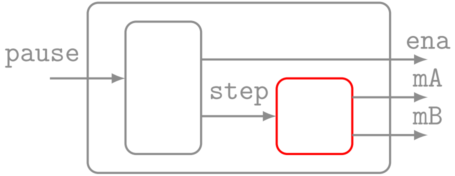


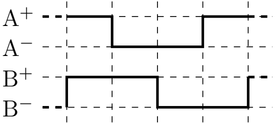


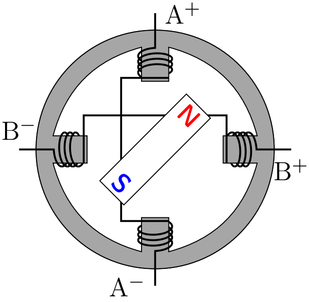
step	F	T	T	F	F	T	F	T	F	T	F	...
last mA	T	T	T									...
last mB	F	F	T									...
mA	T	T	F									...
mB	F	T	T									...

Switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```



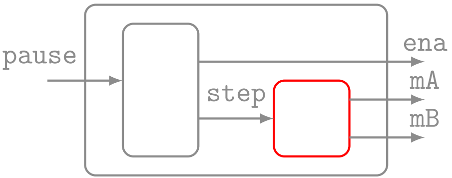


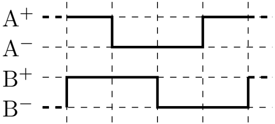


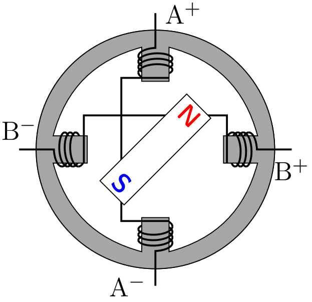
step	F	T	T	F	F	T	F	T	F	T	F	...
last mA	T	T	T	F	F							...
last mB	F	F	T	T	T							...
mA	T	T	F	F	F							...
mB	F	T	T	T	T							...

Switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```

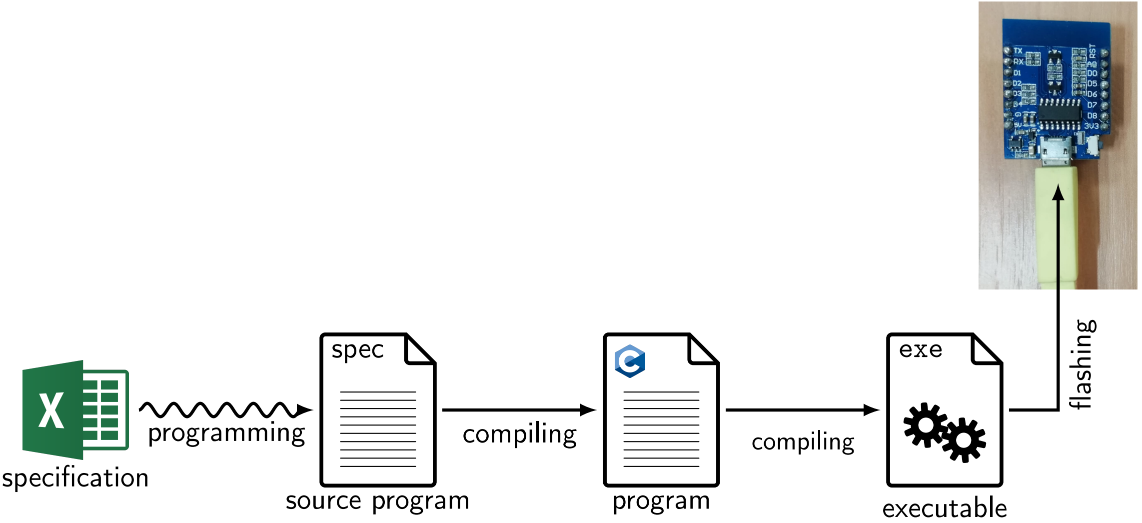




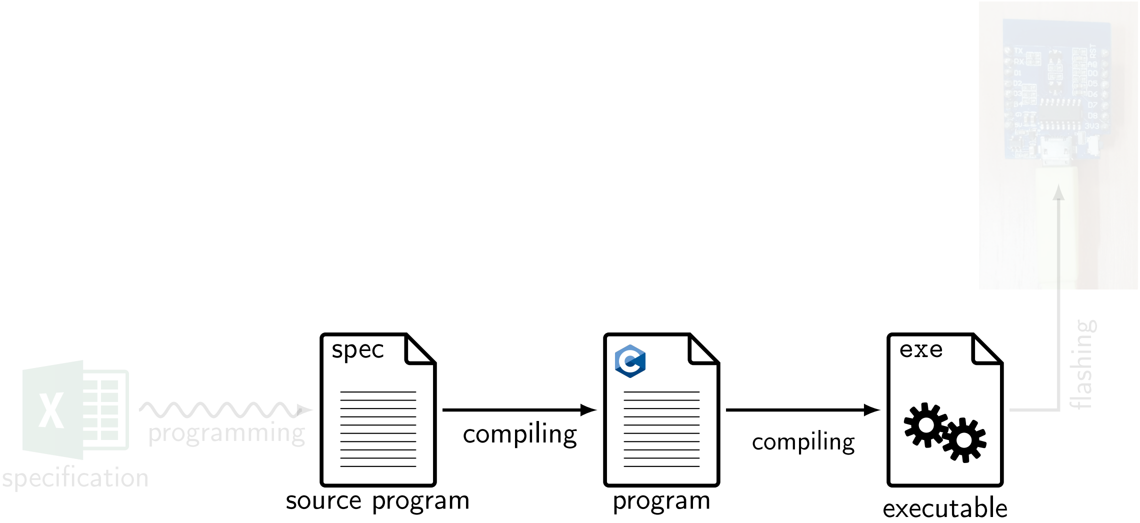


step	F	T	T	F	F	T	F	T	F	T	F	...
last mA	T	T	T	F	F	F	F	F	T	T	T	...
last mB	F	F	T	T	T	T	F	F	F	F	T	...
mA	T	T	F	F	F	F	F	T	T	T	T	...
mB	F	T	T	T	T	F	F	F	F	T	T	...

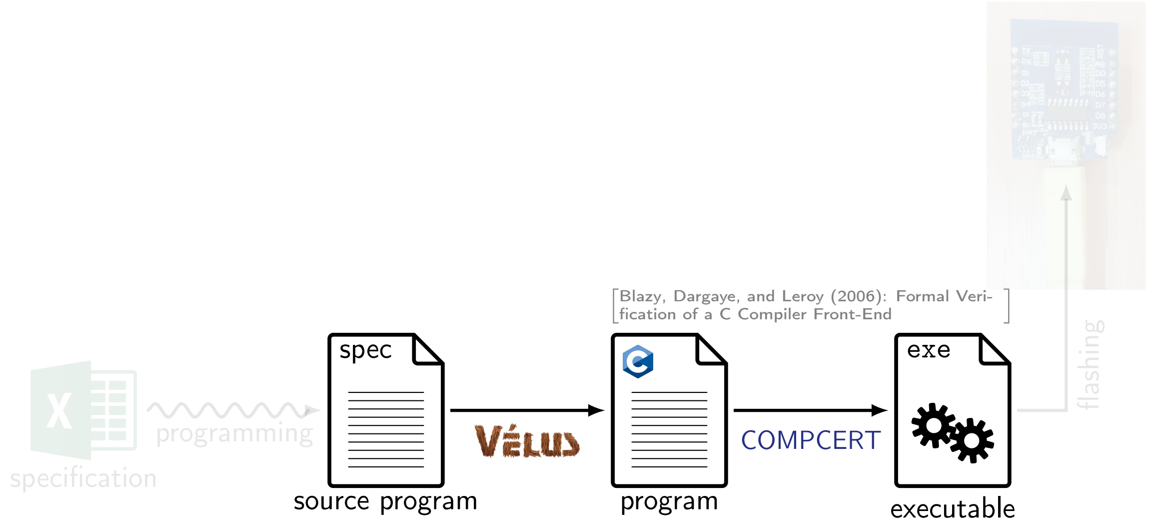
Compiling Lustre to C



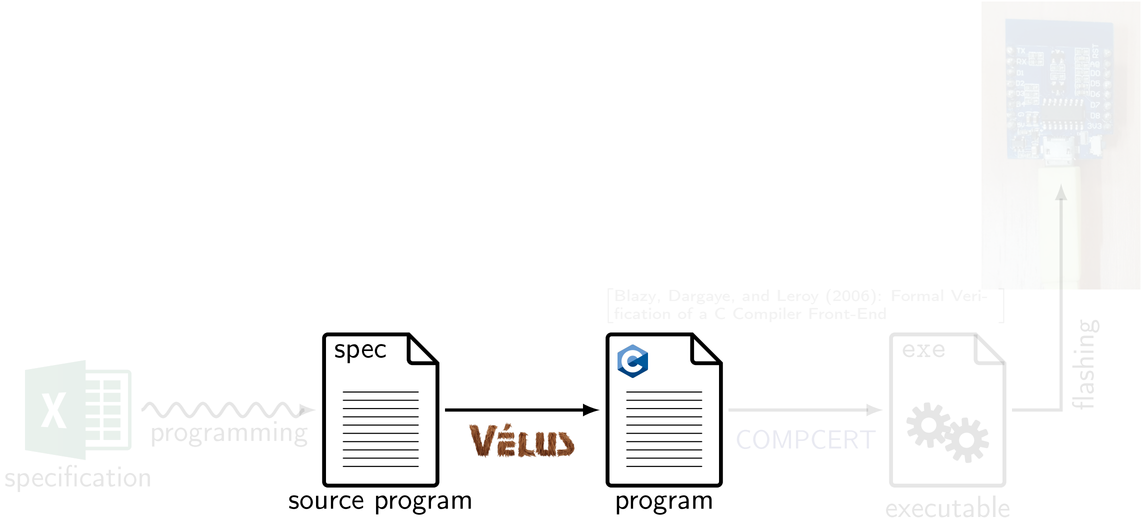
Compiling Lustre to C



Compiling Lustre to C

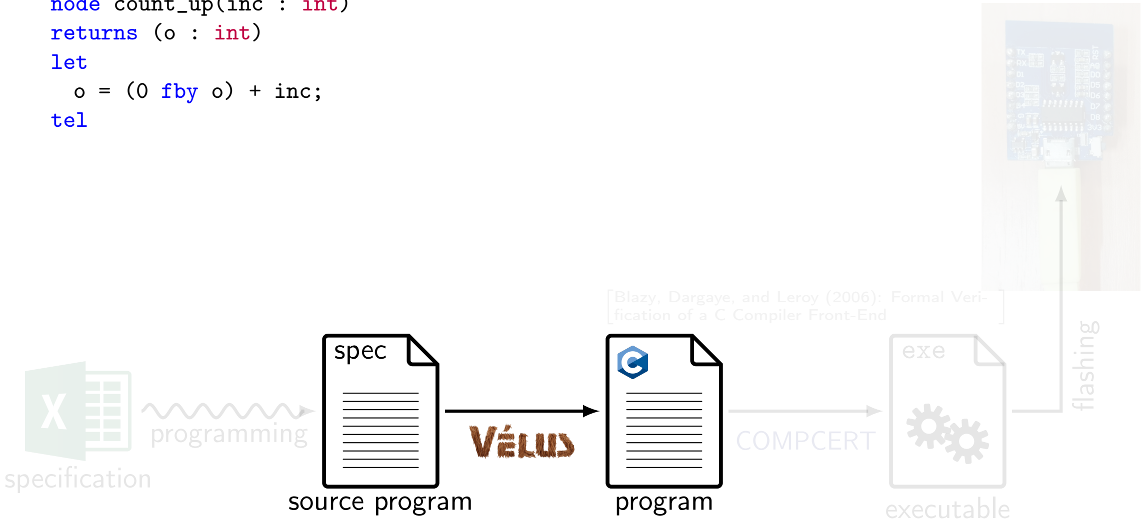


Compiling Lustre to C



Compiling Lustre to C

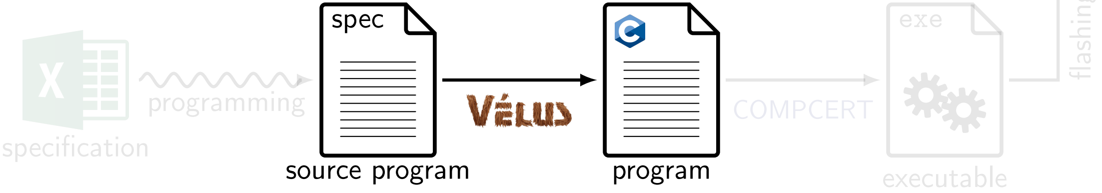
```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```



Compiling Lustre to C

```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

```
struct count_up {
  int norm$1;
};
```

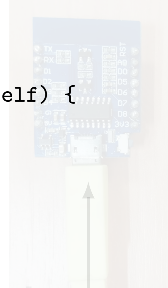



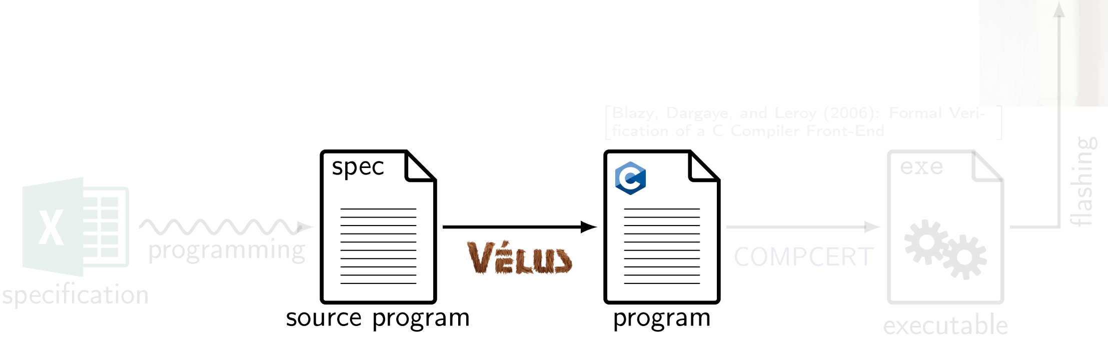
[Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]

Compiling Lustre to C

```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

```
struct count_up {
  int norm$1;
};

void fun$reset$count_up(struct count_up *self) {
  (*self).norm$1 = 0;
}
```

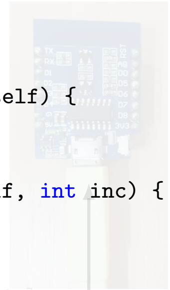




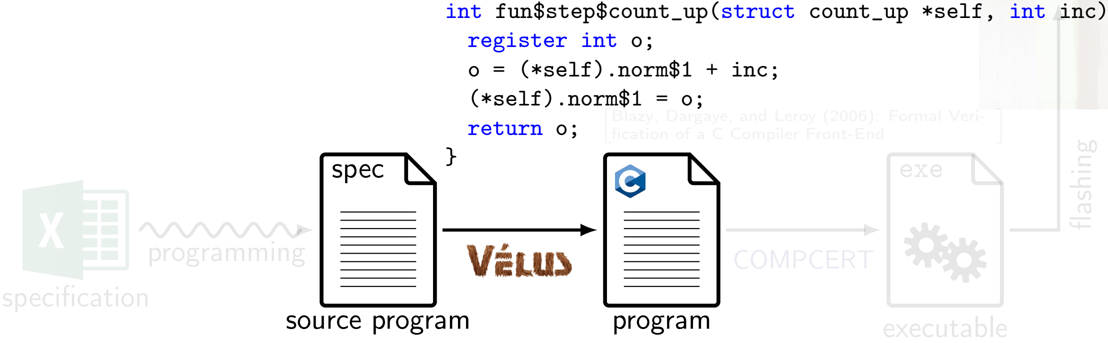
Compiling Lustre to C

```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

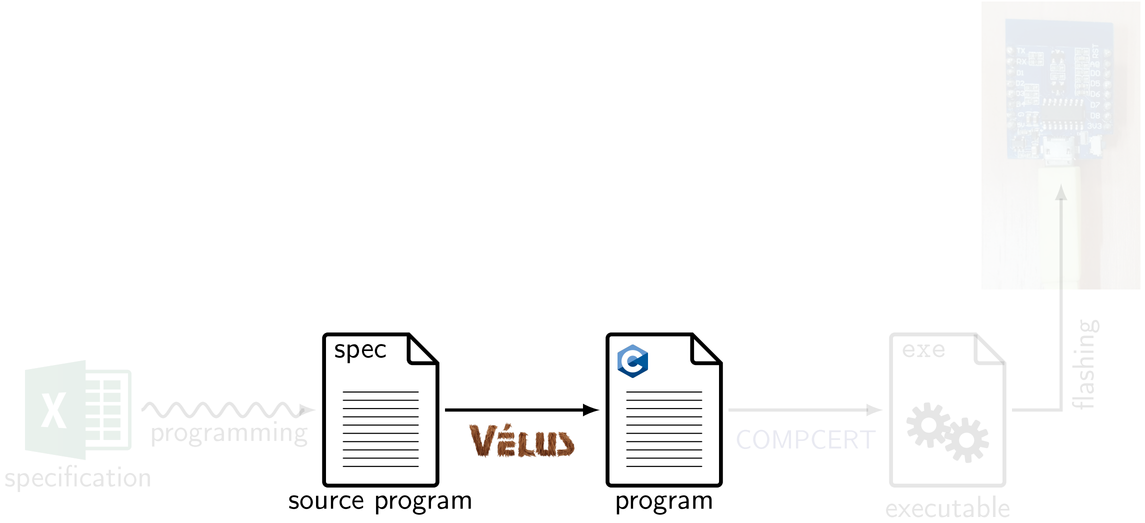
```
struct count_up {
  int norm$1;
};

void fun$reset$count_up(struct count_up *self) {
  (*self).norm$1 = 0;
}

int fun$step$count_up(struct count_up *self, int inc) {
  register int o;
  o = (*self).norm$1 + inc;
  (*self).norm$1 = o;
  return o;
}
```



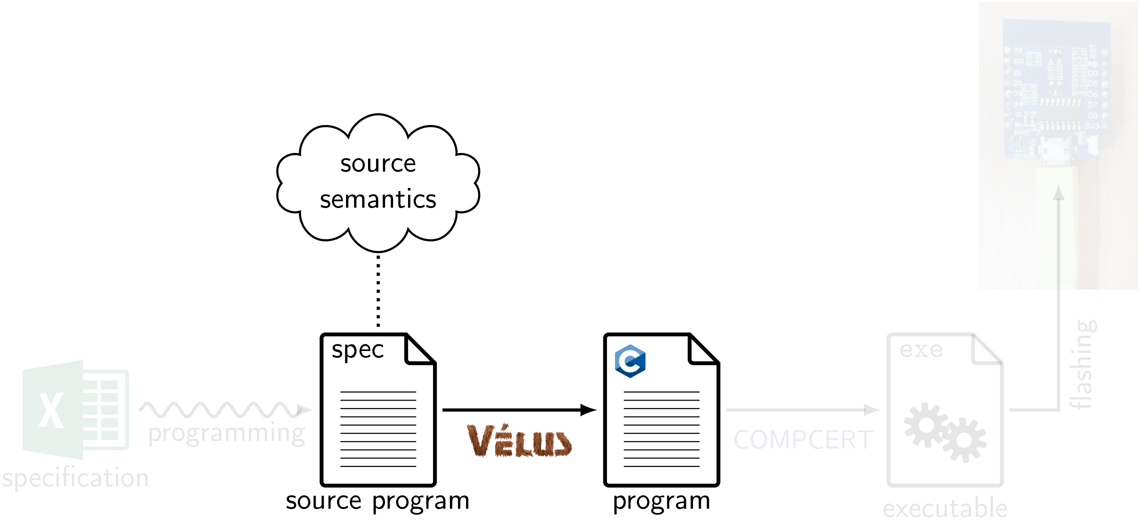
[Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]



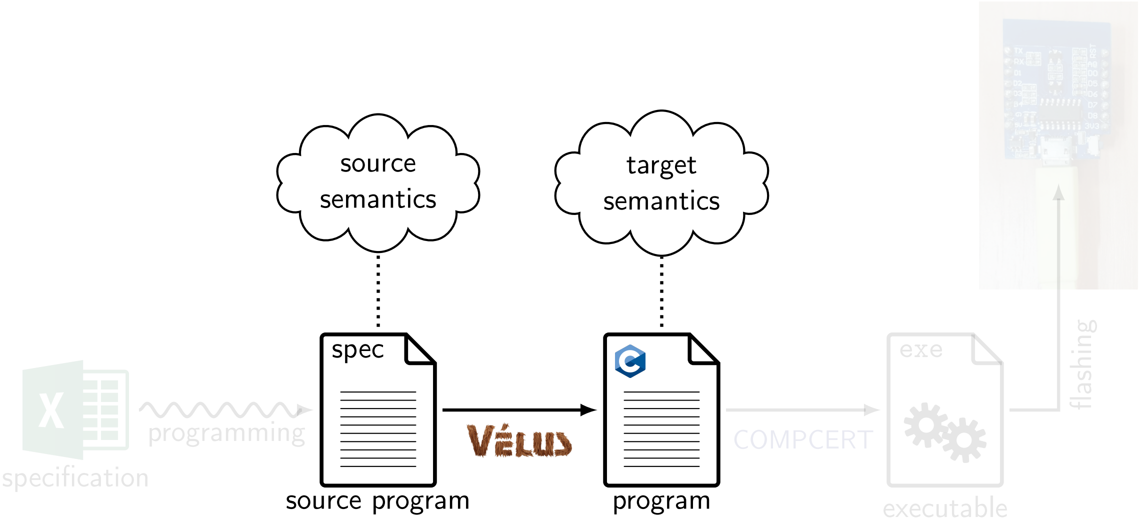
Compiler verification



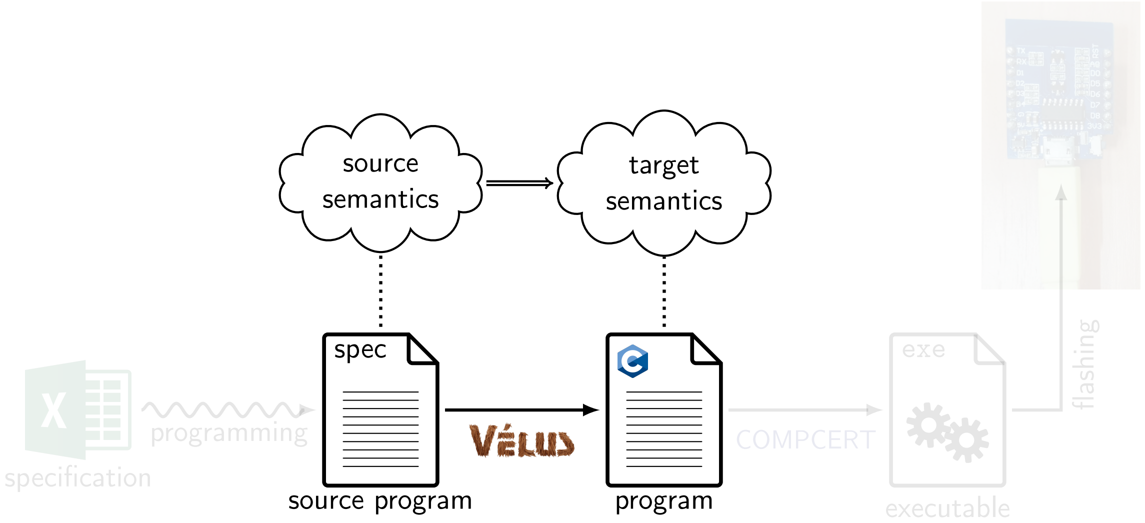
Compiler verification



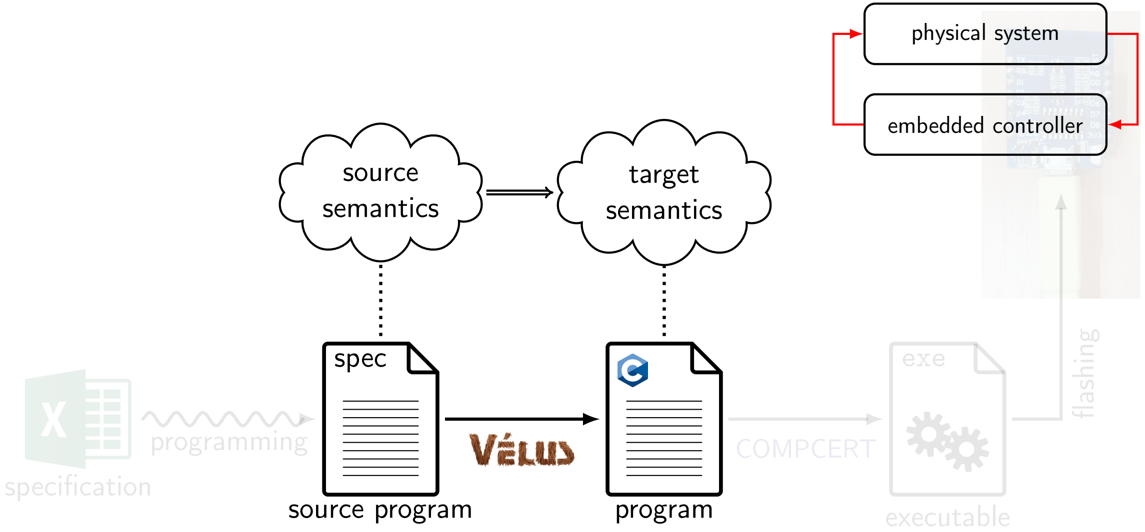
Compiler verification



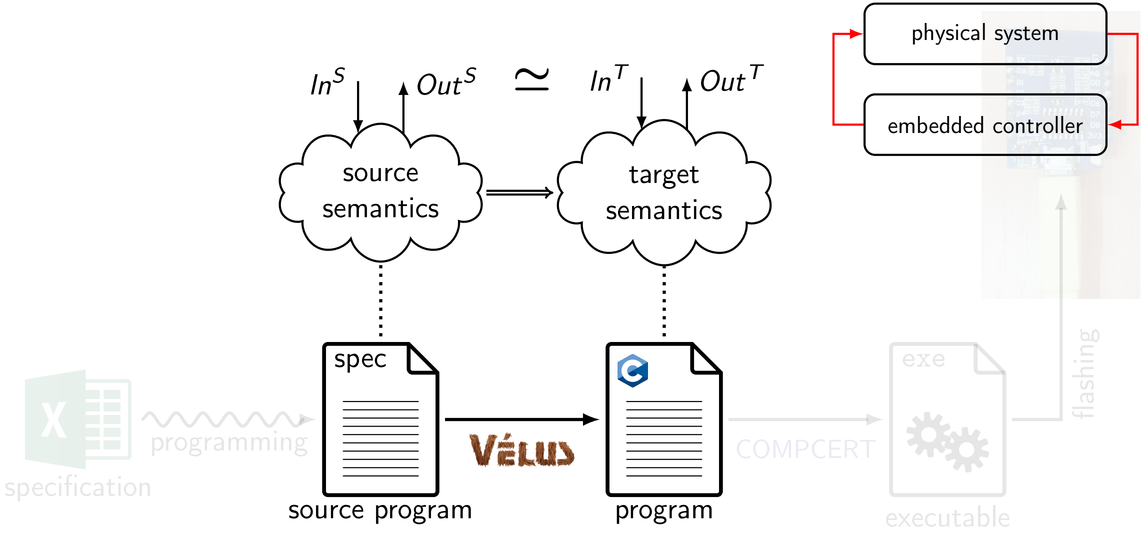
Compiler verification



Compiler verification

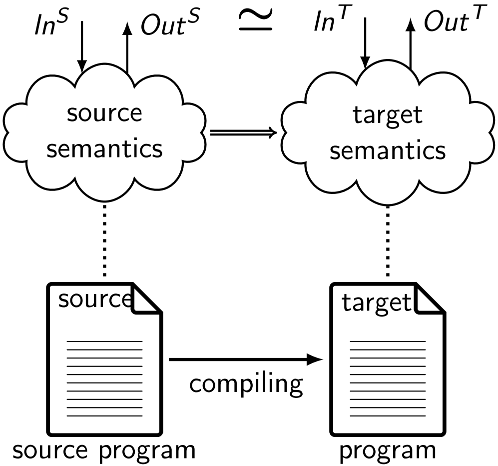


Compiler verification



Compiler verification

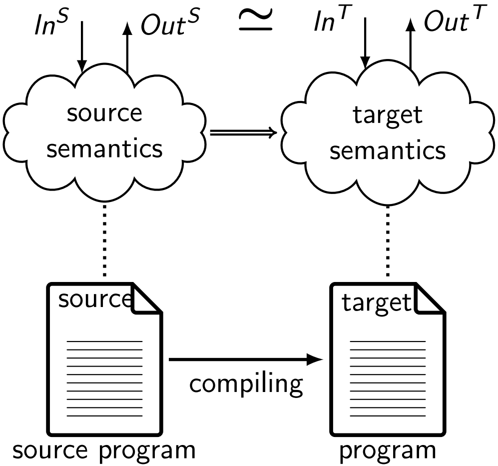
In an Interactive Theorem Prover (recently):



Compiler verification

In an Interactive Theorem Prover (recently):

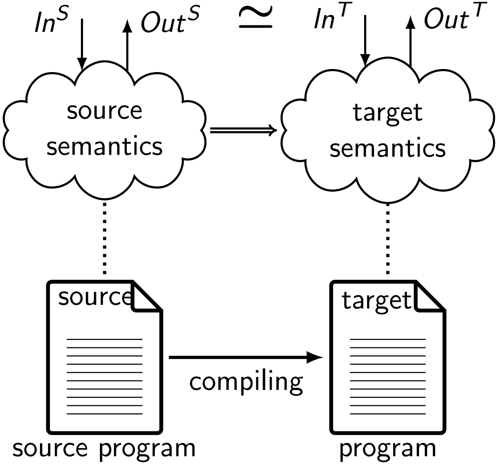
- CompCert: C → machine code
  [Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]
- CakeML: SML → machine code
  [Kumar, Myreen, Norrish, and Owens (2014): CakeML: A Verified Implementation of ML]



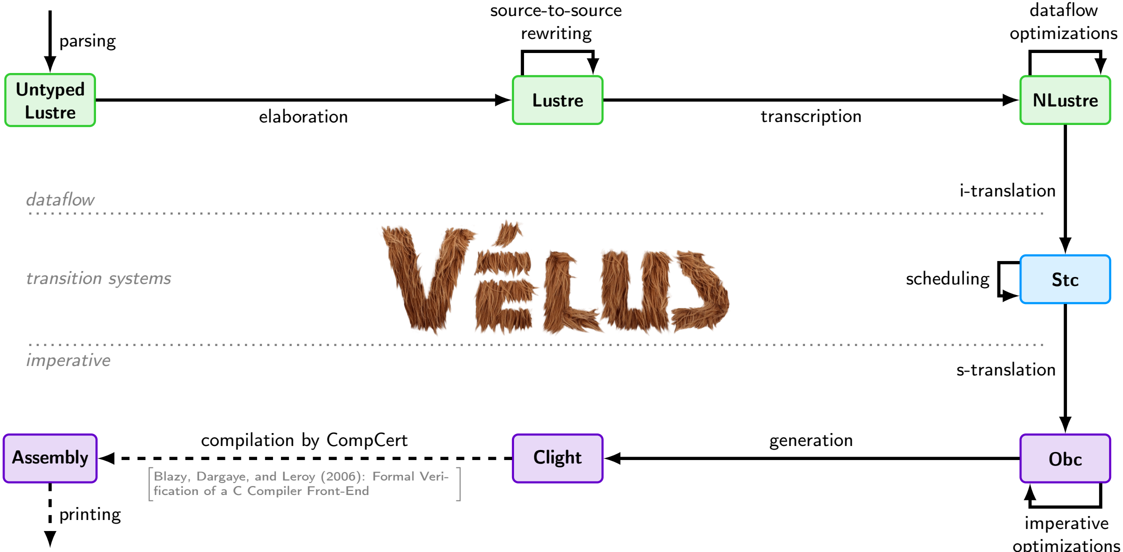
Compiler verification

In an Interactive Theorem Prover (recently):

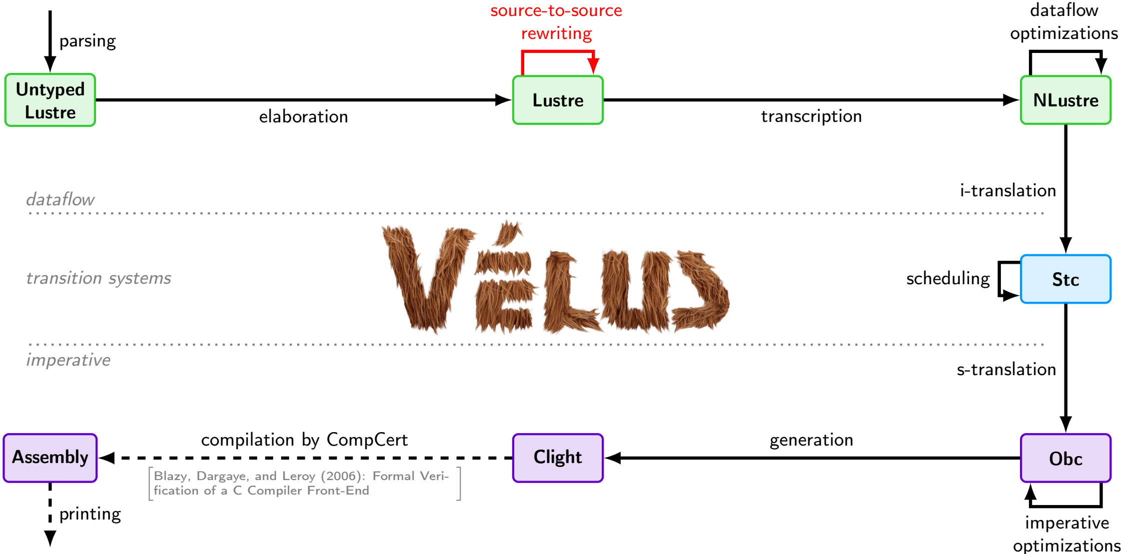
- CompCert: C → machine code
  [Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]
- CakeML: SML → machine code
  [Kumar, Myreen, Norrish, and Owens (2014): CakeML: A Verified Implementation of ML]
- Vélus: Lustre/Scade 6 → C



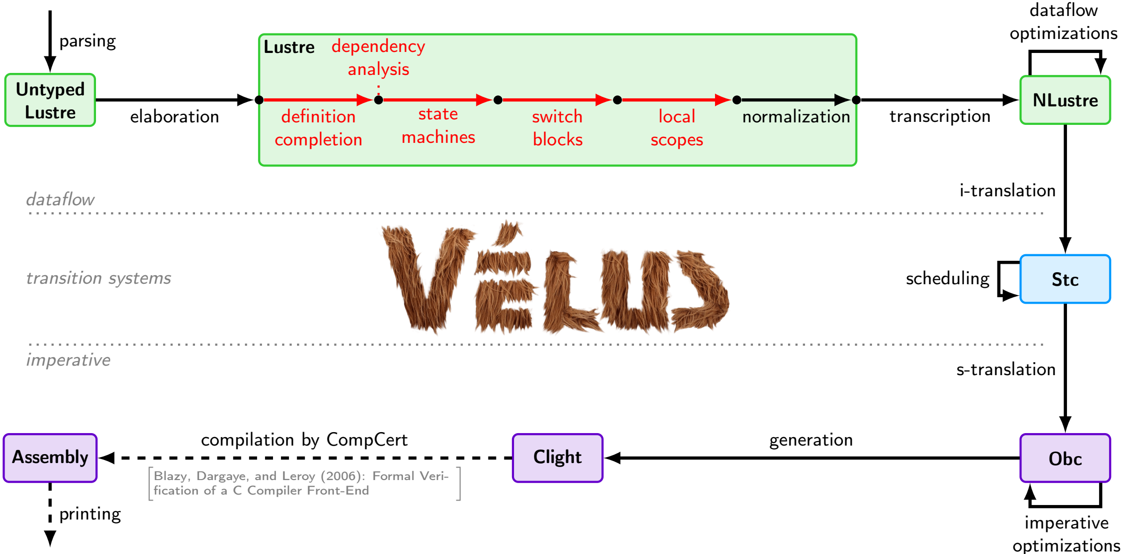
The Vélus Compiler



The Vélus Compiler

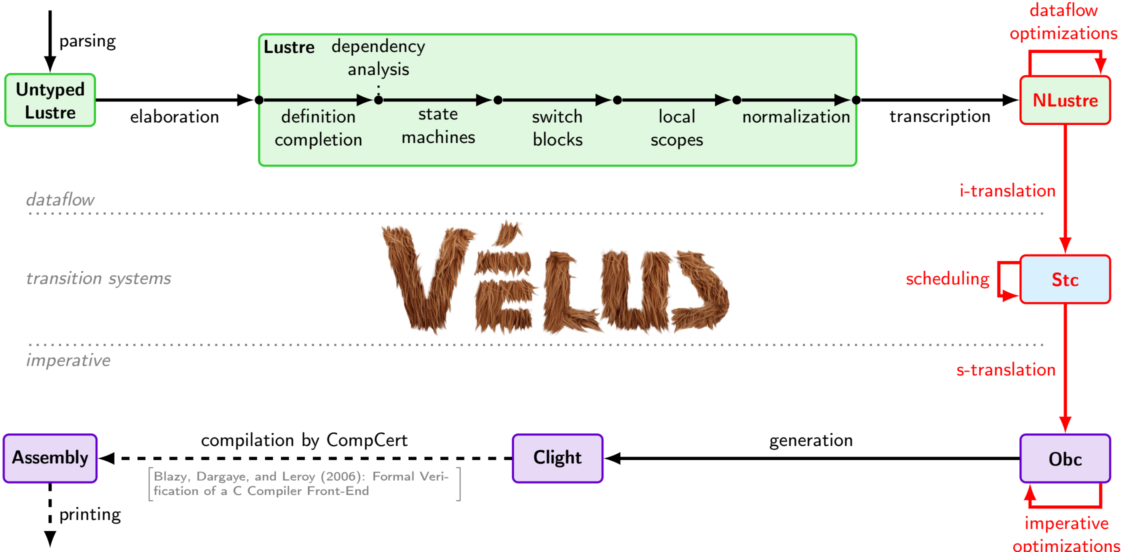


The Vélus Compiler

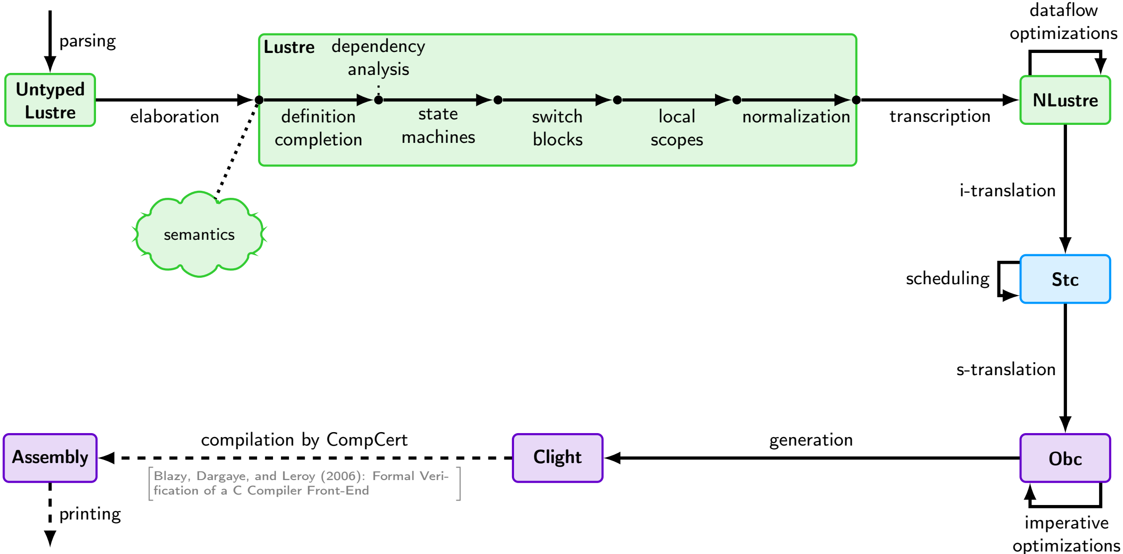


Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End

The Vélus Compiler

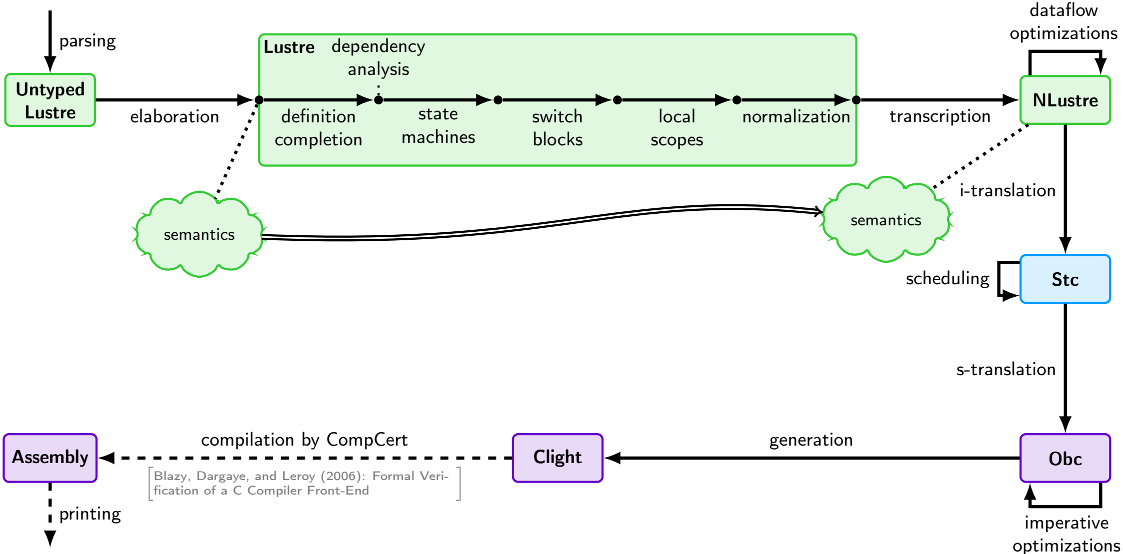


The Vélus Compiler

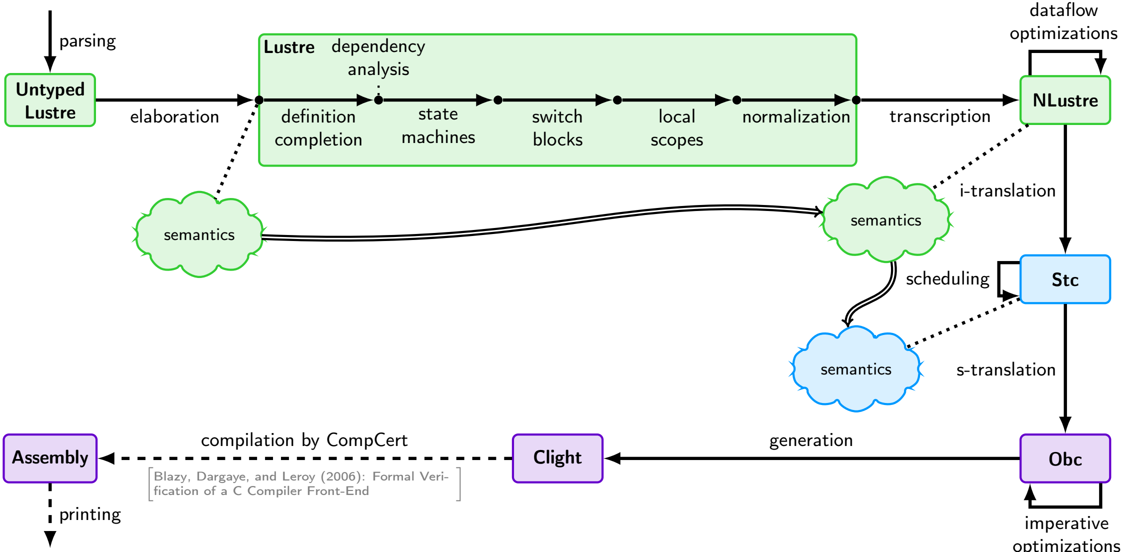


[Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]

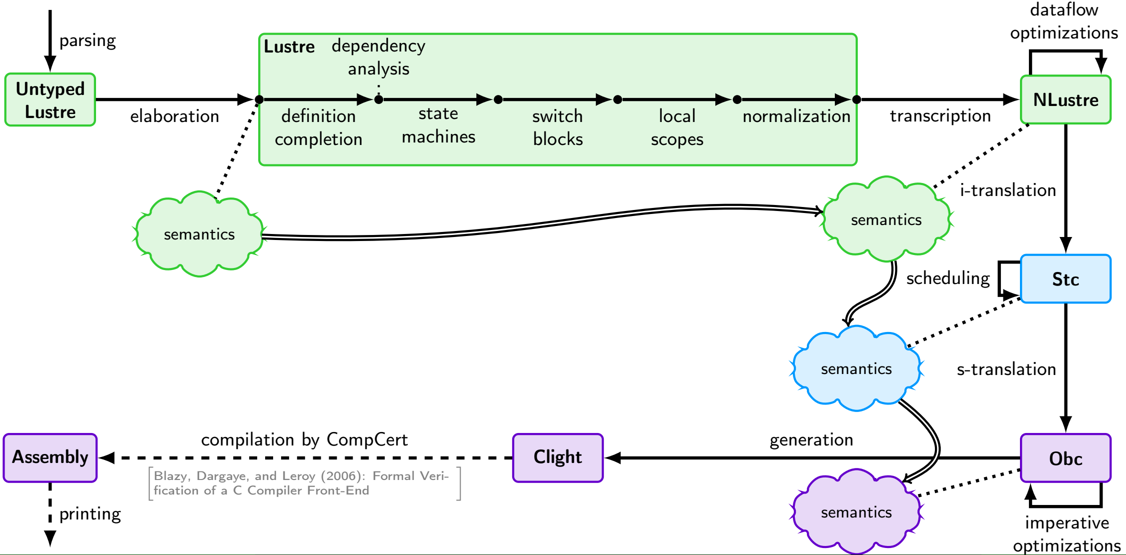
The Vélus Compiler



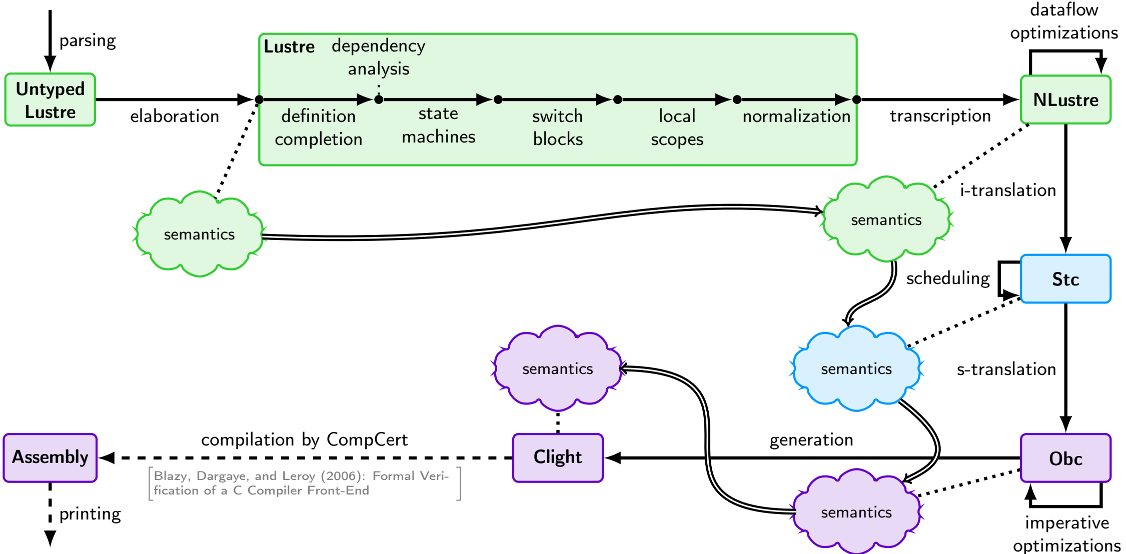
The Vélus Compiler



The Vélus Compiler

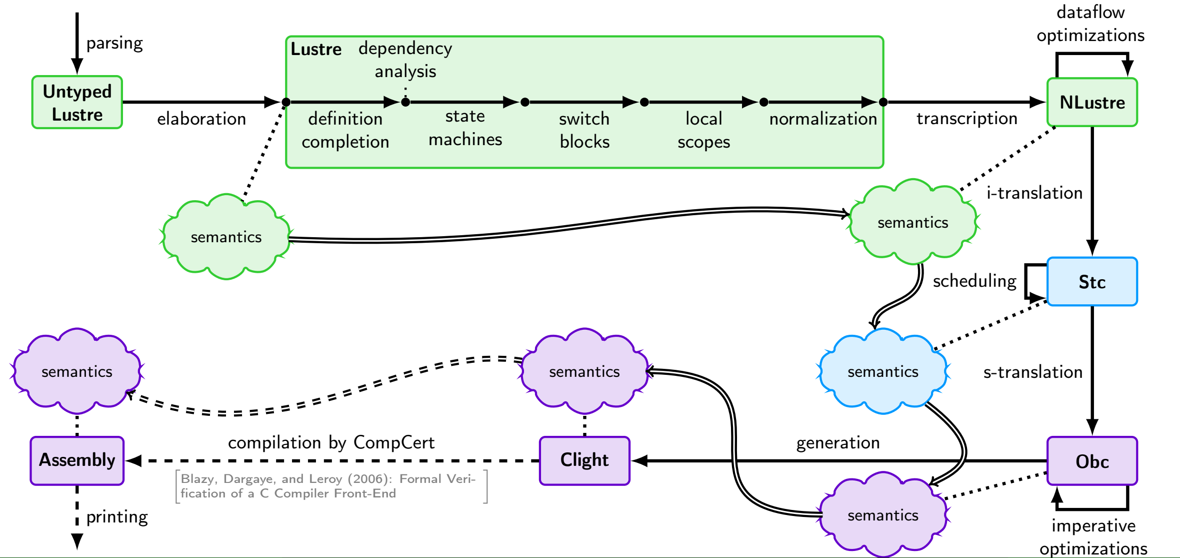


The Vélus Compiler



Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End

The Vélus Compiler



The Coq Interactive Theorem Prover

[Coq Development Team (2020): The Coq proof assistant reference manual]

- A functional programming language
- 'Extraction' to OCaml programs

```
Inductive ℕ :=
| O : ℕ
| S : ℕ → ℕ.

Fixpoint plus n m :=
  match n with
  | O ⇒ m
  | S n ⇒ S (plus n m)
  end.

Fact plus_n_O : ∀ n,
    plus n O = n.
Proof.
  induction n; simpl.
  - reflexivity.
  - now rewrite IHn.
Qed.

Fact plus_n_S : ∀ n m,
    plus n (S m) = S (plus n m).
Proof.
  induction n; intros; simpl.
  - reflexivity.
  - now rewrite IHn.
Qed.

Lemma plus_comm : ∀ n m,
    plus n m = plus m n.
Proof.
  induction n; intros.
  - now rewrite plus_n_O.
  - rewrite plus_n_S; simpl.
    now rewrite IHn.
Qed.
```

```
1 goal (ID 29)

- n : ℕ
- IHn : ∀ m : ℕ,
        plus n m = plus m n
- m : ℕ
______________________________________
plus (S n) m = plus m (S n)
```

550 nat.v 19:3 All Coq | 151 *goals* 9:0 All | 0 *response* 1:0 All

The Coq Interactive Theorem Prover

[Coq Development Team (2020): The Coq proof assistant reference manual]

- A functional programming language
- ‘Extraction’ to OCaml programs
- A specification language

```
Inductive ℕ :=
| O : ℕ
| S : ℕ → ℕ.

Fixpoint plus n m :=
  match n with
  | O ⇒ m
  | S n ⇒ S (plus n m)
  end.

Fact plus_n_O : ∀ n,
    plus n O = n.
Proof.
  induction n; simpl.
  - reflexivity.
  - now rewrite IHn.
Qed.

Fact plus_n_S : ∀ n m,
    plus n (S m) = S (plus n m).
Proof.
  induction n; intros; simpl.
  - reflexivity.
  - now rewrite IHn.
Qed.

Lemma plus_comm : ∀ n m,
    plus n m = plus m n.
Proof.
  induction n; intros.
  - now rewrite plus_n_O.
  - rewrite plus_n_S; simpl.
    now rewrite IHn.
Qed.
```

```
1 goal (ID 29)

n : ℕ
IHn : ∀ m : ℕ,
      plus n m = plus m n
m : ℕ
______________________________________
plus (S n) m = plus m (S n)
```

151 *goals* 9:0 All

550 nat.v 19:3 All Coq

0 *response* 1:0 All

The Coq Interactive Theorem Prover

[Coq Development Team (2020): The Coq proof assistant reference manual]

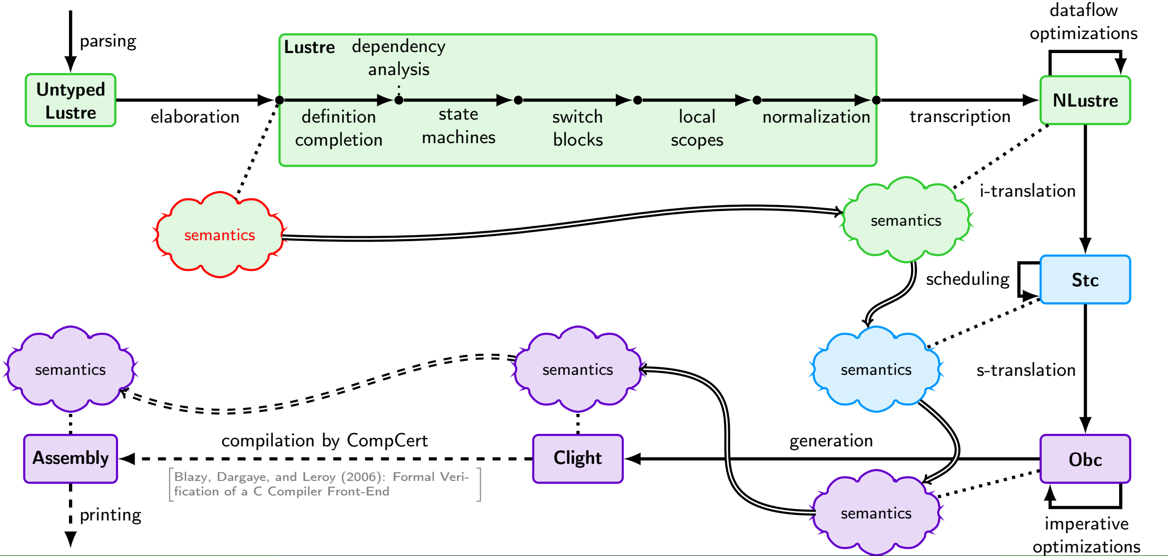
- A functional programming language
- ‘Extraction’ to OCaml programs
- A specification language
- Tactic-based interactive proof

```
Inductive ℕ :=
| O : ℕ
| S : ℕ → ℕ.

Fixpoint plus n m :=
  match n with
  | O ⇒ m
  | S n ⇒ S (plus n m)
  end.

Fact plus_n_O : ∀ n,
    plus n O = n.
Proof.
  induction n; simpl.
  - reflexivity.
  - now rewrite IHn.
Qed.

Fact plus_n_S : ∀ n m,
    plus n (S m) = S (plus n m).
Proof.
  induction n; intros; simpl.
  - reflexivity.
  - now rewrite IHn.
Qed.

Lemma plus_comm : ∀ n m,
    plus n m = plus m n.
Proof.
  induction n; intros.
  - now rewrite plus_n_O.
  - rewrite plus_n_S; simpl.
    now rewrite IHn.
Qed.
```

```
1 goal (ID 29)

n : ℕ
IHn : ∀ m : ℕ,
      plus n m = plus m n
m : ℕ
______________________________________
plus (S n) m = plus m (S n)
```

```
● 151  *goals*  9:0 All
● 550  nat.v  19:3 All  Coq
● 0  *response*  1:0 All
```

Relational Semantics of Vélus



Dataflow relational semantics

$$\frac{G(f) = \texttt{node}\ f\,(x_1,\ldots,x_n)\ \texttt{returns}\ (y_1,\ldots,y_m)\ blk \qquad \forall i,\ H(x_i) \equiv xss_i \qquad \forall j,\ H(y_j) \equiv yss_j \qquad G, H, (\text{base-of}\ (xs_1,\ldots,xs_n)) \vdash blk}{G \vdash f(xss) \Downarrow yss}$$

inc	5	4	1	3	2	8	3	...
o	5	9	10	13	15	23	26	...

Dataflow relational semantics

$$G(f) = \texttt{node}\ f\,(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ \mathit{blk}$$

$$\frac{\forall i,\ H(x_i) \equiv \mathit{xss}_i \qquad \forall j,\ H(y_j) \equiv \mathit{yss}_j \qquad G, H, (\text{base-of}\ (\mathit{xs}_1, \ldots, \mathit{xs}_n)) \vdash \mathit{blk}}{G \vdash f(\mathit{xss}) \Downarrow \mathit{yss}}$$

$$\frac{\forall i,\ H(\mathit{xs}_i) \equiv \mathit{vs}_i \qquad G, H, \mathit{bs} \vdash \mathit{es} \Downarrow [\mathit{vs}_i]^i}{G, H, \mathit{bs} \vdash \mathit{xs} = \mathit{es}}$$

Equations

If the clock is true, the right-hand expression is evaluated and its value is associated with the variable on the left-hand side.

$$\frac{\sigma(\mathtt{ck}) = tt,\ \ \sigma \vdash \mathtt{exp} \xrightarrow{\mathtt{k}} \mathtt{exp}',\ \ \sigma(\mathtt{id}) = \mathtt{k}}{\mathtt{id=(ck)exp} \xrightarrow{\sigma} \mathtt{id=(ck)exp}'}$$

If the clock is not true, the left-hand variable is not evaluated.

$$\frac{\sigma(\mathtt{ck}) \neq tt,\ \ \sigma(\mathtt{id}) = \bot}{\mathtt{id=(ck)exp} \xrightarrow{\sigma} \mathtt{id=(ck)exp}}$$

These rules define σ to be the solution of a fixpoint equation. Moreover, this solution must be unique (otherwise the program contains a *deadlock*; this problem will be detailed in section 4.1).

[Caspi, Pilaud, Halbwachs, and Plaice (1987): LUSTRE: A declarative language for programming synchronous systems]

inc	5	4	1	3	2	8	3	...
o	5	9	10	13	15	23	26	...

Dataflow relational semantics – in Coq

```
Inductive sem_exp:
[...]

with sem_equation:
| Seq:
  Forall2 (sem_exp G H bs) es ss →
  Forall2 (sem_var H) xs (concat ss) →
  sem_equation G H bs (xs, es)
[...]

with sem_node:
| Snode:
  find_node f G = Some n →
  Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_in)) ss →
  Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_out)) os →
  let bs := clocks_of ss in
  sem_block H bs n.(n_block) →
  sem_node f ss os.
```

$$\frac{\forall i,\ H(xs_i) \equiv vs_i \qquad G, H, bs \vdash es \Downarrow [vs_i]^i}{G, H, bs \vdash xs = es}$$

$$\frac{G(f) = \texttt{node}\ f\,(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ blk \qquad \forall i,\ H(x_i) \equiv xss_i \qquad \forall j,\ H(y_j) \equiv yss_j \qquad G, H, (\text{base-of}\ (xs_1, \ldots, xs_n)) \vdash blk}{G \vdash f(xss) \Downarrow yss}$$

Dataflow relational semantics – in Coq

```
Inductive sem_exp:
[...]

with sem_equation:
| Seq:
  Forall2 (sem_exp G H bs) es ss →
  Forall2 (sem_var H) xs (concat ss) →
  sem_equation G H bs (xs, es)
[...]

with sem_node:
| Snode:
  find_node f G = Some n →
  Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_in)) ss →
  Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_out)) os →
  let bs := clocks_of ss in
  sem_block H bs n.(n_block) →
  sem_node f ss os.
```

$$\frac{\forall i,\ H(xs_i) \equiv vs_i \qquad G, H, bs \vdash es \Downarrow [vs_i]^i}{\boxed{G, H, bs \vdash xs = es}}$$

$$\frac{\begin{array}{c} G(f) = \texttt{node}\ f\,(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ blk \\ \forall i,\ H(x_i) \equiv xss_i \qquad \forall j,\ H(y_j) \equiv yss_j \qquad G, H, (\text{base-of}\ (xs_1, \ldots, xs_n)) \vdash blk \end{array}}{G \vdash f(xss) \Downarrow yss}$$

Dataflow relational semantics – in Coq

```
Inductive sem_exp:
[...]

with sem_equation:
| Seq:
  Forall2 (sem_exp G H bs) es ss →
  Forall2 (sem_var H) xs (concat ss) →
  sem_equation G H bs (xs, es)
[...]

with sem_node:
| Snode:
  find_node f G = Some n →
  Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_in)) ss →
  Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_out)) os →
  let bs := clocks_of ss in
  sem_block H bs n.(n_block) →
  sem_node f ss os.
```

$$\frac{\boxed{\forall i,\ H(xs_i) \equiv vs_i} \qquad G, H, bs \vdash es \Downarrow [vs_i]^i}{G, H, bs \vdash xs = es}$$

$$\frac{\begin{array}{c} G(f) = \texttt{node}\ f\,(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ blk \\ \forall i,\ H(x_i) \equiv xss_i \qquad \forall j,\ H(y_j) \equiv yss_j \qquad G, H, (\text{base-of}\ (xs_1, \ldots, xs_n)) \vdash blk \end{array}}{G \vdash f(xss) \Downarrow yss}$$

Dataflow relational semantics – in Coq

```
Inductive sem_exp:
[...]

with sem_equation:
| Seq:
  Forall2 (sem_exp G H bs) es ss →
  Forall2 (sem_var H) xs (concat ss) →
  sem_equation G H bs (xs, es)
[...]

with sem_node:
| Snode:
  find_node f G = Some n →
  Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_in)) ss →
  Forall2 (fun x ⇒ sem_var H (Var x)) (List.map fst n.(n_out)) os →
  let bs := clocks_of ss in
  sem_block H bs n.(n_block) →
  sem_node f ss os.
```

$$\frac{\forall i,\ H(xs_i) \equiv vs_i \qquad \boxed{G, H, bs \vdash es \Downarrow [vs_i]^i}}{G, H, bs \vdash xs = es}$$

$$\frac{\begin{array}{c} G(f) = \textbf{node}\ f\,(x_1, \ldots, x_n)\ \textbf{returns}\ (y_1, \ldots, y_m)\ blk \\ \forall i,\ H(x_i) \equiv xss_i \qquad \forall j,\ H(y_j) \equiv yss_j \qquad G, H, (\text{base-of}\ (xs_1, \ldots, xs_n)) \vdash blk \end{array}}{G \vdash f(xss) \Downarrow yss}$$

fby operator semantics

inc	‹›	‹›	5	‹›	‹›	4	1	3	2	‹›	8	3	...
0 fby o	‹›	‹›	0										...
o = (0 fby o) + inc	‹›	‹›	5										...

```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

$$\mathrm{fby}\,(\langle\rangle \cdot xs)\,(\langle\rangle \cdot ys) \equiv \langle\rangle \cdot \mathrm{fby}\ xs\ ys$$
$$\mathrm{fby}\,(\langle v_1\rangle \cdot xs)\,(\langle v_2\rangle \cdot ys) \equiv \langle v_1\rangle \cdot \mathrm{fby1}\ v_2\ xs\ ys$$

fby operator semantics

`inc`	‹›	‹›	5	‹›	‹›	4	1	3	2	‹›	8	3	...
`0 fby o`	‹›	‹›	0	‹›	‹›	5	9	10	13	‹›	15	23	...
`o = (0 fby o) + inc`	‹›	‹›	5	‹›	‹›	9	10	13	15	‹›	23	26	...

```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

$$\begin{aligned}
\mathrm{fby}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) &\equiv \langle\rangle \cdot \mathrm{fby}\ xs\ ys \\
\mathrm{fby}\ (\langle v_1\rangle \cdot xs)\ (\langle v_2\rangle \cdot ys) &\equiv \langle v_1\rangle \cdot \mathrm{fby1}\ v_2\ xs\ ys \\
\mathrm{fby1}\ v_0\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) &\equiv \langle\rangle \cdot \mathrm{fby1}\ v_0\ xs\ ys \\
\mathrm{fby1}\ v_0\ (\langle v_1\rangle \cdot xs)\ (\langle v_2\rangle \cdot ys) &\equiv \langle v_0\rangle \cdot \mathrm{fby1}\ v_2\ xs\ ys
\end{aligned}$$

fby operator semantics

inc	⟨⟩	⟨⟩	5	⟨⟩	⟨⟩	4	1	3	2	⟨⟩	8	3	...
0 fby o	⟨⟩	⟨⟩	0	⟨⟩	⟨⟩	5	9	10	13	⟨⟩	15	23	...
o = (0 fby o) + inc	⟨⟩	⟨⟩	5	⟨⟩	⟨⟩	9	10	13	15	⟨⟩	23	26	...

```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

$$
\begin{aligned}
\mathrm{fby}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) &\equiv \langle\rangle \cdot \mathrm{fby}\ xs\ ys\\
\mathrm{fby}\ (\langle v_1\rangle \cdot xs)\ (\langle v_2\rangle \cdot ys) &\equiv \langle v_1\rangle \cdot \mathrm{fby1}\ v_2\ xs\ ys\\
\mathrm{fby1}\ v_0\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) &\equiv \langle\rangle \cdot \mathrm{fby1}\ v_0\ xs\ ys\\
\mathrm{fby1}\ v_0\ (\langle v_1\rangle \cdot xs)\ (\langle v_2\rangle \cdot ys) &\equiv \langle v_0\rangle \cdot \mathrm{fby1}\ v_2\ xs\ ys
\end{aligned}
$$

$$
\frac{G, H, bs \vdash es_0 \Downarrow [xs_i]^i \qquad G, H, bs \vdash es_1 \Downarrow [ys_i]^i \qquad \forall i,\ \mathrm{fby}\ xs_i\ ys_i \equiv vs_i}{G, H, bs \vdash es_0\ \texttt{fby}\ es_1 \Downarrow [vs_i]^i}
$$

Stream semantics of switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```

step	...
last mA	...
last mB	...
mA	...
mB	...

Stream semantics of switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```

$$\mathsf{when}^{C}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot cs) \equiv \langle\rangle \cdot \mathsf{when}^{C}\ xs\ cs$$
$$\mathsf{when}^{C}\ (\langle v\rangle \cdot xs)\ (\langle C\rangle \cdot cs) \equiv \langle v\rangle \cdot \mathsf{when}^{C}\ xs\ cs$$
$$\mathsf{when}^{C}\ (\langle v\rangle \cdot xs)\ (\langle C'\rangle \cdot cs) \equiv \langle\rangle \cdot \mathsf{when}^{C}\ xs\ cs$$

step	...
last mA	...
last mB	...
mA	...
mB	...

Stream semantics of switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```

$$\begin{aligned}
\mathsf{when}^{C}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot cs) &\equiv \langle\rangle \cdot \mathsf{when}^{C}\ xs\ cs \\
\mathsf{when}^{C}\ (\langle v\rangle \cdot xs)\ (\langle C\rangle \cdot cs) &\equiv \langle v\rangle \cdot \mathsf{when}^{C}\ xs\ cs \\
\mathsf{when}^{C}\ (\langle v\rangle \cdot xs)\ (\langle C'\rangle \cdot cs) &\equiv \langle\rangle \cdot \mathsf{when}^{C}\ xs\ cs
\end{aligned}$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \qquad \forall i,\ G, \mathsf{when}^{C_i}\ (H, bs)\ cs \vdash blks_i}{G, H, bs \vdash \texttt{switch}\ e\ [C_i\ \texttt{do}\ blks_i]^i\ \texttt{end}}$$

step	...
last mA	...
last mB	...
mA	...
mB	...

Stream semantics of switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```

$$\begin{aligned}
\mathrm{when}^{C}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot cs) &\equiv \langle\rangle \cdot \mathrm{when}^{C}\ xs\ cs\\
\mathrm{when}^{C}\ (\langle v\rangle \cdot xs)\ (\langle C\rangle \cdot cs) &\equiv \langle v\rangle \cdot \mathrm{when}^{C}\ xs\ cs\\
\mathrm{when}^{C}\ (\langle v\rangle \cdot xs)\ (\langle C'\rangle \cdot cs) &\equiv \langle\rangle \cdot \mathrm{when}^{C}\ xs\ cs
\end{aligned}$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \qquad \forall i,\ G, \mathrm{when}^{C_i}\ (H, bs)\ cs \vdash blks_i}{G, H, bs \vdash \texttt{switch}\ e\ [C_i\ \texttt{do}\ blks_i]^i\ \texttt{end}}$$

step	T	T	T	T	T	T	T	...
last mA	T	T	F	F	T	T	F	...
last mB	F	T	T	F	F	T	T	...
mA	T	F	F	T	T	F	F	...
mB	T	T	F	F	T	T	F	...

Stream semantics of switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```

$$\mathsf{when}^{C}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot cs) \equiv \langle\rangle \cdot \mathsf{when}^{C}\ xs\ cs$$

$$\mathsf{when}^{C}\ (\langle v\rangle \cdot xs)\ (\langle C\rangle \cdot cs) \equiv \langle v\rangle \cdot \mathsf{when}^{C}\ xs\ cs$$

$$\mathsf{when}^{C}\ (\langle v\rangle \cdot xs)\ (\langle C'\rangle \cdot cs) \equiv \langle\rangle \cdot \mathsf{when}^{C}\ xs\ cs$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \qquad \forall i,\ G, \mathsf{when}^{C_i}\ (H, bs)\ cs \vdash blks_i}{G, H, bs \vdash \mathtt{switch}\ e\ [C_i\ \mathsf{do}\ blks_i]^i\ \mathsf{end}}$$

step	F		F	F	F	F	F	F	F	...
last mA	T		F	F	F	T	T	F	F	...
last mB	F		T	T	F	F	T	T	T	...
mA	T		F	F	F	T	T	F	F	...
mB	F		T	T	F	F	T	T	T	...

Stream semantics of switch blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```

$$\mathsf{when}^C\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot cs) \equiv \langle\rangle \cdot \mathsf{when}^C\ xs\ cs$$
$$\mathsf{when}^C\ (\langle v\rangle \cdot xs)\ (\langle C\rangle \cdot cs) \equiv \langle v\rangle \cdot \mathsf{when}^C\ xs\ cs$$
$$\mathsf{when}^C\ (\langle v\rangle \cdot xs)\ (\langle C'\rangle \cdot cs) \equiv \langle\rangle \cdot \mathsf{when}^C\ xs\ cs$$

$$\frac{G, H, bs \vdash e \Downarrow [cs] \qquad \forall i,\ G, \mathsf{when}^{C_i}\ (H, bs)\ cs \vdash blks_i}{G, H, bs \vdash \texttt{switch}\ e\ [C_i\ \texttt{do}\ blks_i]^i\ \texttt{end}}$$

step	F	T	T	F	F	T	F	T	F	T	F	T	F	F	T	...
last mA	T	T	T	F	F	F	F	F	T	T	T	T	F	F	F	...
last mB	F	F	T	T	T	T	F	F	F	F	T	T	T	T	T	...
mA	T	T	F	F	F	F	F	T	T	T	T	F	F	F	F	...
mB	F	T	T	T	T	F	F	F	F	T	T	T	T	T	F	...

Stream semantics of reset blocks and state machines

$$\mathsf{mask}^{k}_{k'}\ (\mathtt{F} \cdot rs)\ (sv \cdot xs) \equiv (\mathsf{if}\ k' = k\ \mathsf{then}\ sv\ \mathsf{else}\ \langle\rangle) \cdot \mathsf{mask}^{k}_{k'}\ rs\ xs$$

$$\mathsf{mask}^{k}_{k'}\ (\mathtt{T} \cdot rs)\ (sv \cdot xs) \equiv (\mathsf{if}\ k' + 1 = k\ \mathsf{then}\ sv\ \mathsf{else}\ \langle\rangle) \cdot \mathsf{mask}^{k}_{k'+1}\ rs\ xs$$

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

$$\frac{G, H, bs \vdash e \Downarrow [ys] \qquad \mathsf{bools\text{-}of}\ ys \equiv rs \qquad \forall k,\ G, \mathsf{mask}^{k}\ rs\ (H, bs) \vdash blks}{G, H, bs \vdash \mathtt{reset}\ blks\ \mathtt{every}\ e}$$

- `reset` block $\mapsto$ mask operator

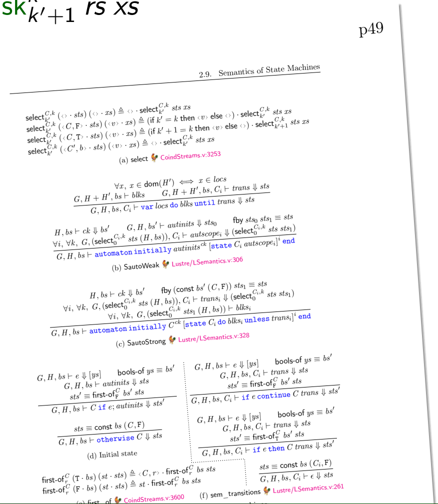
Stream semantics of reset blocks and state machines

$$\mathsf{mask}^{k}_{k'}\ (\mathtt{F}\cdot rs)\ (sv\cdot xs) \equiv (\mathsf{if}\ k' = k\ \mathsf{then}\ sv\ \mathsf{else}\ \langle\rangle)\cdot \mathsf{mask}^{k}_{k'}\ rs\ xs$$
$$\mathsf{mask}^{k}_{k'}\ (\mathtt{T}\cdot rs)\ (sv\cdot xs) \equiv (\mathsf{if}\ k'+1 = k\ \mathsf{then}\ sv\ \mathsf{else}\ \langle\rangle)\cdot \mathsf{mask}^{k}_{k'+1}\ rs\ xs$$

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

$$\frac{G, H, bs \vdash e \Downarrow [ys] \qquad \mathsf{bools\text{-}of}\ ys \equiv rs \qquad \forall k,\ G, \mathsf{mask}^{k}\ rs\ (H, bs) \vdash blks}{G, H, bs \vdash \mathtt{reset}\ blks\ \mathtt{every}\ e}$$

- $\mathtt{reset}$ block $\mapsto$ mask operator
- state machines $\mapsto$ select operator


p49

2.9. Semantics of State Machines

$$\begin{aligned}
\mathsf{select}^{C,k}_{k'}\ (\langle\rangle\cdot sts)\ (\langle\rangle\cdot xs) &\triangleq \langle\rangle\cdot\mathsf{select}^{C,k}_{k'}\ sts\ xs\\
\mathsf{select}^{C,k}_{k'}\ (\langle C,\mathtt{F}\rangle\cdot sts)\ (\langle v\rangle\cdot xs) &\triangleq (\mathsf{if}\ k'=k\ \mathsf{then}\ \langle v\rangle\ \mathsf{else}\ \langle\rangle)\cdot\mathsf{select}^{C,k}_{k'}\ sts\ xs\\
\mathsf{select}^{C,k}_{k'}\ (\langle C,\mathtt{T}\rangle\cdot sts)\ (\langle v\rangle\cdot xs) &\triangleq (\mathsf{if}\ k'+1=k\ \mathsf{then}\ \langle v\rangle\ \mathsf{else}\ \langle\rangle)\cdot\mathsf{select}^{C,k}_{k'+1}\ sts\ xs\\
\mathsf{select}^{C,k}_{k'}\ (\langle C',b\rangle\cdot sts)\ (\langle v\rangle\cdot xs) &\triangleq \langle\rangle\cdot\mathsf{select}^{C,k}_{k'}\ sts\ xs
\end{aligned}$$

(a) select CoindStreams.v:3253

$$\frac{\forall x,\ x\in\mathsf{dom}(H') \iff x\in locs \qquad G, H+H', bs\vdash blks \qquad G, H+H', bs, C_i\vdash trans\Downarrow sts}{G, H, bs, C_i\vdash \mathtt{var}\ locs\ \mathtt{do}\ blks\ \mathtt{until}\ trans\Downarrow sts}$$

$$\frac{H, bs\vdash ck\Downarrow bs' \qquad G, H, bs'\vdash autinits\Downarrow sts_0 \qquad \mathsf{fby}\ sts_0\ sts_1\equiv sts \qquad \forall i,\forall k,\ G,(\mathsf{select}^{C_i,k}_0\ sts\ (H,bs)), C_i\vdash autscope_i\Downarrow(\mathsf{select}^{C_i,k}_0\ sts\ sts_1)}{G, H, bs\vdash \mathtt{automaton}\ \mathtt{initially}\ autinits^{ck}\ [\mathtt{state}\ C_i\ autscope_i]^i\ \mathtt{end}}$$

(b) SautoWeak Lustre/LSemantics.v:306

$$\frac{H, bs\vdash ck\Downarrow bs' \qquad \mathsf{fby}\ (\mathsf{const}\ bs'\ (C,\mathtt{F}))\ sts_1\equiv sts \qquad \forall i,\forall k,\ G,(\mathsf{select}^{C_i,k}_0\ sts\ (H,bs)), C_i\vdash trans_i\Downarrow(\mathsf{select}^{C_i,k}_0\ sts\ sts_1) \qquad \forall i,\forall k,\ G,(\mathsf{select}^{C_i,k}_0\ sts_1\ (H,bs))\vdash blks_i}{G, H, bs\vdash \mathtt{automaton}\ \mathtt{initially}\ C^{ck}\ [\mathtt{state}\ C_i\ \mathtt{do}\ blks_i\ \mathtt{unless}\ trans_i]^i\ \mathtt{end}}$$

(c) SautoStrong Lustre/LSemantics.v:328

$$\frac{G, H, bs\vdash e\Downarrow[ys] \qquad \mathsf{bools\text{-}of}\ ys\equiv bs' \qquad G, H, bs\vdash autinits\Downarrow sts \qquad sts'\equiv\mathsf{first\text{-}of}^{C}_{\mathtt{F}}\ bs'\ sts}{G, H, bs\vdash C\ \mathtt{if}\ e; autinits\Downarrow sts'}$$

$$\frac{sts\equiv\mathsf{const}\ bs\ (C,\mathtt{F})}{G, H, bs\vdash \mathtt{otherwise}\ C\Downarrow sts}$$

(d) Initial state

$$\frac{G, H, bs\vdash e\Downarrow[ys] \qquad \mathsf{bools\text{-}of}\ ys\equiv bs' \qquad G, H, bs, C_i\vdash trans\Downarrow sts \qquad sts'\equiv\mathsf{first\text{-}of}^{C}_{\mathtt{F}}\ bs'\ sts}{G, H, bs, C_i\vdash \mathtt{if}\ e\ \mathtt{continue}\ C\ trans\Downarrow sts'}$$

$$\frac{G, H, bs\vdash e\Downarrow[ys] \qquad \mathsf{bools\text{-}of}\ ys\equiv bs' \qquad G, H, bs, C_i\vdash trans\Downarrow sts \qquad sts'\equiv\mathsf{first\text{-}of}^{C}_{\mathtt{T}}\ bs'\ sts}{G, H, bs, C_i\vdash \mathtt{if}\ e\ \mathtt{then}\ C\ trans\Downarrow sts'}$$

$$\begin{aligned}
\mathsf{first\text{-}of}^{C}_{r}\ (\mathtt{T}\cdot bs)\ (st\cdot sts) &\triangleq \langle C, r\rangle\cdot\mathsf{first\text{-}of}^{C}_{r}\ bs\ sts\\
\mathsf{first\text{-}of}^{C}_{r}\ (\mathtt{F}\cdot bs)\ (st\cdot sts) &\triangleq st\cdot\mathsf{first\text{-}of}^{C}_{r}\ bs\ sts
\end{aligned}$$

(e) first_of CoindStreams.v:3600

$$\frac{sts\equiv\mathsf{const}\ bs\ (C_i,\mathtt{F})}{G, H, bs, C_i\vdash\epsilon\Downarrow sts}$$

(f) sem_transitions Lustre/LSemantics.v:261

Stream semantics of reset blocks and state machines

$$\text{mask}^{k}_{k'}\ (\texttt{F} \cdot rs)\ (sv \cdot xs) \equiv (\text{if } k' = k \text{ then } sv \text{ else } \langle\rangle) \cdot \text{mask}^{k}_{k'}\ rs\ xs$$

$$\text{mask}^{k}_{k'}\ (\texttt{T} \cdot rs)\ (sv \cdot xs) \equiv (\text{if } k' + 1 = k \text{ then } sv \text{ else } \langle\rangle) \cdot \text{mask}^{k}_{k'+1}\ rs\ xs$$

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

$$\frac{G, H, bs \vdash e \Downarrow [ys] \qquad \text{bools-of } ys \equiv rs \qquad \forall k,\ G, \text{mask}^{k}\ rs\ (H, bs) \vdash blks}{G, H, bs \vdash \texttt{reset } blks \texttt{ every } e}$$

- `reset` block $\mapsto$ mask operator
- state machines $\mapsto$ select operator

Proving semantic meta-properties

Prove properties of the semantic model:

- Determinism of the semantics:
  if $G \vdash f(xs) \Downarrow ys_1$ **and** $G \vdash f(xs) \Downarrow ys_2$ **then** $ys_1 \equiv ys_2$

Proving semantic meta-properties

Prove properties of the semantic model:

- Determinism of the semantics:
  if $G \vdash f(xs) \Downarrow ys_1$ **and** $G \vdash f(xs) \Downarrow ys_2$ **then** $ys_1 \equiv ys_2$
- Clock-system correctness:
  if $\Gamma \vdash e : ck$ **and** $G, H, bs \vdash e \Downarrow vs$ **then** $H, bs \vdash ck \Downarrow (\text{clock-of } vs)$

Proving semantic meta-properties

Prove properties of the semantic model:

- Determinism of the semantics:
  if $G \vdash f(xs) \Downarrow ys_1$ **and** $G \vdash f(xs) \Downarrow ys_2$ **then** $ys_1 \equiv ys_2$
- Clock-system correctness:
  if $\Gamma \vdash e : ck$ **and** $G, H, bs \vdash e \Downarrow vs$ **then** $H, bs \vdash ck \Downarrow (\text{clock-of } vs)$

Proof by induction on the syntax, inversion of the semantics:

- ...
- variable: inverting $G, H, bs \vdash x \Downarrow [vs]$ tells us $H(x) \equiv vs$. What now ?
- ...

Dependency Analysis

Consider a program with the following definitions:

- `x = x;` admits all value
- `x = x + 1;` admits no value

Dependency Analysis

Consider a program with the following definitions:

- `x = x;` admits all value
- `x = x + 1;` admits no value

Not possible to prove any property of the stream of `x`.
We can only reason on program without dependency cycle.

Dependency Analysis

Consider a program with the following definitions:

- `x = x;` admits all value
- `x = x + 1;` admits no value

Not possible to prove any property of the stream of `x`.
We can only reason on program without dependency cycle.

Solution: dependency analysis [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- node-by-node graph analysis (no type system [Cuoq and Pouzet (2001): Modular Causality in a Synchronous Stream Language])

Dependency Analysis

Consider a program with the following definitions:

- `x = x;` admits all value
- `x = x + 1;` admits no value

Not possible to prove any property of the stream of `x`.
We can only reason on program without dependency cycle.

Solution: dependency analysis [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- node-by-node graph analysis (no type system [Cuoq and Pouzet (2001): Modular Causality in a Synchronous Stream Language])
- extended to handle control blocks (using labels)

Dependency Analysis

Consider a program with the following definitions:

- `x = x;` admits all value
- `x = x + 1;` admits no value

Not possible to prove any property of the stream of `x`.
We can only reason on program without dependency cycle.

Solution: dependency analysis [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- node-by-node graph analysis (no type system [Cuoq and Pouzet (2001): Modular Causality in a Synchronous Stream Language])
- extended to handle control blocks (using labels)
- verified graph analysis algorithm: produces a witness of acyclicity

Dependency Analysis

Consider a program with the following definitions:

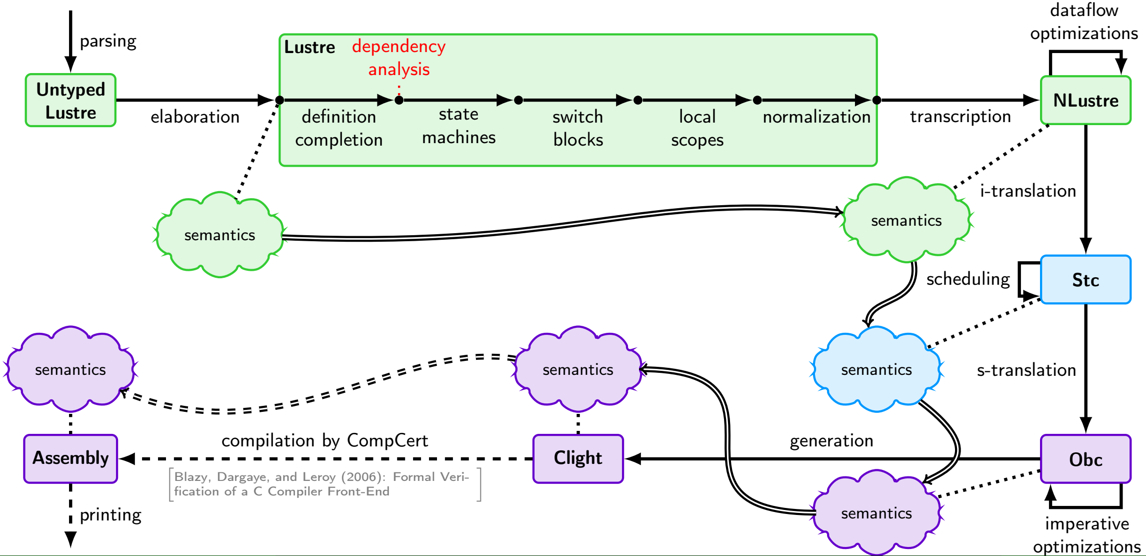
- `x = x;` admits all value
- `x = x + 1;` admits no value

Not possible to prove any property of the stream of `x`.
We can only reason on program without dependency cycle.

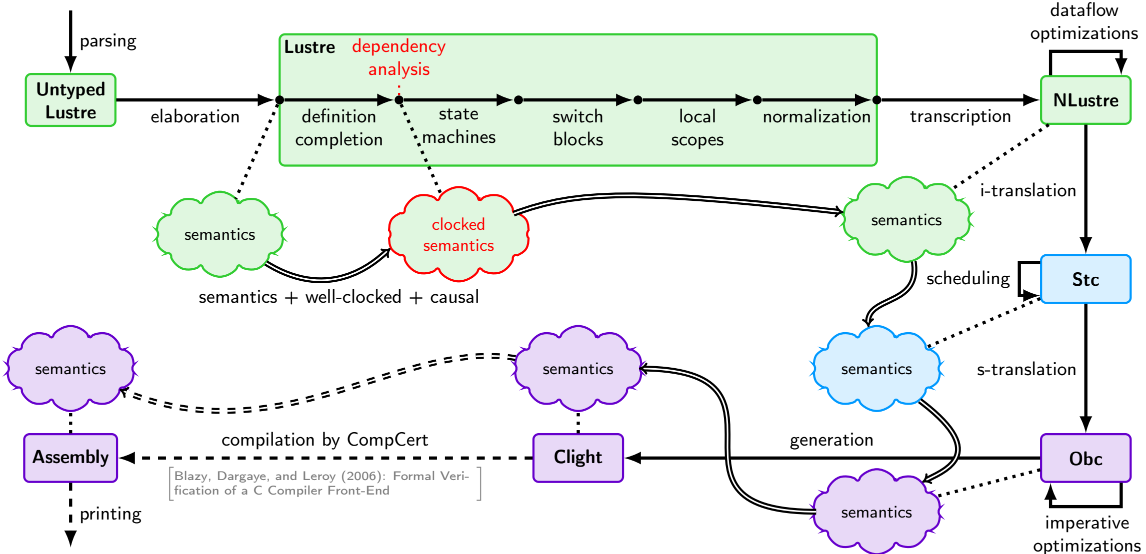
Solution: dependency analysis [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

- node-by-node graph analysis (no type system [Cuoq and Pouzet (2001): Modular Causality in a Synchronous Stream Language])
- extended to handle control blocks (using labels)
- verified graph analysis algorithm: produces a witness of acyclicity
- Used to prove properties of the semantics (clock-system correctness, determinism)

Instrumented Semantic Model

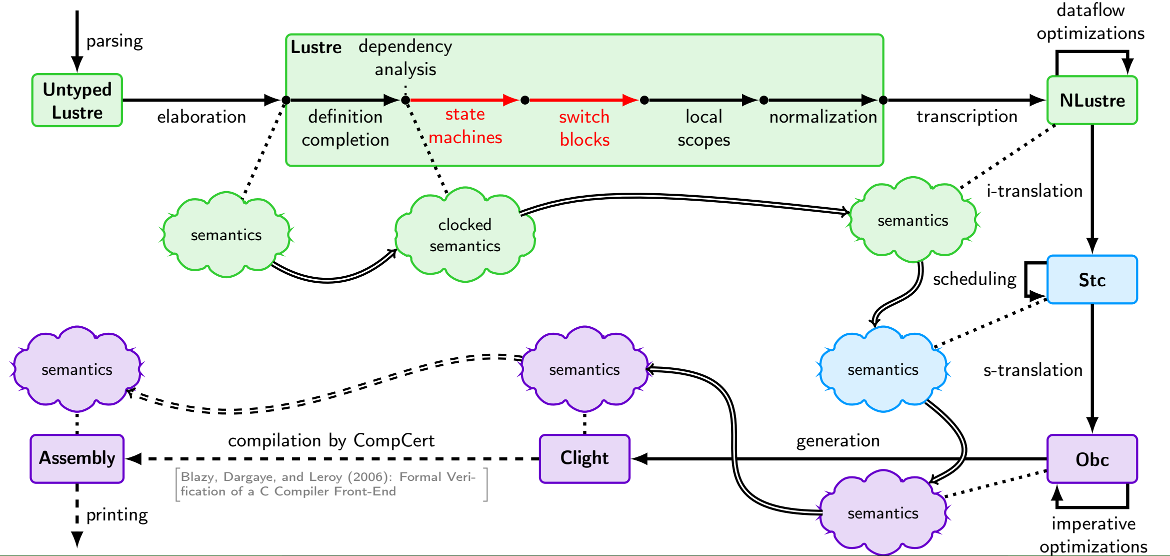


Instrumented Semantic Model



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

Compilation of State Machines and Switch Blocks



Compilation of State Machines

```
node feed_pause(pause : bool) returns (ena, step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);

  automaton initially Feeding
  state Feeding do
    ena = true;
    automaton initially Starting
      state Starting do
        step = true fby false
      unless time >= 750 then Moving

      state Moving do
        step = true fby false
      unless time >= 500 then Moving

    end;
  unless pause then Holding

  state Holding do
    step = false;
    automaton initially Waiting
      state Waiting do
        ena = true
      unless time >= 500 then Modulating

      state Modulating do
        ena = pwm(true)

    end;
  unless
    | not pause and time >= 750 then Feeding
    | not pause continue Feeding
  end
tel
```

H*

Compilation of State Machines

```
node feed_pause(pause : bool) returns (ena, step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);

  automaton initially Feeding
    state Feeding do
      ena = true;
      automaton initially Starting
        state Starting do
          step = true fby false
        unless time >= 750 then Moving

        state Moving do
          step = true fby false
        unless time >= 500 then Moving

      end;
    unless pause then Holding

    state Holding do
      step = false;
      automaton initially Waiting
        state Waiting do
          ena = true
        unless time >= 500 then Modulating

        state Modulating do
          ena = pwm(true)

      end;
    unless
      | not pause and time >= 750 then Feeding
      | not pause continue Feeding
  end
tel
```

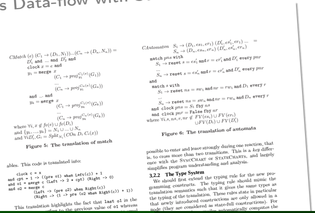
Compilation of State Machines

```
automaton initially Starting
  state Starting do
    step = true fby false
  unless time >= 750 then Moving
  state Moving do
    step = true fby false
  unless time >= 500 then Moving
end
```

Compilation of State Machines

```
automaton initially Starting
  state Starting do
    step = true fby false
  unless time >= 750 then Moving

  state Moving do
    step = true fby false
  unless time >= 500 then Moving
end
```

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



$$CMatch\ (e)\ (C_1 \to (D_1, N_1))...(C_n \to (D_n, N_n)) = D'_1 \text{ and } ... \text{ and } D'_2 \text{ and clock } x = e \text{ and } y_1 = \text{merge } x\ (C_1 \to proj_{y_1}^{C_1(x)}(G_1)) ... (C_n \to proj_{y_1}^{C_n(x)}(G_n)) \text{ and } ... \text{ and } y_k = \text{merge } x\ (C_1 \to proj_{y_k}^{C_1(x)}(G_k)) ... (C_n \to proj_{y_k}^{C_n(x)}(G_k))$$

where $\forall i, x \notin fv(e) \cup fv(D_i)$ and $\{y_1, ..., y_k\} = N_1 \cup ... \cup N_n$ and $\forall i D'_i, G_i = Split_{N_i}(COn\ D_i\ C_i(x))$

Figure 5: The translation of match

ables. This code is translated into:

```
    clock c = x
and cpt = 1 -> ((pre o1) when Left(c)) + 1
and o1 = merge c (Left -> 2 * cpt) (Right -> 0)
and o2 = merge c
        (Left -> (pre o2) when Right(c))
        (Right -> (1 -> pre (o2 when Right(c)) + 1))
```

This translation highlights the fact that last o1 in the ... refers to the previous value of o1 whereas ...

$$CAutomaton\ S_1 \to (D_1, es_1, er_1)\ (D'_1, es'_1, er_1) ... S_n \to (D_n, es_n, er_n)\ (D'_n, es'_n, er_n) =$$

match pns with $S_1 \to$ reset $s = es'_1$ and $r = er'_1$ and D'_1 every pnr ... $S_n \to$ reset $s = es'_n$ and $r = er'_n$ and D'_n every pnr
and
match s with $S_1 \to$ reset $ns = sw_1$ and $nr = rw_1$ and D_1 every r ... $S_n \to$ reset $ns = sw_n$ and $nr = rw_n$ and D_n every r
and clock $pns = S_1$ fby ns
and clock pnr = False fby nr
where $\forall i.s, ns, r, nr \notin FV(es_i) \cup FV(er_i) \cup FV(D_i) \cup FV(D'_i)$

Figure 6: The translation of automata

possible to enter and leave strongly during one reaction, that is, to cross more than two transitions. This is a *key* difference with the SyncChart or StateCharts, and largely simplifies program understanding and analysis.

3.2.2 The Type System

We should first extend the typing rule for the new programming constructs. The typing rule should mimic the translation semantics such that it gives the same types as the typing of the translation. These rules state in particular that newly introduced constructions are only allowed in a node (they are considered as state-full constructions). For ... compiler automatically computes the ...

Compilation of State Machines

```
automaton initially Starting
  state Starting do
    step = true fby false
  unless time >= 750 then Moving

  state Moving do
    step = true fby false
  unless time >= 500 then Moving

end
```

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

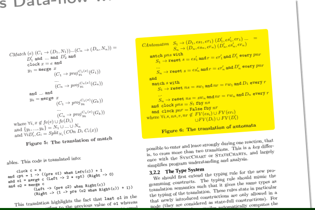


Figure 5: The translation of match

Figure 6: The translation of automata

3.2.2 The Type System

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines

```
automaton initially Starting
  state Starting do
    step = true fby false
  unless time >= 750 then Moving

  state Moving do
    step = true fby false
  unless time >= 500 then Moving

end
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

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

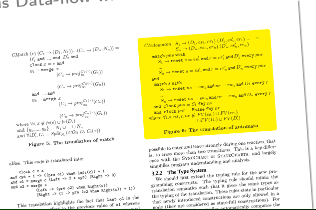


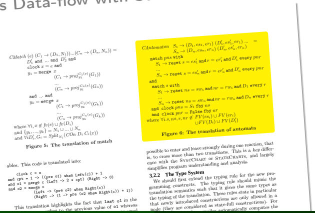
Figure 5: The translation of match

Figure 6: The translation of automata

Compilation of State Machines

```
automaton initially Starting
  state Starting do
    step = true fby false
  unless time >= 750 then Moving
  state Moving do
    step = true fby false
  unless time >= 500 then Moving
end
```

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




```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines

```
automaton initially Starting
  state Starting do
    step = true fby false
  unless time >= 750 then Moving

  state Moving do
    step = true fby false
  unless time >= 500 then Moving

end
```

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

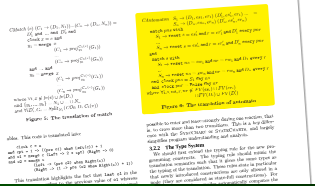


Figure 5: The translation of match
Figure 6: The translation of automata

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines

```
automaton initially Starting
  state Starting do
    step = true fby false
  unless time >= 750 then Moving

  state Moving do
    step = true fby false
  unless time >= 500 then Moving
end
```

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

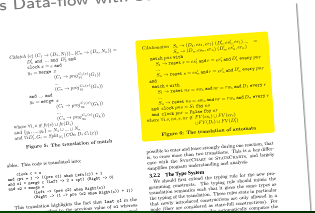


Figure 5: The translation of match

Figure 6: The translation of automata

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines

```
automaton initially Starting
  state Starting do
    step = true fby false
  unless time >= 750 then Moving

  state Moving do
    step = true fby false
  unless time >= 500 then Moving

end
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

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

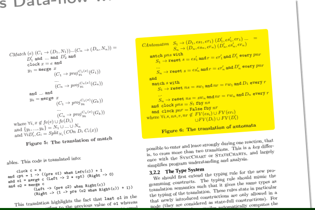


Figure 5: The translation of match

Figure 6: The translation of automata

3.2.2 The Type System

Generating Fresh Identifiers during Compilation

generating new identifiers?

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Generating Fresh Identifiers during Compilation

generating new identifiers?

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

In OCaml:

```
let fresh =
  let cnt = ref 0 in
  fun () ->
    cnt := !cnt + 1; !cnt
```

Generating Fresh Identifiers during Compilation

generating new identifiers?

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

In OCaml:

```
let fresh =
  let cnt = ref 0 in
  fun () ->
    cnt := !cnt + 1; !cnt
```

But Coq is a pure functional language!

Generating Fresh Identifiers during Compilation

generating new identifiers?

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

In OCaml:

```
let fresh =
  let cnt = ref 0 in
  fun () ->
    cnt := !cnt + 1; !cnt
```

But Coq is a pure functional language!

- Monadic approach: `Fresh`

Generating Fresh Identifiers during Compilation

generating new identifiers?

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

In OCaml:

```
let fresh =
  let cnt = ref 0 in
  fun () ->
    cnt := !cnt + 1; !cnt
```

But Coq is a pure functional language!

- Monadic approach: `Fresh`
- Access OCaml code: `gensym`

Compilation of State Machines – Coq Implementation

```
Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states ⇒
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
     Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
                [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) ⇒
     let transeq := Beq ([st; res], trans_exp unl e) in
     (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) ⇒
     do blks' ← mmap auto_block blks;
     ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines – Coq Implementation

```
Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states ⇒
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
     Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
                [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) ⇒
     let transeq := Beq ([st; res], trans_exp unl e) in
     (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) ⇒
     do blks' ← mmap auto_block blks;
     ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines – Coq Implementation

```
Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states ⇒
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
     Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
                [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) ⇒
     let transeq := Beq ([st; res], trans_exp unl e) in
     (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) ⇒
     do blks' ← mmap auto_block blks;
     ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Common monadic notation:

do x ← e1; e2 ∼ let x := e1 in e2

Compilation of State Machines – Coq Implementation

```
Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states ⇒
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
     Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
                [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) ⇒
     let transeq := Beq ([st; res], trans_exp unl e) in
     (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) ⇒
     do blks' ← mmap auto_block blks;
     ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines – Coq Implementation

```
Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states ⇒
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
     Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
                [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) ⇒
     let transeq := Beq ([st; res], trans_exp unl e) in
     (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) ⇒
     do blks' ← mmap auto_block blks;
     ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines – Coq Implementation

```
Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states ⇒
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
     Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
                [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) ⇒
     let transeq := Beq ([st; res], trans_exp unl e) in
     (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) ⇒
     do blks' ← mmap auto_block blks;
     ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])
```

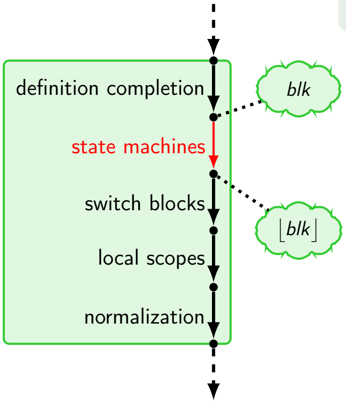
```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines – Coq Implementation

```
Fixpoint auto_block (blk: block) : Fresh block :=
match blk with
| ...
| Bauto Strong ck (_, oth) states ⇒
  do pst ← fresh_ident; do pres ← fresh_ident;
  do st ← fresh_ident; do res ← fresh_ident;
  let stateq :=
     Beq ([pst; pres],
          [Efby [Eenum oth; Eenum false]
                [Evar st; Evar res]]) in
  let branches := map (fun '((e, _), (unl, _)) ⇒
     let transeq := Beq ([st; res], trans_exp unl e) in
     (e, [Breset [transeq] (Evar pres)])) states in
  let sw1 := Bswitch (Evar pst) branches in
  do branches ← mmap (fun '((e, _), (_, (blks, _))) ⇒
     do blks' ← mmap auto_block blks;
     ret (e, ([Breset blks' (Evar res)]))) states;
  let sw2 := Bswitch (Evar st) branches in
  ret (Blocal [pst; pres; st; res] [stateq; sw1; sw2])
```

```
var pst, pres, st, res; let
  (pst, pres) = (Starting, false) fby (st, res);
  switch pst
  | Starting do
    reset
      (st, res) =
        if time >= 750
        then (Moving, true)
        else (Starting, false)
    every pres
  | Moving do ...
  end;
  switch st
  | Starting do
    reset
      step = true fby false
    every res
  | Moving do ...
  end
tel
```

Compilation of State Machines – Proof Intuition



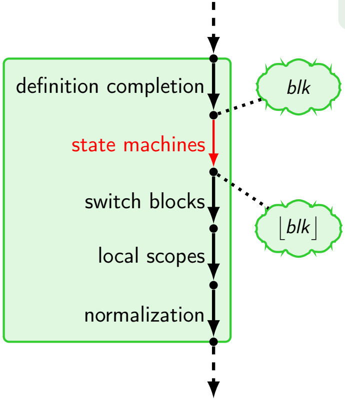
Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$

Compilation of State Machines – Proof Intuition

Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$



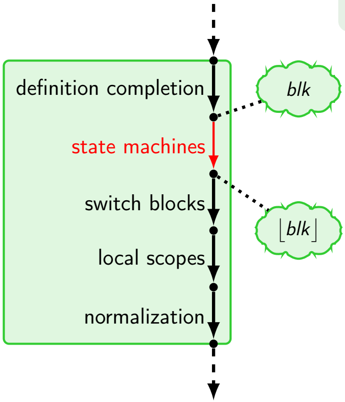
Works well:

- local transformation and reasoning
- correspondence between select, mask and when

Compilation of State Machines – Proof Intuition

Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$



Works well:

- local transformation and reasoning
- correspondence between select, mask and when

Works less well:

- static invariants (typing, clock-typing, . . .)
- fresh identifiers

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Γty Γck Hi bs blk' tys st st',
    (∀ x, IsVar Γty x → AtomOrGensym elab_prefs x) →
    (∀ x, IsVar Γck x → IsVar Γty x) →
    (∀ x, IsLast Γck x → IsLast Γty x) →
    NoDupLocals (List.map fst Γty) blk →
    GoodLocals elab_prefs blk →
    wt_block G1 Γty blk →
    wc_block G1 Γck blk →
    dom_ub Hi Γty →
    sc_vars Γck Hi bs →
    sem_block_ck G1 Hi bs blk →
    auto_block blk st = (blk', tys, st') →
    sem_block_ck G2 Hi bs blk'.
Proof.
  induction blk using block_ind2;
```

Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Γty Γck Hi bs blk' tys st st',
    (∀ x, IsVar Γty x → AtomOrGensym elab_prefs x) →
    (∀ x, IsVar Γck x → IsVar Γty x) →
    (∀ x, IsLast Γck x → IsLast Γty x) →
    NoDupLocals (List.map fst Γty) blk →
    GoodLocals elab_prefs blk →
    wt_block G1 Γty blk →
    wc_block G1 Γck blk →
    dom_ub Hi Γty →
    sc_vars Γck Hi bs →
    sem_block_ck G1 Hi bs blk →
    auto_block blk st = (blk', tys, st') →
    sem_block_ck G2 Hi bs blk'.
Proof.
  induction blk using block_ind2;
```

Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Γty Γck Hi bs blk' tys st st',
    (∀ x, IsVar Γty x → AtomOrGensym elab_prefs x) →
    (∀ x, IsVar Γck x → IsVar Γty x) →
    (∀ x, IsLast Γck x → IsLast Γty x) →
    NoDupLocals (List.map fst Γty) blk →
    GoodLocals elab_prefs blk →
    wt_block G1 Γty blk →
    wc_block G1 Γck blk →
    dom_ub Hi Γty →
    sc_vars Γck Hi bs →
    sem_block_ck G1 Hi bs blk →
    auto_block blk st = (blk', tys, st') →
    sem_block_ck G2 Hi bs blk'.
Proof.
  induction blk using block_ind2;
```

Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Γty Γck Hi bs blk' tys st st',
    (∀ x, IsVar Γty x → AtomOrGensym elab_prefs x) →
    (∀ x, IsVar Γck x → IsVar Γty x) →
    (∀ x, IsLast Γck x → IsLast Γty x) →
    NoDupLocals (List.map fst Γty) blk →
    GoodLocals elab_prefs blk →
    wt_block G1 Γty blk →
    wc_block G1 Γck blk →
    dom_ub Hi Γty →
    sc_vars Γck Hi bs →
    sem_block_ck G1 Hi bs blk →
    auto_block blk st = (blk', tys, st') →
    sem_block_ck G2 Hi bs blk'.
Proof.
  induction blk using block_ind2;
```

Lemma (State machines correctness)

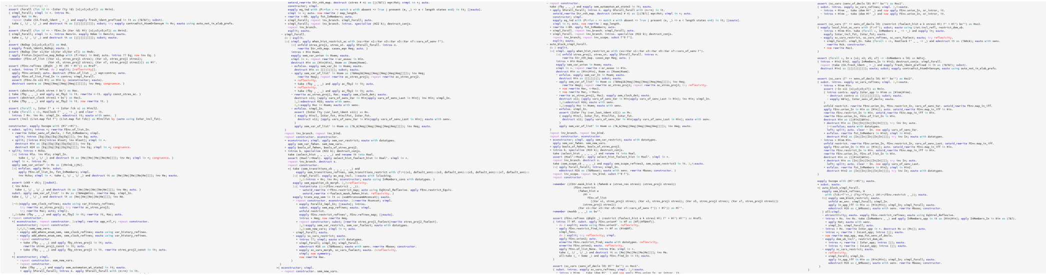
$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$

Compilation of State Machines – Coq Proof

```
Lemma auto_block_sem : ∀ blk Γty Γck Hi bs blk' tys st st',
    (∀ x, IsVar Γty x → AtomOrGensym elab_prefs x) →
    (∀ x, IsVar Γck x → IsVar Γty x) →
    (∀ x, IsLast Γck x → IsLast Γty x) →
    NoDupLocals (List.map fst Γty) blk →
    GoodLocals elab_prefs blk →
    wt_block G1 Γty blk →
    wc_block G1 Γck blk →
    dom_ub Hi Γty →
    sc_vars Γck Hi bs →
    sem_block_ck G1 Hi bs blk →
    auto_block blk st = (blk', tys, st') →
    sem_block_ck G2 Hi bs blk'.
Proof.
  induction blk using block_ind2;
```

Lemma (State machines correctness)

$$\text{if} \quad G, H \vdash blk \quad \text{then} \quad G, H \vdash \lfloor blk \rfloor$$



Compilation of Switch Blocks

```
switch st
| Starting do
  reset
    step = true fby false
  every res
| Holding do ...
end
```

```
resS = res when (st=Starting);
resM = res when (st=Moving);
step = merge st (Starting => stepS) (Moving => stepM);
reset
  stepS = true when (st=Starting) fby false when (st=Starting)
every resS;
```

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

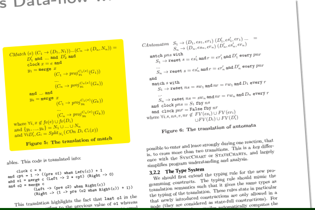


Figure 5: The translation of match

Figure 6: The translation of automata

Compilation of Switch Blocks

```
switch st
| Starting do
  reset
    step = true fby false
  every res
| Holding do ...
end
```

```
resS = res when (st=Starting);
resM = res when (st=Moving);
step = merge st (Starting => stepS) (Moving => stepM);
reset
  stepS = true when (st=Starting) fby false when (st=Starting)
every resS;
```

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

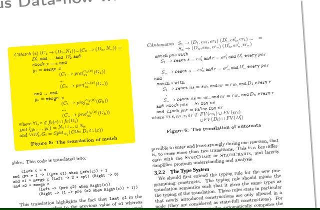


Figure 5: The translation of match

Figure 6: The translation of automata

- sampling explicited by `when`

Compilation of Switch Blocks

```
switch st
| Starting do
  reset
    step = true fby false
  every res
| Holding do ...
end
```

```
resS = res when (st=Starting);
resM = res when (st=Moving);
step = merge st (Starting => stepS) (Moving => stepM);
reset
  stepS = true when (st=Starting) fby false when (st=Starting)
every resS;
```

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

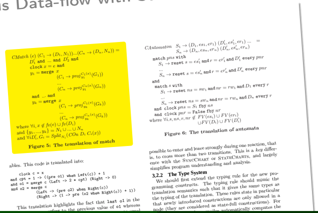


Figure 5: The translation of match

Figure 6: The translation of automata

3.2.2 The Type System

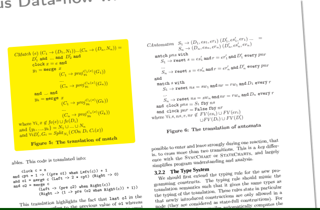
- sampling explicited by `when`
- choice explicited by `merge`

Compilation of Switch Blocks

```
switch st
| Starting do
  reset
    step = true fby false
  every res
| Holding do ...
end
```

```
resS = res when (st=Starting);
resM = res when (st=Moving);
step = merge st (Starting => stepS) (Moving => stepM);
reset
  stepS = true when (st=Starting) fby false when (st=Starting)
every resS;
```

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

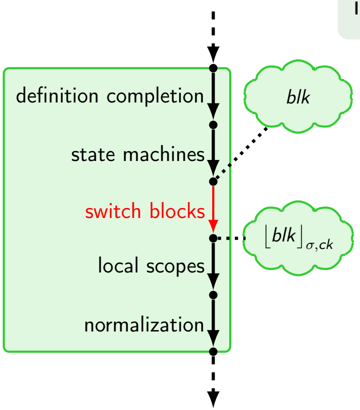


- sampling explicited by `when`
- choice explicited by `merge`
- constants are also sampled

Compilation of Switch Blocks – Proof Intuition

Lemma (Switch correctness)

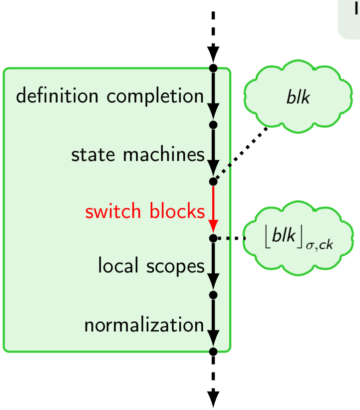
if $\quad G, H_1 \vdash blk \quad$ and $\quad H_1 \sqsubseteq_\sigma H_2 \quad$ then $\quad G, H_2 \vdash \lfloor blk \rfloor_{\sigma,ck}$



Compilation of Switch Blocks – Proof Intuition

Lemma (Switch correctness)

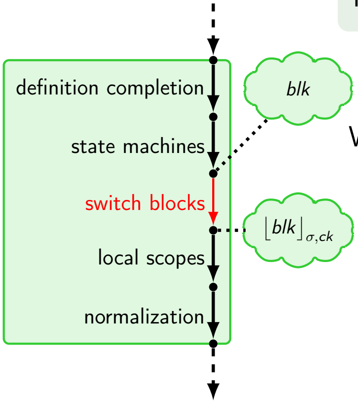
if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_\sigma H_2$ then $G, H_2 \vdash \lfloor blk \rfloor_{\sigma,ck}$



Works less well:

- reasoning is not local: renaming propagates to sub-blocks
- static invariants, in particular clock-typing

Compilation of Switch Blocks – Proof Intuition



Lemma (Switch correctness)

if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_\sigma H_2$ then $G, H_2 \vdash \lfloor blk \rfloor_{\sigma,ck}$

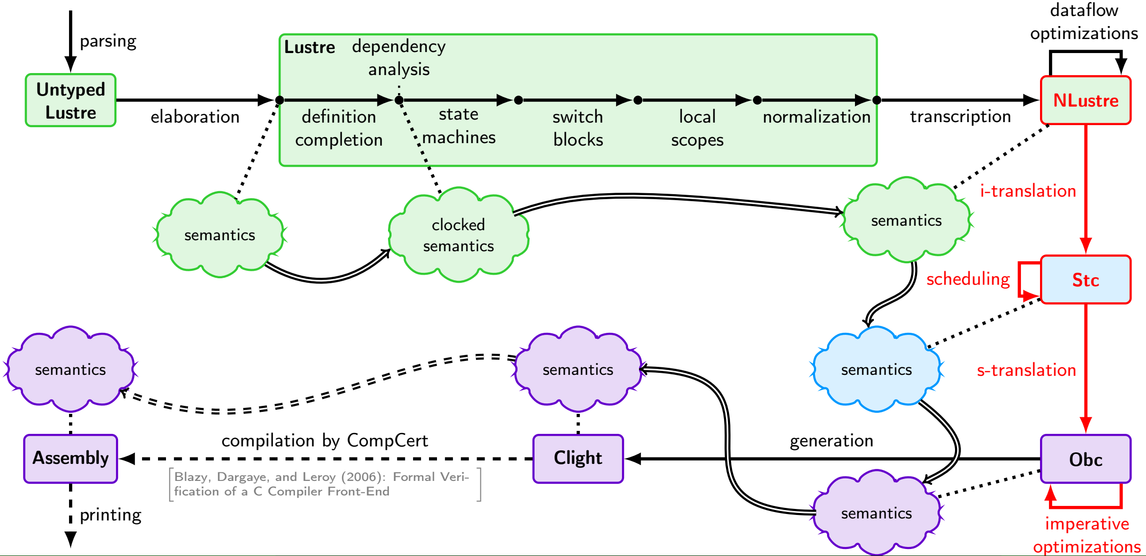
Works well:

- correspondence between `switch` and `when`/`merge`: implicit to explicit sampling
- less cases to handle

Works less well:

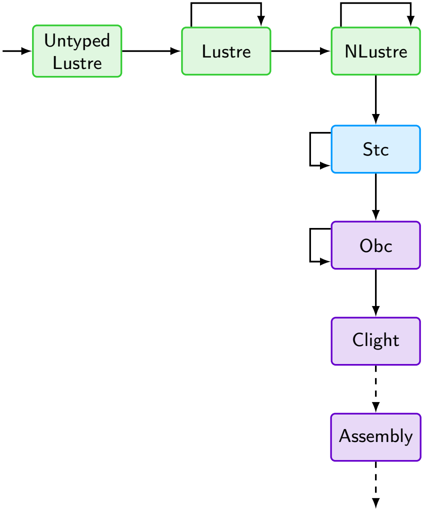
- reasoning is not local: renaming propagates to sub-blocks
- static invariants, in particular clock-typing

Compilation to Imperative Code



Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```

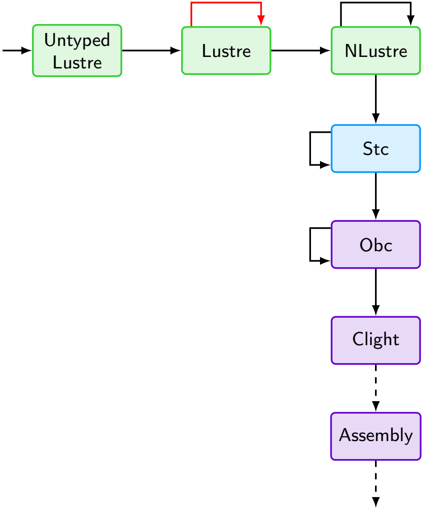


Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```

↓

```
switch step
| true do
  mA = not last$mB;
  mB = last$mA;
| false do (mA, mB) = (last$mA, last$mB)
end;
last$mA = true fby mA;
last$mB = false fby mB;
```

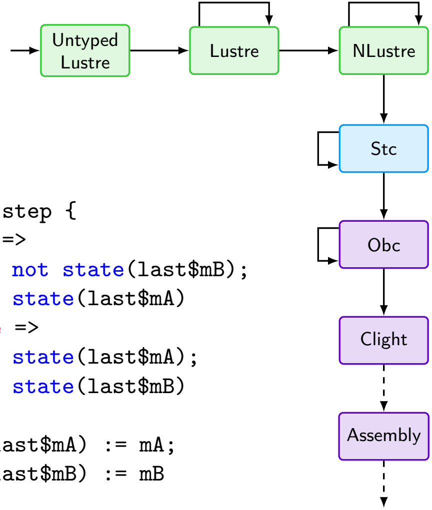


Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```

```
switch step
| true do
  mA = not last$mB;
  mB = last$mA;
| false do (mA, mB) = (last$mA, last$mB)
end;
last$mA = true fby mA;
last$mB = false fby mB;
```

```
switch step {
| true =>
  mA := not state(last$mB);
  mB := state(last$mA)
| false =>
  mA := state(last$mA);
  mB := state(last$mB)
};
state(last$mA) := mA;
state(last$mB) := mB
```

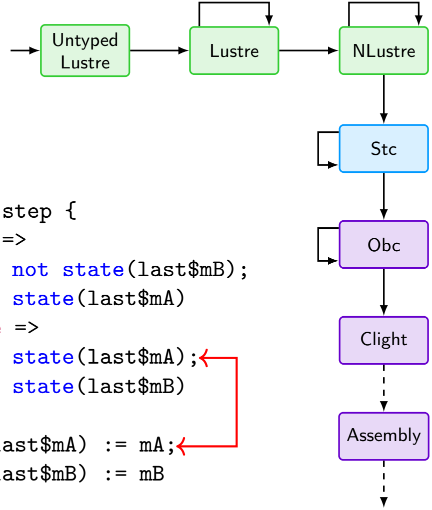


Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```

```
switch step
| true do
  mA = not last$mB;
  mB = last$mA;
| false do (mA, mB) = (last$mA, last$mB)
end;
last$mA = true fby mA;
last$mB = false fby mB;
```

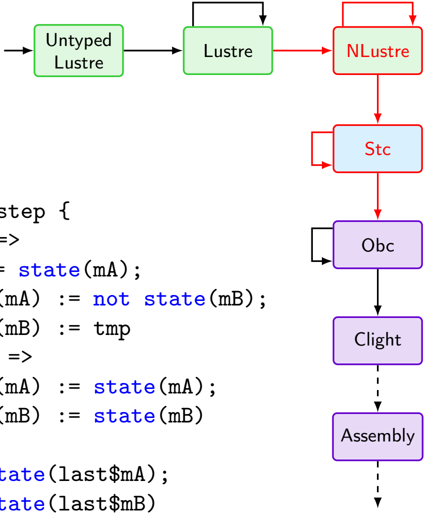
```
switch step {
| true =>
  mA := not state(last$mB);
  mB := state(last$mA)
| false =>
  mA := state(last$mA);
  mB := state(last$mB)
};
state(last$mA) := mA;
state(last$mB) := mB
```



Compiling Last Variables

```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```

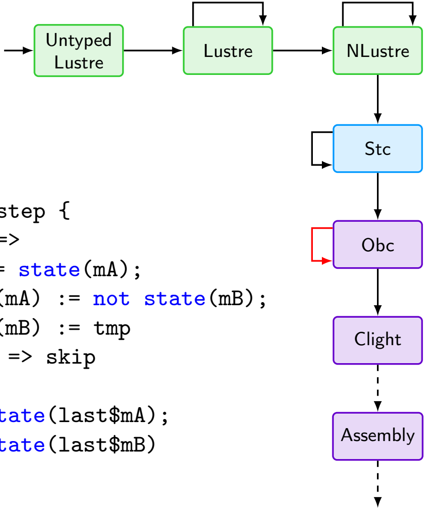
```
switch step {
| true =>
  tmp := state(mA);
  state(mA) := not state(mB);
  state(mB) := tmp
| false =>
  state(mA) := state(mA);
  state(mB) := state(mB)
};
mA := state(last$mA);
mB := state(last$mB)
```



Compiling Last Variables

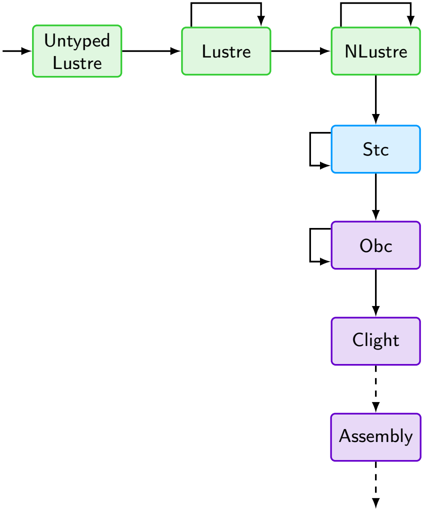
```
switch step
| true do
  mA = not (last mB);
  mB = last mA;
| false do (mA, mB) = (last mA, last mB)
end;
last mA = true;
last mB = false;
```

```
switch step {
| true =>
  tmp := state(mA);
  state(mA) := not state(mB);
  state(mB) := tmp
| false => skip
};
mA := state(last$mA);
mB := state(last$mB)
```

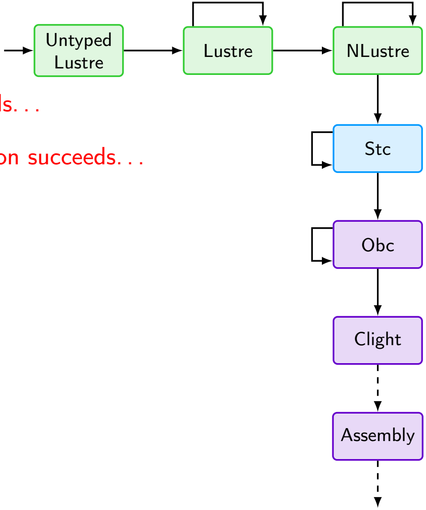


Main Correctness Theorem

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



Main Correctness Theorem

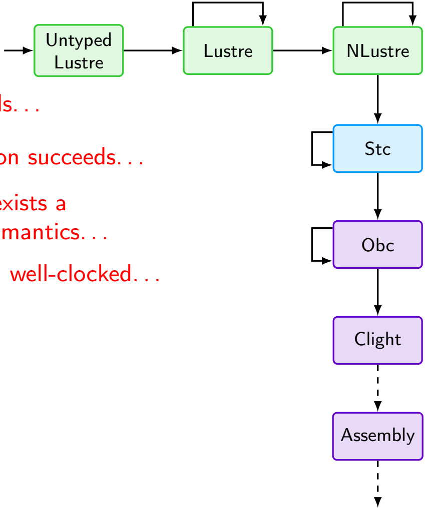


if typing/elaboration succeeds. . .

and compilation succeeds. . .

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

Main Correctness Theorem




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

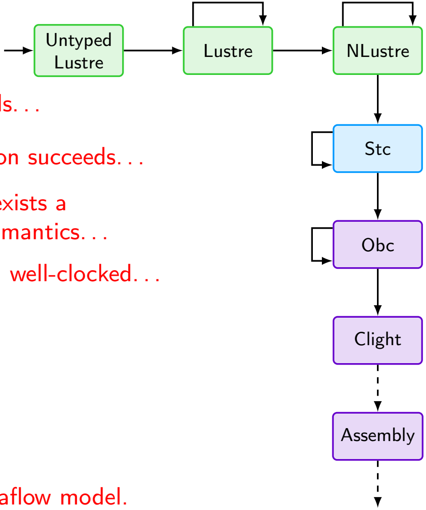
if typing/elaboration succeeds...

and compilation succeeds...

and there exists a dataflow semantics...

and input streams are well-typed and well-clocked...

Main Correctness Theorem




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

if typing/elaboration succeeds...

and compilation succeeds...

and there exists a dataflow semantics...

and input streams are well-typed and well-clocked...

then the generated assembly produces an infinite trace

and the trace corresponds to the dataflow model.

Conclusion

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
  - `switch` blocks
  - `reset` blocks
  - state machines
  - `last` variables
- a verified dependency analysis used to prove meta-properties of the model
- a verified implementation of an efficient compilation scheme for these blocks

Conclusion

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
  - `switch` blocks
  - `reset` blocks
  - state machines
  - `last` variables
- a verified dependency analysis used to prove meta-properties of the model
- a verified implementation of an efficient compilation scheme for these blocks

Future work:

- proof automation?
- missing Scade 6 features:
  - inlining and modular dependency analysis
  - `pre` operator and initialization analysis
  - arrays

Conclusion

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
  - `switch` blocks
  - `reset` blocks
  - state machines
  - `last` variables
- a verified dependency analysis used to prove meta-properties of the model
- a verified implementation of an efficient compilation scheme for these blocks

Future work:

- proof automation?
- missing Scade 6 features:
  - inlining and modular dependency analysis
  - `pre` operator and initialization analysis
  - arrays

`https://velus.inria.fr/phd-pesin`

Semantics – switch blocks

$$
\begin{aligned}
\mathsf{when}^{C}\,(\langle\rangle \cdot xs)\,(\langle\rangle \cdot cs) &\equiv \langle\rangle \cdot \mathsf{when}^{C}\; xs\; cs\\
\mathsf{when}^{C}\,(\langle v\rangle \cdot xs)\,(\langle C\rangle \cdot cs) &\equiv \langle v\rangle \cdot \mathsf{when}^{C}\; xs\; cs\\
\mathsf{when}^{C}\,(\langle v\rangle \cdot xs)\,(\langle C'\rangle \cdot cs) &\equiv \langle\rangle \cdot \mathsf{when}^{C}\; xs\; cs
\end{aligned}
$$

$$
(\mathsf{when}^{C}\; H\; cs)(x) \equiv \mathsf{when}^{C}\; (H(x))\; cs
$$

$$
\frac{G, H, bs \vdash e \Downarrow [cs] \qquad \forall i,\; G, \mathsf{when}^{C_i}\,(H, bs)\; cs \vdash blks_i}{G, H, bs \vdash \texttt{switch}\; e \left[C_i\; \texttt{do}\; blks_i\right]^{i}\; \texttt{end}}
$$

Semantics – reset blocks

$$\mathsf{mask}^k_{k'}\ (\mathtt{F} \cdot rs)\ (sv \cdot xs) \equiv (\mathsf{if}\ k' = k\ \mathsf{then}\ sv\ \mathsf{else}\ \langle\rangle) \cdot \mathsf{mask}^k_{k'}\ rs\ xs$$
$$\mathsf{mask}^k_{k'}\ (\mathtt{T} \cdot rs)\ (sv \cdot xs) \equiv (\mathsf{if}\ k' + 1 = k\ \mathsf{then}\ sv\ \mathsf{else}\ \langle\rangle) \cdot \mathsf{mask}^k_{k'+1}\ rs\ xs$$

$$\frac{\begin{array}{c} G, H, bs \vdash es \Downarrow xss \\ G, H, bs \vdash e \Downarrow [ys] \qquad \mathsf{bools\text{-}of}\ ys \equiv rs \\ \forall k,\ G \vdash f(\mathsf{mask}^k\ rs\ xss) \Downarrow (\mathsf{mask}^k\ rs\ yss) \end{array}}{G, H, bs \vdash (\,\mathtt{reset}\ f\ \mathtt{every}\ e)(es) \Downarrow yss}$$

$$\frac{\begin{array}{c} G, H, bs \vdash e \Downarrow [ys] \\ \mathsf{bools\text{-}of}\ ys \equiv rs \\ \forall k,\ G, \mathsf{mask}^k\ rs\ (H, bs) \vdash blks \end{array}}{G, H, bs \vdash \mathtt{reset}\ blks\ \mathtt{every}\ e}$$

Semantics – Hierarchical State Machines

$$\frac{\begin{array}{c} H, bs \vdash ck \Downarrow bs' \qquad G, H, bs' \vdash_{\mathrm{I}} autinits \Downarrow sts_0 \qquad \mathsf{fby}\ sts_0\ sts_1 \equiv sts \\ \forall i, \forall k,\ G, (\mathsf{select}_0^{C_i,k}\ sts\ (H, bs)), C_i \vdash_{\mathrm{W}} autscope_i \Downarrow (\mathsf{select}_0^{C_i,k}\ sts\ sts_1) \end{array}}{G, H, bs \vdash \texttt{automaton initially}\ autinits^{ck}\ [\texttt{state}\ C_i\ autscope_i]^i\ \texttt{end}}$$

$$\frac{\begin{array}{c} \forall x,\ x \in \mathsf{dom}(H') \iff x \in locs \\ \forall x\, e,\ (\texttt{last}\, x = e) \in locs \implies G, H + H', bs \vdash_{\mathrm{L}} \texttt{last}\, x = e \\ G, H + H', bs \vdash blks \qquad G, H + H', bs, C_i \vdash_{\mathrm{TR}} trans \Downarrow sts \end{array}}{G, H, bs, C_i \vdash_{\mathrm{W}} \texttt{var}\ locs\ \texttt{do}\ blks\ \texttt{until}\ trans \Downarrow sts}$$

$$\frac{\begin{array}{c} H, bs \vdash ck \Downarrow bs' \qquad \mathsf{fby}\ (\mathsf{const}\ bs'\ (C, \mathtt{F}))\ sts_1 \equiv sts \\ \forall i, \forall k,\ G, (\mathsf{select}_0^{C_i,k}\ sts\ (H, bs)), C_i \vdash_{\mathrm{TR}} trans_i \Downarrow (\mathsf{select}_0^{C_i,k}\ sts\ sts_1) \\ \forall i, \forall k,\ G, (\mathsf{select}_0^{C_i,k}\ sts_1\ (H, bs)) \vdash blks_i \end{array}}{G, H, bs \vdash \texttt{automaton initially}\ C^{ck}\ [\texttt{state}\ C_i\ \texttt{do}\ blks_i\ \texttt{unless}\ trans_i]^i\ \texttt{end}}$$

Semantics – Transitions

$$\frac{G, H, bs \vdash e \Downarrow [ys] \qquad \text{bools-of } ys \equiv bs' \qquad G, H, bs \vdash_{\mathrm{I}} autinits \Downarrow sts \qquad sts' \equiv \text{first-of}^{C}_{\mathtt{F}}\ bs'\ sts}{G, H, bs \vdash_{\mathrm{I}} C\ \mathtt{if}\ e; autinits \Downarrow sts'}$$

$$\frac{sts \equiv \text{const}\ bs\ (C, \mathtt{F})}{G, H, bs \vdash_{\mathrm{I}} \mathtt{otherwise}\ C \Downarrow sts}$$

$$\text{first-of}^{C}_{r}\ (\mathtt{T} \cdot bs)\ (st \cdot sts) \equiv \langle C, r \rangle \cdot \text{first-of}^{C}_{r}\ bs\ sts$$
$$\text{first-of}^{C}_{r}\ (\mathtt{F} \cdot bs)\ (st \cdot sts) \equiv st \cdot \text{first-of}^{C}_{r}\ bs\ sts$$

$$\frac{sts \equiv \text{const}\ bs\ (C_i, \mathtt{F})}{G, H, bs, C_i \vdash_{\mathrm{TR}} \epsilon \Downarrow sts}$$

$$\frac{G, H, bs \vdash e \Downarrow [ys] \qquad \text{bools-of } ys \equiv bs' \qquad G, H, bs, C_i \vdash_{\mathrm{TR}} trans \Downarrow sts \qquad sts' \equiv \text{first-of}^{C}_{\mathtt{F}}\ bs'\ sts}{G, H, bs, C_i \vdash_{\mathrm{TR}} \mathtt{if}\ e\ \mathtt{continue}\ C\ trans \Downarrow sts'}$$

$$\frac{G, H, bs \vdash e \Downarrow [ys] \qquad \text{bools-of } ys \equiv bs' \qquad G, H, bs, C_i \vdash_{\mathrm{TR}} trans \Downarrow sts \qquad sts' \equiv \text{first-of}^{C}_{\mathtt{T}}\ bs'\ sts}{G, H, bs, C_i \vdash_{\mathrm{TR}} \mathtt{if}\ e\ \mathtt{then}\ C\ trans \Downarrow sts'}$$

Semantics – local blocks and last variables

$$\frac{H(\texttt{last}\, x) \equiv vs}{G, H, bs \vdash \texttt{last}\, x \Downarrow [vs]}$$

$$\frac{\begin{array}{c}\forall x,\ x \in \mathrm{dom}(H') \iff x \in locs \\ \forall x\, e,\ (\texttt{last}\, x = e) \in locs \implies G, H + H', bs \vdash_{\mathrm{L}} \texttt{last}\, x = e \\ G, H + H', bs \vdash blks\end{array}}{G, H, bs \vdash \texttt{var}\ locs\ \texttt{let}\ blks\ \texttt{tel}}$$

$$\frac{G, H, bs \vdash e \Downarrow [vs_0] \qquad H(x) \equiv vs_1 \qquad H(\texttt{last}\, x) \equiv \mathrm{fby}\ vs_0\ vs_1}{G, H, bs \vdash_{\mathrm{L}} \texttt{last}\, x = e}$$

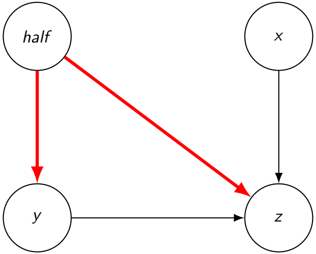
$$(H_1 + H_2)(x) = \begin{cases} H_2(x) & \text{if } x \in H_2 \\ H_1(x) & \text{otherwise.}\end{cases}$$

Dependency analysis of dataflow equations

```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```

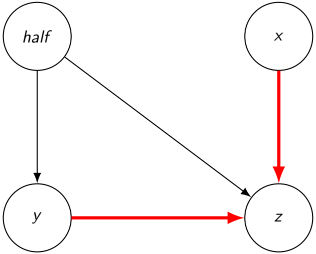
Dependency analysis of dataflow equations

```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```



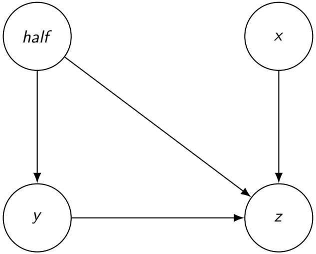
Dependency analysis of dataflow equations

```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```



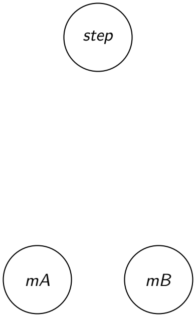
Dependency analysis of dataflow equations

```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```



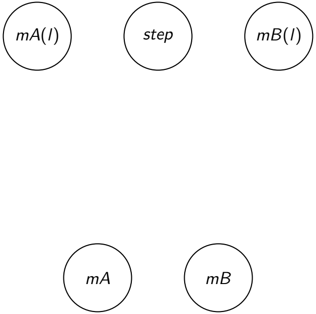
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA = true;
  last mB = false;
tel
```



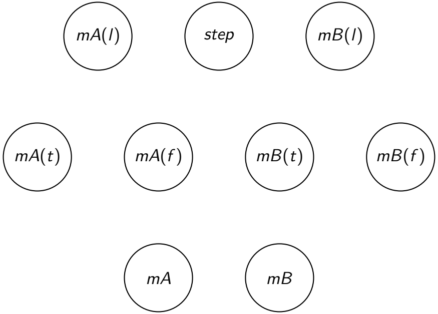
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA = not (last mB);
    mB = last mA;
  | false do (mA, mB) = (last mA, last mB)
  end;
  last mA^{mA(I)} = true;
  last mB^{mB(I)} = false;
tel
```



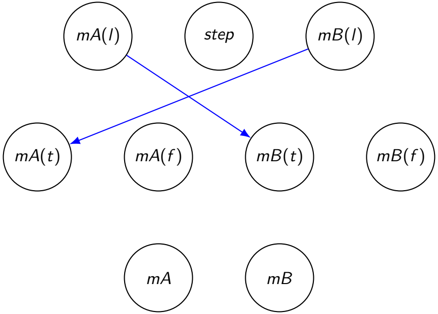
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(I)} = true;
  last mB^{mB(I)} = false;
tel
```



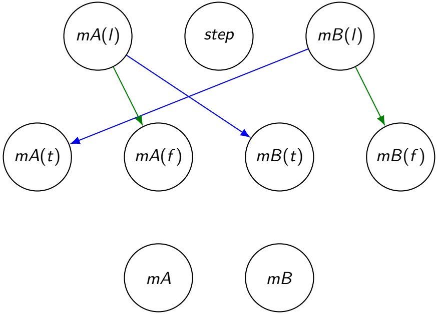
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(I)} = true;
  last mB^{mB(I)} = false;
tel
```



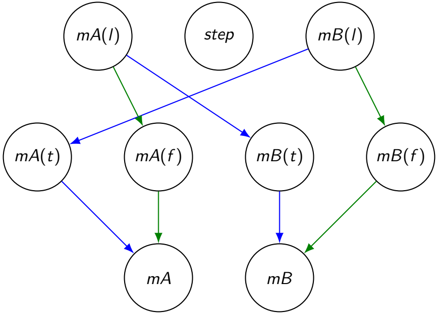
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^mA(t) = not (last mB);
    mB^mB(t) = last mA;
  | false do (mA^mA(f), mB^mB(f)) = (last mA, last mB)
  end;
  last mA^mA(l) = true;
  last mB^mB(l) = false;
tel
```



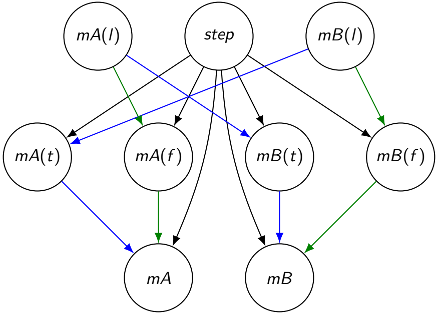
Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```



Dependency analysis of control blocks

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```



Dependency graph analysis

$$\frac{}{\text{AcyGraph}\ \emptyset\ \emptyset} \qquad \frac{\text{AcyGraph}\ V\ E}{\text{AcyGraph}\ (V \cup \{x\})\ E} \qquad \frac{\text{AcyGraph}\ V\ E \qquad x, y \in V \qquad y \not\to_E^* x}{\text{AcyGraph}\ V\ (E \cup \{x \to y\})}$$

- Simple graph analysis, based on DFS
- Produces a witness that the graph is acyclic (`AcyGraph`) that we will reason on
- More difficult to show termination in Coq

Dependency graph analysis

$$\frac{}{\text{AcyGraph } \emptyset\ \emptyset} \qquad \frac{\text{AcyGraph } V\ E}{\text{AcyGraph } (V \cup \{x\})\ E} \qquad \frac{\text{AcyGraph } V\ E \qquad x, y \in V \qquad y \not\to^*_E x}{\text{AcyGraph } V\ (E \cup \{x \to y\})}$$

```
Definition visited (p : set) (v : set) : Prop :=
    (∀ x, x ∈ p → ¬(x ∈ v))
  ∧ ∃ a, AcyGraph v a
        ∧ (∀ x, x ∈ v → ∃ zs, graph(x) = Some zs
                               ∧ (∀ y, y ∈ zs → has_arc a y x)).

Program Fixpoint dfs'
  (s : { p | ∀ x, x ∈ p → x ∈ graph }) (x : ident)
  (v : { v | visited s v }) {measure (|graph| - |s|)}
  : option { v' | visited s v' & x ∈ v' ∧ v ⊆ v' } := ...
```

Dependency graph analysis

$$\frac{}{\text{AcyGraph } \emptyset\ \emptyset} \qquad \frac{\text{AcyGraph } V\ E}{\text{AcyGraph } (V \cup \{x\})\ E} \qquad \frac{\text{AcyGraph } V\ E \qquad x, y \in V \qquad y \not\to^*_E x}{\text{AcyGraph } V\ (E \cup \{x \to y\})}$$

```
Definition visited (p : set) (v : set) : Prop :=
    (∀ x, x ∈ p → ¬(x ∈ v))
  ∧ ∃ a, AcyGraph v a
        ∧ (∀ x, x ∈ v → ∃ zs, graph(x) = Some zs
                                ∧ (∀ y, y ∈ zs → has_arc a y x)).

Program Fixpoint dfs'
  (s : { p | ∀ x, x ∈ p → x ∈ graph }) (x : ident)
  (v : { v | visited s v }) {measure (|graph| - |s|)}
  : option { v' | visited s v' & x ∈ v' ∧ v ⊆ v' } := ...
```

Dependency graph analysis

$$\frac{}{\text{AcyGraph}\ \emptyset\ \emptyset} \qquad \frac{\text{AcyGraph}\ V\ E}{\text{AcyGraph}\ (V \cup \{x\})\ E} \qquad \frac{\text{AcyGraph}\ V\ E \qquad x, y \in V \qquad y \not\to^*_E x}{\text{AcyGraph}\ V\ (E \cup \{x \to y\})}$$

```
Definition visited (p : set) (v : set) : Prop :=
    (∀ x, x ∈ p → ¬(x ∈ v))
  ∧ ∃ a, AcyGraph v a
        ∧ (∀ x, x ∈ v → ∃ zs, graph(x) = Some zs
                               ∧ (∀ y, y ∈ zs → has_arc a y x)).

Program Fixpoint dfs'
  (s : { p | ∀ x, x ∈ p → x ∈ graph }) (x : ident)
  (v : { v | visited s v }) {measure (|graph| - |s|)}
  : option { v' | visited s v' & x ∈ v' ∧ v ⊆ v' } := ...
```

Dependency graph analysis

$$\frac{}{\text{AcyGraph}\ \emptyset\ \emptyset} \qquad \frac{\text{AcyGraph}\ V\ E}{\text{AcyGraph}\ (V \cup \{x\})\ E} \qquad \frac{\text{AcyGraph}\ V\ E \qquad x, y \in V \qquad y \not\to_E^* x}{\text{AcyGraph}\ V\ (E \cup \{x \to y\})}$$

```
Definition visited (p : set) (v : set) : Prop :=
    (∀ x, x ∈ p → ¬(x ∈ v))
  ∧ ∃ a, AcyGraph v a
      ∧ (∀ x, x ∈ v → ∃ zs, graph(x) = Some zs
                          ∧ (∀ y, y ∈ zs → has_arc a y x)).

Program Fixpoint dfs'
  (s : { p | ∀ x, x ∈ p → x ∈ graph }) (x : ident)
  (v : { v | visited s v }) {measure (|graph| - |s|)}
  : option { v' | visited s v' & x ∈ v' ∧ v ⊆ v' } := ...
```

Proving with dependencies

$$\frac{}{\text{TopoOrder (AcyGraph } V\ E)\ []} \qquad \frac{\text{TopoOrder (AcyGraph } V\ E)\ l \qquad x \in V \qquad \neg\text{In } x\ l \qquad (\forall y,\ y \to_E^* x \implies \text{In } y\ l)}{\text{TopoOrder (AcyGraph } V\ E)\ (x :: l)}$$

Proving with dependencies

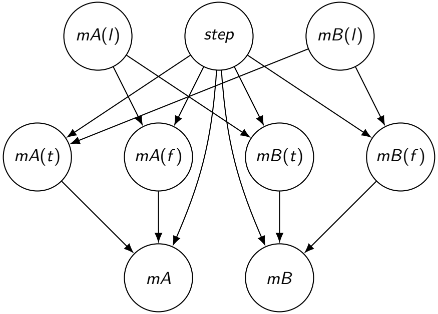
$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []} \qquad \frac{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ l \\ x \in V \qquad \neg \text{In}\ x\ l \qquad (\forall y,\ y \rightarrow_E^* x \implies \text{In}\ y\ l)}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ (x :: l)}$$

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```

Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []} \qquad \frac{\begin{array}{c}\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ l \\ x \in V \qquad \neg\text{In}\ x\ l \qquad (\forall y,\ y \to_E^* x \implies \text{In}\ y\ l)\end{array}}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ (x :: l)}$$

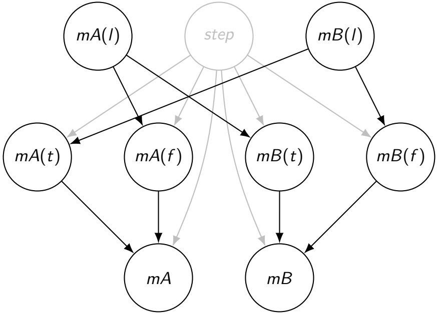
```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^mA(t) = not (last mB);
    mB^mB(t) = last mA;
  | false do (mA^mA(f), mB^mB(f)) = (last mA, last mB)
  end;
  last mA^mA(l) = true;
  last mB^mB(l) = false;
tel
```

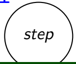


Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []} \qquad \frac{\begin{array}{c}\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ l \\ x \in V \qquad \neg\text{In}\ x\ l \qquad (\forall y,\ y \rightarrow_E^* x \implies \text{In}\ y\ l)\end{array}}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ (x :: l)}$$

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```

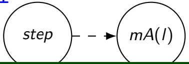


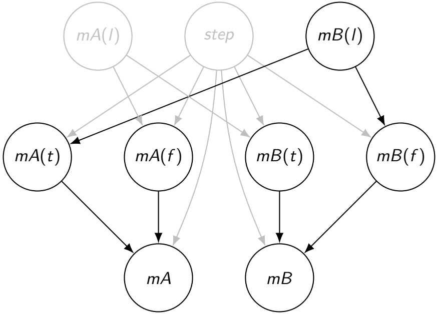


Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []} \qquad \frac{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ l \qquad x \in V \qquad \neg\text{In}\ x\ l \qquad (\forall y,\ y \to_E^* x \implies \text{In}\ y\ l)}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ (x :: l)}$$

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```

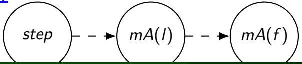


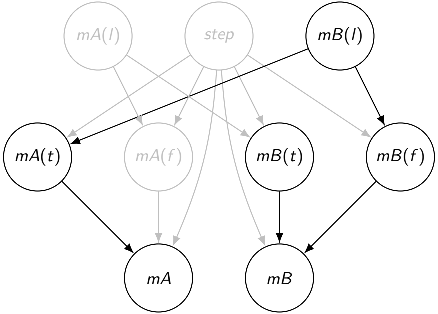


Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}\qquad\frac{\begin{array}{c}\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ l\\ x \in V \qquad \neg\text{In}\ x\ l \qquad (\forall y,\ y \to_E^* x \implies \text{In}\ y\ l)\end{array}}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ (x :: l)}$$

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```

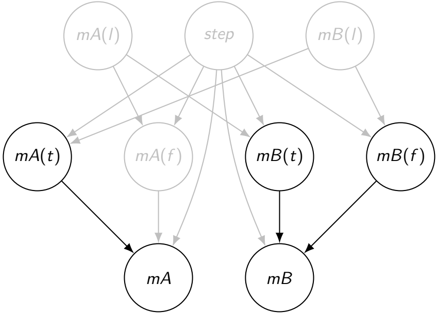


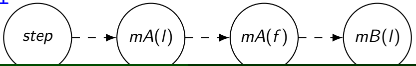


Proving with dependencies

$$\frac{}{\text{TopoOrder (AcyGraph } V\ E)\ []} \qquad \frac{\text{TopoOrder (AcyGraph } V\ E)\ l \qquad x \in V \qquad \neg\text{In } x\ l \qquad (\forall y,\ y \rightarrow_E^* x \implies \text{In } y\ l)}{\text{TopoOrder (AcyGraph } V\ E)\ (x :: l)}$$

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```

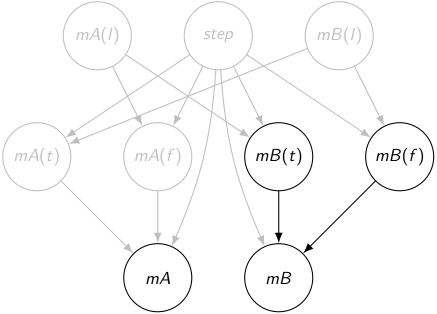




Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}
\qquad
\frac{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ l \quad x \in V \quad \neg\text{In}\ x\ l \quad (\forall y,\ y \to_E^* x \implies \text{In}\ y\ l)}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ (x :: l)}$$

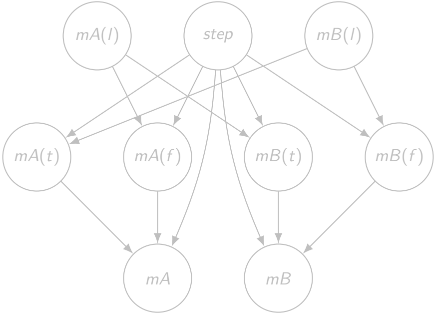
```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```

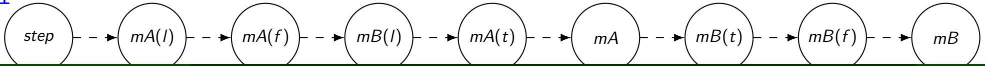


Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []} \qquad \frac{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ l \qquad x \in V \qquad \neg\text{In}\ x\ l \qquad (\forall y,\ y \to_E^* x \implies \text{In}\ y\ l)}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ (x :: l)}$$

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```





Performances

	Vélus	*Hept+CompCert*		*Hept+gcc*		*Hept+gcci*	
stepper_motor	930	1185	(+27 %)	610	(−34 %)	535	(−42 %)
chrono	505	970	(+92 %)	570	(+12 %)	570	(+12 %)
cruisecontrol	1405	1745	(+24 %)	960	(−31 %)	895	(−36 %)
heater	2415	3125	(+29 %)	730	(−69 %)	515	(−78 %)
buttons	1015	1430	(+40 %)	625	(−38 %)	625	(−38 %)
stopwatch	1305	1970	(+50 %)	1290	(−1 %)	1290	(−1 %)

WCET estimated by OTAWA 2 [Ballabriga, Cassé, Rochange, and Sainrat (2010): OTAWA: An Open Toolbox for Adaptive WCET Analysis] for an armv7

- Vélus generally better than Heptagon, but worse than Heptagon+GCC

Performances

	Vélus	*Hept+CompCert*		*Hept+gcc*		*Hept+gcci*	
stepper_motor	930	1185	(+27 %)	610	(−34 %)	535	(−42 %)
chrono	505	970	(+92 %)	570	(+12 %)	570	(+12 %)
cruisecontrol	1405	1745	(+24 %)	960	(−31 %)	895	(−36 %)
heater	2415	3125	(+29 %)	730	(−69 %)	515	(−78 %)
buttons	1015	1430	(+40 %)	625	(−38 %)	625	(−38 %)
stopwatch	1305	1970	(+50 %)	1290	(−1 %)	1290	(−1 %)

WCET estimated by OTAWA 2 [Ballabriga, Cassé, Rochange, and Sainrat (2010): OTAWA: An Open Toolbox for Adaptive WCET Analysis] for an armv7

- Vélus generally better than Heptagon, but worse than Heptagon+GCC
- Inlining of CompCert not fine tuned to small functions generated by Vélus

Performances

	Vélus	*Hept+CompCert*		*Hept+gcc*		*Hept+gcci*	
stepper_motor	930	1185	(+27 %)	610	(−34 %)	535	(−42 %)
chrono	505	970	(+92 %)	570	(+12 %)	570	(+12 %)
cruisecontrol	1405	1745	(+24 %)	960	(−31 %)	895	(−36 %)
heater	2415	3125	(+29 %)	730	(−69 %)	515	(−78 %)
buttons	1015	1430	(+40 %)	625	(−38 %)	625	(−38 %)
stopwatch	1305	1970	(+50 %)	1290	(−1 %)	1290	(−1 %)

WCET estimated by OTAWA 2 [Ballabriga, Cassé, Rochange, and Sainrat (2010): OTAWA: An Open Toolbox for Adaptive WCET Analysis] for an armv7

- Vélus generally better than Heptagon, but worse than Heptagon+GCC
- Inlining of CompCert not fine tuned to small functions generated by Vélus
- Some possible improvements
  - Better compilation of `last` to reduce useless updates (done in unpublished version)
  - Memory optimization: state variables in mutually exclusive states can be be reused