Verifying a compiler for a synchronous dataflow language with state machines in Coq

Basile Pesin, Timothy Bourke and Marc Pouzet

Inria - PARKAS

September 13, 2023

Introduction

Formalizing Lustre in Coq

- Lustre: A synchronous dataflow language [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]
- Scade 3: certified industrial implementation of Lustre v6

Formalizing Lustre in Coq

- Lustre: A synchronous dataflow language [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]
- Scade 3: certified industrial implementation of Lustre v6
- Verifying a Lustre compiler in a proof assistant
  - [Boulmé and Hamon (2001): Certifying Synchrony for Free]
    Shallow embedding of the semantics of Lucid-Synchrone in Coq
  - [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
    Translation validation of the frontend of a Lustre compiler
  - [Shi, Gan, Shang, Wang, Dong, and Yew (2017): A Formally Verified Sequentializer for Lustre-Like Concurrent Synchronous Data-Flow Programs]
    Verified Compiler for Lustre (with arrays) with a more imperative semantics

Formalizing Lustre in Coq

- Lustre: A synchronous dataflow language [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]
- Scade 3: certified industrial implementation of Lustre v6
- Verifying a Lustre compiler in a proof assistant
  - [Boulmé and Hamon (2001): Certifying Synchrony for Free]
    Shallow embedding of the semantics of Lucid-Synchrone in Coq
  - [Auger (2013): Compilation certifiée de SCADE/LUSTRE]
    Translation validation of the frontend of a Lustre compiler
  - [Shi, Gan, Shang, Wang, Dong, and Yew (2017): A Formally Verified Sequentializer for Lustre-Like Concurrent Synchronous Data-Flow Programs]
    Verified Compiler for Lustre (with arrays) with a more imperative semantics
- [Bourke, Brun, Dagand, Leroy, Pouzet, and Rieg (2017): A Formally Verified Compiler for Lustre]
  Vélus: a formally verified compiler for Lustre with a dataflow semantics

Extending Vélus with Control Structures

- My thesis: extend Vélus with Scade 6/Lucid Synchrone control structures
  [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]
  - Hierarchical State Machines

Extending Vélus with Control Structures

- My thesis: extend Vélus with Scade 6/Lucid Synchrone control structures
  [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]
  - Hierarchical State Machines
  - `switch` blocks
  - `reset` blocks
  - nested local scopes
  - shared variables (`last`)

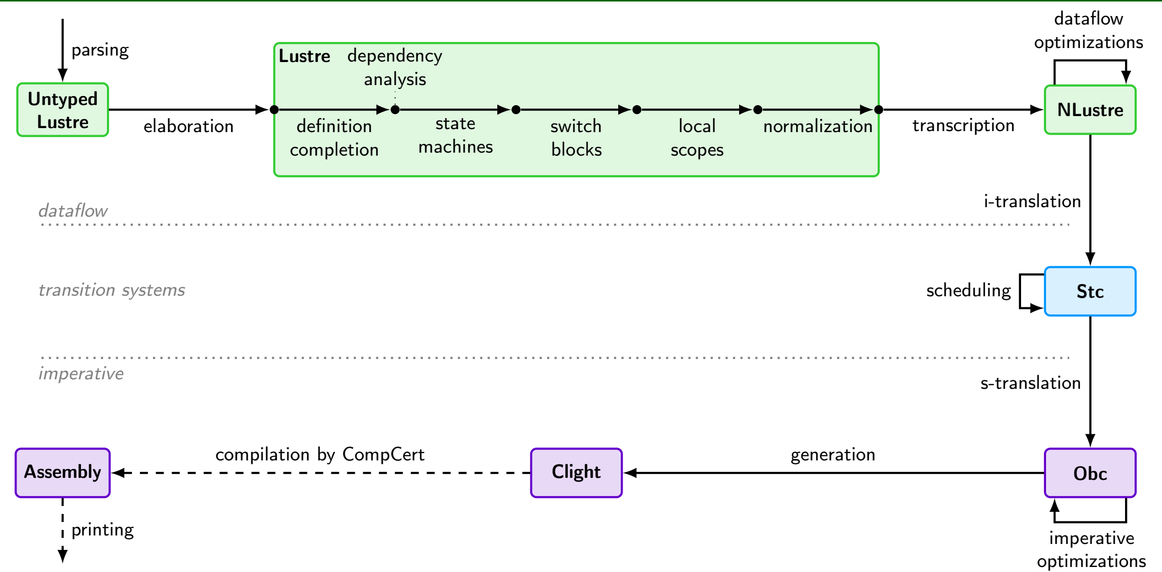
Extending Vélus with Control Structures

- My thesis: extend Vélus with Scade 6/Lucid Synchrone control structures
  [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]
  - Hierarchical State Machines
  - `switch` blocks
  - `reset` blocks
  - nested local scopes
  - shared variables (`last`)
- These constructions are already well-defined
  [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines], [Colaço, Hamon, and Pouzet (2006): Mixing Signals and Modes in Synchronous Data-flow Systems]

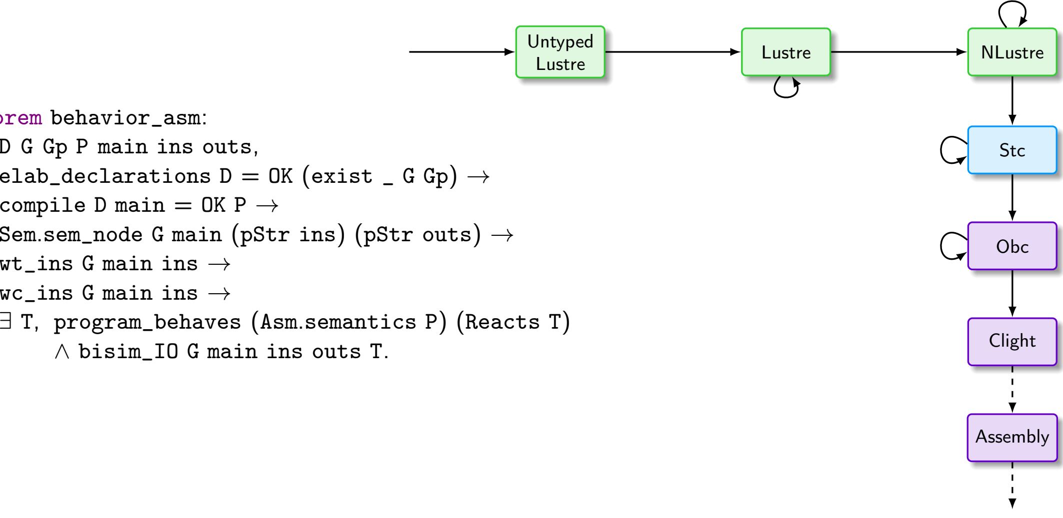
Extending Vélus with Control Structures

- My thesis: extend Vélus with Scade 6/Lucid Synchrone control structures
  [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]
  - Hierarchical State Machines
  - `switch` blocks
  - `reset` blocks
  - nested local scopes
  - shared variables (`last`)
- These constructions are already well-defined
  [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines], [Colaço, Hamon, and Pouzet (2006): Mixing Signals and Modes in Synchronous Data-flow Systems]
- Challenges:
  - Integrate their semantics in Vélus
  - Prove the correctness of their compilation with every last detail

The Vélus Compiler

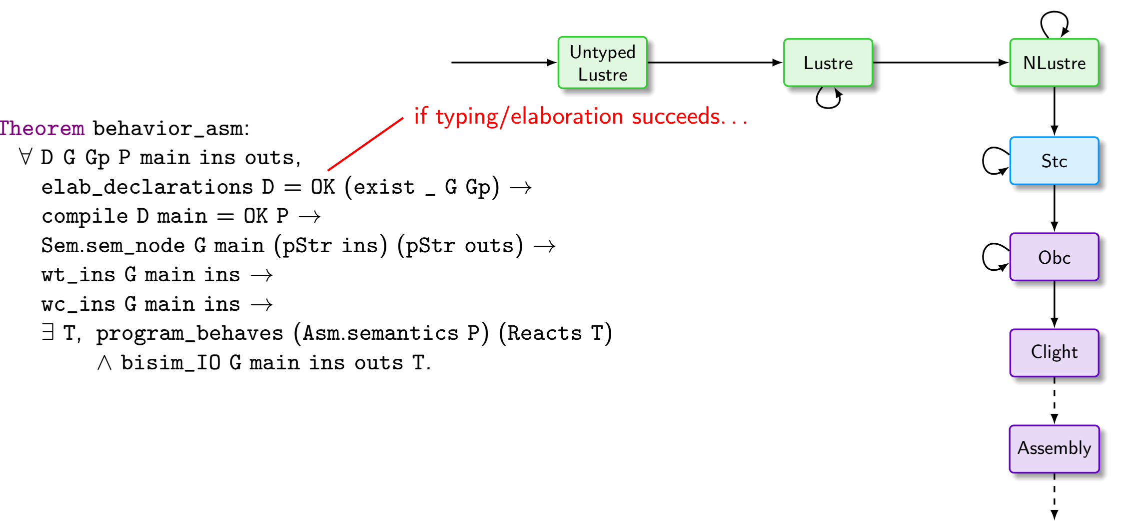


Main Correctness Theorem

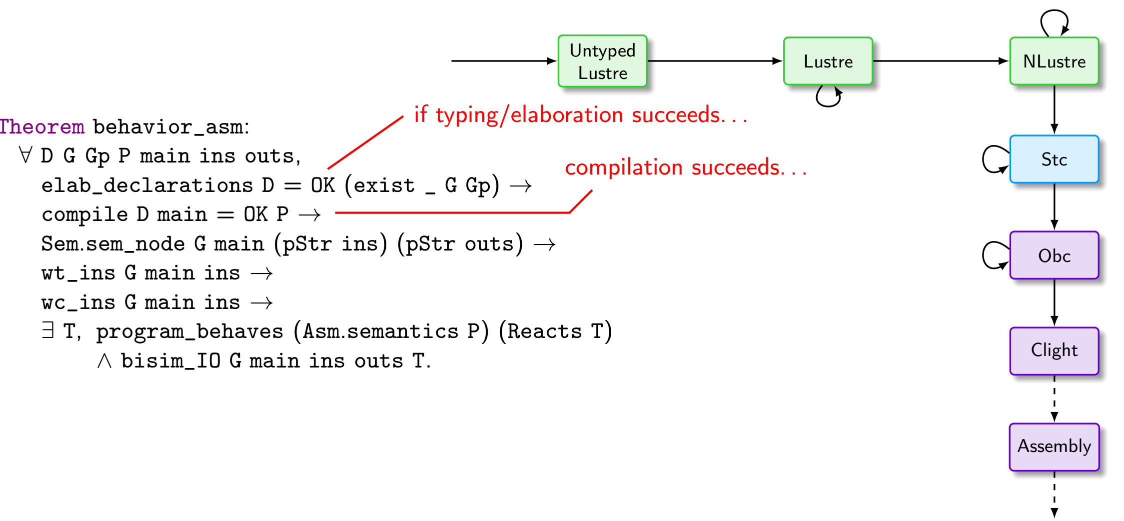



```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    compile D main = OK P →
    Sem.sem_node G main (pStr ins) (pStr outs) →
    wt_ins G main ins →
    wc_ins G main ins →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_IO G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

Main Correctness Theorem

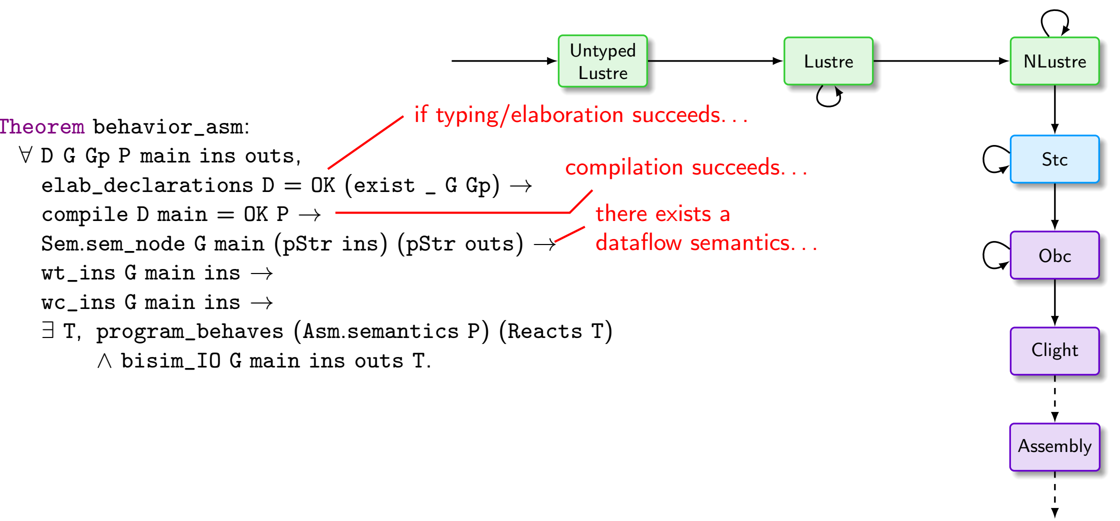


Main Correctness Theorem



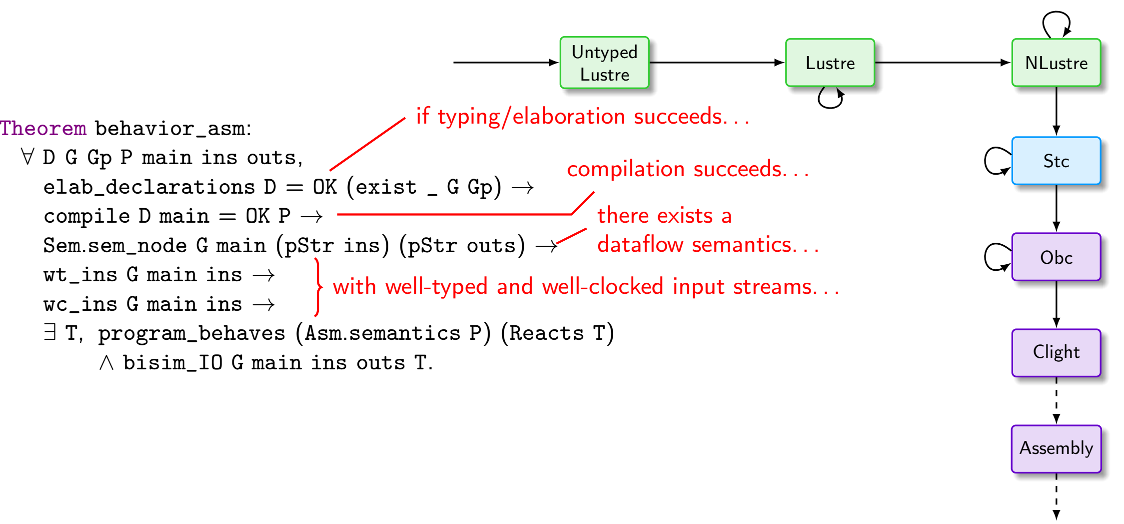

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    compile D main = OK P →
    Sem.sem_node G main (pStr ins) (pStr outs) →
    wt_ins G main ins →
    wc_ins G main ins →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
          ∧ bisim_IO G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

Main Correctness Theorem

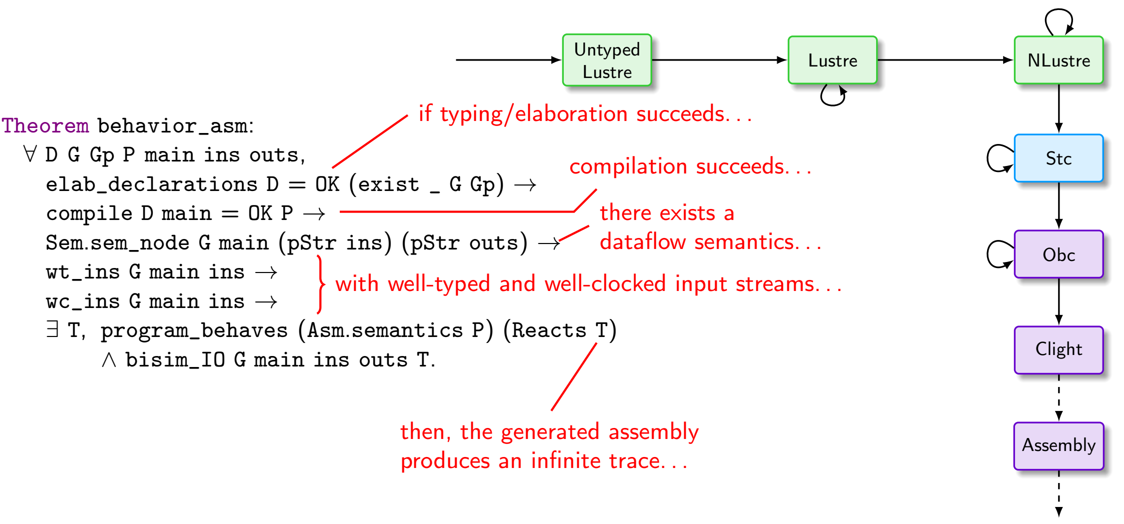



```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    compile D main = OK P →
    Sem.sem_node G main (pStr ins) (pStr outs) →
    wt_ins G main ins →
    wc_ins G main ins →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_IO G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

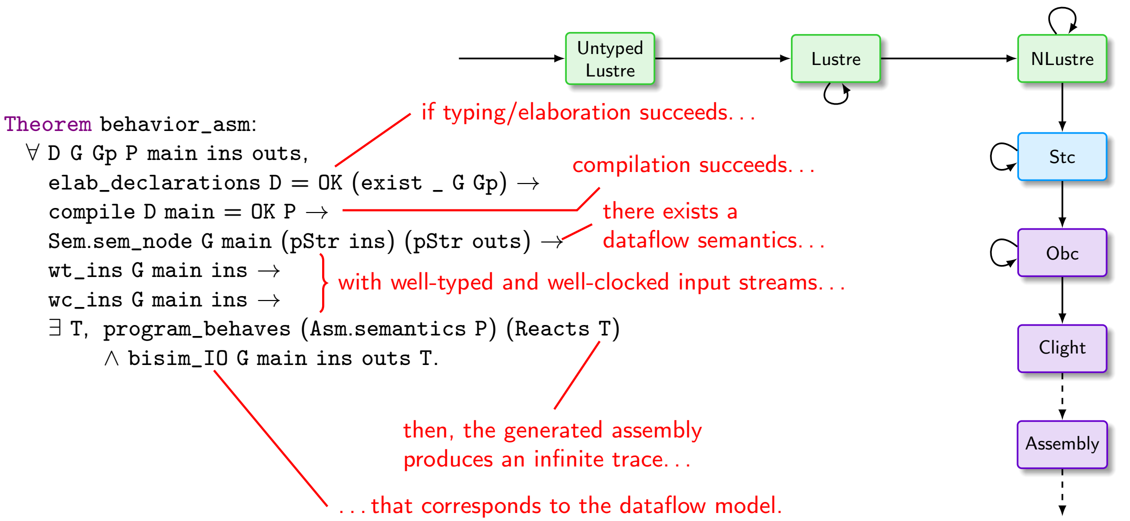
Main Correctness Theorem



Main Correctness Theorem

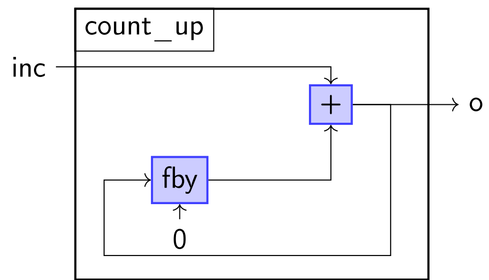


Main Correctness Theorem



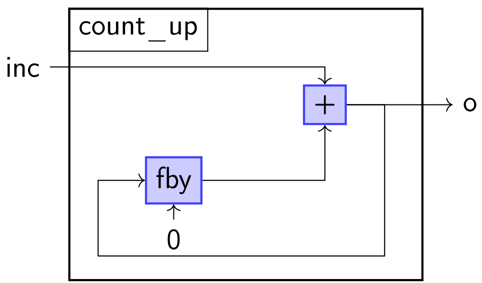
The Vélus Core Dataflow Language

The Vélus Dataflow Core



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

The Vélus Dataflow Core



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

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

Dataflow Semantics of Vélus

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

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

Dataflow Semantics of Vélus

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.

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

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

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

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	...

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

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

Dataflow Semantics of Vélus

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.

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

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

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

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	...

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

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

$$\frac{\begin{array}{c} G(f) = \texttt{node}\ f(x_1, \ldots, x_n)\ \texttt{returns}\ (y_1, \ldots, y_m)\ blk \\ \forall i \in 1 \ldots n,\ H(x_i) \equiv xs_i \qquad \forall i \in 1 \ldots m,\ H(y_i) \equiv ys_i \qquad G, H, (\text{base-of}\,(xs_1, \ldots, xs_n)) \vdash blk \end{array}}{G \vdash f(xs_1, \ldots, xs_n) \Downarrow (ys_1, \ldots, ys_m)}$$

Dataflow Semantics of Vélus – in Coq

```
Inductive sem_exp:
| Svar: sem_var H x vs →
      sem_exp G H bs (Evar x ann) [vs]
[...]
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: ident → list (Stream svalue) → list (Stream svalue) → Prop :=
| 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.
```

Lustre fby operator semantics

`inc`	⟨⟩	⟨⟩	5	⟨⟩	⟨⟩	4	1	3	2	⟨⟩	8	3	...
`0 fby o`	⟨⟩	⟨⟩	0										...
`o = (0 fby o) + inc`	⟨⟩	⟨⟩	5										...

$$\text{fby}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) \equiv \langle\rangle \cdot \text{fby}\ xs\ ys$$

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

Lustre 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	...

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

Lustre 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	...

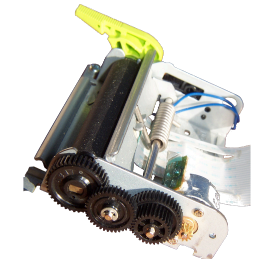
$$
\begin{aligned}
\text{fby}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) &\equiv \langle\rangle \cdot \text{fby}\ xs\ ys \\
\text{fby}\ (\langle v_1\rangle \cdot xs)\ (\langle v_2\rangle \cdot ys) &\equiv \langle v_1\rangle \cdot \text{fby1}\ v_2\ xs\ ys \\
\text{fby1}\ v_0\ (\langle\rangle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

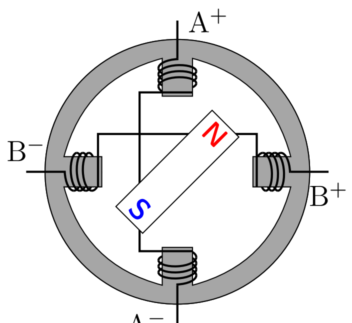
$$
\frac{G, H, bs \vdash es_0 \Downarrow [xs_i]^i \qquad G, H, bs \vdash es_1 \Downarrow [ys_i]^i \qquad \forall i,\ \text{fby}\ xs_i\ ys_i \equiv vs_i}{G, H, bs \vdash es_0\ \texttt{fby}\ es_1 \Downarrow [vs_i]^i}
$$

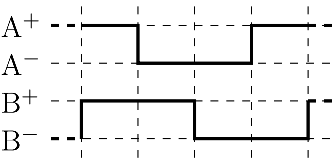
Control Blocks and their Stream Semantics

An embedded system example : stepper motor for a small printer

- 4 windings , energized by `enable` and controlled by `mA` and `mB`
- can be paused: the motor needs to stop turning
- when starting/restarting, needs to speed up first
- when paused, less energy should be sent to the windings : PWM



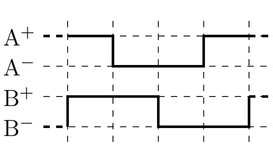


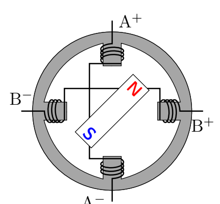


Switch [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines] – Stream Semantics

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...

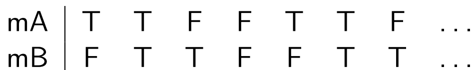
```
mA = true fby not mB;
mB = false fby mA;
```



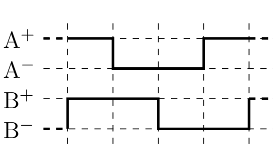


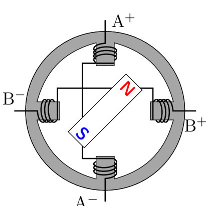
Switch [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines] – Stream Semantics

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



mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...





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

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...

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

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

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

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

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...

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

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

step	F			F	F		F		F		F		F	F		...
mA	F			F	F		F		F		F		F	F		...
mB	F			F	F		F		F		F		F	F		...

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

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

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...

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

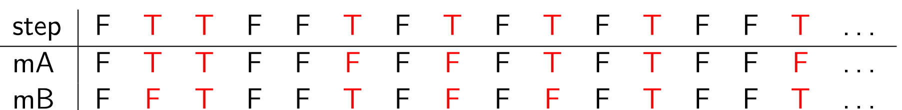
$$\frac{G, H, bs \vdash e \Downarrow [cs] \qquad \forall i,\ G, \text{when}^{C_i}\ (H, bs)\ cs \vdash blks_i}{G, H, bs \vdash \texttt{switch}\ e\ [C_i\ \texttt{do}\ blks_i]^i\ \texttt{end}}$$

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

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

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

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...



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

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

```
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
```

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...

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

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

```
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
```

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...

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

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

```
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
```

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...

A+
A−
B+
B−
N
S

step	F	T	T	F	F	T	F	T	F	T	F	T	F	F	T	...
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											...

```
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
```

mA	T	T	F	F	T	T	F	...
mB	F	T	T	F	F	T	T	...

A+
A−
B+
B−
N
S

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	...

Shared variables – Semantic rules

- The semantics of `last x` depends on the global stream of `x`, not its sampled view
- Solution: add the stream of `last x` in the history [Pouzet (2021): ZRun]

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

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	...

Local Scopes - Semantic rules

```
node f(x : int) returns (z : bool)
var y : int;
let
  var t : int;
  let t = x fby (t + 1);
      y = t;
  tel;
  var t : int;
  let t = y + 1;
      z = t > 0;
  tel
tel
```

Local Scopes - Semantic rules

```
node f(x : int) returns (z : bool)
var y : int;
let
  var t : int;
  let t = x fby (t + 1);
      y = t;
  tel;
  var t : int;
  let t = y + 1;
      z = t > 0;
  tel
tel
```

$$\frac{\forall x,\ x \in \mathrm{dom}(H') \iff x \in \mathit{locs} \qquad G, H + H', \mathit{bs} \vdash \mathit{blks}}{G, H, \mathit{bs} \vdash \texttt{var}\ \mathit{locs}\ \texttt{let}\ \mathit{blks}\ \texttt{tel}}$$

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

Reset [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs] – Stream Semantics

```
node feed_pause()
returns (step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);
  step = time > 125;
tel
```

Reset [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs] – Stream Semantics

$$
\begin{aligned}
\mathrm{mask}^k_{k'}\,(\mathtt{F}\cdot rs)\,(sv\cdot xs) &\equiv (\text{if } k' = k \text{ then } sv \text{ else } \langle\rangle)\cdot \mathrm{mask}^k_{k'}\ rs\ xs\\
\mathrm{mask}^k_{k'}\,(\mathtt{T}\cdot rs)\,(sv\cdot xs) &\equiv (\text{if } k' + 1 = k \text{ then } sv \text{ else } \langle\rangle)\cdot \mathrm{mask}^k_{k'+1}\ rs\ xs
\end{aligned}
$$

```
node feed_pause()
returns (step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);
  step = time > 125;
tel
```

time $\Downarrow$ ts	...
step	...
false fby step $\Downarrow$ rs	...
$\mathrm{mask}^0\ rs\ ts$	...
$\mathrm{mask}^1\ rs\ ts$	...
$\mathrm{mask}^2\ rs\ ts$	...

Reset [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs] – Stream Semantics

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

```
node feed_pause()
returns (step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);
  step = time > 125;
tel
```

`time` $\Downarrow$ *ts*	50	100	150	...
`step`	F	F	T	...
`false fby step` $\Downarrow$ *rs*	F	F	F	...
$\text{mask}^0\ rs\ ts$	50	100	150	...
$\text{mask}^1\ rs\ ts$				...
$\text{mask}^2\ rs\ ts$				...

Reset [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs] – Stream Semantics

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

```
node feed_pause()
returns (step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);
  step = time > 125;
tel
```

time $\Downarrow$ ts				50	100	150	...
step	F	F	T	F	F	T	...
false fby step $\Downarrow$ rs	F	F	F	T	F	F	...
$\mathsf{mask}^0\ rs\ ts$							...
$\mathsf{mask}^1\ rs\ ts$				50	100	150	...
$\mathsf{mask}^2\ rs\ ts$							...

$$
\begin{aligned}
\mathrm{mask}^k_{k'}\,(\mathtt{F}\cdot rs)\,(sv\cdot xs) &\equiv (\text{if } k' = k \text{ then } sv \text{ else } \langle\rangle)\cdot \mathrm{mask}^k_{k'}\; rs\; xs\\
\mathrm{mask}^k_{k'}\,(\mathtt{T}\cdot rs)\,(sv\cdot xs) &\equiv (\text{if } k'+1 = k \text{ then } sv \text{ else } \langle\rangle)\cdot \mathrm{mask}^k_{k'+1}\; rs\; xs
\end{aligned}
$$

```
node feed_pause()
returns (step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);
  step = time > 125;
tel
```

time ⇓ *ts*							50	...
step	F	F	T	F	F	T	F	...
false fby step ⇓ *rs*	F	F	F	T	F	F	T	...
mask^0 *rs ts*								...
mask^1 *rs ts*								...
mask^2 *rs ts*							50	...

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

```
node feed_pause()
returns (step : bool)
var time : int;
let
  reset
    time = count_up(50)
  every (false fby step);
  step = time > 125;
tel
```

time $\Downarrow$ *ts*	50	100	150	50	100	150	50	...
step	F	F	T	F	F	T	F	...
false fby step $\Downarrow$ *rs*	F	F	F	T	F	F	T	...
mask^0 *rs ts*	50	100	150	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	...
mask^1 *rs ts*	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	50	100	150	$\langle\rangle$	...
mask^2 *rs ts*	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	50	...

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

```
node pwm(d : bool) returns (ena : bool)
let
  automaton initially Off
  state Off do
    ena = false;
  unless count_up(15) > 50 then On
  state On do
    ena = true;
  unless count_up(15) > 50 then Off
  end
tel
```

- Allow for the specification of sequential, modal behaviors
- Weak (`until`) or strong (`unless`) transitions we do not mix the two
- Entry by reset (`then`) or by history (`continue`)
- May be nested

Hierarchical State Machines – Example

```
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 -> false;
    unless false -> time >= 750 then Moving

    state Moving do
      step = true -> 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*

Hierarchical State Machines – Stream semantics

`automaton` = `switch` + `reset`; translation semantics ? [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]

Hierarchical State Machines – Stream semantics

`automaton` = `switch` + `reset`; translation semantics ? [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]

$$\text{select}_{k'}^{C,k}\ sts\ xs \equiv \text{mask}_{k'}^{k}\ (\text{when}^{C}\ \pi_2(sts)\ \pi_1(sts))\ (\text{when}^{C}\ xs\ \pi_1(sts))$$

Hierarchical State Machines – Stream semantics

`automaton` = `switch` + `reset`; translation semantics ? [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]

$$\text{select}_{k'}^{C,k}\ sts\ xs \equiv \text{mask}_{k'}^{k}\ (\text{when}^{C}\ \pi_2(sts)\ \pi_1(sts))\ (\text{when}^{C}\ xs\ \pi_1(sts))$$

Case-by-base coinductive definition:

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

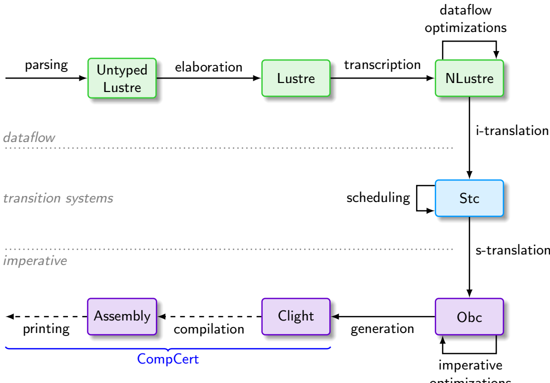
Compilation

Compilation to imperative code : Obc

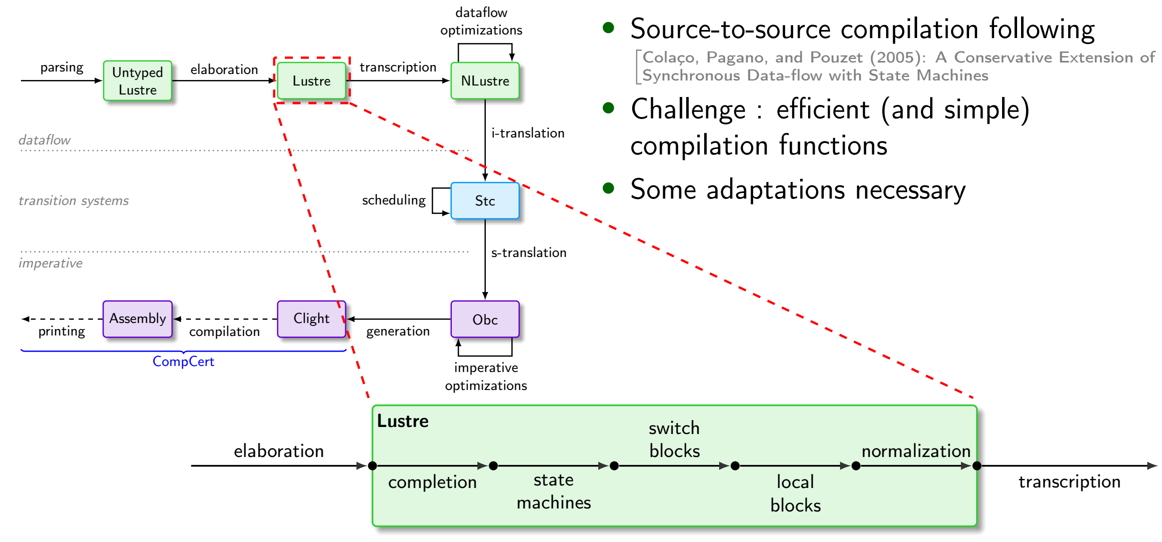
```
node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel
```

```
class count_up {
  state norm2$1 : int;

  method reset() {
    state(norm2$1) := 0
  }

  method step(inc : int) returns (o : int) {
    o := state(norm2$1) + inc;
    state(norm2$1) := o
  }
}
```

Compilation to imperative code – Front-end



Compilation to imperative code – Front-end



- Source-to-source compilation following [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]
- Challenge : efficient (and simple) compilation functions
- Some adaptations necessary

Hierarchical State Machines – Compilation

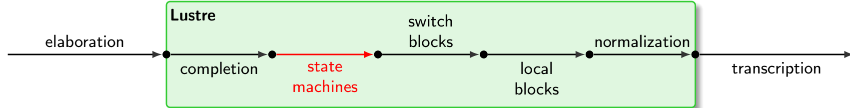
Lustre

elaboration → completion → state machines → switch blocks → local blocks → normalization → transcription

```
node feed_pause(pause : bool)
returns (ena, step : bool)
var time : int;
let
  ...
  automaton initially Feed
  state Feed do
    ena = true; ...
  unless pause then Pause

  state Pause do
    step = false; ...
  unless (not pause) and time >= 750 then Feed
       | (not pause) continue Feed
  end
tel
```

```
node feed_pause (pause : bool)
returns (ena, step : bool)
var time : int32;
let ...
  var auto$2, auto$3 : auto$1; auto$4, auto$5 : bool;
  let
    (auto$2, auto$4) = (Feed, false) fby (auto$3, auto$5);
    switch auto$2
    | Feed do
      reset
        (auto$3, auto$5) =
          if pause then (Pause, true) else (Feed, false)
      every auto$4
    | Pause do ...
    end;
    switch auto$3
    | Feed do reset ena = true; ... every auto$5
    | Pause do ...
    end
```

Hierarchical State Machines – Compilation



Lemma (Correctness invariant for the compilation of state machines)

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

The proof is simple for two reasons

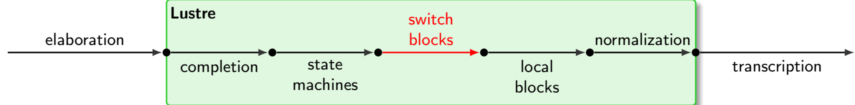
- semantic correspondence between `select`, `mask` and `when` (translation-like semantics)
- reasoning is local (introduction of local blocks)

Switch – Compilation


```
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
```

```
node drive_sequence (step : bool) returns (mA, mB : bool)
let
  var swi$mA$2 : bool when true(step); ...;
  let
    mA = merge step (true => swi$mA$4) (false => swi$mA$9);
    mB = merge step (true => swi$mB$5) (false => swi$mB$10);
    swi$mA$4 = not swi$mB$3;
    swi$mB$5 = swi$mA$2;
    swi$mA$2 = last mA when true(step);
    swi$mB$3 = last mB when true(step);
    (swi$mA$9, swi$mB$10) = (swi$mA$7, swi$mB$8);
    swi$mA$7 = last mA when false(step);
    swi$mB$8 = last mA when false(step);
  tel
  last mA = true;
  last mB = false;
tel
```

Switch – Compilation



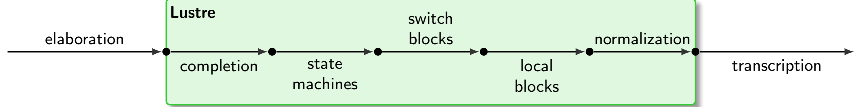
Lemma (Correctness invariant for the compilation of switch blocks)

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

The proof is more involved

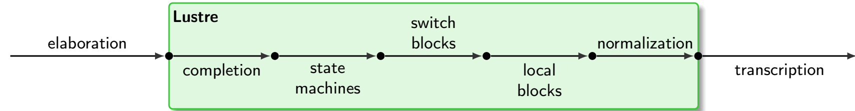
- renaming/resampling of sub-expressions (parameters σ and ck)
- same semantic operators for `switch` and `when` simplify the "core" of the proof

Shared variables – Simple Compilation



- Simple approach (taken by [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines], Heptagon)
- Eliminate `last` early, transforms into `fby` equations
- Same semantic operator : proof of correctness easy

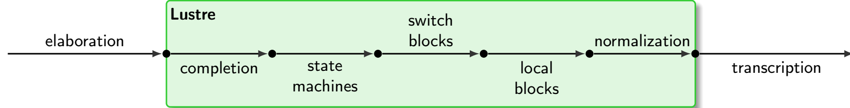
Shared variables – Simple Compilation




```
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
```

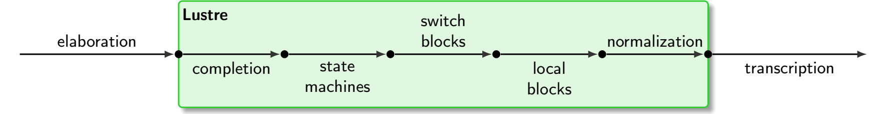
```
node drive_sequence (step : bool)
returns (mA, mB : bool)
var last$mA, last$mB : bool)
let
  switch step
  | true do
    mA = not last$mB;
    mB = last$mA;
  | false do (mA, mB) = (last$mA, last$mB)
  end;
  last$mA = true fby mA;
  last$mB = false fby mB;
tel
```

Shared variables – Limitations




```
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
```

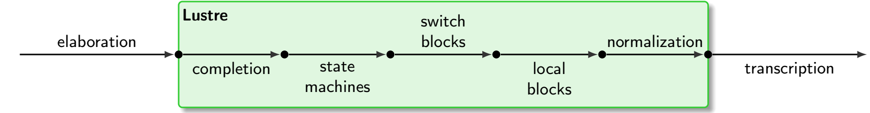
Shared variables – Limitations




```
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
```

```
class drive_sequence {
  state last$mB : bool;
  state last$mA : bool;
  method step(step : bool) returns (mA, mB : bool) {
    switch step {
    | true => mB := state(last$mA);
              mA := not state(last$mB)
    | false => mB := state(last$mB);
               mA := state(last$mA)
    };
    state(last$mB) := mB;
    state(last$mA) := mA
  }
  method reset() {
    state(last$mB) := false;
    state(last$mA) := true
  }
}
```

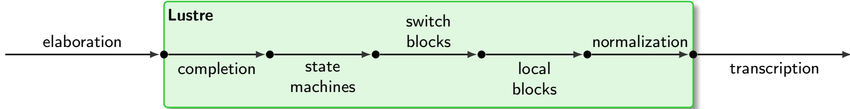
Shared variables – Limitations




```
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
```

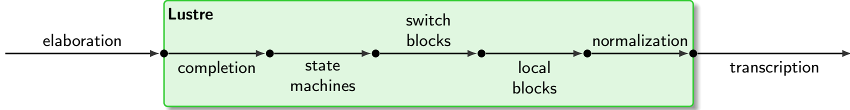
```
class drive_sequence {
  state last$mB : bool;
  state last$mA : bool;
  method step(step : bool) returns (mA, mB : bool) {
    switch step {
    | true => tmp := state(last$mA);
              state(last$mA) := not state(last$mB)
              state(last$mB) := tmp
    | false => skip
    };
    mB := state(last$mB)
    mA := state(last$mA)
  }
  method reset() {
    state(last$mB) := false;
    state(last$mA) := true
  }
}
```

Shared variables – More involved compilation



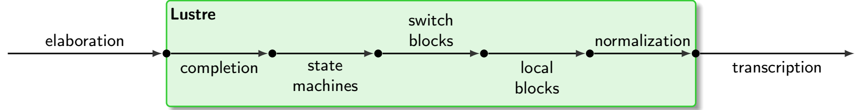
- Keep `last` until Obc
  - In Lustre: no syntax/semantics change, but more work in all compilation correctness proof
  - In NLustre: restricted compared to Lustre (no `last` on outputs, initialized by constants, defined by simple expressions)
  - In Stc: two type of states/updates: `last` (updates during the cycle) and `next` (updates at the end of the cycle)

Shared variables – More involved compilation



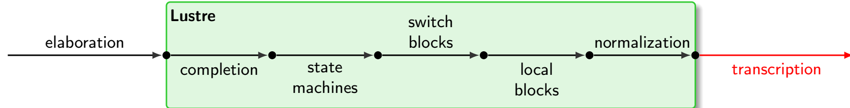
- Keep `last` until Obc
  - In Lustre: no syntax/semantics change, but more work in all compilation correctness proof
  - In NLustre: restricted compared to Lustre (no `last` on outputs, initialized by constants, defined by simple expressions)
  - In Stc: two type of states/updates: `last` (updates during the cycle) and `next` (updates at the end of the cycle)
- This makes `last` a primary construction in the compiler

Shared variables – More involved compilation



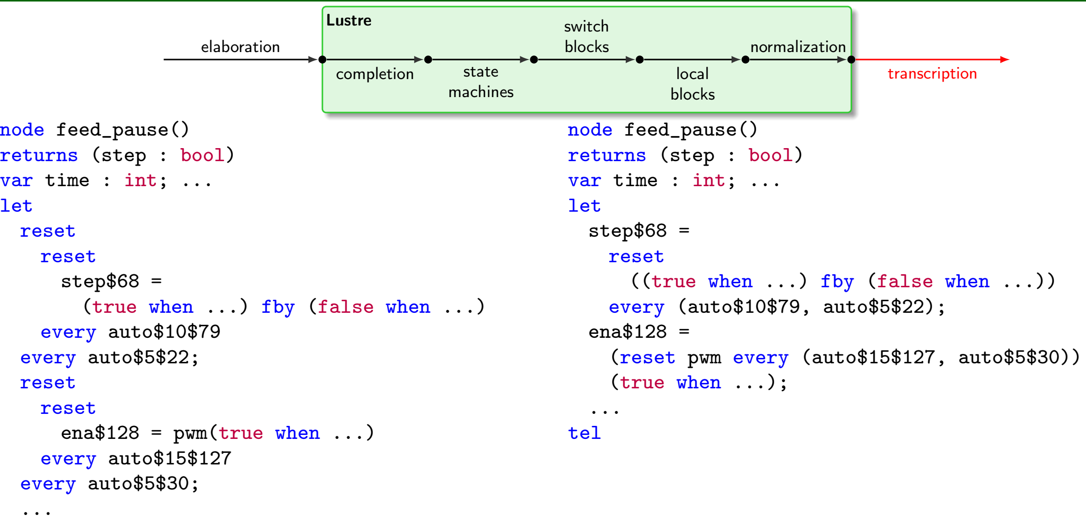
- Keep `last` until Obc
  - In Lustre: no syntax/semantics change, but more work in all compilation correctness proof
  - In NLustre: restricted compared to Lustre (no `last` on outputs, initialized by constants, defined by simple expressions)
  - In Stc: two type of states/updates: `last` (updates during the cycle) and `next` (updates at the end of the cycle)
- This makes `last` a primary construction in the compiler
- Add an optimizing pass to remove `state(x) := state(x)` instructions in Obc

Reset – Compilation



- Cannot be done directly in Lustre
- Flatten `reset` blocks, keep them until Obc
- Separate `reset` constraints to improve scheduling

Reset – Compilation



NLustre → Stc

Stc : intermediate language designed to facilitate scheduling

```
node feed_pause()
returns (step : bool)
var time : int; ...
let
  step$68 =
    reset
      ((true when ...) fby (false when ...))
    every (auto$10$79, auto$5$22);
  ena$128 =
    (reset pwm every (auto$15$127, auto$5$30))
    (true when ...);
  ...
tel
```

```
system feed_pause {
  init step$68 = true;
  sub <ena$128> : pwm;

  transition(pause : bool) returns (ena, step : bool)
  var ...;
  reset step$68 = true
    every Feed(auto$3) on auto$10$79;
  reset step$68 = true
    every Feed(auto$3) on SpeedUp(auto$8) on auto$5$22;
  next step$68 =
    false when Feed(auto$3) when SpeedUp(auto$8);
  ...
  reset pwm<ena$128>
    every Pause(auto$3) on auto$10$79;
  reset pwm<ena$128>
    every Pause(auto$3) on PWM(auto$13) on auto$5$22;
  ena$128 = pwm<ena$128>(true when ...);
  ...
}
```

Stc → Obc

```
system feed_pause {
  init step$68 = true;
  sub <ena$128> : pwm;

  transition(pause : bool) returns (ena, step : bool)
  var ...;
  reset step$68 = true
    every Feed(auto$3) on auto$10$79;
  reset step$68 = true
    every Feed(auto$3) on SpeedUp(auto$8) on auto$5$22;
  next step$68 =
    false when Feed(auto$3) when SpeedUp(auto$8);
  ...
  reset pwm<ena$128>
    every Pause(auto$3) on auto$10$79;
  reset pwm<ena$128>
    every Pause(auto$3) on PWM(auto$13) on auto$5$22;
  ena$128 = pwm<ena$128>(true when ...);
  ...
}
```

```
class feed_pause {
  state step$68 : int;
  instance ena$128 : pwm;
  method step(pause : bool)
  returns (ena, step : bool)
  var ···; {
    switch auto$3 {
    | Feed =>
      switch auto$10$79 {
      | true => step$68 = false
      | false => skip
      };
      switch auto$8 {
      ···
      }
      step$68 = false;
    | Pause => ···
    }
  }
  method reset() {
    step$68 = true;
    pwm<ena$128>.reset();
  }
```

Dependency Analysis

Proving semantic properties

We want to prove properties of the semantic model, for any well-formed program

- 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-Type preservation:
  if $\Gamma \vdash e : ck$ **and** $G, H, bs \vdash e \Downarrow vs$ **then** $H, bs \vdash ck \Downarrow (\text{abstract-clock } vs)$
- ...

The proof would usually proceed by induction on the syntax, and inversion of the semantic hypothesis. What are the base cases ?

- constant: inversing $G, H, bs \vdash c \Downarrow [vs]$ tells us vs is a constant stream
- variable: inversing $G, H, bs \vdash x \Downarrow [vs]$ tells us $H(x) \equiv vs$. What now ?

Dependency Analysis

Consider a program with the following definitions:

- `x = x;` could take any value
- `x = x + 1;` doesn't have any value

Dependency Analysis

Consider a program with the following definitions:

- `x = x;` could take any value
- `x = x + 1;` doesn't have any value

In both of these cases, not possible to prove any interesting property of the stream associated with `x`. We can only reason on program without dependency cycle.

Dependency Analysis

Consider a program with the following definitions:

- `x = x;` could take any value
- `x = x + 1;` doesn't have any value

In both of these cases, not possible to prove any interesting property of the stream associated with `x`. We can only reason on program without dependency cycle.

In NLustre, program without cycle = schedulable. In Lustre, we implement a dedicated dependency analysis, à la [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE], extended to handle control structures.

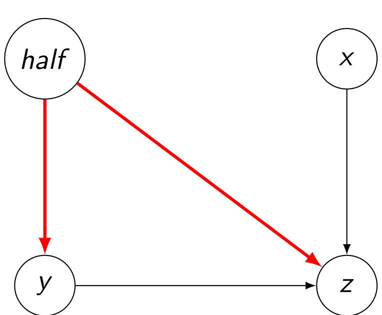
- Treatment of activation structures using labels
- Improved graph analysis algorithm
- Used to prove more properties of the semantics (clock system correctness, determinism)

Dependency Analysis of the Dataflow Language

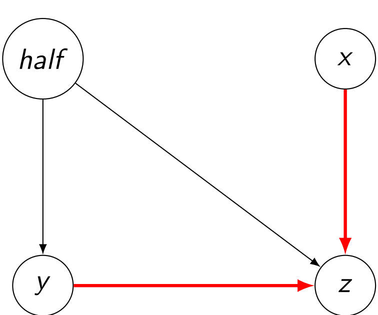
```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```

Dependency Analysis of the Dataflow Language

```
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
```

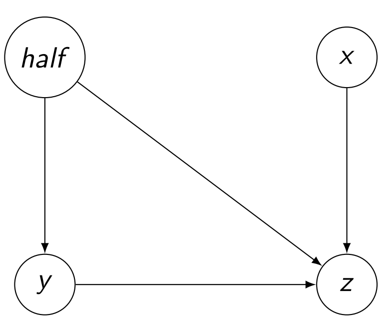


```
node f(x : int) returns (y, z : int)
var half : bool;
let
    half = true fby (not half);
    (y, z) = if half then (0, x) else (1, y);
tel
```



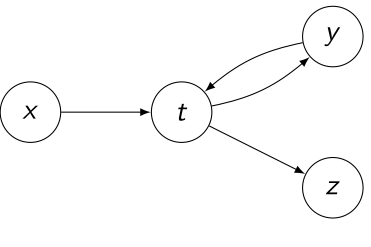
Dependency Analysis of the Dataflow Language

```
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
```



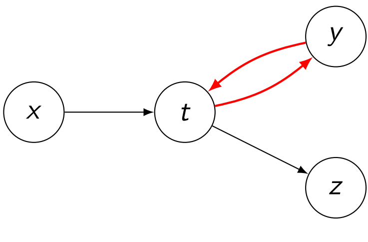
Dependencies in Local Scopes

```
node f(x : int) returns (z : bool)
var y : int;
let
  var t : int;
  let t = x fby (t + 1);
      y = t;
  tel;
  var t : int;
  let t = y + 1;
      z = t > 0;
  tel
tel
```



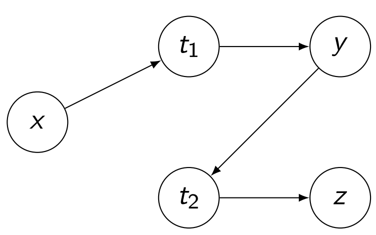
Dependencies in Local Scopes

```
node f(x : int) returns (z : bool)
var y : int;
let
  var t : int;
  let t = x fby (t + 1);
      y = t;
  tel;
  var t : int;
  let t = y + 1;
      z = t > 0;
  tel
tel
```



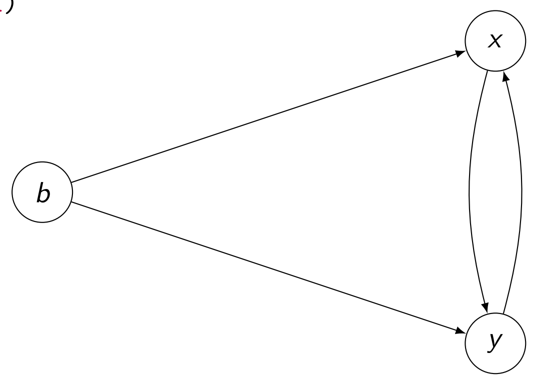
Dependencies in Local Scopes

```
node f(x(x1) : int) returns (z(z1) : bool)
var y(y1) : int;
let
  var t(t1) : int;
  let t = x fby (t + 1);
      y = t;
  tel;
  var t(t2) : int;
  let t = y + 1;
      z = t > 0;
  tel
tel
```



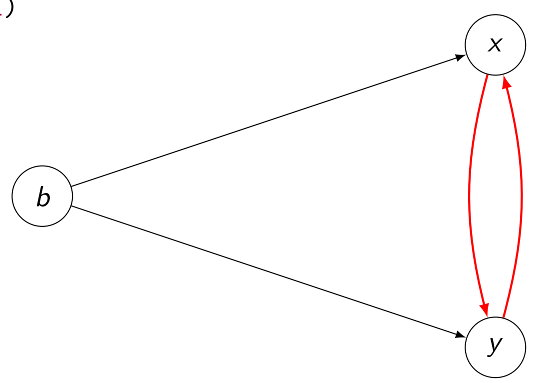
Dependencies in branches

```
node f(b : bool) returns (x, y : bool)
let
    switch b
    | true do
        x = 0 fby (x - 1);
        y = x * 2;
    | false do
        y = 0 fby (y + 1);
        x = y * 2;
    end
tel
```



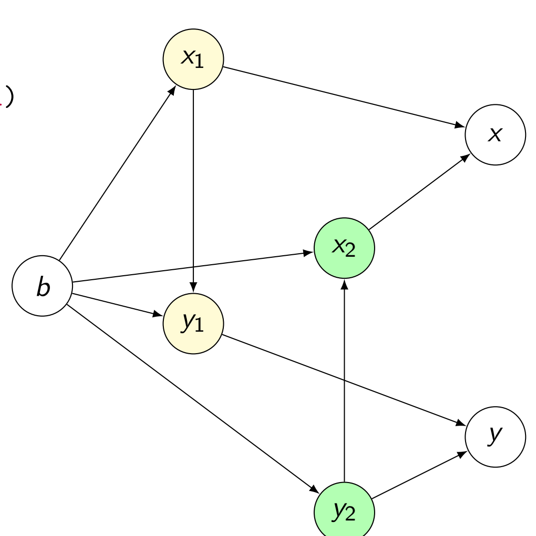
Dependencies in branches

```
node f(b : bool) returns (x, y : bool)
let
    switch b
    | true do
        x = 0 fby (x - 1);
        y = x * 2;
    | false do
        y = 0 fby (y + 1);
        x = y * 2;
    end
tel
```



Dependencies in branches

```
node f(b : bool) returns (x, y : bool)
let
    switch b
    | true do
        x^x1 = 0 fby (x^x1 - 1);
        y^y1 = x^x1 * 2;
    | false do
        y^y2 = 0 fby (y^y2 + 1);
        x^x2 = y^y2 * 2;
    end
tel
```



Dependencies and shared variables

```
var last x = 0;
let x = last x + 1;
tel
```

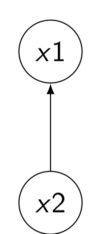
last x	0	1	2	3	...
x	1	2	3	4	...

```
var last x = x + 1;
let x = 0;
tel
```

x	0	0	0	0	...
last x	1	1	1	1	...

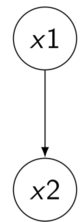
Dependencies and shared variables

```
var last x^x2 = 0;
let x^x1 = last x^x2 + 1;
tel
```



last x	0	1	2	3	...
x	1	2	3	4	...

```
var last x^x2 = x^x1 + 1;
let x^x1 = 0;
tel
```



x	0	0	0	0	...
last x	1	1	1	1	...

Dependency graph analysis

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

- 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

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}$$

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

Proving with dependencies

$$\frac{}{\text{TopoOrder}\,(\text{AcyGraph}\ V\ E)\ []}$$

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

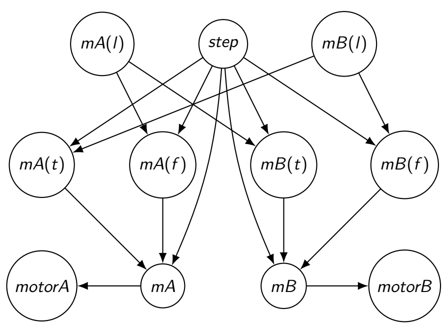
```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  switch step
  | true do
    mA^{mA(t)} = not (last mB);
    mB^{mB(t)} = last mA;
  | false do (mA^{mA(f)}, mB^{mB(f)}) = (last mA, last mB)
  end;
  last mA^{mA(l)} = true;
  last mB^{mB(l)} = false;
tel
```

Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}$$

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

```
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
```



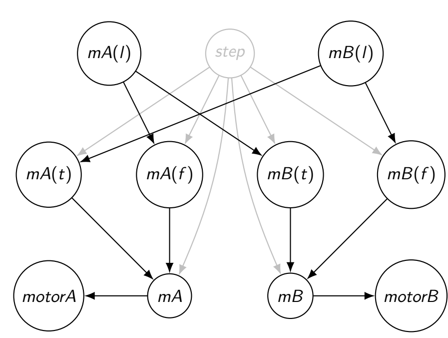
Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}$$

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

```
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
```

step



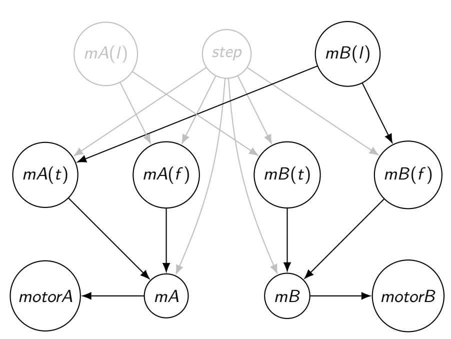
Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}$$

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

```
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
```

step - - -> mA(l)



Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}$$

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

```
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
```

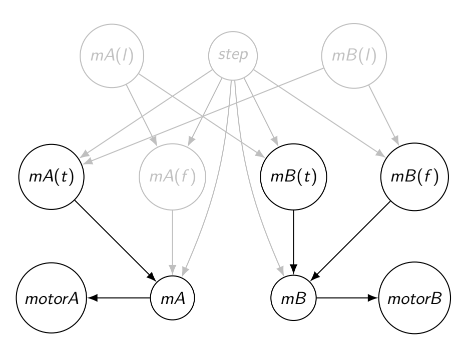
Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}$$

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

```
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
```

step - - → mA(l) - - → mA(f) - - → mB(l)



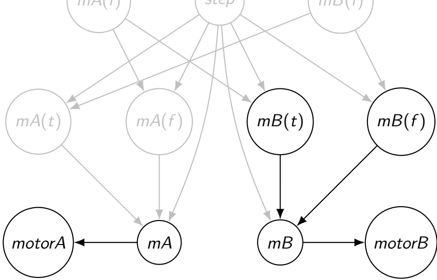
Proving with dependencies

$$\frac{}{\text{TopoOrder}\,(\text{AcyGraph}\ V\ E)\ []}$$

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

```
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
```

step → $mA(l)$ → $mA(f)$ → $mB(l)$ → $mA(t)$



Proving with dependencies

$$\frac{}{\text{TopoOrder}\ (\text{AcyGraph}\ V\ E)\ []}$$

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

```
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
```

Conclusion

Conclusion

Current version of Vélus:

- A relational semantics for the Vélus language with control structures
  - Hierarchical State Machines
  - `switch` blocks
  - `reset` blocks
  - nested local scopes
  - shared variables (`last`) with implicit completion
- An end-to-end verified compiler
- A generic approach to prove complex properties of the semantics of well-formed programs

Future works

What we would like to improve and add to Vélus:

- Shorter proofs : automate or simplify definitions ?
- Other semantic models (see the work of Paul Jeanmaire)
  - Executable
  - Prove program properties
- Causality as a type system
- Arrays ?

References

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

Boulmé, S. and G. Hamon (Dec. 2001). "Certifying Synchrony for Free". In: Proc. 8th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001). Ed. by R. Nieuwenhuis and A. Voronkov. Vol. 2250. LNCS. Havana, Cuba: Springer, pp. 495–506.

Bourke, T., L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg (June 2017). "A Formally Verified Compiler for Lustre". In: Proc. 2017 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). Barcelona, Spain: ACM Press, pp. 586–601.

Caspi, P., D. Pilaud, N. Halbwachs, and J. A. Plaice (Jan. 1987). "LUSTRE: A declarative language for programming synchronous systems". In: Proc. 14th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1987). Munich, Germany: ACM Press, pp. 178–188.

Colaço, J.-L., G. Hamon, and M. Pouzet (Oct. 2006). "Mixing Signals and Modes in Synchronous Data-flow Systems". In: Proc. 6th ACM Int. Conf. on Embedded Software (EMSOFT 2006). Ed. by S. L. Min and Y. Wang. Seoul, South Korea: ACM Press, pp. 73–82.

Colaço, J.-L., B. Pagano, and M. Pouzet (Sept. 2005). "A Conservative Extension of Synchronous Data-flow with State Machines". In: Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005). Ed. by W. Wolf. Jersey City

Semantic rules

Local Scopes and Shared Variables – Semantic rules

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

$$\frac{\forall x,\ x \in \mathrm{dom}(H') \iff x \in locs \qquad G, H + H', bs \vdash blks}{G, H, bs \vdash \texttt{var}\ locs\ \texttt{let}\ blks\ \texttt{tel}}$$

Local Scopes and Shared Variables – Semantic rules

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

$$\frac{\forall x,\ x \in \mathrm{dom}(H') \iff x \in locs \qquad G, H + H', bs \vdash blks}{G, H, bs \vdash \texttt{var}\ locs\ \texttt{let}\ blks\ \texttt{tel}}$$

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

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

Switch – Semantic rules

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

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

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

Reset – Semantics rules

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

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

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

Hierarchical State Machines – Semantic rules

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

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

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

Hierarchical State Machines – Transitions

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

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

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

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

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

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