Optimizing memory representation of state machines

Grégoire Bussone

ENS-PSL, Inria Paris, Parkas team

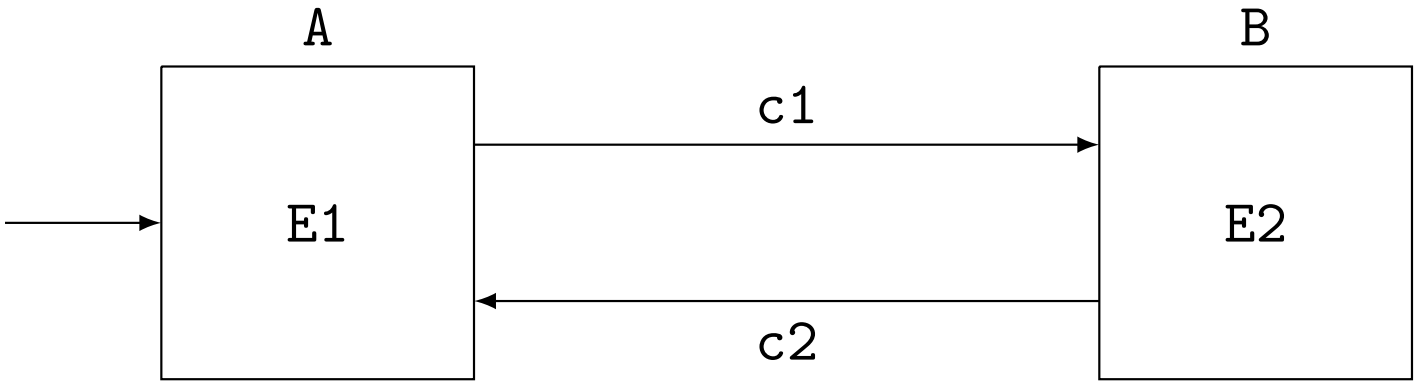
November 24, 2025

Table of contents

Table of contents

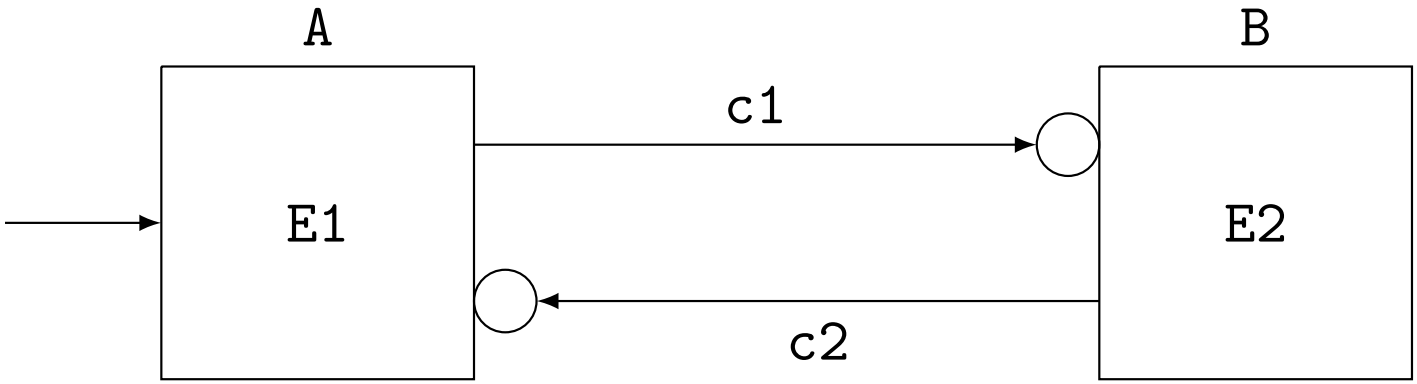
Table of contents

High-level construct for Lustre/Velus



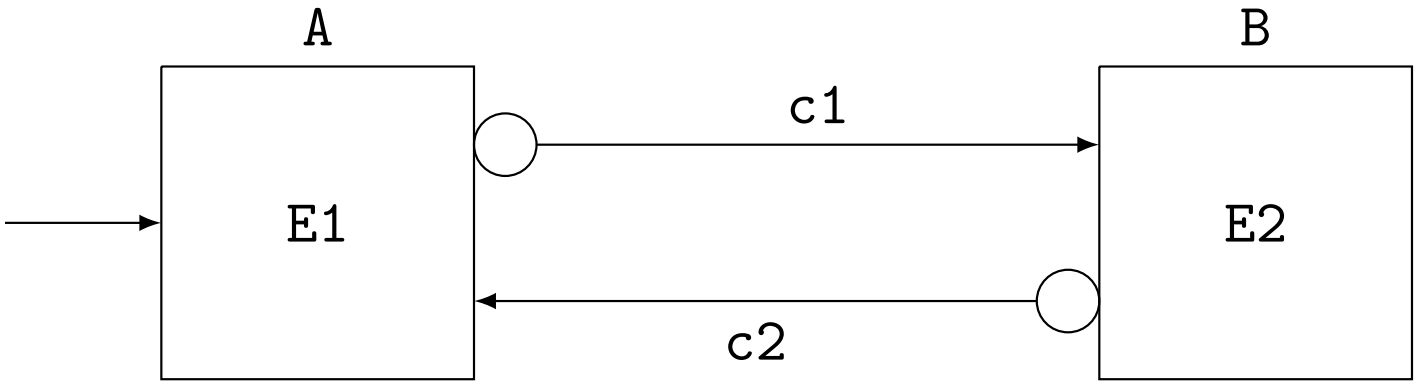

```
automaton initially A
  state A do E1
  unless/until
  | c1 then/continue B
  state B do E2
  unless/until
  | c2 then/continue A
```

High-level construct for Lustre/Velus



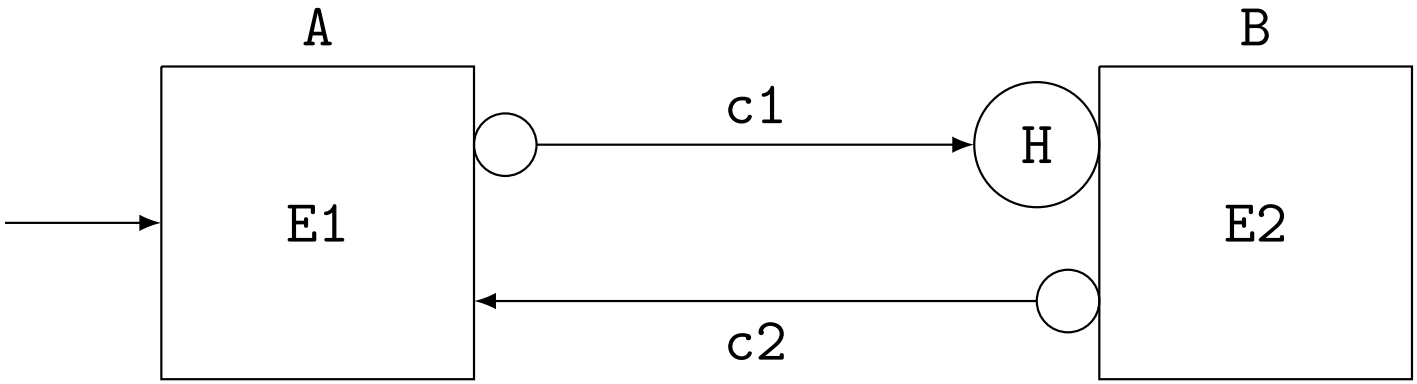

```
automaton initially A
  state A do E1
  until
  | c1 then/continue B
  state B do E2
  until
  | c2 then/continue A
```

High-level construct for Lustre/Velus




```
automaton initially A
  state A do E1
  unless
  | c1 then/continue B
  state B do E2
  unless
  | c2 then/continue A
```

High-level construct for Lustre/Velus




```
automaton initially A
  state A do E1
  unless
  | c1 continue B
  state B do E2
  unless
  | c2 then A
```

Table of contents

Compilation into switches

```
automaton initially A
  state A do E1 unless | c1 continue B
  state B do E2 unless | c2 then A

(s, r) = (A, false) fby (ns, nr)
switch s
| A do reset
    (ns, nr) = if c1 then (B, false) else (A, false)
  every r
| B do reset
    (ns, nr) = if c2 then (A, true) else (B, false)
  every r
switch ns
| A do reset E1 every nr
| B do reset E2 every nr
```

Compilation to imperative code directly

```
automaton initially A
  state A do E1 unless | c1 continue B
  state B do E2 unless | c2 then A

switch (self.s)
  case A:
    if (self.r)              self.c1.reset();
    if (self.c1.step(...)) self.r = false; self.s = B;
    else                     self.r = false; self.s = A;
  case B: ...
switch (self.s)
  case A:
    if (self.r)              self.E1.reset();
    ... = self.E1.step(...);
  case B: ...
```

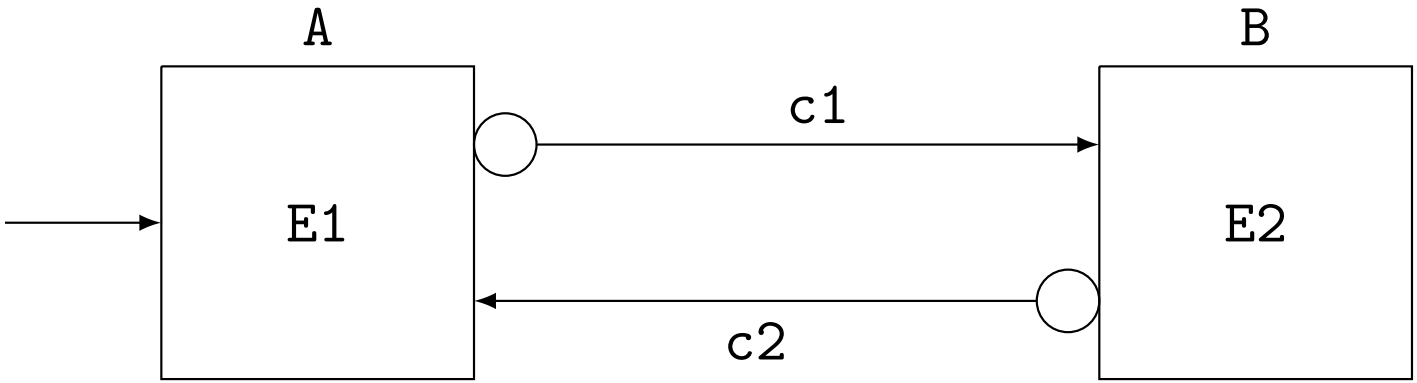
Compilation to imperative code directly

```
automaton initially A
  state A do E1 unless | c1 continue B
  state B do E2 unless | c2 then A

switch (self.s)
  case A:
    if (self.c1.step(...)) self.s = B;
    else                   self.s = A;
  case B:
    if (self.c2.step(...))
      self.c1.reset(); self.E1.reset(); self.s = A;
    else                   self.s = B;
switch (self.s)
  case A: ... = self.E1.step(...);
  case B: ... = self.E2.step(...);
```

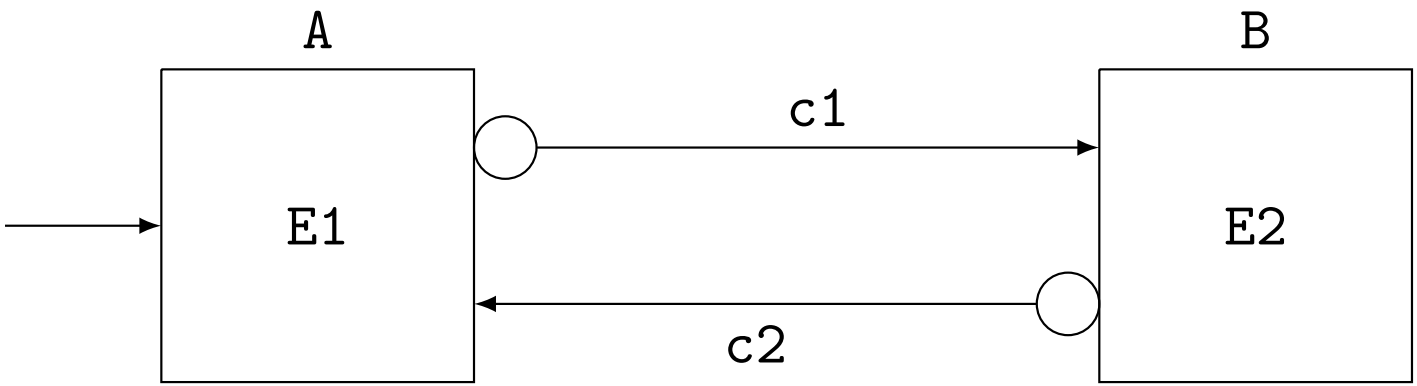
Table of contents

All transitions are by reset



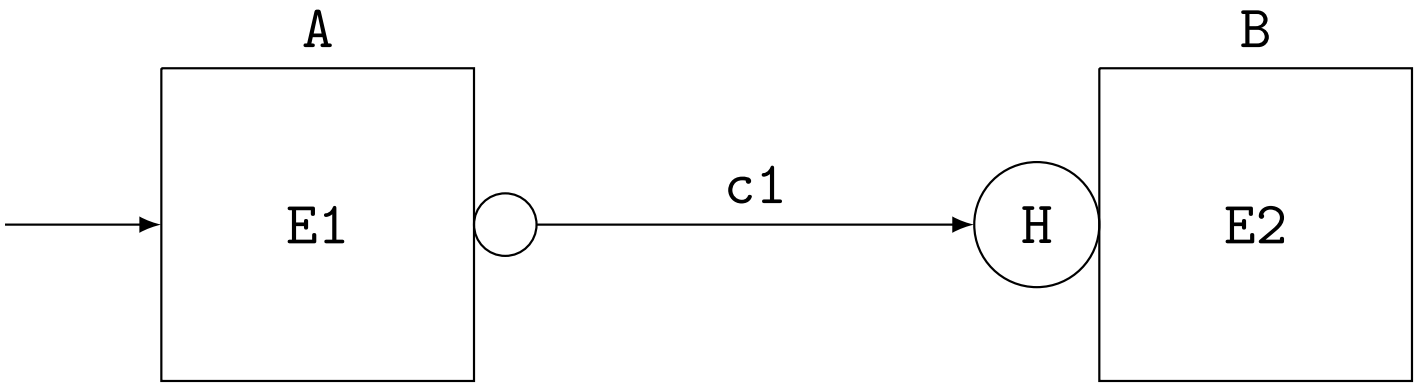

```
switch (self.s)
  case A:
    if (self.c1.step(...))
      self.c2.reset(); self.E2.reset(); self.s = B;
    else                              self.s = A;
  case B: ...
switch (self.s)
  case A: ... = self.E1.step(...);
  case B: ... = self.E2.step(...);
```

All transitions are by reset



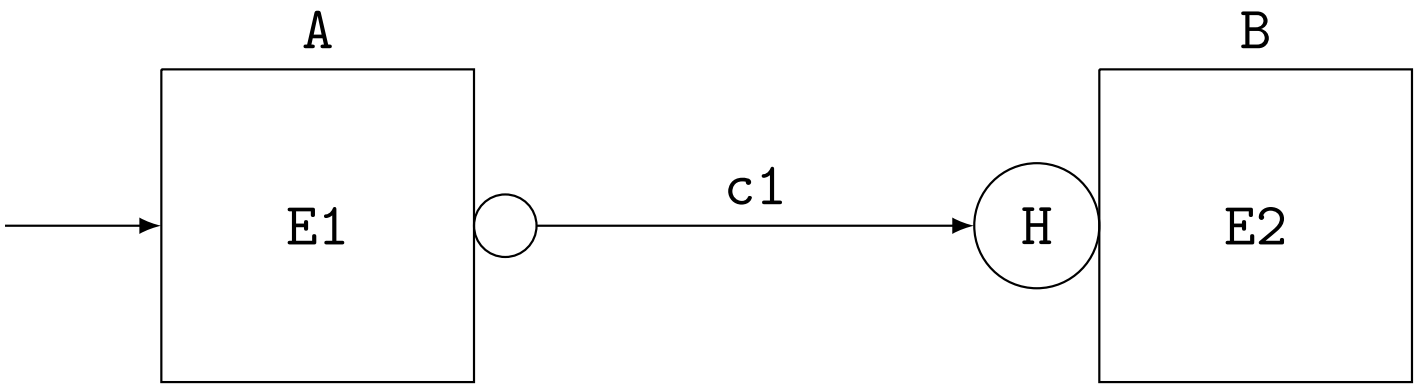

```
switch (self.s)
  case A:
    if (self.u1.c1.step(...))
      self.u2.c2.reset(); self.u2.E2.reset(); self.s = B;
    else                                      self.s = A;
  case B: ...
switch (self.s)
  case A: ... = self.u1.E1.step(...);
  case B: ... = self.u2.E2.step(...);
```

Strongly connected components




```
switch (self.s)
  case A:
    if (self.c1.step(...)) self.s = B;
    else                   self.s = A;
  case B: self.s = B;
switch (self.s)
  case A: ... = self.E1.step(...);
  case B: ... = self.E2.step(...);
```

Strongly connected components

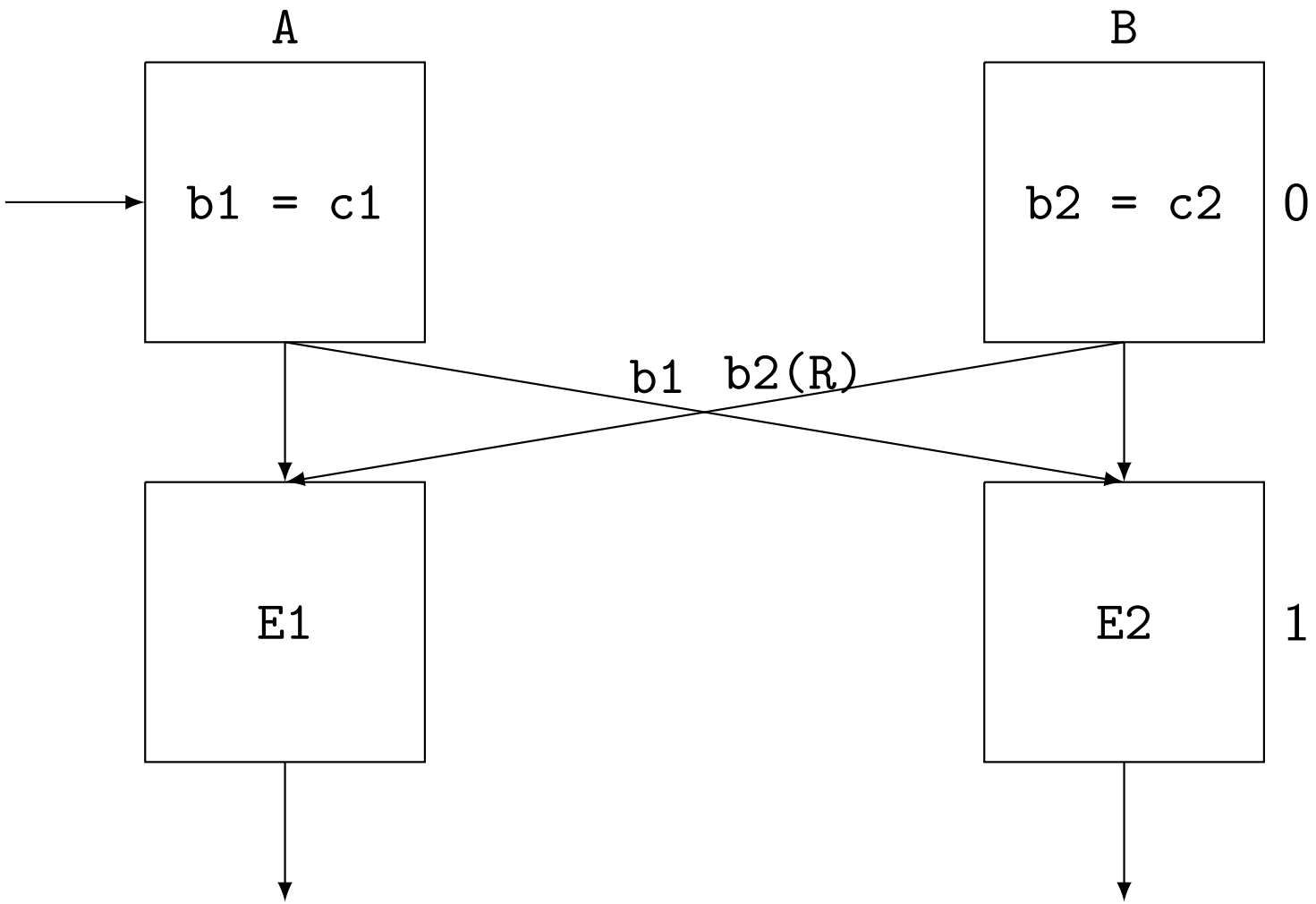



```
switch (self.s)
  case A:
    if (self.u1.c1.step(...))
      self.u2.c2.reset(); self.u2.E2.reset(); self.s = B;
    else                                self.s = A;
  case B: self.s = B;
switch (self.s)
  case A: ... = self.u1.E1.step(...);
  case B: ... = self.u2.E2.step(...);
```

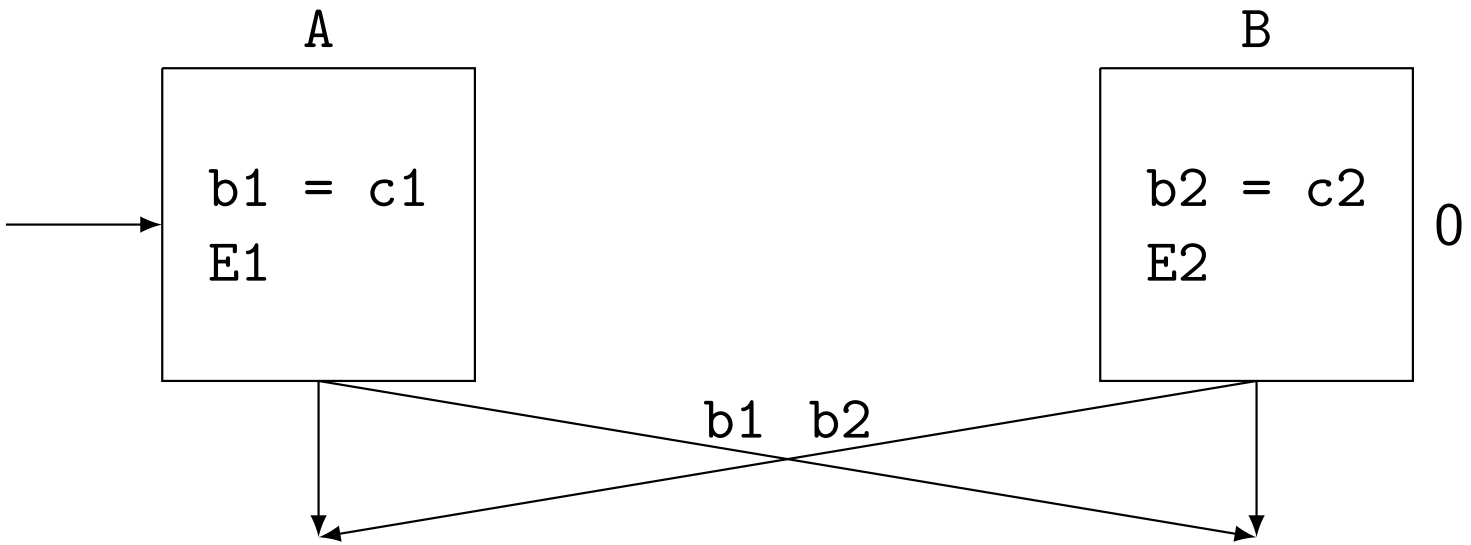
Table of contents

Table of contents

Unified internal representation (strong transitions)



Unified internal representation (weak transitions)



Unified internal representation (dynamic initialisation)

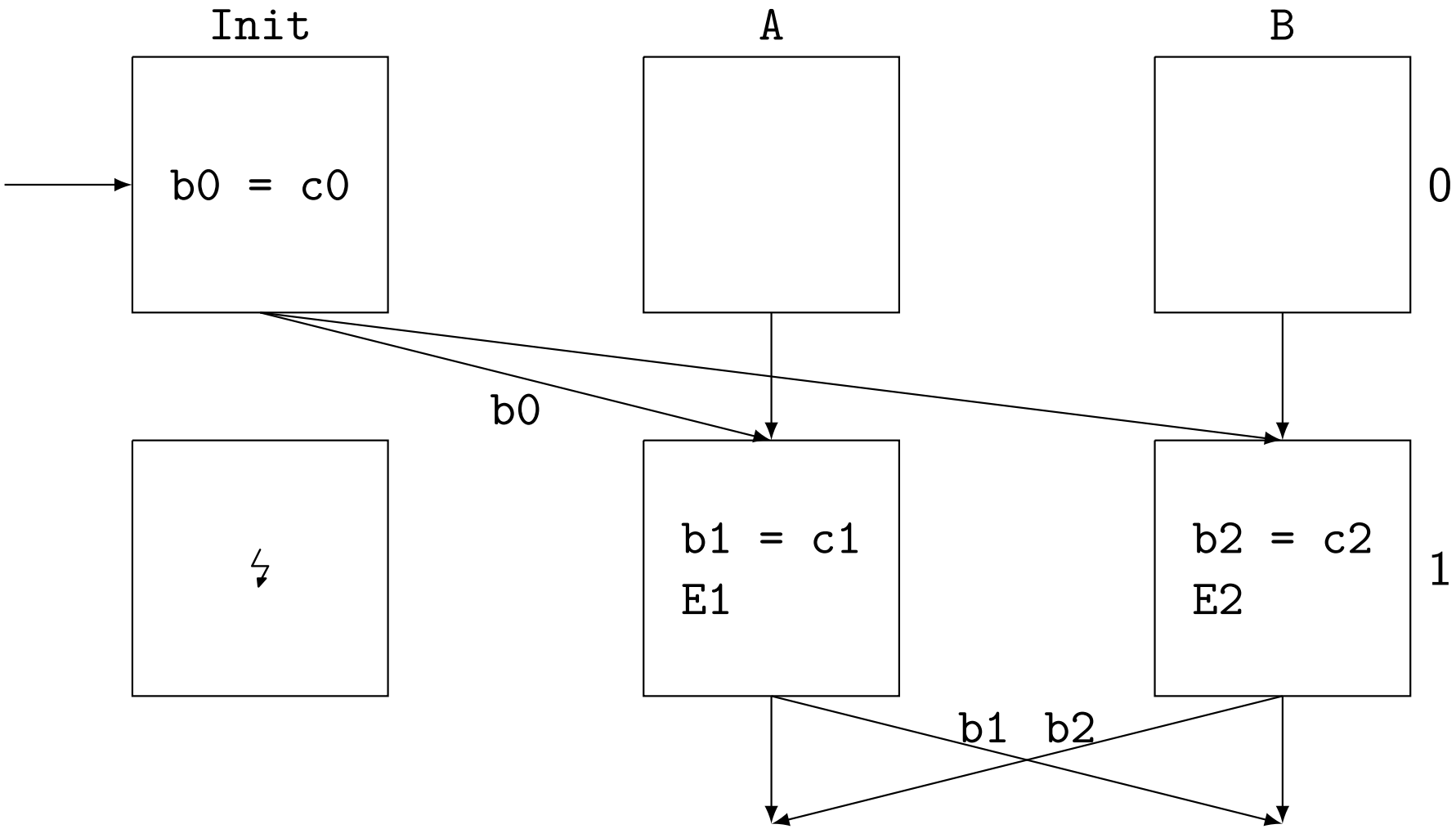
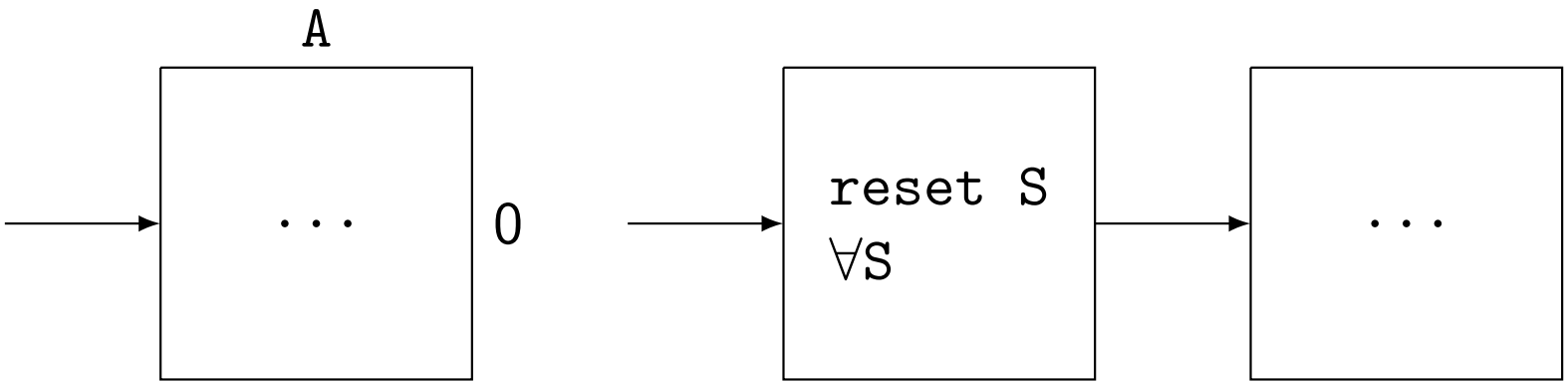
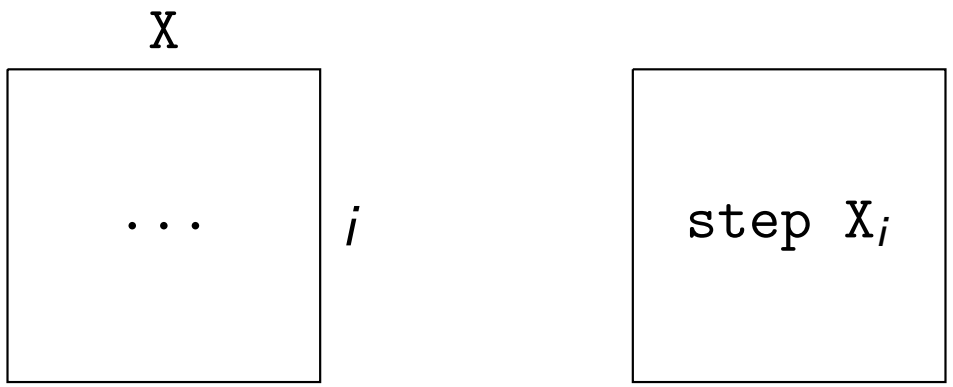


Table of contents

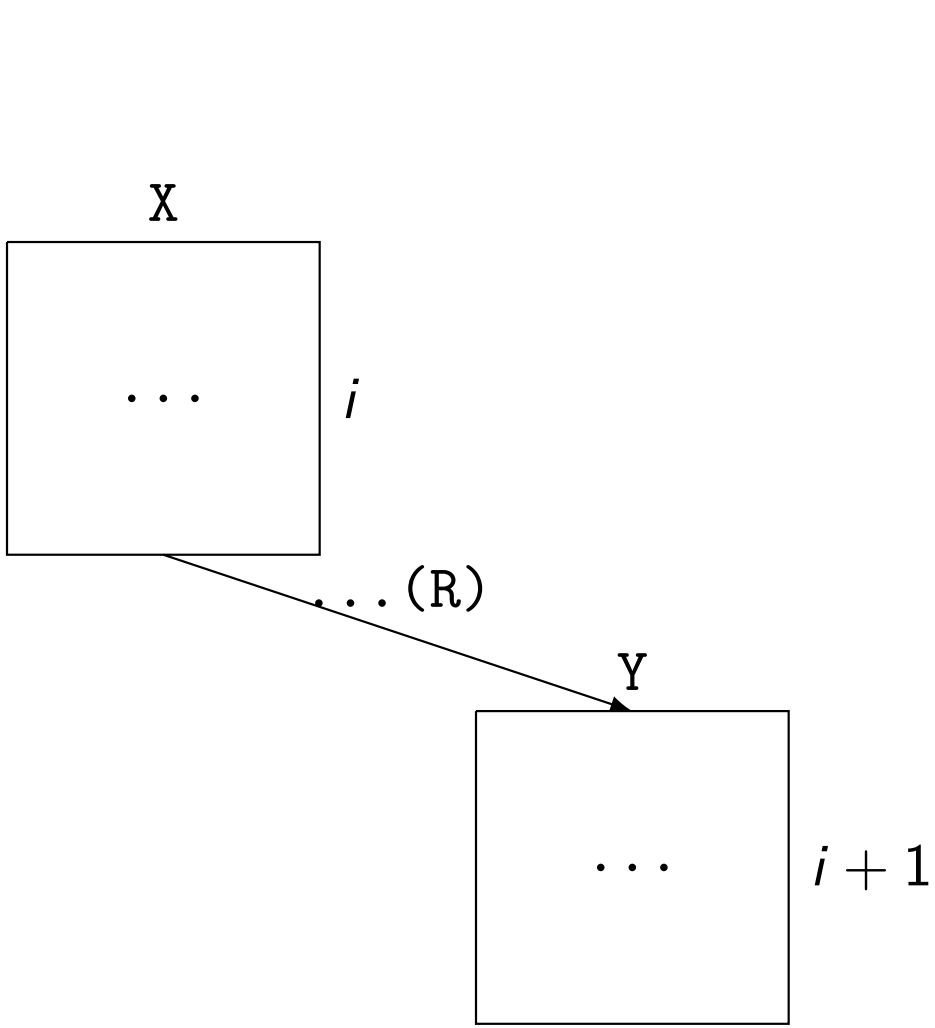
Translation (initialization)

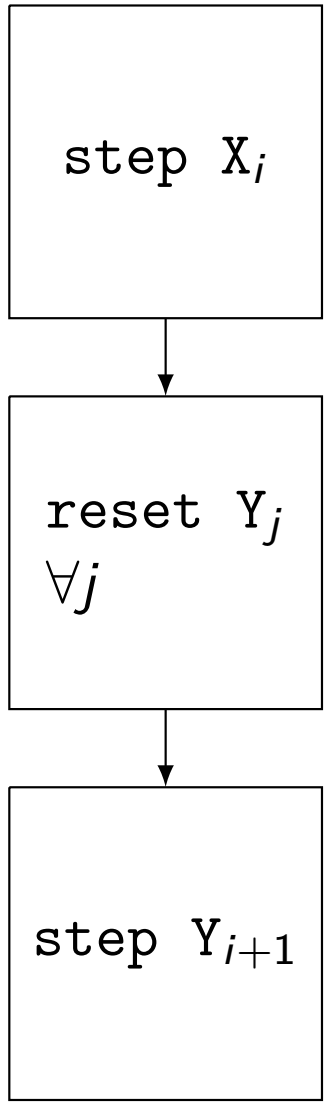


Translation (step)

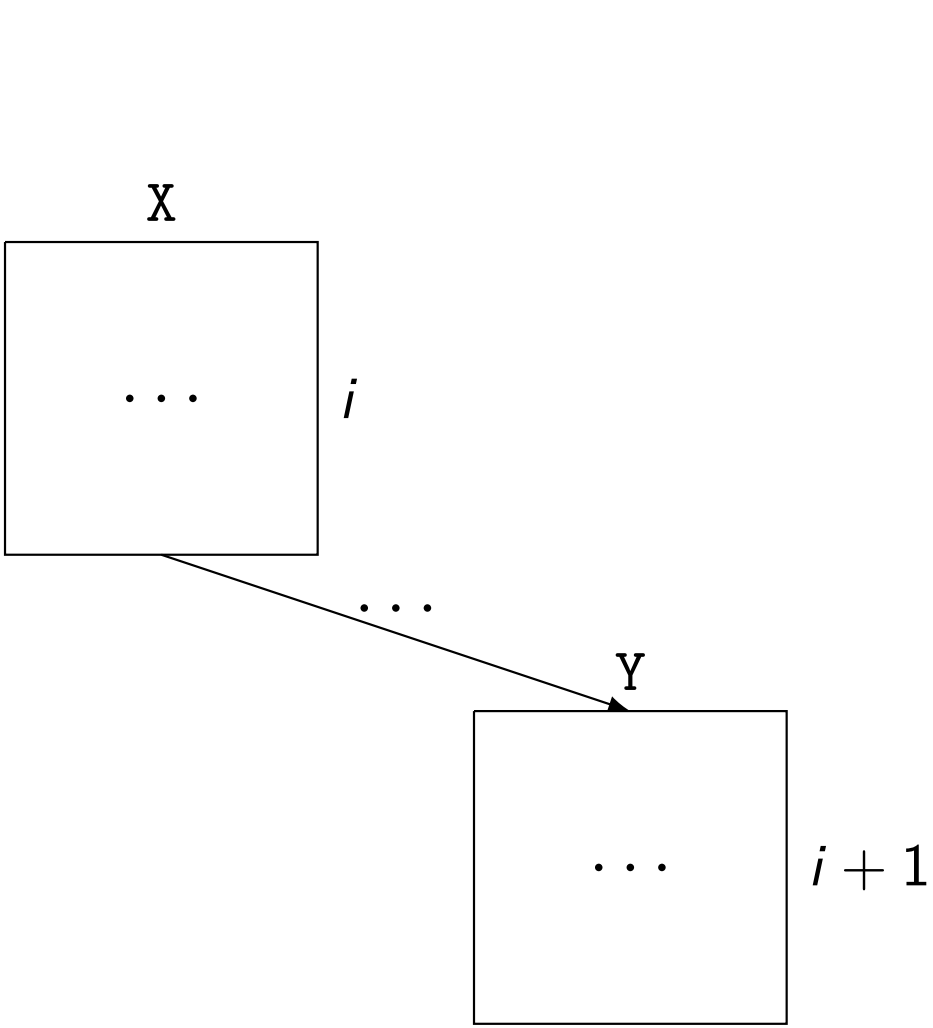


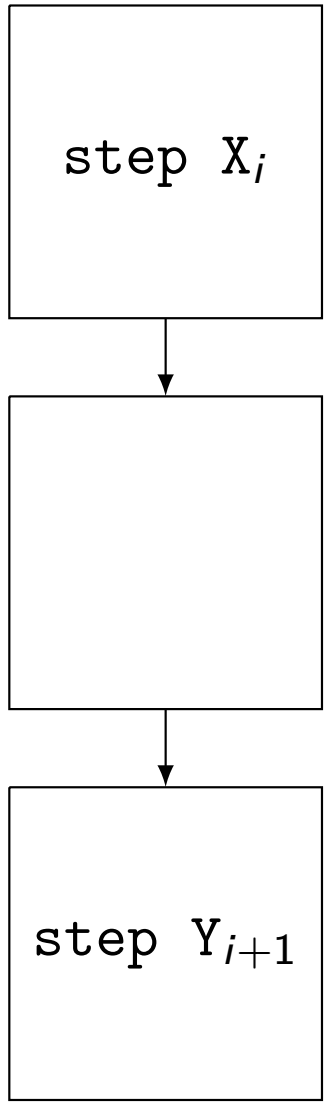
Translation (transition)



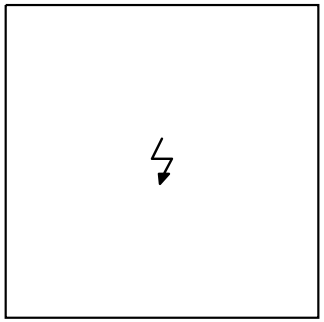


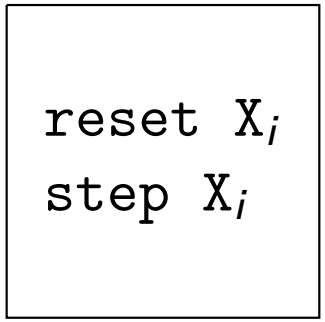
Translation (transition)

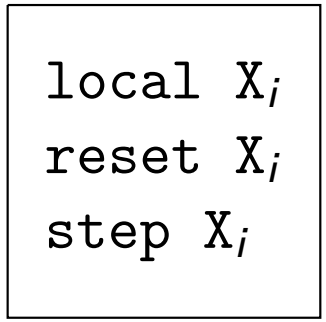


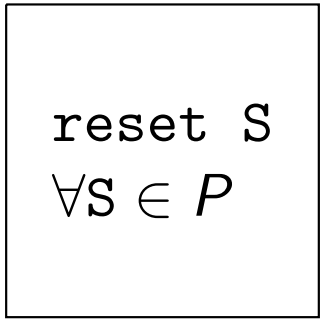


Instructions









Layout

$$
\begin{array}{rcl}
L & ::= & \mathtt{x}_i \\
 & & \text{(a single state)} \\
 & | & \prod L \\
 & & \text{(a list of layouts with distinct memory locations)} \\
 & | & \sum L \\
 & & \text{(a list of layouts sharing the same memory location)}
\end{array}
$$

At the beginning, we have $\prod_{\mathtt{s}} \mathtt{S}$

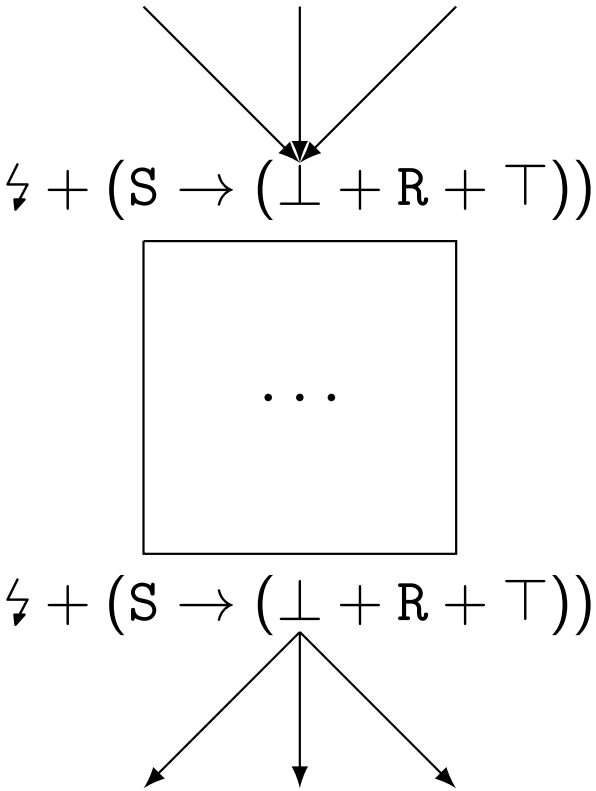
Table of contents

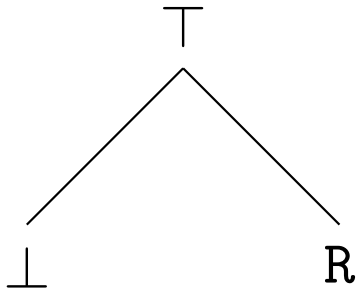
Interferences

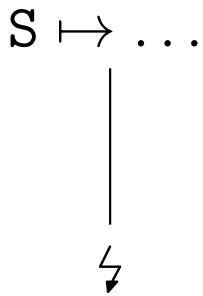
$\mathtt{X}_i$ and $\mathtt{Y}_j$ interfere in L if they are both stored in L, are different and

- $L = \prod_k L_k$ and they interfere in some L_k
- or $L = \sum_k L_k$ and they interfere in some L_k
- or $L = \sum_k L_k$ and they are stored in different L_k

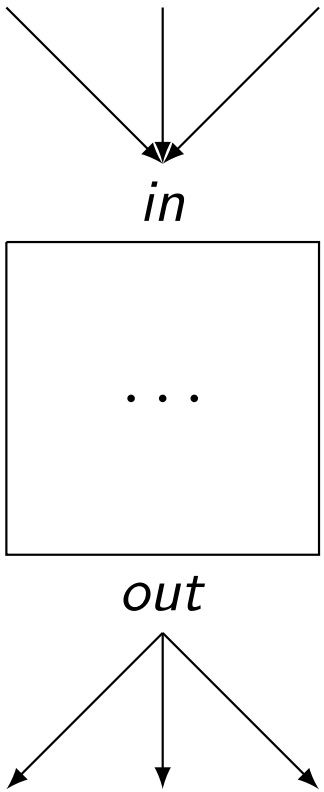
Value analysis

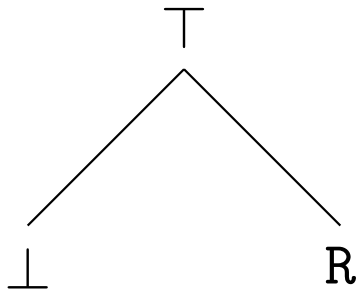


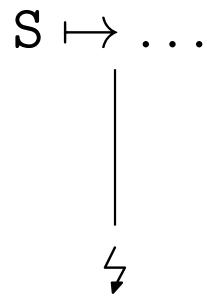




Value analysis

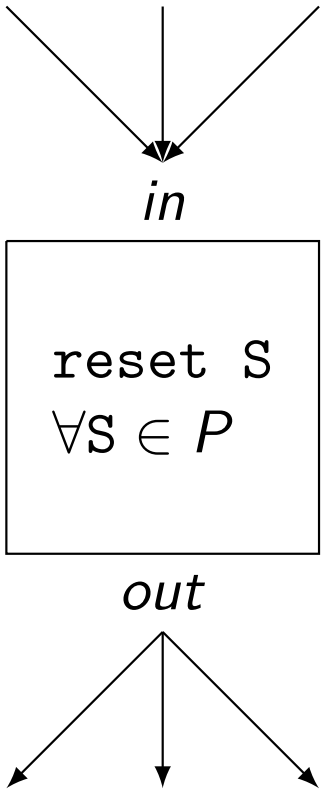


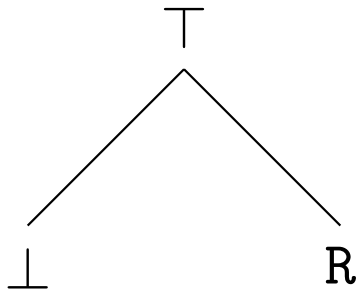


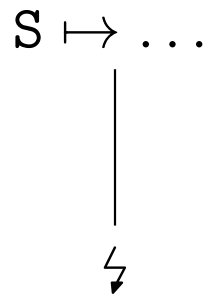


$$in = \begin{cases} \mathtt{S} \mapsto \bot & \text{if it is the initial state} \\ \bigsqcup_{p \in preds} p.out & \text{otherwise} \end{cases}$$

Value analysis

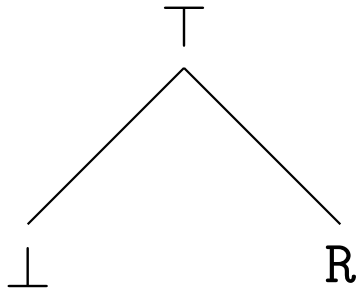


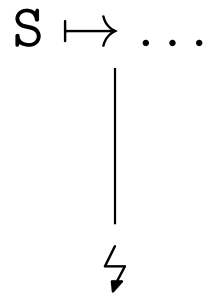




$$
\begin{array}{lcll}
out & = & \lightning & \text{if } in = \lightning \\
out(\mathtt{S}) & = & \mathtt{R} & \text{if } \mathtt{S} \in \mathtt{P} \\
 & & \bot & \text{if } \mathtt{S} \text{ interferes with a state in } \mathtt{P} \\
 & & in(\mathtt{S}) & \text{otherwise}
\end{array}
$$

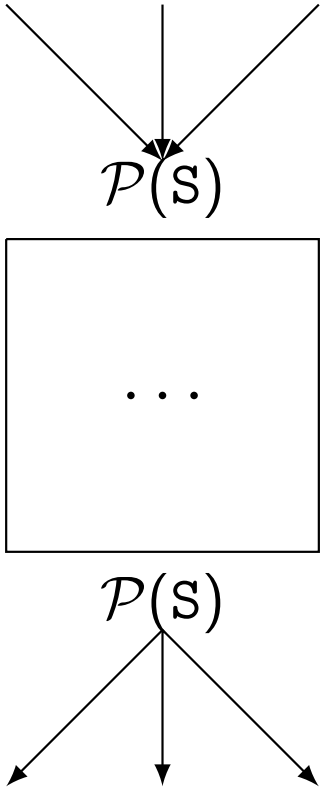
Value analysis





$$
\begin{array}{lcll}
out & = & \lightning & \text{if } in = \lightning \\
out & = & \lightning & \text{if } in(\mathtt{S}) = \bot \\
out(\mathtt{S'}) & = & \top & \text{if } \mathtt{S} = \mathtt{S'} \\
 & & \bot & \text{if } \mathtt{S} \text{ and } \mathtt{S'} \text{ interfere} \\
 & & in(\mathtt{S'}) & \text{otherwise}
\end{array}
$$

Lifetime analysis



Lifetime analysis

$$
\textit{use} \quad = \quad \begin{cases} \emptyset & \text{if } \texttt{op} = \texttt{reset S } \forall \texttt{S} \in \texttt{P} \\ \{\texttt{S}\} & \text{if } \texttt{op} = \texttt{step S} \\ \emptyset & \text{if } \texttt{op} = \texttt{reset S step S} \end{cases}
$$

Lifetime analysis

$$
\begin{array}{rcll}
\mathit{def} & = & \mathtt{P} & \text{if } \mathtt{op} = \mathtt{reset\ S}\ \forall \mathtt{S} \in \mathtt{P} \\
 & & \{\mathtt{S}\} & \text{if } \mathtt{op} = \mathtt{step\ S} \\
 & & \{\mathtt{S}\} & \text{if } \mathtt{op} = \mathtt{reset\ S\ step\ S}
\end{array}
$$

Lifetime analysis

$$
\begin{aligned}
in &= use \cup (out \setminus def) \\
out &= \bigcup_{s \in succs} s.in
\end{aligned}
$$

Add resets

Before any instruction, if the value analysis says that a state will be already reset (and that every state which interferes with it will be already corrupted), we can safely reset again this state during that instruction.

Remove resets

After a reset instruction, if the lifetime analysis says that one of the reset states is already dead, we can safely remove the reset of that state.

Remove unreachable states

A state is unreachable if it is never reset and the value analysis says that its step is unreachable. An unreachable state is removed from the layout and its step is replaced by an unreachable instruction.

Make states local

If a state appears only in a `reset step` and is dead after that, we can remove the state from the layout and replace its `reset step` by a `local reset step`.

Change layout

Two states need to have distinct memory locations if they are different and there is an instruction defining one while the other is alive after. We can replace a layout by another valid one.

Final pipeline

- remove resets
- remove unreachable states
- add resets
- remove resets
- make states local
- change layout

The rest of the compilation

- rewrite this imperative code into an (even more general) automaton form
- typing: every line should define the same variables (a sufficient condition would be that every path from top to bottom defines the same variables)
- causality and scheduling: an automaton is seen as an atomic equation
- code generation: a series of switches

Possible improvements

- find a way to express sharing in a compilation scheme à la Velus
- refine the static analyses to take values into account
- find good layout heuristics, or prove that simple ones are sufficient to achieve optimality

Thank you for your attention!
Any questions?