

Verified Compilation of Synchronous Dataflow with State Machines

Timothy Bourke Basile Pesin Marc Pouzet

Inria Paris

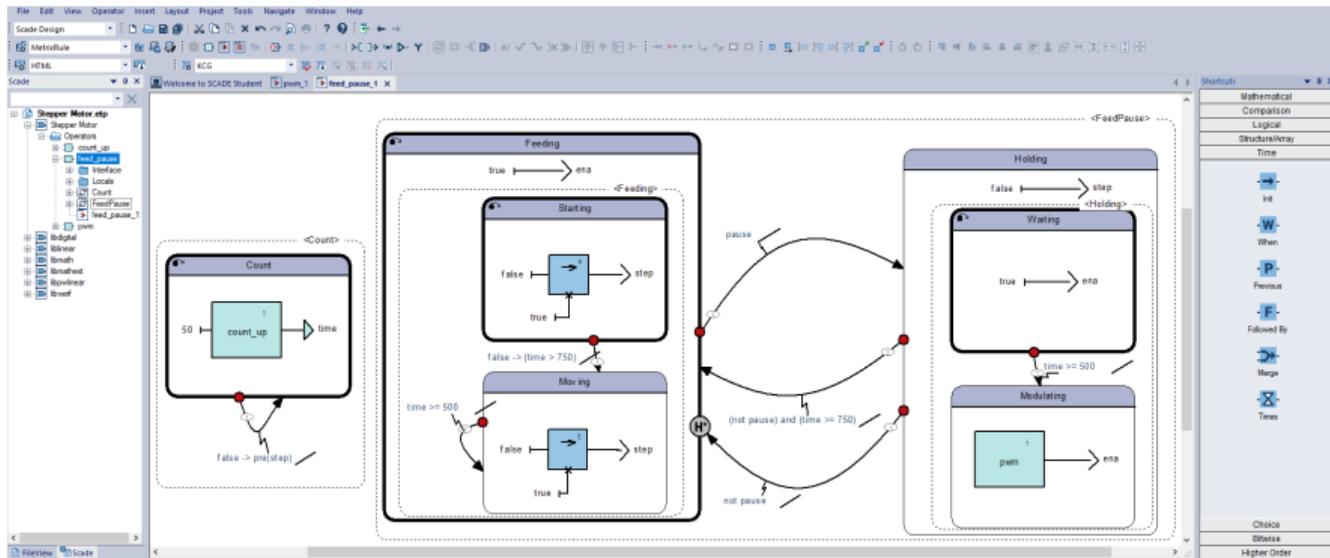
École normale supérieure, CNRS, PSL University

ESWEEK 2023 - EMSOFT

Monday, September 18

11:22am - 11:47am CET

Block-Diagram Languages for Embedded Systems

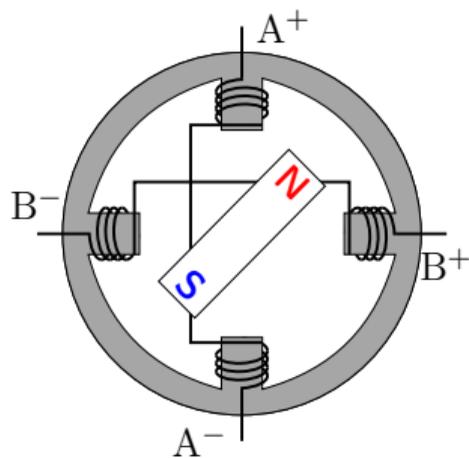
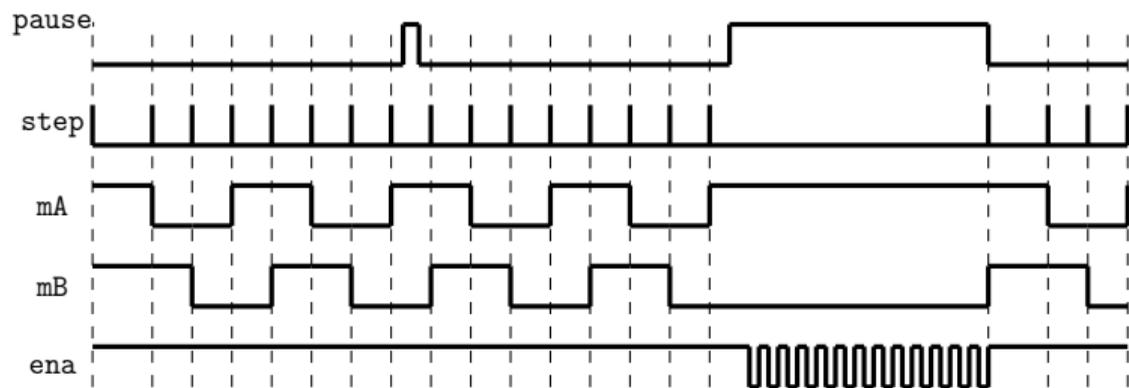


- Widely used in safety-critical applications:
Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear...
- Scade 6: Qualified compiler for Lustre + Control Structures
- Our work: Verified compilation in an Interactive Theorem Prover (Coq)

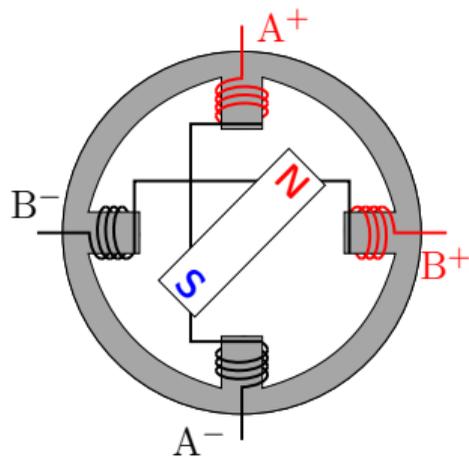
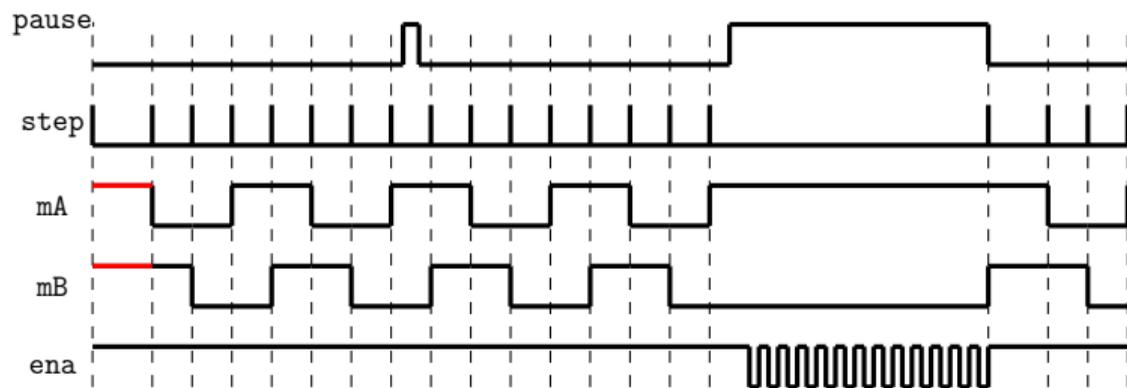
An system example: stepper motor for a small printer



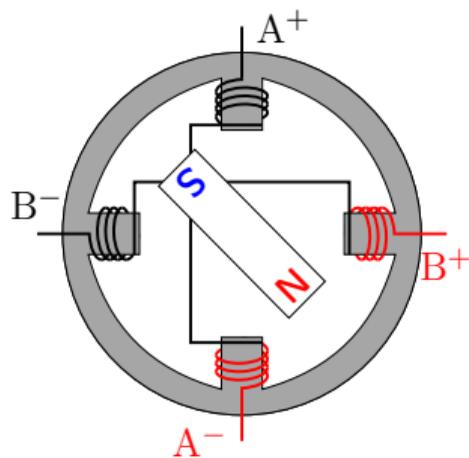
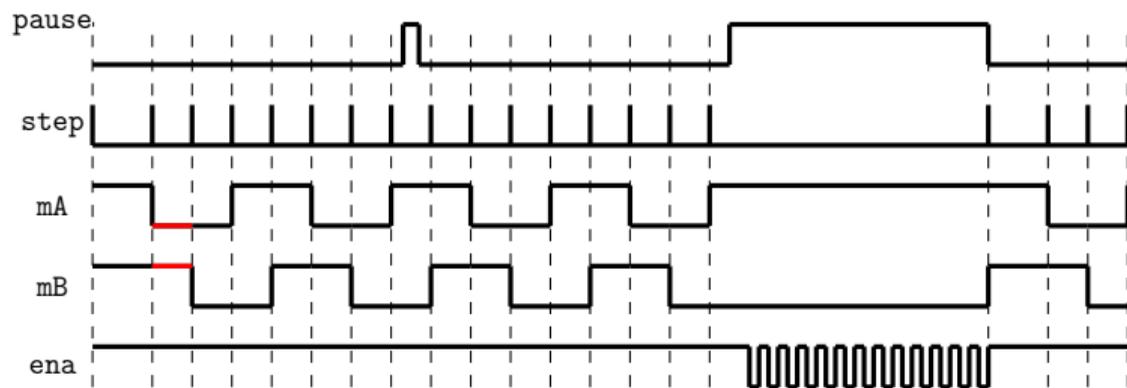
An system example: stepper motor for a small printer



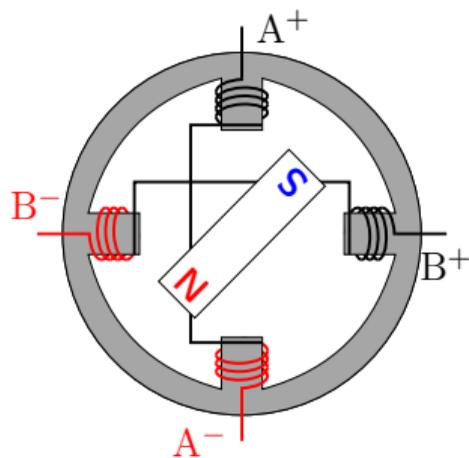
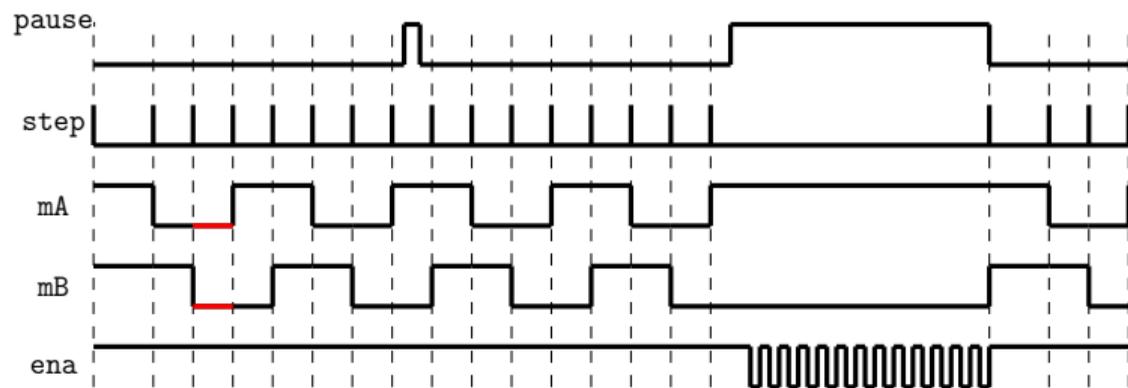
An system example: stepper motor for a small printer



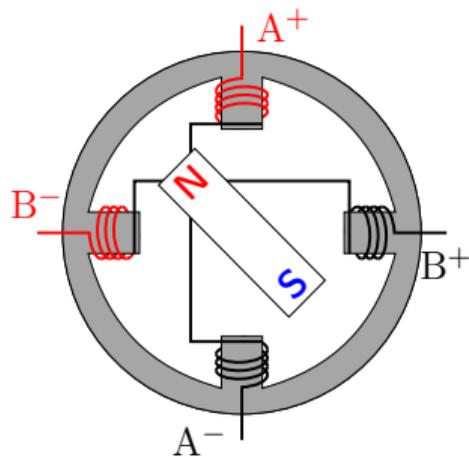
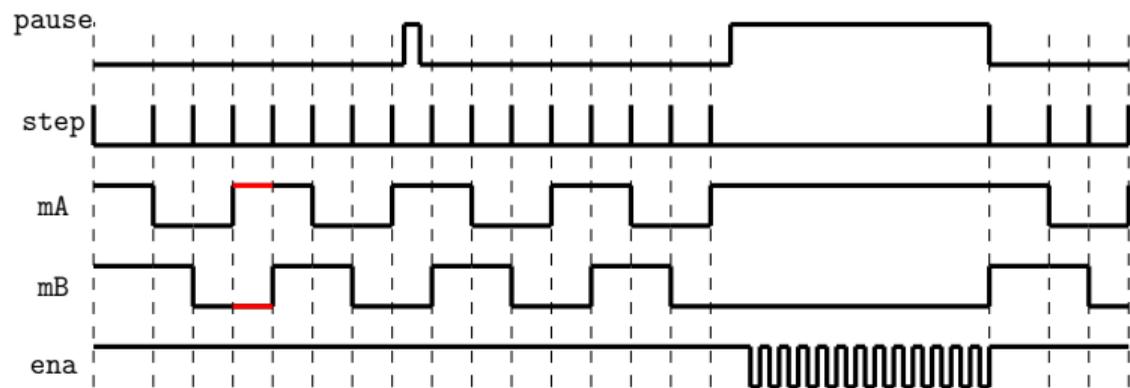
An system example: stepper motor for a small printer



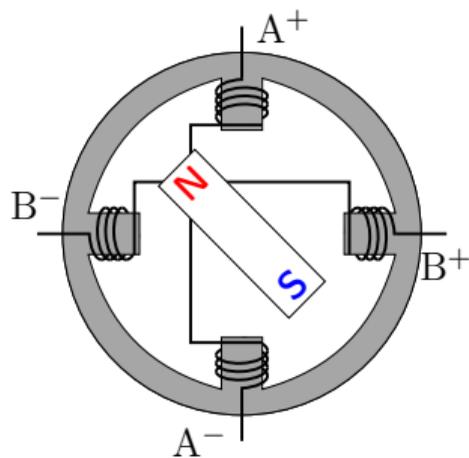
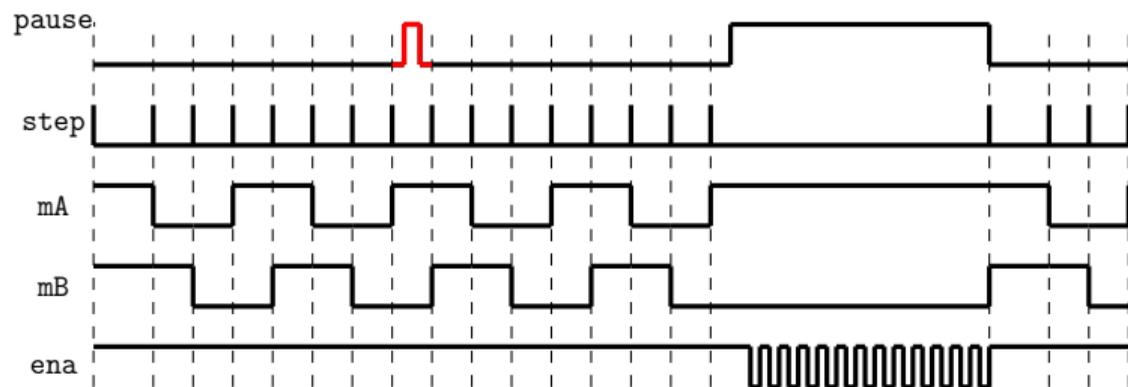
An system example: stepper motor for a small printer



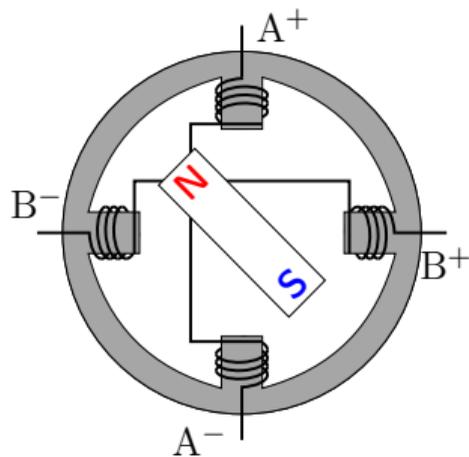
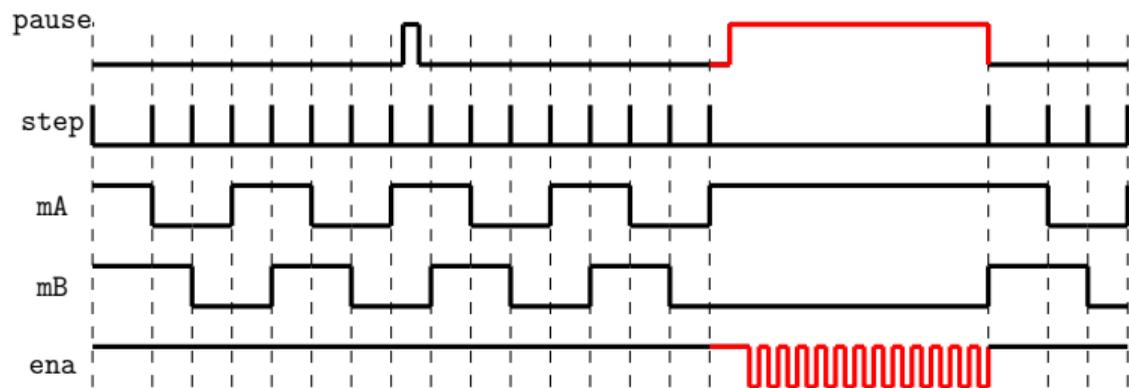
An system example: stepper motor for a small printer



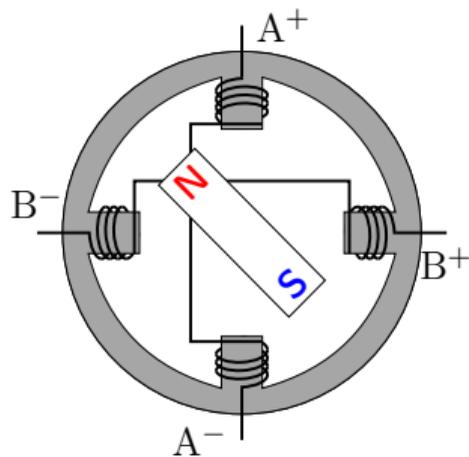
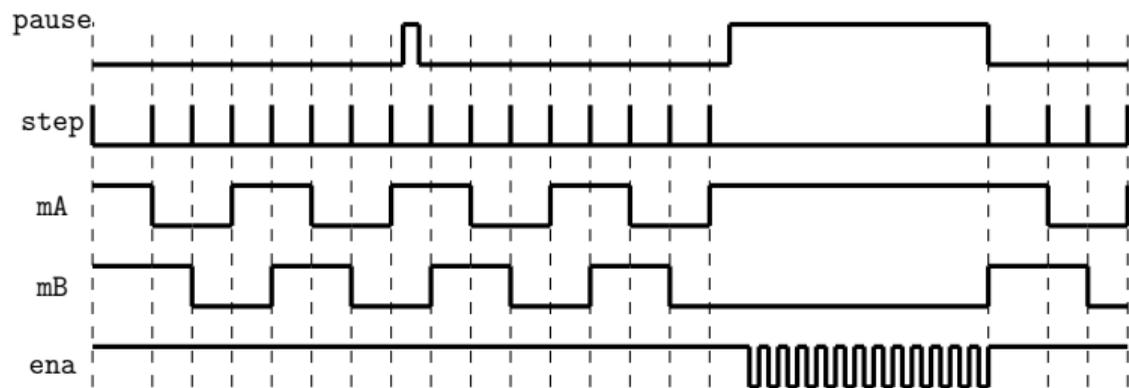
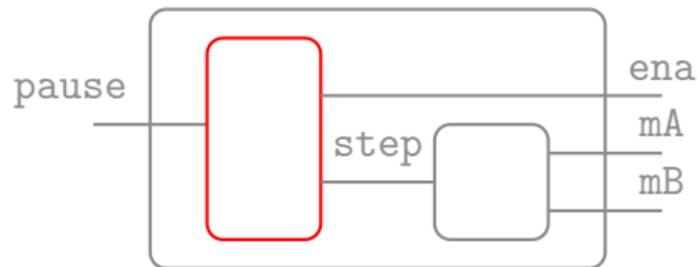
An system example: stepper motor for a small printer



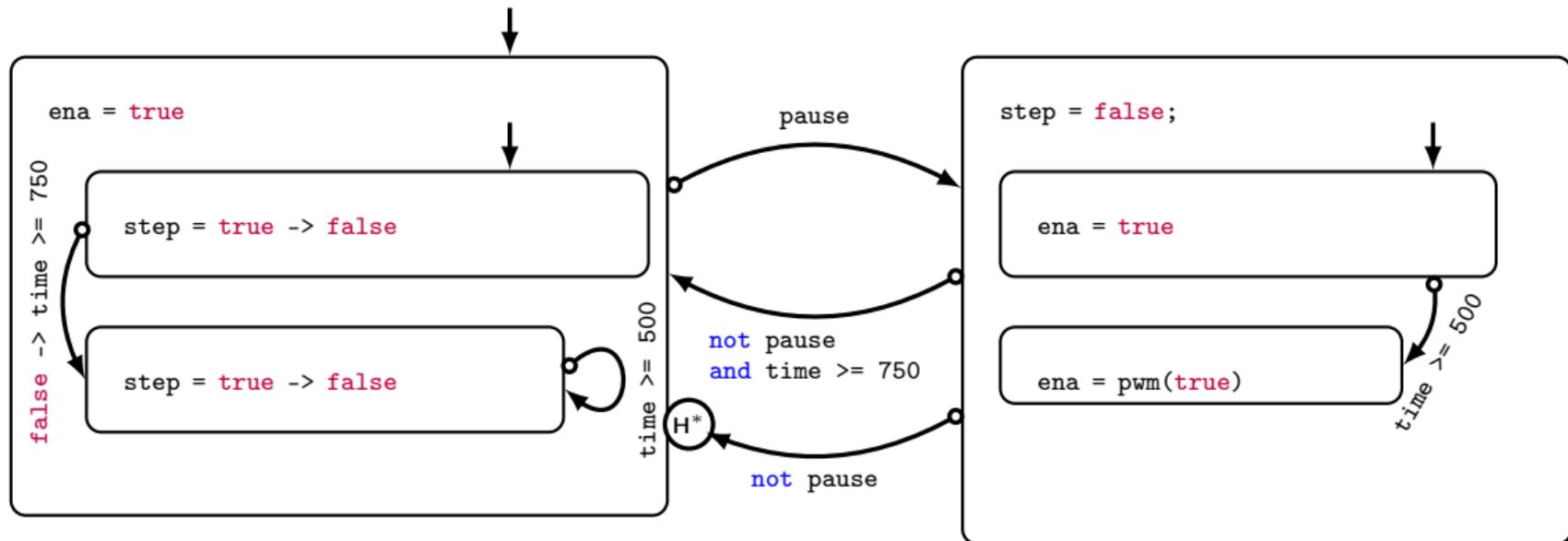
An system example: stepper motor for a small printer



An system example: stepper motor for a small printer

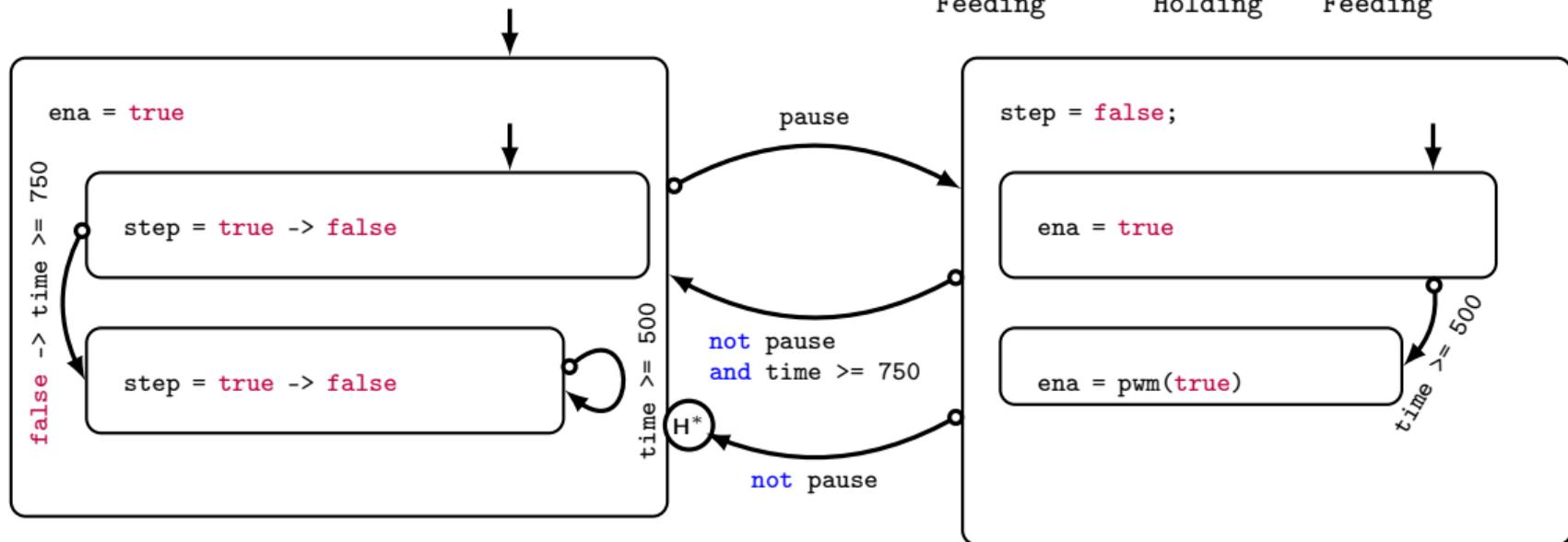


Hierarchical State Machines – Example



Hierarchical State Machines – Example

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 – Example

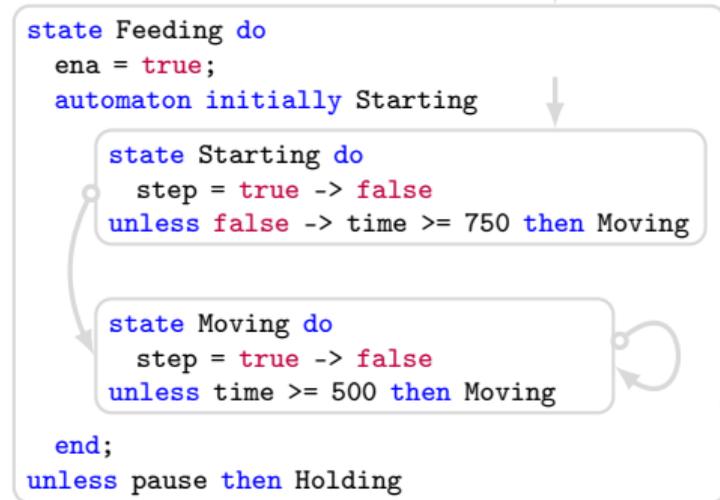
```

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

```

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			

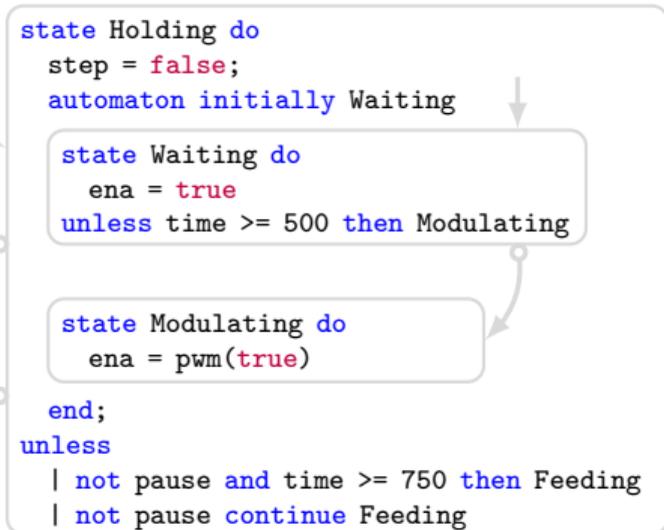
automaton initially Feeding



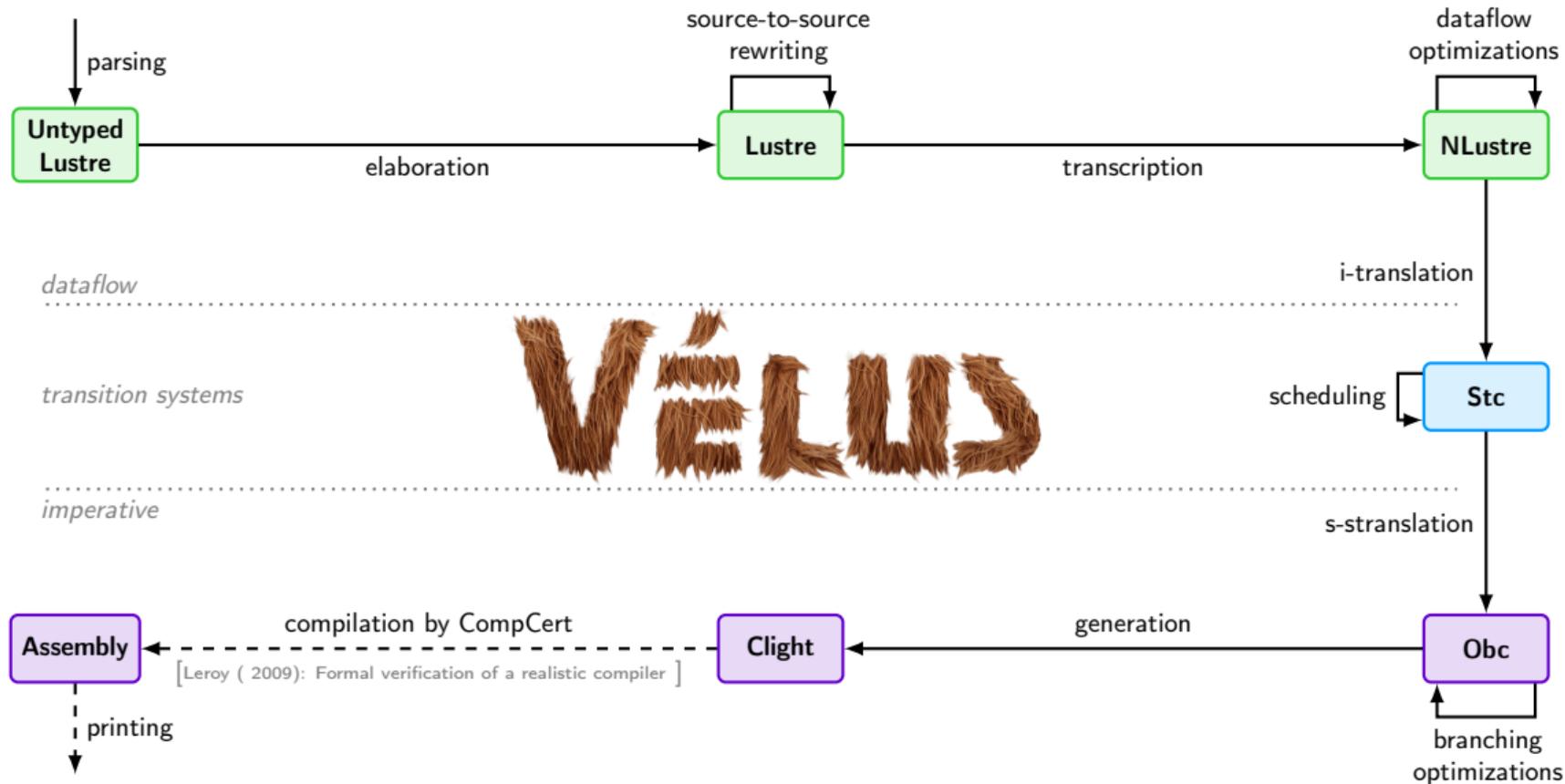
```

end
tel

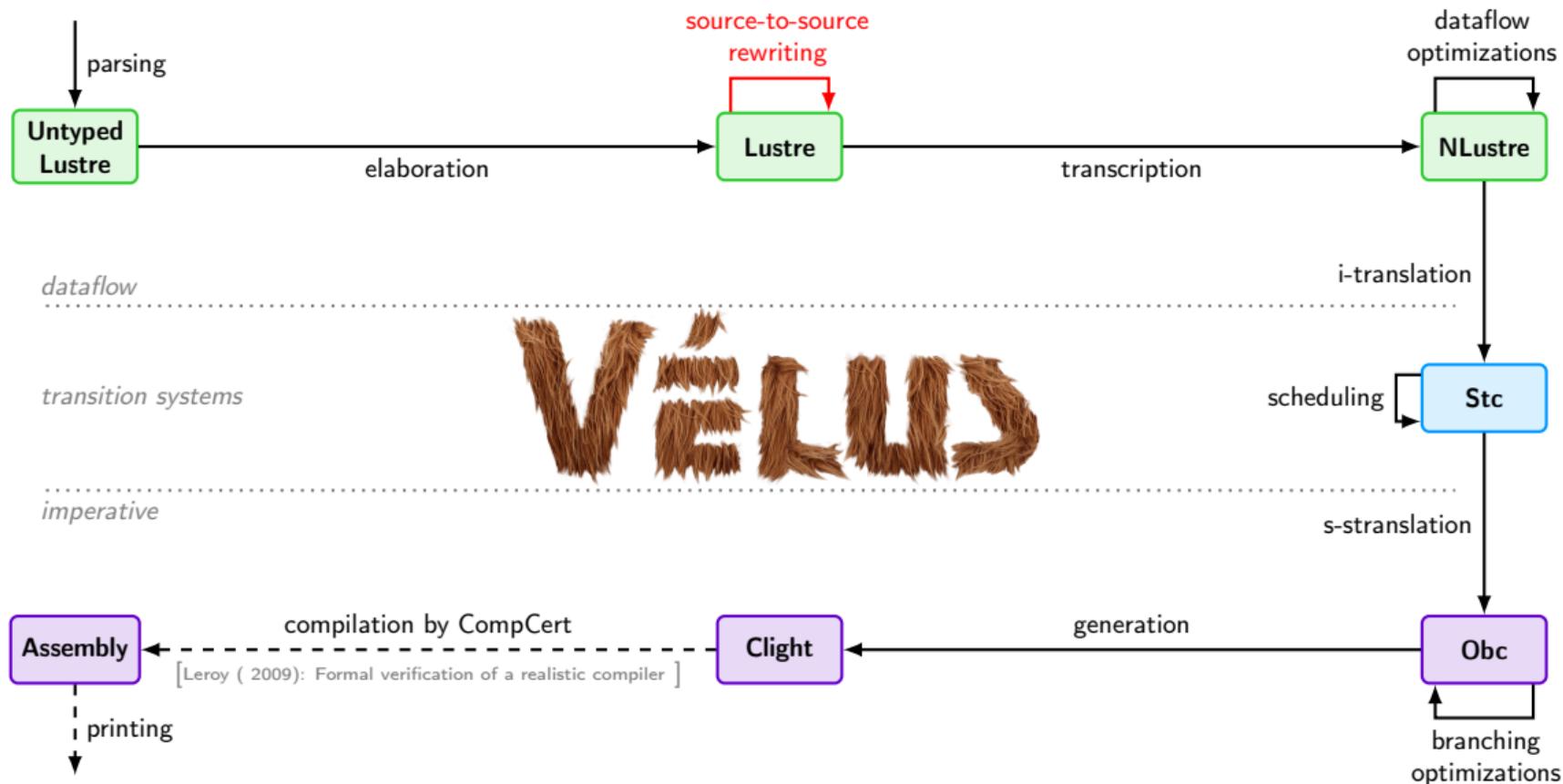
```



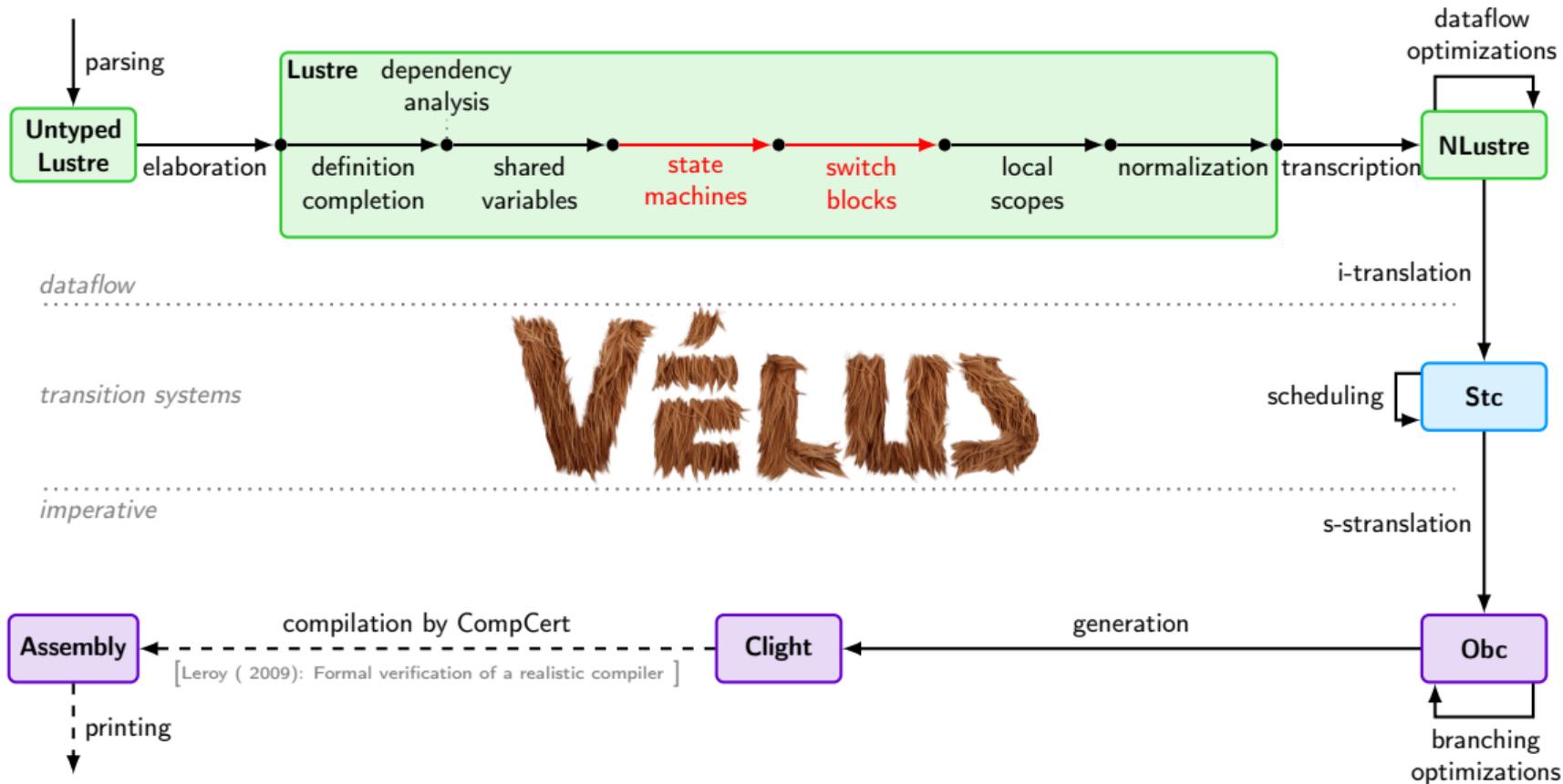
The Vélus Compiler



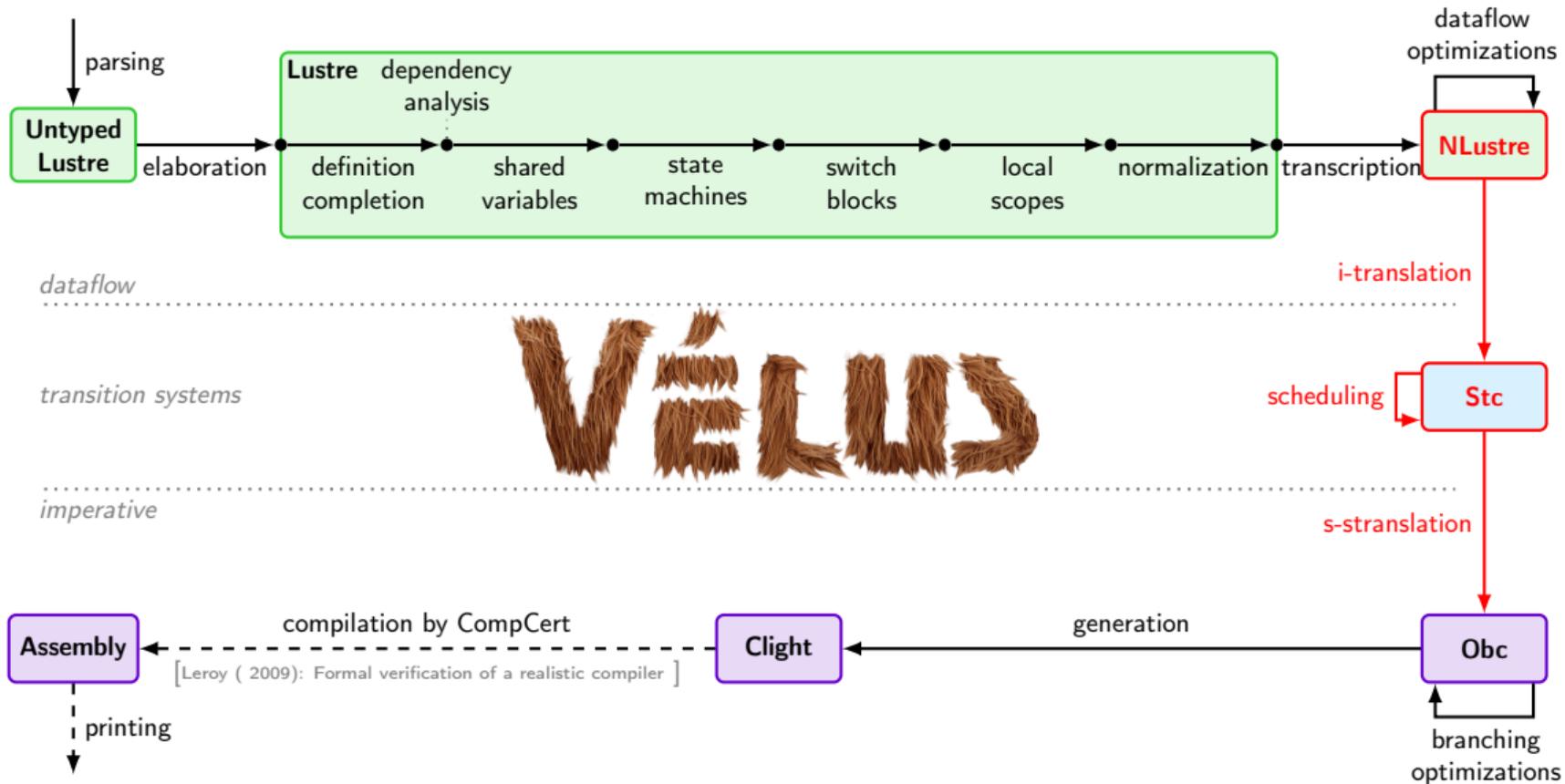
The Vélu Compiler



The Vélu Compiler



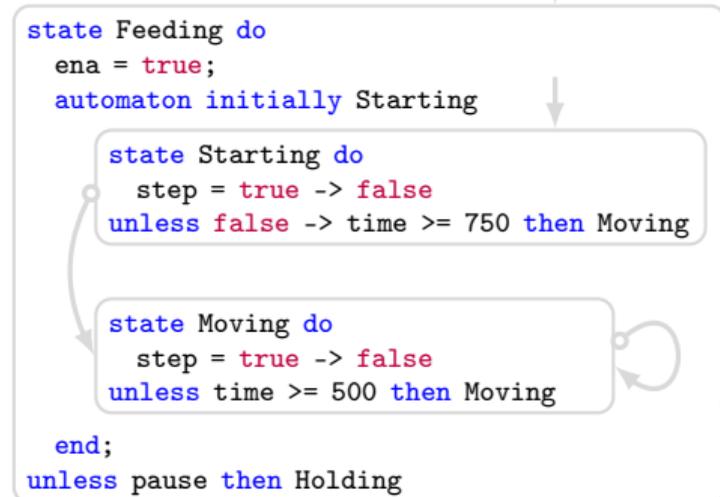
The Vélu Compiler



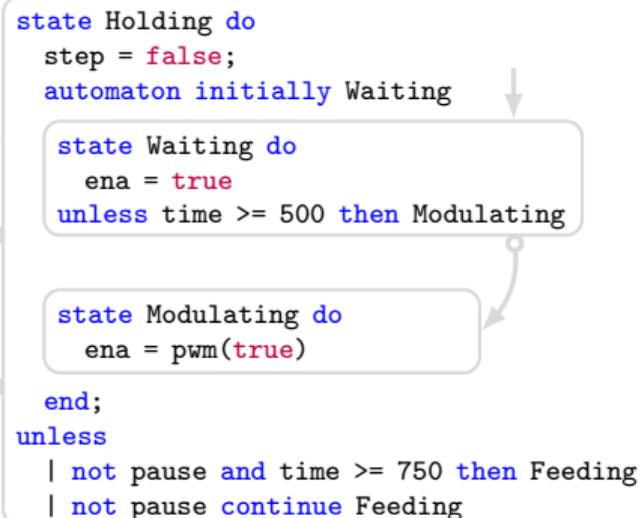
Compilation of state machines

```
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



H⁺



```
end
tel
```

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 -> false
      unless false -> time >= 750 then Moving
    end
    state Moving do
      step = true -> false
      unless time >= 500 then Moving
    end
  end;
  unless pause then Holding
```

```
end
tel
```

```
state Holding do
  step = false;
  automaton initially Waiting
    state Waiting do
      ena = true
      unless time >= 500 then Modulating
    end
    state Modulating do
      ena = pwm(true)
    end;
  unless
    | not pause and time >= 750 then Feeding
    | not pause continue Feeding
```

H*

Compilation of state machines

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

Compilation of state machines

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

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

```

state S1 do
  step = true -> false
unless false -> time >= 750 then S2
end
state S2 do
  step = true -> false
unless time >= 500 then S1
end

```

Figure 5: The translation of switch

```

state S1 do
  step = true -> false
unless false -> time >= 750 then S2
end
state S2 do
  step = true -> false
unless time >= 500 then S1
end

```

```

state S1 do
  step = true -> false
unless false -> time >= 750 then S2
end
state S2 do
  step = true -> false
unless time >= 500 then S1
end

```

Figure 6: The translation of assignment

```

state S1 do
  step = true -> false
unless false -> time >= 750 then S2
end
state S2 do
  step = true -> false
unless time >= 500 then S1
end

```

```

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

```

Compilation of state machines

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

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

```

state Starting do
  step = true -> false
  unless false -> time >= 750 then Moving
end
state Moving do
  step = true -> false
  unless time >= 500 then Moving
end

```

Figure 5: The translation of switch

```

state Starting do
  step = true -> false
  unless false -> time >= 750 then Moving
end
state Moving do
  step = true -> false
  unless time >= 500 then Moving
end

```

Figure 6: The translation of assignment

```

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

```

Compilation of state machines

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

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

```

state S1 do
  if (x < 0) then
    S2
  else
    S3
  end
end
state S2 do
  x = x + 1
end
state S3 do
  x = x - 1
end

```

```

x = y
state S1 do
  if (y < 0) then
    x = y + 1
  else
    x = y - 1
  end
end

```

```

x = y
state S1 do
  if (y < 0) then
    x = y + 1
  else
    x = y - 1
  end
end

```

possible to split and have strongly disjoint code sections that
 in, it runs more than one computation. That is a key differ-
 ence with the State-Machine or SystemC models, and largely
 simplifies program modeling/analysis and synthesis.

3.1.2 The Type System
 We should first extend the typing rules for the new pro-
 cessing constructs. The typing rules should assume the
 synchronous automaton model that it gives the same type on
 the output of the transitions. (There is a subtle but important
 difference between the two models: in the latter, the output of
 the transitions is not necessarily a value, but a signal. This
 means they are not necessarily a protected representation of
 the value.)

```

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

```

Compilation of state machines

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

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

```

state Starting do
  step = true -> false
unless false -> time >= 750 then Moving
end
state Moving do
  step = true -> false
unless time >= 500 then Moving
end
end

```

Figure 5: The translation of switch

```

state Starting do
  step = true -> false
unless false -> time >= 750 then Moving
end
state Moving do
  step = true -> false
unless time >= 500 then Moving
end
end

```

Figure 6: The translation of assignment

```

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

```


Compilation of state machines

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

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

```

state S1 do
  step = true -> false
  unless false -> time >= 750 then S2
end
state S2 do
  step = true -> false
  unless time >= 500 then S2
end

```

Figure 5: The translation of switch

```

state S1 do
  step = true -> false
  unless false -> time >= 750 then S2
end
state S2 do
  step = true -> false
  unless time >= 500 then S2
end

```

```

state S1 do
  step = true -> false
  unless false -> time >= 750 then S2
end
state S2 do
  step = true -> false
  unless time >= 500 then S2
end

```

Figure 6: The translation of assignment

```

state S1 do
  step = true -> false
  unless false -> time >= 750 then S2
end
state S2 do
  step = true -> false
  unless time >= 500 then S2
end

```

```

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

```

Compilation of switch blocks

```
switch st
| Starting do
  reset
  step = true -> false
  every res stepS = true when (st=Starting) -> false when (st=Starting)
| Holding do ...
  every resS;
end
```

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

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

```
switch (st) => (S1, S2, S3, S4) => (S1, S2, S3) =>
  S1: res = res;
  S2: res = res;
  S3: res = res;
  S4: res = res;
end
```

also, the code is translated into:

```
when st = S1 => res = res;
when st = S2 => res = res;
when st = S3 => res = res;
when st = S4 => res = res;
```

This construct highlights the fact that being set in the previous state is the previous value of st when the current state is the previous state.

```
(StateMachine S1 => (S1, S2, S3, S4) => (S1, S2, S3) =>
  S1: res = res;
  S2: res = res;
  S3: res = res;
  S4: res = res;
end
```

Figure 6: The translation of assignment

possible to write and have strongly dualistic structure, that is, it runs more than one computation. That is a key difference with the StateMachine or SystemC models, and largely simplifies program modeling and analysis.

3.1.2 The Type System

We should first extend the typing rule for the new programming construct. The typing rule should assign the type of the computation. There is a key difference in particular: the typing rule should be able to handle the fact that the same variable is used in multiple computations, but that they are not necessarily connected by the same data flow.

- sampling explicited by `when`

Compilation of switch blocks

```
switch st
| Starting do
  reset
  step = true -> 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) -> false when (st=Starting)
every resS;
```

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

```
switch (st) => (S1, S2, S3) => (S1, S2, S3) =>
  S1: res = res and S1;
  S2: res = res and S2;
  S3: res = res and S3;
  S4: res = res and S4;
  S5: res = res and S5;
  S6: res = res and S6;
  S7: res = res and S7;
  S8: res = res and S8;
  S9: res = res and S9;
  S10: res = res and S10;
  S11: res = res and S11;
  S12: res = res and S12;
  S13: res = res and S13;
  S14: res = res and S14;
  S15: res = res and S15;
  S16: res = res and S16;
  S17: res = res and S17;
  S18: res = res and S18;
  S19: res = res and S19;
  S20: res = res and S20;
  S21: res = res and S21;
  S22: res = res and S22;
  S23: res = res and S23;
  S24: res = res and S24;
  S25: res = res and S25;
  S26: res = res and S26;
  S27: res = res and S27;
  S28: res = res and S28;
  S29: res = res and S29;
  S30: res = res and S30;
  S31: res = res and S31;
  S32: res = res and S32;
  S33: res = res and S33;
  S34: res = res and S34;
  S35: res = res and S35;
  S36: res = res and S36;
  S37: res = res and S37;
  S38: res = res and S38;
  S39: res = res and S39;
  S40: res = res and S40;
  S41: res = res and S41;
  S42: res = res and S42;
  S43: res = res and S43;
  S44: res = res and S44;
  S45: res = res and S45;
  S46: res = res and S46;
  S47: res = res and S47;
  S48: res = res and S48;
  S49: res = res and S49;
  S50: res = res and S50;
  S51: res = res and S51;
  S52: res = res and S52;
  S53: res = res and S53;
  S54: res = res and S54;
  S55: res = res and S55;
  S56: res = res and S56;
  S57: res = res and S57;
  S58: res = res and S58;
  S59: res = res and S59;
  S60: res = res and S60;
  S61: res = res and S61;
  S62: res = res and S62;
  S63: res = res and S63;
  S64: res = res and S64;
  S65: res = res and S65;
  S66: res = res and S66;
  S67: res = res and S67;
  S68: res = res and S68;
  S69: res = res and S69;
  S70: res = res and S70;
  S71: res = res and S71;
  S72: res = res and S72;
  S73: res = res and S73;
  S74: res = res and S74;
  S75: res = res and S75;
  S76: res = res and S76;
  S77: res = res and S77;
  S78: res = res and S78;
  S79: res = res and S79;
  S80: res = res and S80;
  S81: res = res and S81;
  S82: res = res and S82;
  S83: res = res and S83;
  S84: res = res and S84;
  S85: res = res and S85;
  S86: res = res and S86;
  S87: res = res and S87;
  S88: res = res and S88;
  S89: res = res and S89;
  S90: res = res and S90;
  S91: res = res and S91;
  S92: res = res and S92;
  S93: res = res and S93;
  S94: res = res and S94;
  S95: res = res and S95;
  S96: res = res and S96;
  S97: res = res and S97;
  S98: res = res and S98;
  S99: res = res and S99;
  S100: res = res and S100;
```

Figure 6: The translation of switch.

```
(Assignment S1 => (S1, S2, S3) => (S1, S2, S3) =>
  S1: res = res and S1;
  S2: res = res and S2;
  S3: res = res and S3;
  S4: res = res and S4;
  S5: res = res and S5;
  S6: res = res and S6;
  S7: res = res and S7;
  S8: res = res and S8;
  S9: res = res and S9;
  S10: res = res and S10;
  S11: res = res and S11;
  S12: res = res and S12;
  S13: res = res and S13;
  S14: res = res and S14;
  S15: res = res and S15;
  S16: res = res and S16;
  S17: res = res and S17;
  S18: res = res and S18;
  S19: res = res and S19;
  S20: res = res and S20;
  S21: res = res and S21;
  S22: res = res and S22;
  S23: res = res and S23;
  S24: res = res and S24;
  S25: res = res and S25;
  S26: res = res and S26;
  S27: res = res and S27;
  S28: res = res and S28;
  S29: res = res and S29;
  S30: res = res and S30;
  S31: res = res and S31;
  S32: res = res and S32;
  S33: res = res and S33;
  S34: res = res and S34;
  S35: res = res and S35;
  S36: res = res and S36;
  S37: res = res and S37;
  S38: res = res and S38;
  S39: res = res and S39;
  S40: res = res and S40;
  S41: res = res and S41;
  S42: res = res and S42;
  S43: res = res and S43;
  S44: res = res and S44;
  S45: res = res and S45;
  S46: res = res and S46;
  S47: res = res and S47;
  S48: res = res and S48;
  S49: res = res and S49;
  S50: res = res and S50;
  S51: res = res and S51;
  S52: res = res and S52;
  S53: res = res and S53;
  S54: res = res and S54;
  S55: res = res and S55;
  S56: res = res and S56;
  S57: res = res and S57;
  S58: res = res and S58;
  S59: res = res and S59;
  S60: res = res and S60;
  S61: res = res and S61;
  S62: res = res and S62;
  S63: res = res and S63;
  S64: res = res and S64;
  S65: res = res and S65;
  S66: res = res and S66;
  S67: res = res and S67;
  S68: res = res and S68;
  S69: res = res and S69;
  S70: res = res and S70;
  S71: res = res and S71;
  S72: res = res and S72;
  S73: res = res and S73;
  S74: res = res and S74;
  S75: res = res and S75;
  S76: res = res and S76;
  S77: res = res and S77;
  S78: res = res and S78;
  S79: res = res and S79;
  S80: res = res and S80;
  S81: res = res and S81;
  S82: res = res and S82;
  S83: res = res and S83;
  S84: res = res and S84;
  S85: res = res and S85;
  S86: res = res and S86;
  S87: res = res and S87;
  S88: res = res and S88;
  S89: res = res and S89;
  S90: res = res and S90;
  S91: res = res and S91;
  S92: res = res and S92;
  S93: res = res and S93;
  S94: res = res and S94;
  S95: res = res and S95;
  S96: res = res and S96;
  S97: res = res and S97;
  S98: res = res and S98;
  S99: res = res and S99;
  S100: res = res and S100;
```

Figure 6: The translation of assignment.

- sampling explicited by **when**
- choice explicited by **merge**

Compilation of switch blocks

```

switch st
| Starting do
  reset
  step = true -> 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) -> false when (st=Starting)
every resS;
    
```

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

```

switch (st) => (S, M, H) => (st, res) =>
  S => reset;
  M => merge;
  H => merge;
end
    
```

Figure 6: The translation of switch

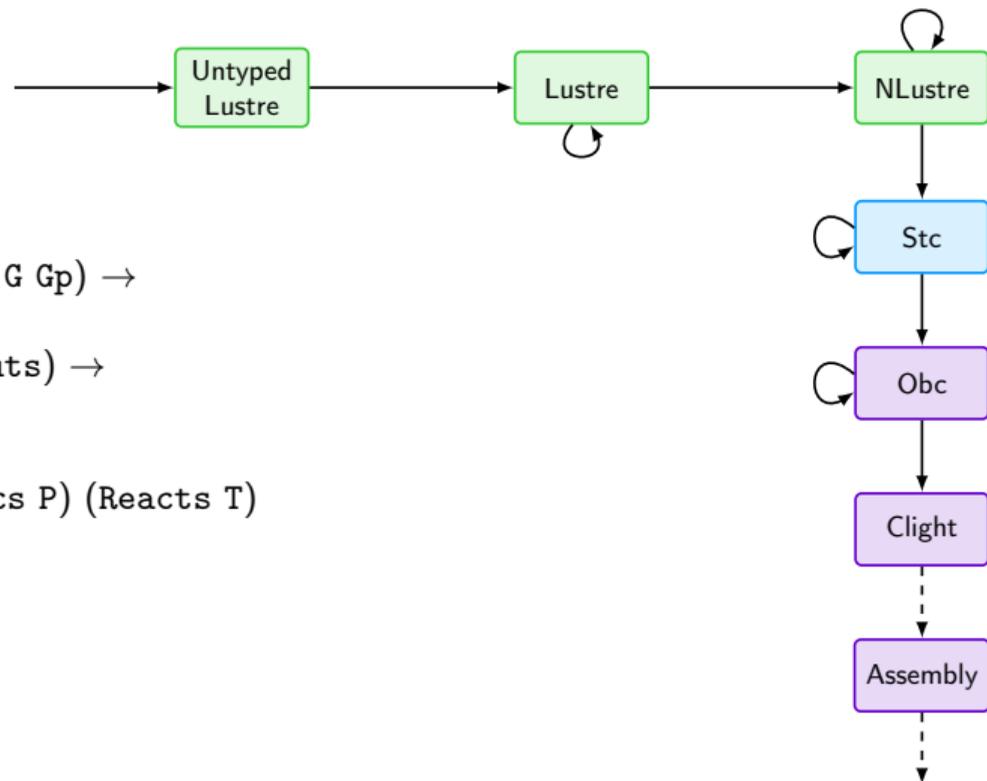
```

(translation: Si => (Si, res, res); (Mi, res, res) =>
  Si => reset;
  Mi => merge;
  Hi => merge;
end
    
```

Figure 6: The translation of assignment

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

Main Correctness Theorem



Theorem `behavior_asm`:

$$\begin{aligned} &\forall D \ G \ Gp \ P \ \text{main} \ \text{ins} \ \text{outs}, \\ &\text{elab_declarations } D = \text{OK} \ (\text{exist } _ \ G \ Gp) \rightarrow \\ &\text{compile } D \ \text{main} = \text{OK} \ P \rightarrow \\ &\text{sem_node } G \ \text{main} \ (\text{pStr} \ \text{ins}) \ (\text{pStr} \ \text{outs}) \rightarrow \\ &\text{wt_ins } G \ \text{main} \ \text{ins} \rightarrow \\ &\text{wc_ins } G \ \text{main} \ \text{ins} \rightarrow \\ &\exists T, \ \text{program_behaves} \ (\text{Asm.semantics } P) \ (\text{Reacts } T) \\ &\quad \wedge \ \text{bisim_IO } G \ \text{main} \ \text{ins} \ \text{outs} \ T. \end{aligned}$$

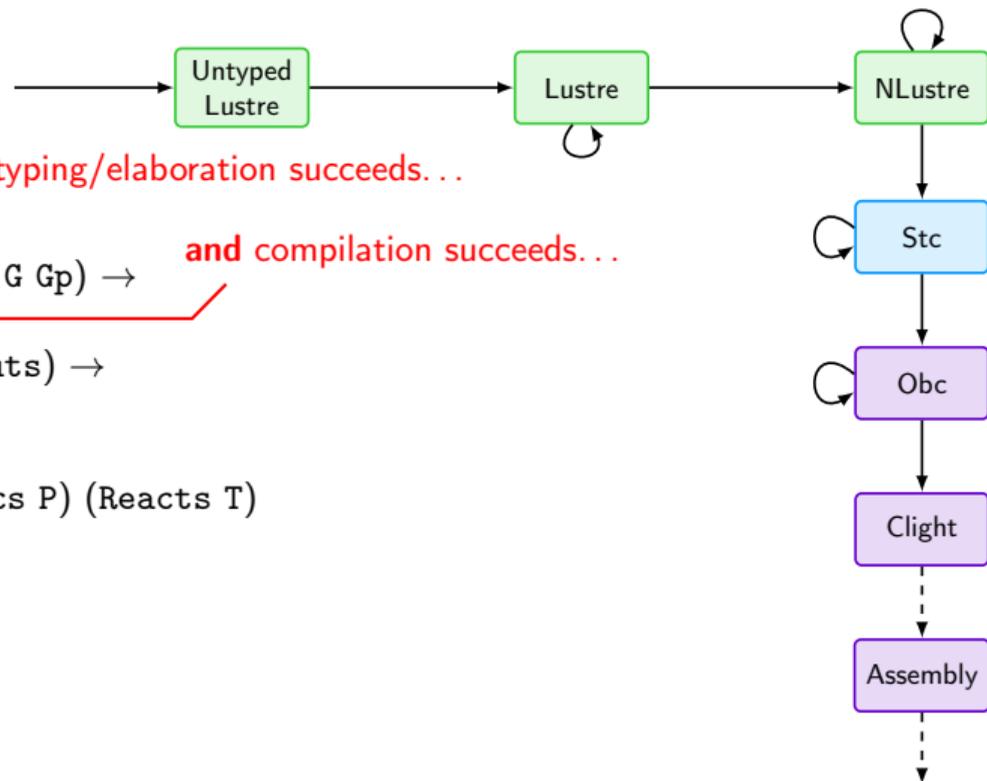
Main Correctness Theorem

Theorem `behavior_asm`:

$\forall D G Gp P \text{ main ins outs,}$
 $\text{elab_declarations } D = \text{OK } (\text{exist } _ G Gp) \rightarrow$
 $\text{compile } D \text{ main} = \text{OK } P \rightarrow$
 $\text{sem_node } G \text{ main } (\text{pStr ins}) (\text{pStr outs}) \rightarrow$
 $\text{wt_ins } G \text{ main ins} \rightarrow$
 $\text{wc_ins } G \text{ main ins} \rightarrow$
 $\exists T, \text{program_behaves } (\text{Asm.semantics } P) (\text{Reacts } T)$
 $\wedge \text{bisim_IO } G \text{ main ins outs } T.$

if typing/elaboration succeeds...

and compilation succeeds...



Main Correctness Theorem

Theorem `behavior_asm`:

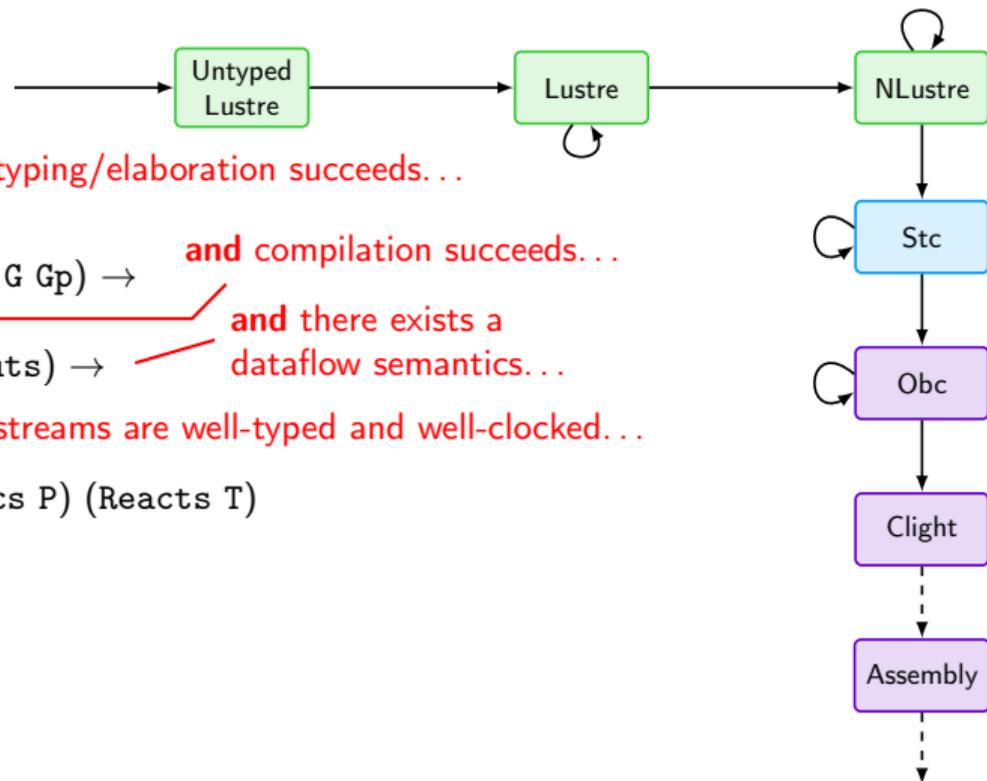
$\forall D G Gp P \text{ main ins outs,}$
 $\text{elab_declarations } D = \text{OK } (\text{exist } _ G Gp) \rightarrow$
 $\text{compile } D \text{ main} = \text{OK } P \rightarrow$
 $\text{sem_node } G \text{ main } (\text{pStr ins}) (\text{pStr outs}) \rightarrow$
 $\text{wt_ins } G \text{ main ins} \rightarrow$
 $\text{wc_ins } G \text{ main ins} \rightarrow$
 $\exists T, \text{program_behaves } (\text{Asm.semantics } P) (\text{Reacts } T)$
 $\wedge \text{bisim_IO } G \text{ main ins outs } T.$

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:

$\forall D G Gp P \text{ main ins outs,}$
 $\text{elab_declarations } D = \text{OK } (\text{exist } _ G Gp) \rightarrow$
 $\text{compile } D \text{ main} = \text{OK } P \rightarrow$
 $\text{sem_node } G \text{ main } (\text{pStr ins}) (\text{pStr outs}) \rightarrow$
 $\text{wt_ins } G \text{ main ins} \rightarrow$
 $\text{wc_ins } G \text{ main ins} \rightarrow$
 $\exists T, \text{program_behaves } (\text{Asm.semantics } P) (\text{Reacts } T)$
 $\wedge \text{bisim_IO } G \text{ main ins outs } T.$

if typing/elaboration succeeds...

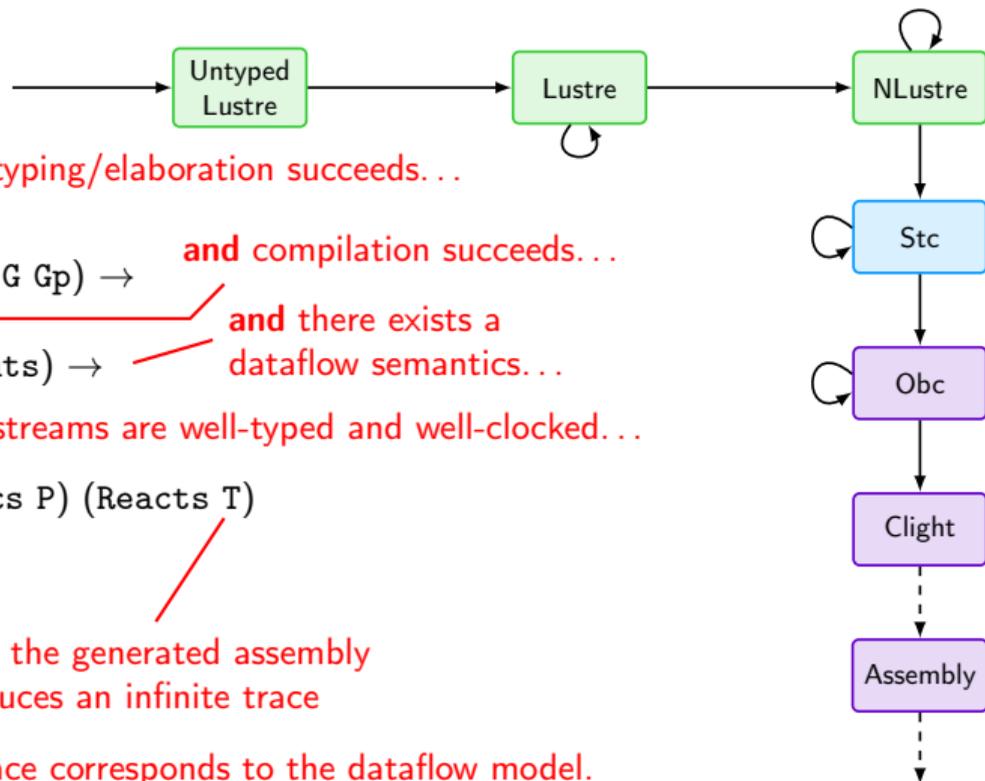
and compilation succeeds...

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.



Dataflow relational semantics

$$\frac{G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk} \quad \forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H \vdash \text{blk}}{G \vdash f(xss) \Downarrow yss} \text{sem_node}$$

Dataflow relational semantics

$$\frac{G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk} \quad \forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H \vdash \text{blk}}{G \vdash f(xss) \Downarrow yss} \text{sem_node}$$

<i>pause</i>	F	F	F	...	F	F	...	T	...	F	...	F	...
<i>time</i>	0	0	50	...	750	0	...	150	...	350	...	500	...
<i>step</i>	T	F	F	...	T	F	...	F	...	F	...	T	...

Dataflow relational semantics

$$\frac{G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk} \quad \forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H \vdash \text{blk}}{G \vdash f(xss) \Downarrow yss} \text{sem_node}$$

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	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \quad G, H \vdash es \Downarrow vss}{G, H \vdash xs = es} \text{sem_equation}$$

Dataflow relational semantics

$$\frac{G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk} \quad \forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H \vdash \text{blk}}{G \vdash f(xss) \Downarrow yss} \text{sem_node}$$

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	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \quad G, H \vdash es \Downarrow vss}{G, H \vdash xs = es} \text{sem_equation}$$

$$\frac{G, H \vdash e \Downarrow [vs] \quad \forall i, G, (\text{when}^{C_i} \text{ vs } H) \vdash \text{blks}_i}{G, H \vdash \text{switch } e [C_i \text{ do } \text{blks}_i]^i \text{ end}} \text{sem_switch}$$

Dataflow relational semantics

$$\frac{G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk} \quad \forall i, H(x_i) \equiv xss_i \quad \forall j, H(y_j) \equiv yss_j \quad G, H \vdash \text{blk}}{G \vdash f(xss) \Downarrow yss} \text{sem_node}$$

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	...

$$\frac{\forall i, H(xs_i) \equiv vss_i \quad G, H \vdash es \Downarrow vss}{G, H \vdash xs = es} \text{sem_equation}$$

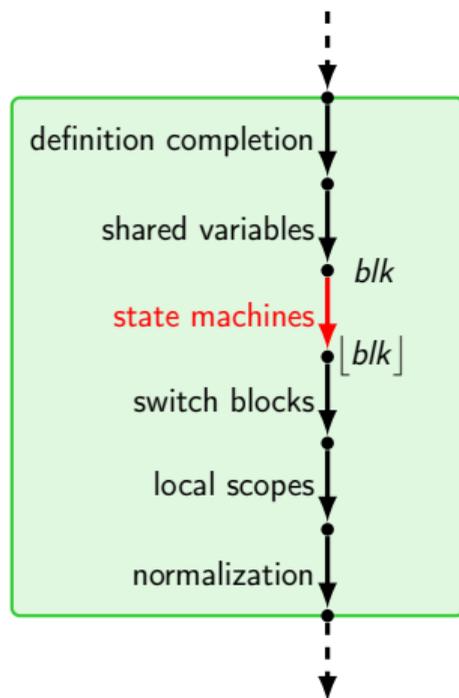
$$\frac{G, H \vdash e \Downarrow [vs] \quad \forall i, G, (\text{when}^{C_i} \text{ vs } H) \vdash \text{blks}_i}{G, H \vdash \text{switch } e [C_i \text{ do } \text{blks}_i]^i \text{ end}} \text{sem_switch}$$

- when for **switch** blocks
- mask for **reset** blocks
- select for state machines

Compilation correctness – state machines

Lemma (State machines correctness)

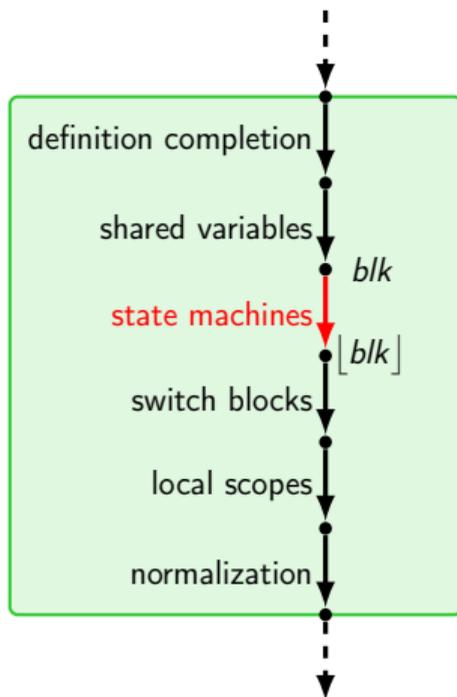
if $G, H \vdash blk$ then $G, H \vdash [blk]$



Compilation correctness – state machines

Lemma (State machines correctness)

if $G, H \vdash blk$ then $G, H \vdash [blk]$

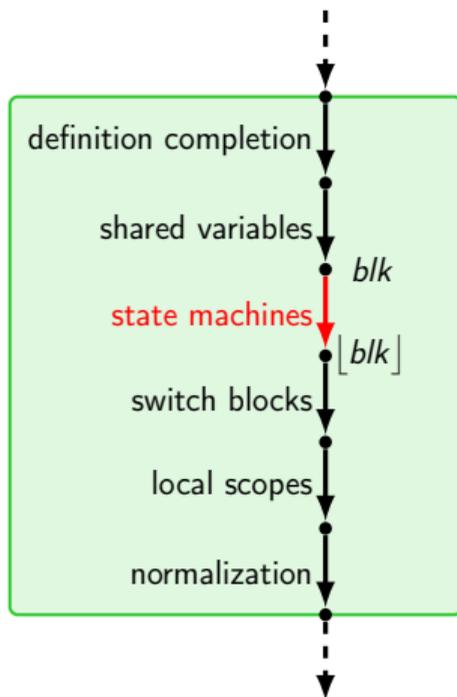


Works well:

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

Lemma (State machines correctness)

if $G, H \vdash blk$ then $G, H \vdash [blk]$



Works well:

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

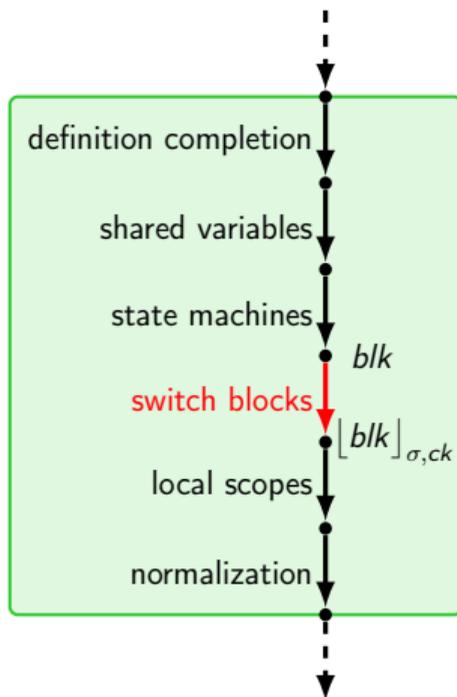
Works less well:

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

Compilation correctness – switch blocks

Lemma (Switch correctness)

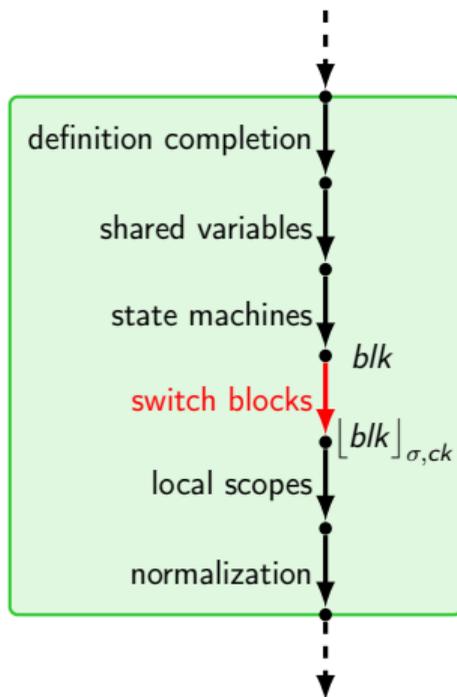
if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_{\sigma} H_2$ then $G, H_2 \vdash [blk]_{\sigma, ck}$



Compilation correctness – switch blocks

Lemma (Switch correctness)

if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_{\sigma} H_2$ then $G, H_2 \vdash [blk]_{\sigma, ck}$



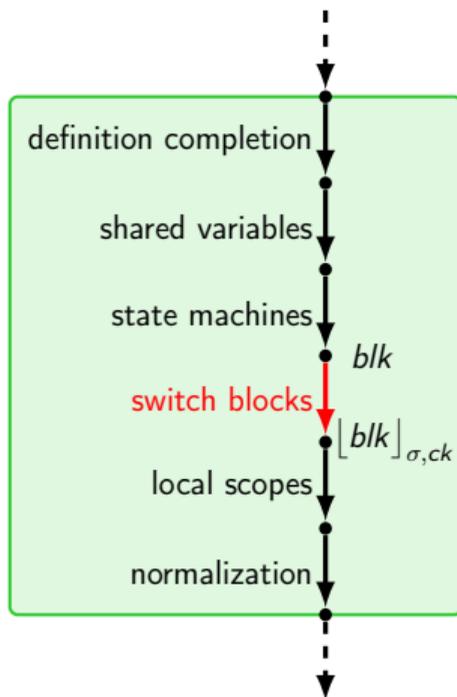
Works less well:

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

Compilation correctness – switch blocks

Lemma (Switch correctness)

if $G, H_1 \vdash blk$ and $H_1 \sqsubseteq_{\sigma} H_2$ then $G, H_2 \vdash [blk]_{\sigma, ck}$



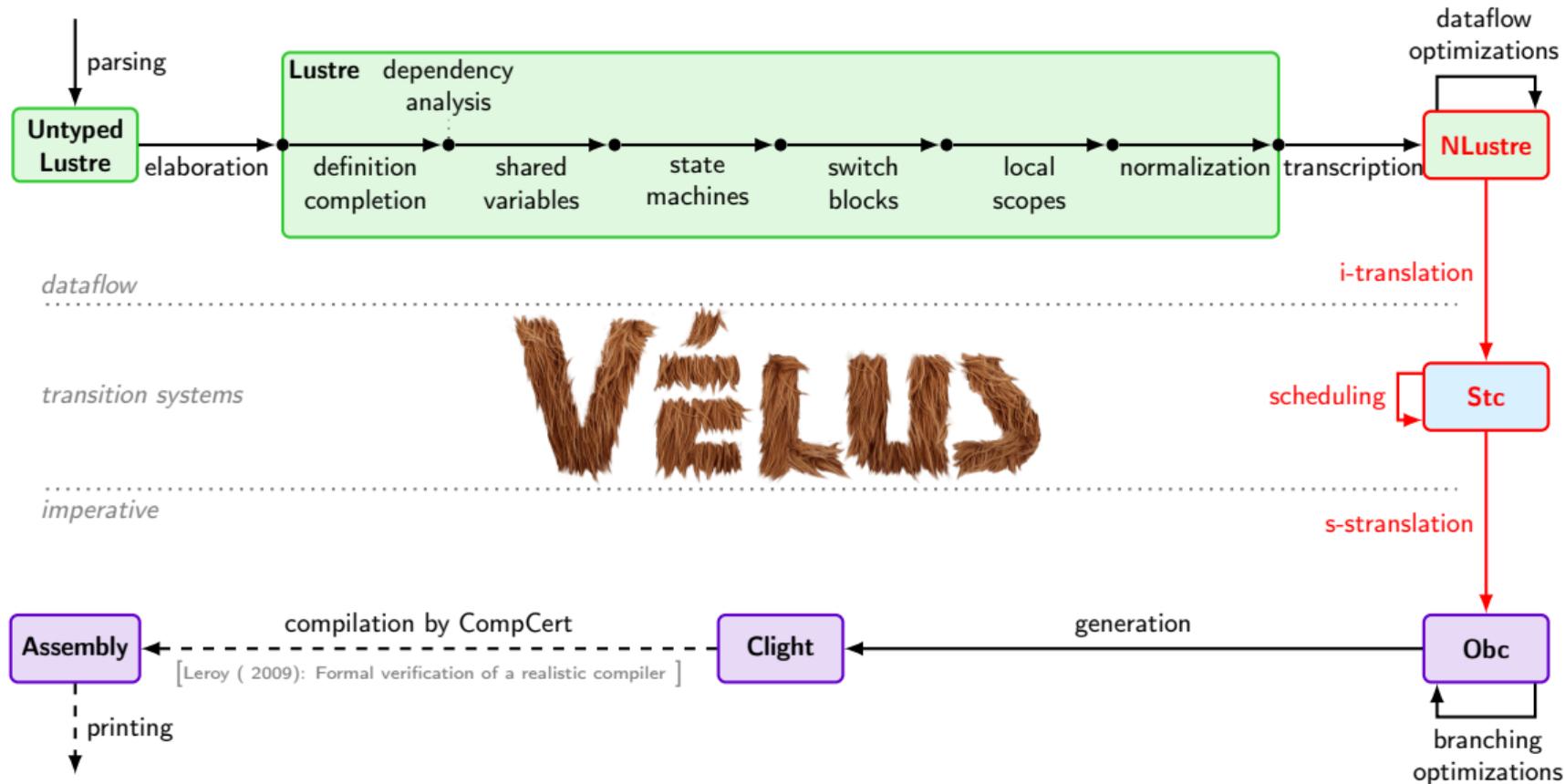
Works well:

- 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

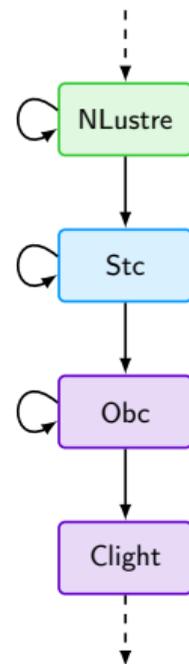
The Vélu Compiler



Generation of imperative code

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

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



Generation of imperative code

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

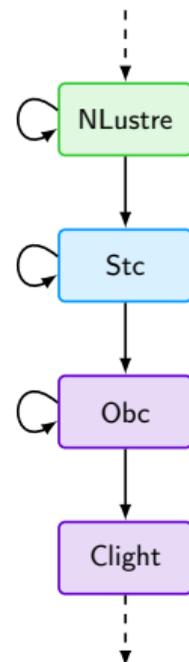
```
switch(st) { case Starting: resS = res; }
```

```
switch(st) {  
  case Starting:  
    if(resS) st.stepS = true;  
}
```

```
ena = true;  
switch(st) {  
  case Starting:  
    step = st.stepS;  
    break;  
  case Moving: ...  
}
```

```
switch(st) { case Starting: st.stepS = false; }
```

- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction



Generation of imperative code

```
resS = res when (st=Starting);
```

```
reset
```

```
  stepS = (true when (st=Starting)) fby (false when (st=Starting))
```

```
every resS;
```

```
ena = true;
```

```
step = merge st (Starting => stepS) (Moving => stepM);
```

```
switch(st) { case Starting: resS = res; }
```

```
switch(st) {
```

```
  case Starting:
```

```
    if(resS) st.stepS = true;
```

```
}
```

```
ena = true;
```

```
switch(st) {
```

```
  case Starting:
```

```
    step = st.stepS;
```

```
    break;
```

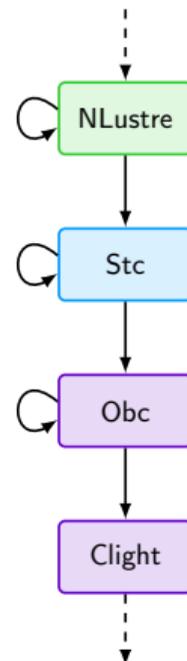
```
  case Moving: ...
```

```
}
```

```
switch(st) { case Starting: st.stepS = false; }
```

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

- Each controlled expression/reset produces a `switch` instruction



Generation of imperative code

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

```
switch(st) { case Starting: resS = res; }
```

```
switch(st) {
```

```
  case Starting:
```

```
    if(resS) st.stepS = true;
```

```
}
```

```
ena = true;
```

```
switch(st) {
```

```
  case Starting:
```

```
    step = st.stepS;
```

```
    break;
```

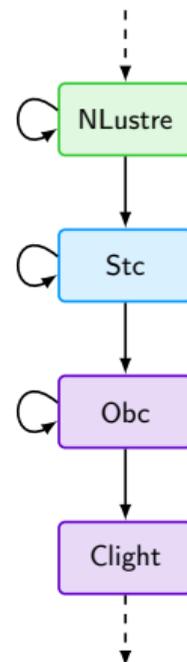
```
  case Moving: ...
```

```
}
```

```
switch(st) { case Starting: st.stepS = false; }
```

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

● Each controlled expression/reset produces a `switch` instruction



Generation of imperative code

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

```
switch(st) { case Starting: resS = res; }
```

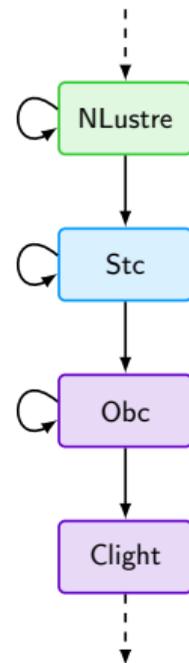
```
switch(st) {  
  case Starting:  
    if(resS) st.stepS = true;  
}
```

```
ena = true;
```

```
switch(st) {  
  case Starting:  
    step = st.stepS;  
    break;  
  case Moving: ...  
}
```

```
switch(st) { case Starting: st.stepS = false; }
```

- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction

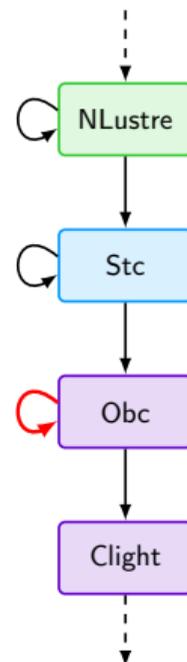


Generation of imperative code

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

```
switch(st) {  
  case Starting:  
    resS = res;  
    if(resS) st.stepS = true;  
}  
ena = true;  
switch(st) {  
  case Starting:  
    step = st.stepS;  
    st.stepS = false;  
    break;  
  case Moving: ...  
}
```

- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction

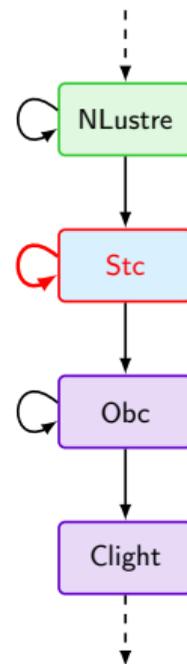


Generation of imperative code

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

```
switch(st) {  
  case Starting:  
    resS = res;  
    if(resS) st.stepS = true;  
}  
ena = true;  
switch(st) {  
  case Starting:  
    step = st.stepS;  
    st.stepS = false;  
    break;  
  case Moving: ...  
}
```

- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction

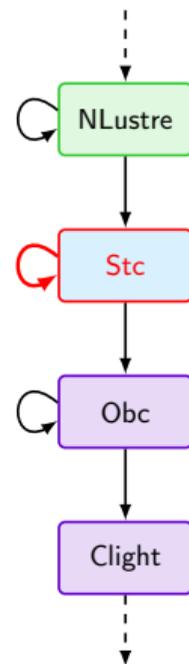


Generation of imperative code

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

```
ena = true;  
switch(st) {  
  case Starting:  
    resS = res;  
    if(resS) st.stepS = true;  
    step = st.stepS;  
    st.stepS = false;  
    break;  
  case Moving: ...  
}
```

- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction
- Quality of fusion depends on the scheduling

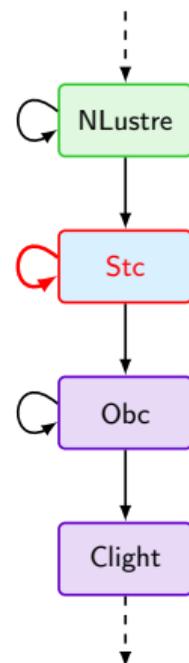


Generation of imperative code

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

```
ena = true;  
switch(st) {  
  case Starting:  
    resS = res;  
    if(resS) st.stepS = true;  
    step = st.stepS;  
    st.stepS = false;  
    break;  
  case Moving: ...  
}
```

- [Biernacki, Colaço, Hamon, and Pouzet (LCTES 2008): Clock-directed modular code generation for synchronous data-flow languages]
- Each controlled expression/reset produces a `switch` instruction
- Quality of fusion depends on the scheduling
- Extensions of `Stc`: reset on state variables, multiple reset conditions



	<i>Vélus</i>	<i>Hept+CompCert</i>	<i>Hept+gcc</i>	<i>Hept+gcci</i>
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 (LNCS 2010): OTAWA: An Open Toolbox for Adaptive WCET Analysis] for an armv7

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

	<i>Vélus</i>	<i>Hept+CompCert</i>	<i>Hept+gcc</i>	<i>Hept+gcci</i>
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 (LNCS 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

	<i>Vélus</i>	<i>Hept+CompCert</i>	<i>Hept+gcc</i>	<i>Hept+gcc</i>
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 (LNCS 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 reused

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
 - `switch` blocks
 - `reset` blocks
 - state machines
- a verified implementation of an efficient compilation scheme for these blocks

Our contributions:

- a Coq-based semantics for the control blocks of Scade 6
 - `switch` blocks
 - `reset` blocks
 - state machines
- a verified implementation of an efficient compilation scheme for these blocks

<https://velus.inria.fr/emsoft2023>

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

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

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

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

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

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

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

Semantics – Hierarchical State Machines

$$\frac{\text{fby } sts_0 \text{ } sts_1 \equiv sts \quad \begin{array}{l} H, bs \vdash ck \Downarrow bs' \quad G, H, bs' \vdash_{\text{I}} \text{autinits} \Downarrow sts_0 \\ \forall i, \forall k, G, (\text{select}_0^{C_i, k} sts (H, bs)), C_i \vdash_{\text{W}} \text{autscope}_i \Downarrow (\text{select}_0^{C_i, k} sts sts_1) \end{array}}{G, H, bs \vdash \text{automaton initially autinits}^{ck} [\text{state } C_i \text{ autscope}_i]^i \text{ end}}$$

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

$$\frac{\begin{array}{l} H, bs \vdash ck \Downarrow bs' \quad \text{fby } (\text{const } bs' (C, F)) \text{ } sts_1 \equiv sts \\ \forall i, \forall k, G, (\text{select}_0^{C_i, k} sts (H, bs)), C_i \vdash_{\text{TR}} \text{trans}_i \Downarrow (\text{select}_0^{C_i, k} sts sts_1) \\ \forall i, \forall k, G, (\text{select}_0^{C_i, k} sts_1 (H, bs)) \vdash \text{blks}_i \end{array}}{G, H, bs \vdash \text{automaton initially } C^{ck} [\text{state } C_i \text{ do blks}_i \text{ unless trans}_i]^i \text{ end}}$$

Semantics – Transitions

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

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

$$\begin{array}{l} \text{first-of}_r^{\text{C}} (\text{T} \cdot bs) (st \cdot sts) \equiv \langle C, r \rangle \cdot \text{first-of}_r^{\text{C}} bs sts \\ \text{first-of}_r^{\text{C}} (\text{F} \cdot bs) (st \cdot sts) \equiv st \cdot \text{first-of}_r^{\text{C}} bs sts \end{array}$$

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

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

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

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

$$\frac{\forall x, x \in \text{dom}(H') \iff x \in \text{locs} \quad \forall x e, (\text{last } x = e) \in \text{locs} \implies G, H + H', bs \vdash_{\perp} \text{last } x = e \quad G, H + H', bs \vdash \text{blks}}{G, H, bs \vdash \text{var locs let blks tel}}$$

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

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