Proving a Lustre compiler

Lélio Brun[1,2] – PARKAS Team

Timothy Bourke[1,2] Pierre-Évariste Dagand[4,3,1] Xavier Leroy[1]
Marc Pouzet[4,2,1] Lionel Rieg[5]

[1]Inria Paris

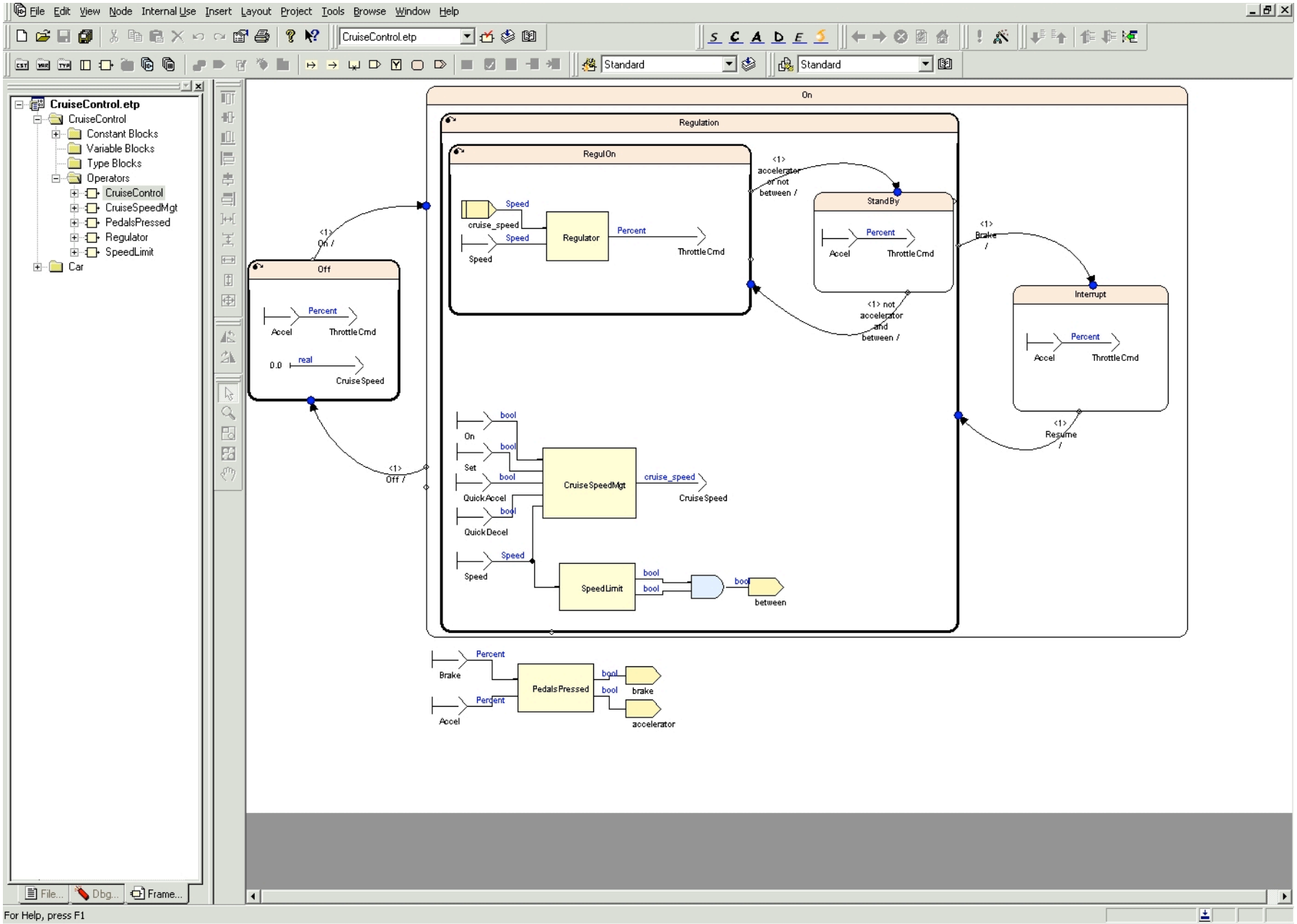
[2]DI ENS

[3]CNRS

[4]UPMC

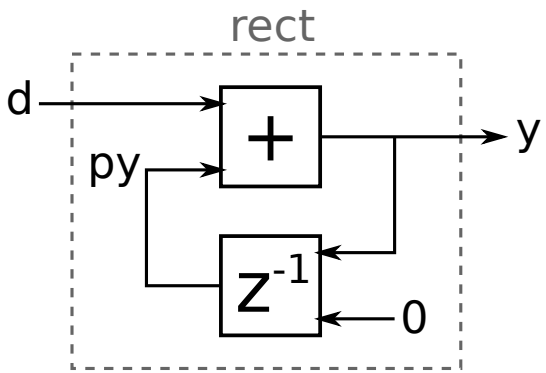
[5]Yale University

Inria's Junior Seminar — May 16, 2017



Screenshot from ANSYS/Esterel Techologies SCADE Suite

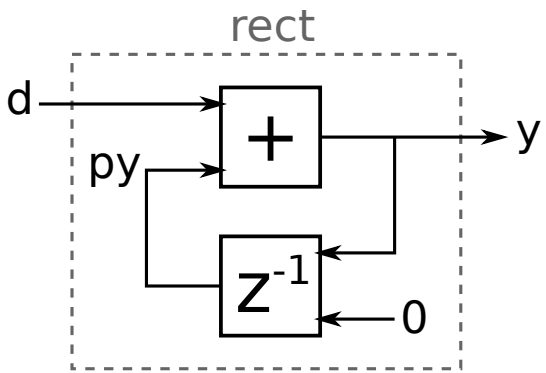
Lustre[1]: example




```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

[1] Caspi et al. (1987): "LUSTRE: A declarative language for programming synchronous systems"

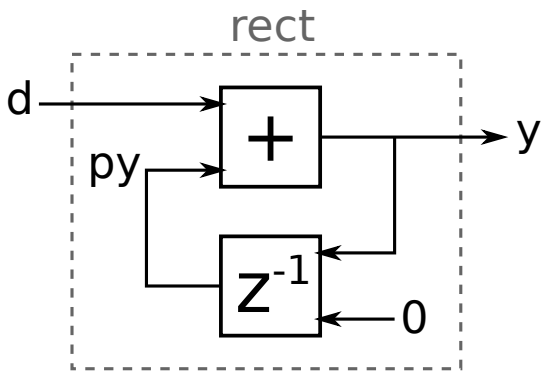
Lustre: example




```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

t	0
d	3
py	0
y	3

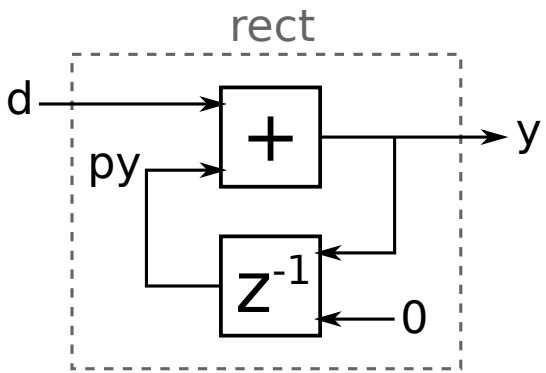
Lustre: example




```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

t	0	1
d	3	1
py	0	3
y	3	4

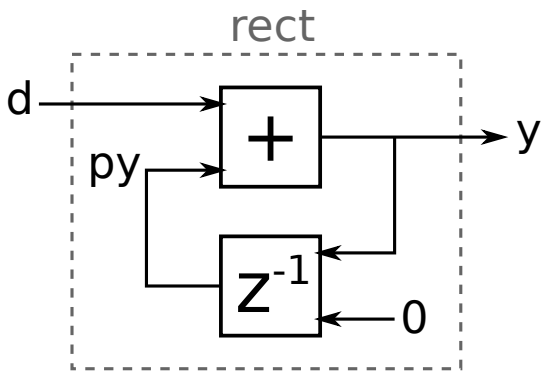
Lustre: example




```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

t	0	1	2
d	3	1	5
py	0	3	4
y	3	4	9

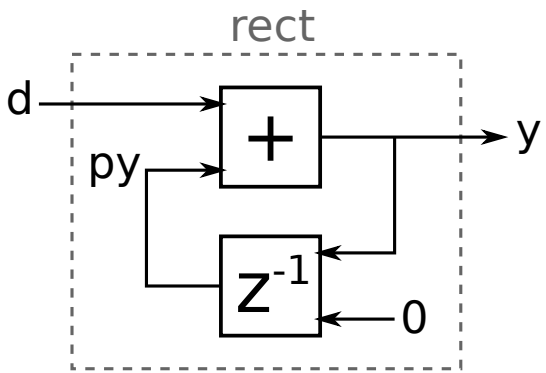
Lustre: example




```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

t	0	1	2	4
d	3	1	5	2
py	0	3	4	9
y	3	4	9	11

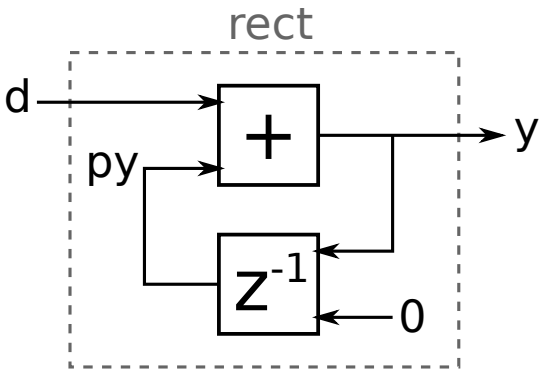
Lustre: example



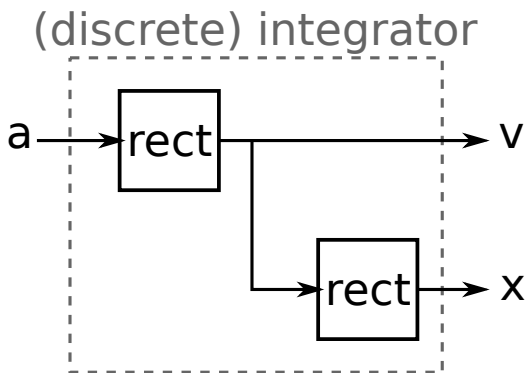

```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

t	0	1	2	4	...
d	3	1	5	2	...
py	0	3	4	9	...
y	3	4	9	11	...

Lustre: example

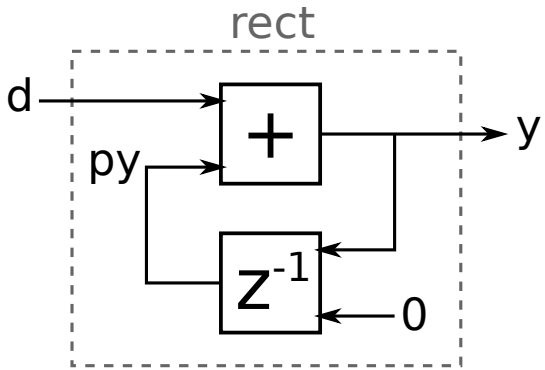



```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

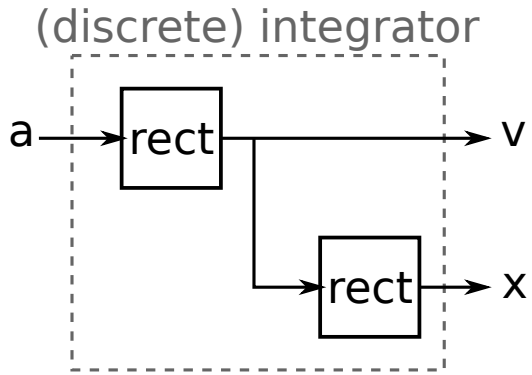



```
node integrator(a: int) returns (v, x: int)
let
  v = rect(a);
  x = rect(v);
tel
```

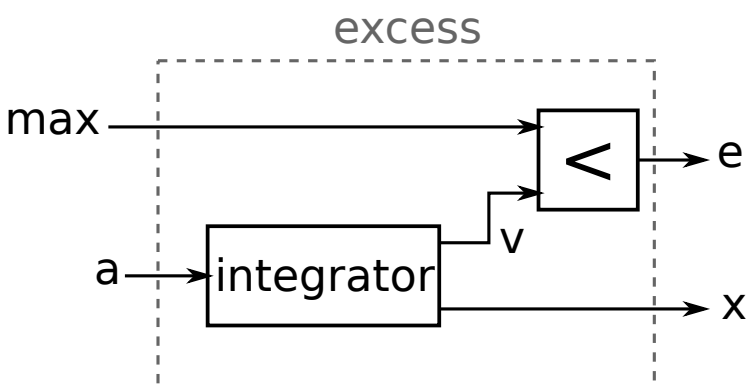
Lustre: example




```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```




```
node integrator(a: int) returns (v, x: int)
let
  v = rect(a);
  x = rect(v);
tel
```




```
node excess(max, a: int)
  returns (e: bool; x: int)
  var v: int;
let
  (v, x) = integrator(a);
  e = max < v;
tel
```

Context

Critical aspect

- specification norms (DO-178B), industrial certification

Context

Critical aspect

- specification norms (DO-178B), industrial certification
- formal verification, mechanized proofs, proof assistant (eg. Coq[1])

[1] The Coq Development Team (2016): *The Coq proof assistant reference manual*

Context

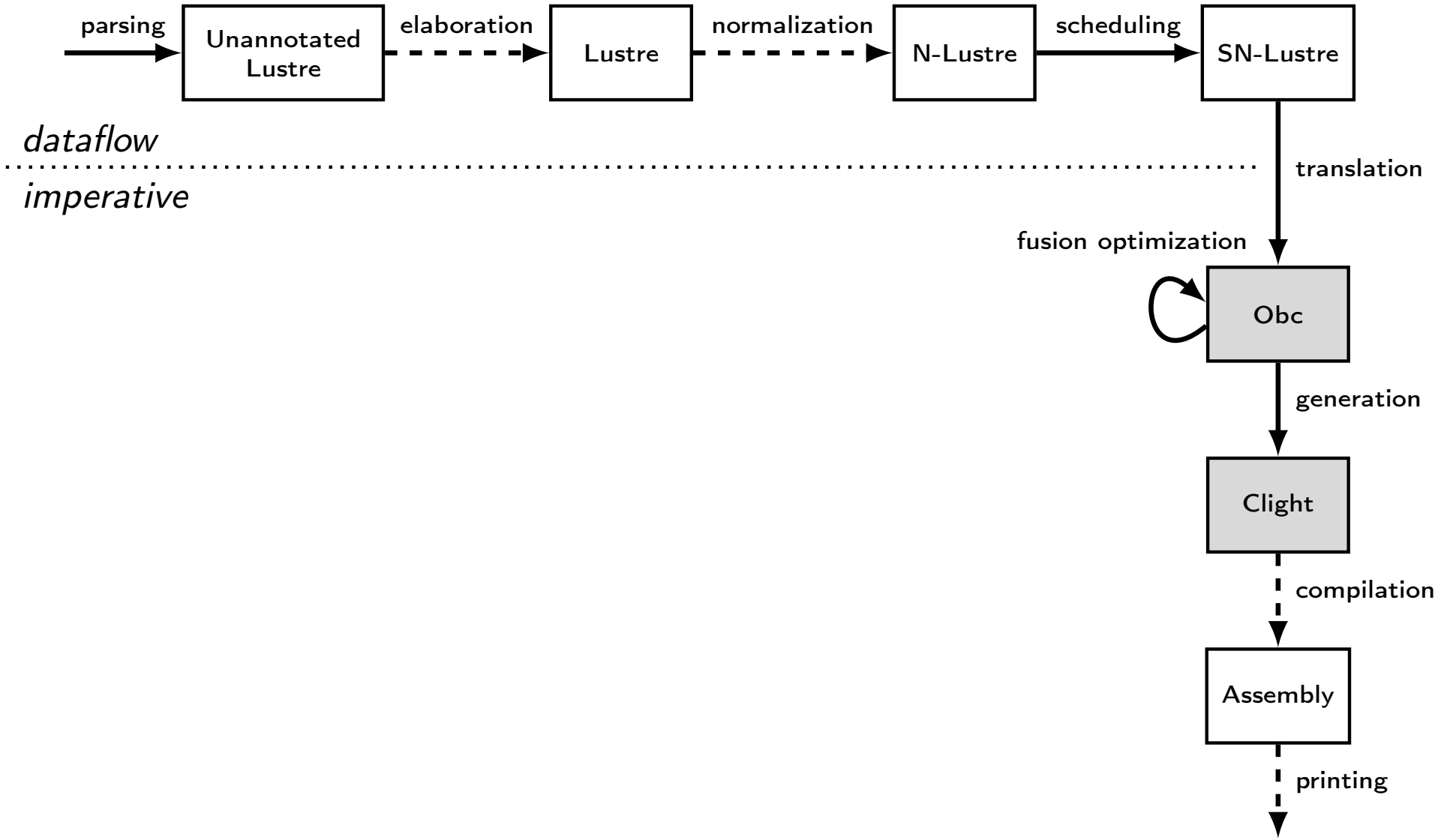
Critical aspect

- specification norms (DO-178B), industrial certification
- formal verification, mechanized proofs, proof assistant (eg. Coq)

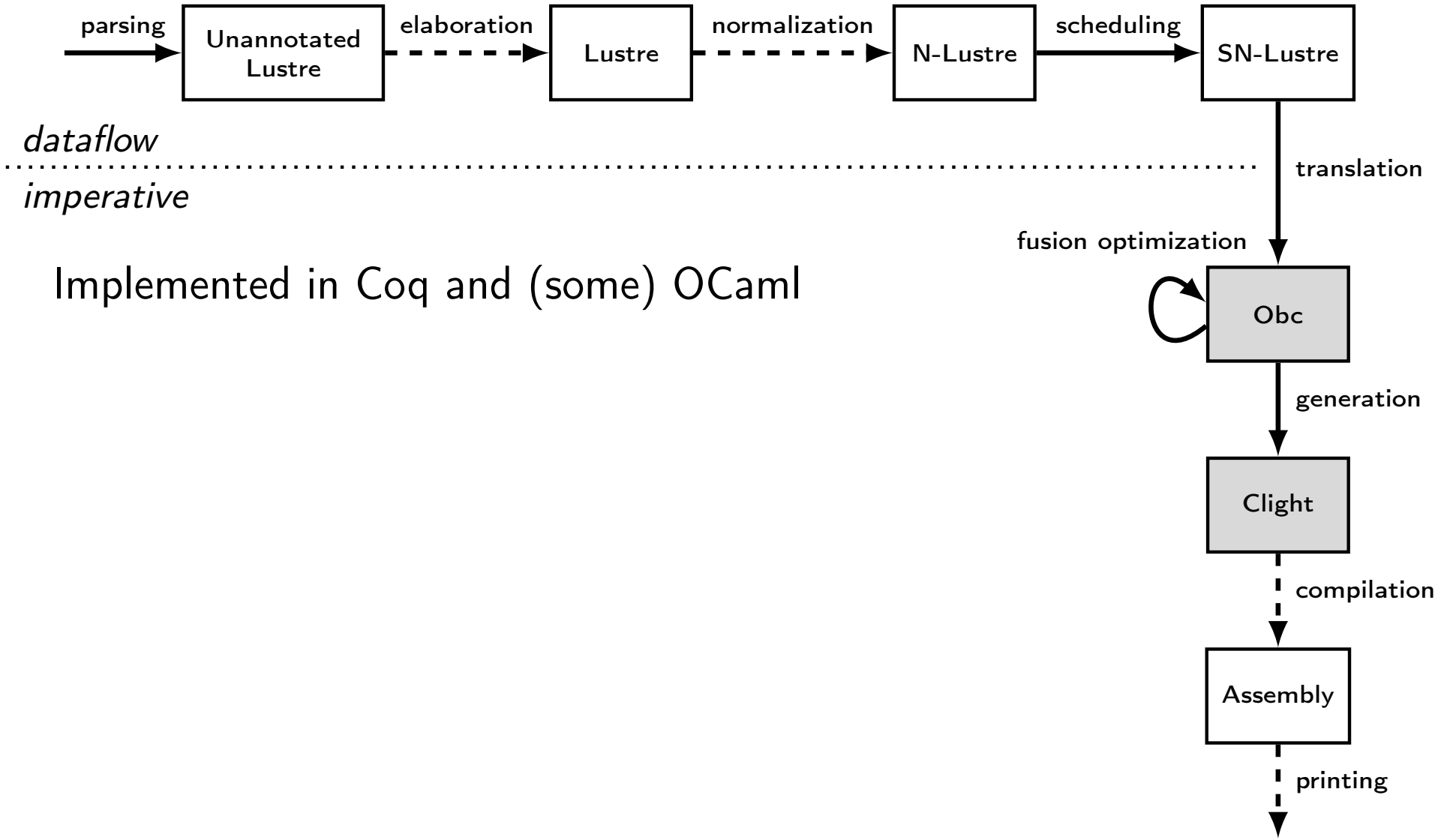
Goal

Develop a formally verified code generator

Vélus: a verified compiler

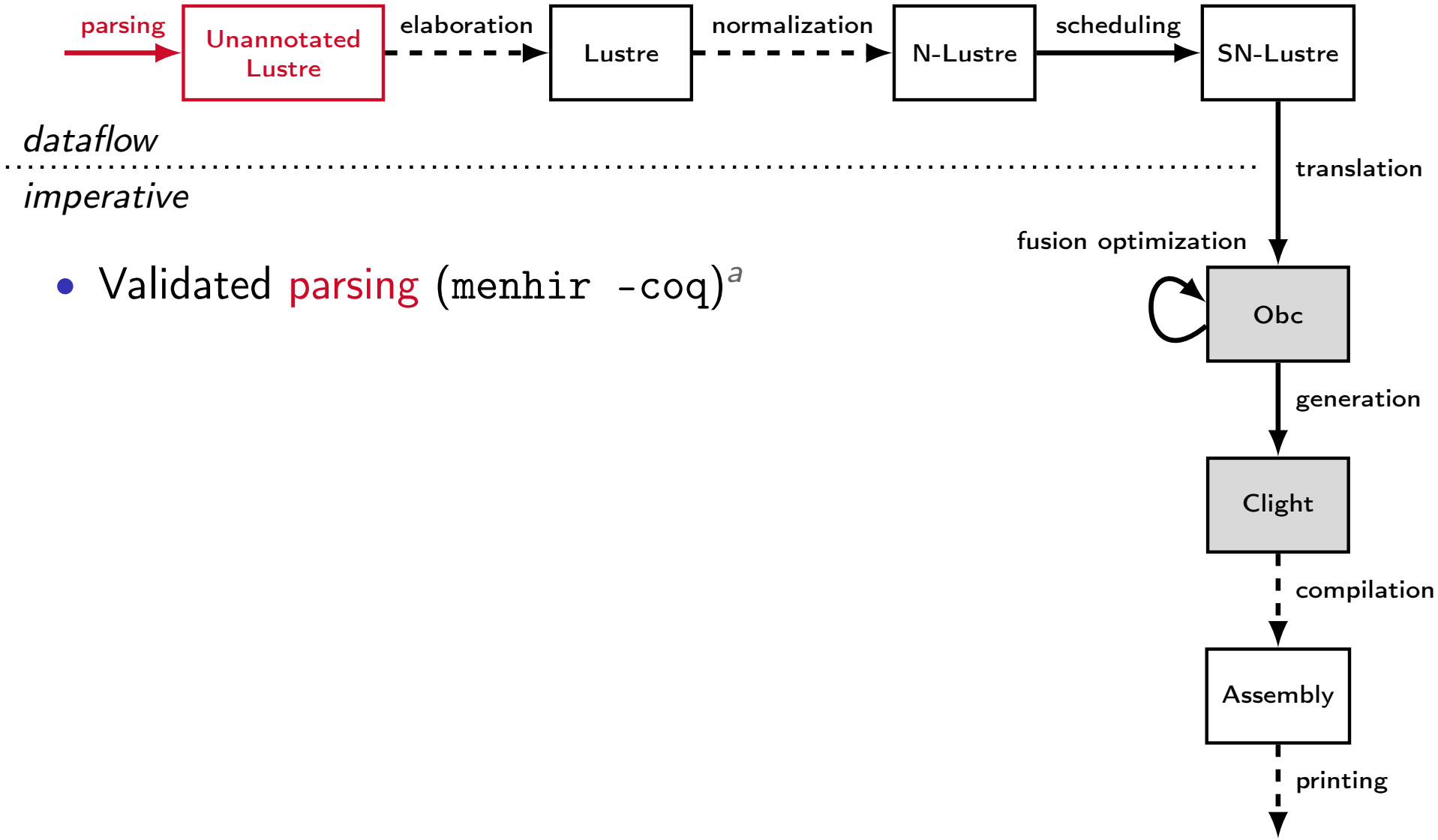


Vélus: a verified compiler



Implemented in Coq and (some) OCaml

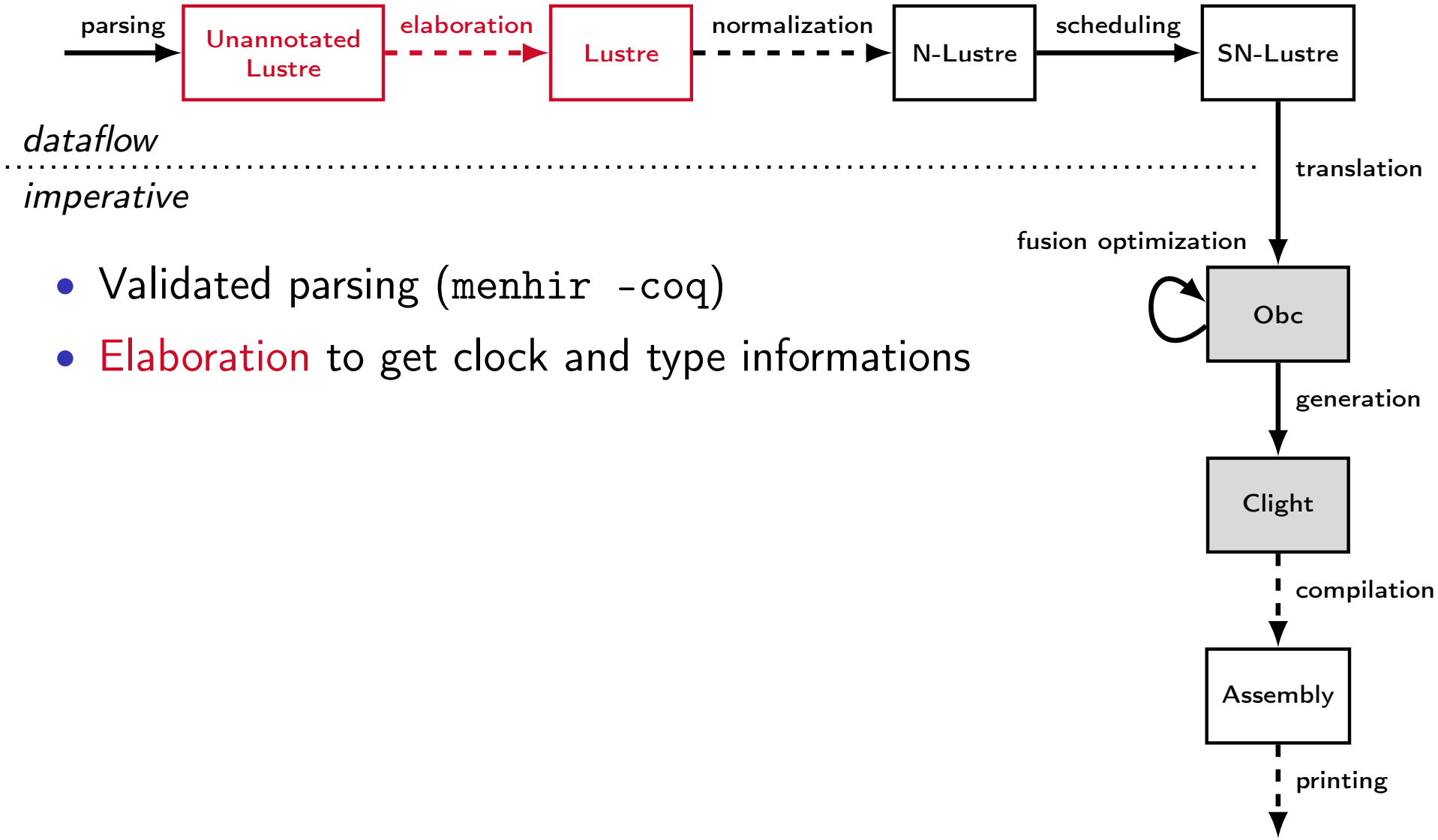
Vélus: a verified compiler



- Validated parsing (`menhir -coq`)[a]

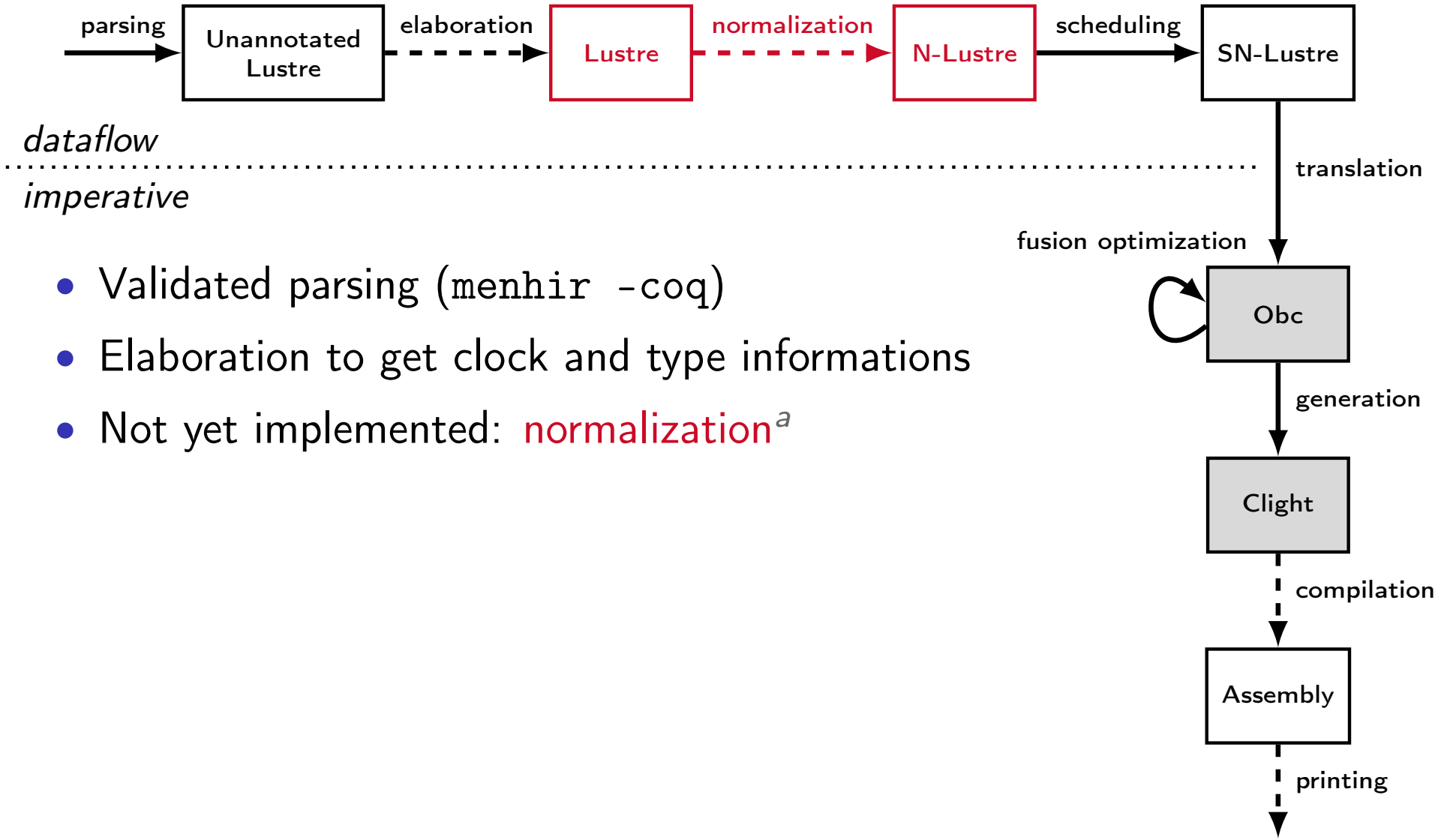
[a] Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"

Vélus: a verified compiler



- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations

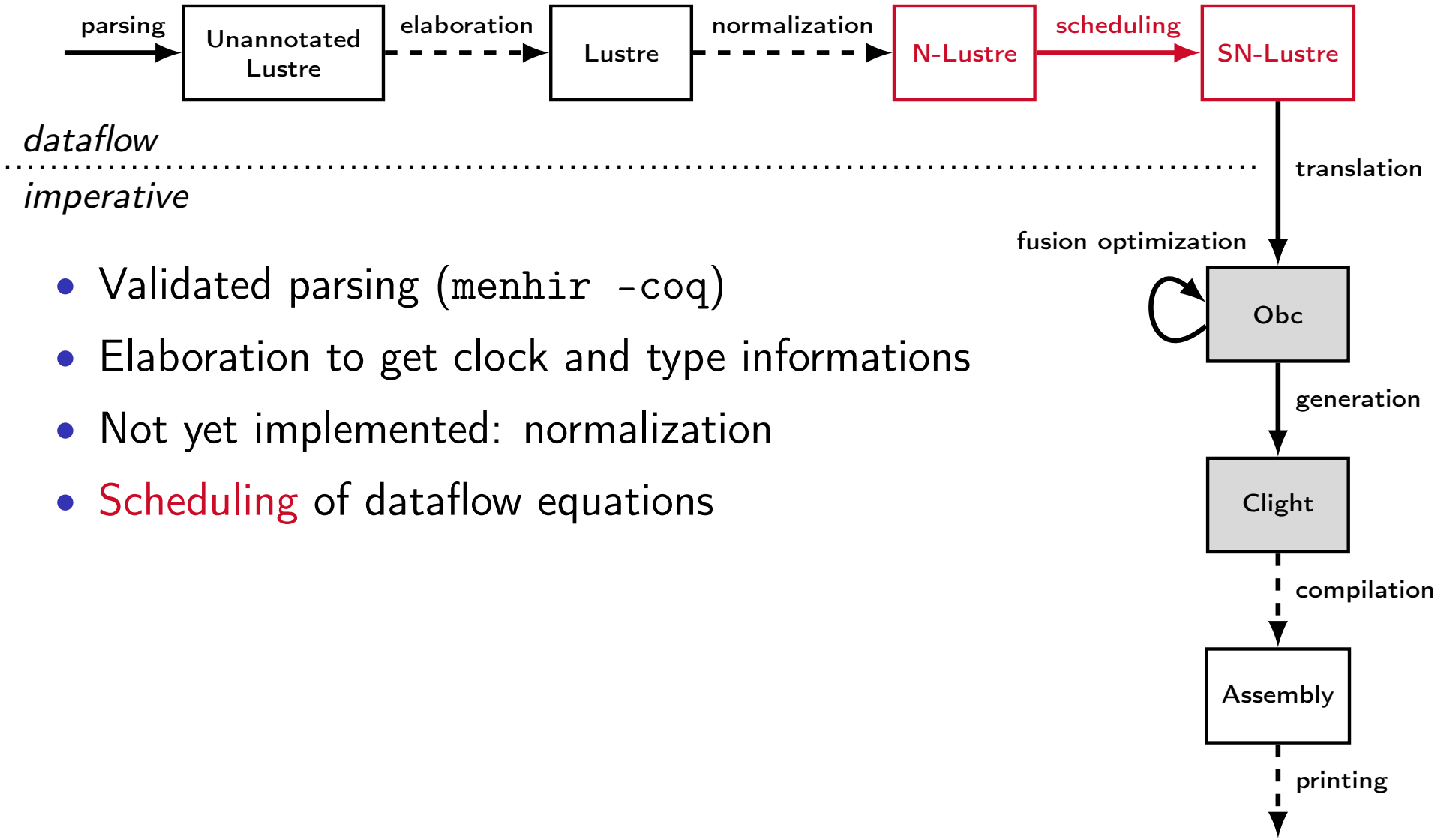
Vélus: a verified compiler



- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization[a]

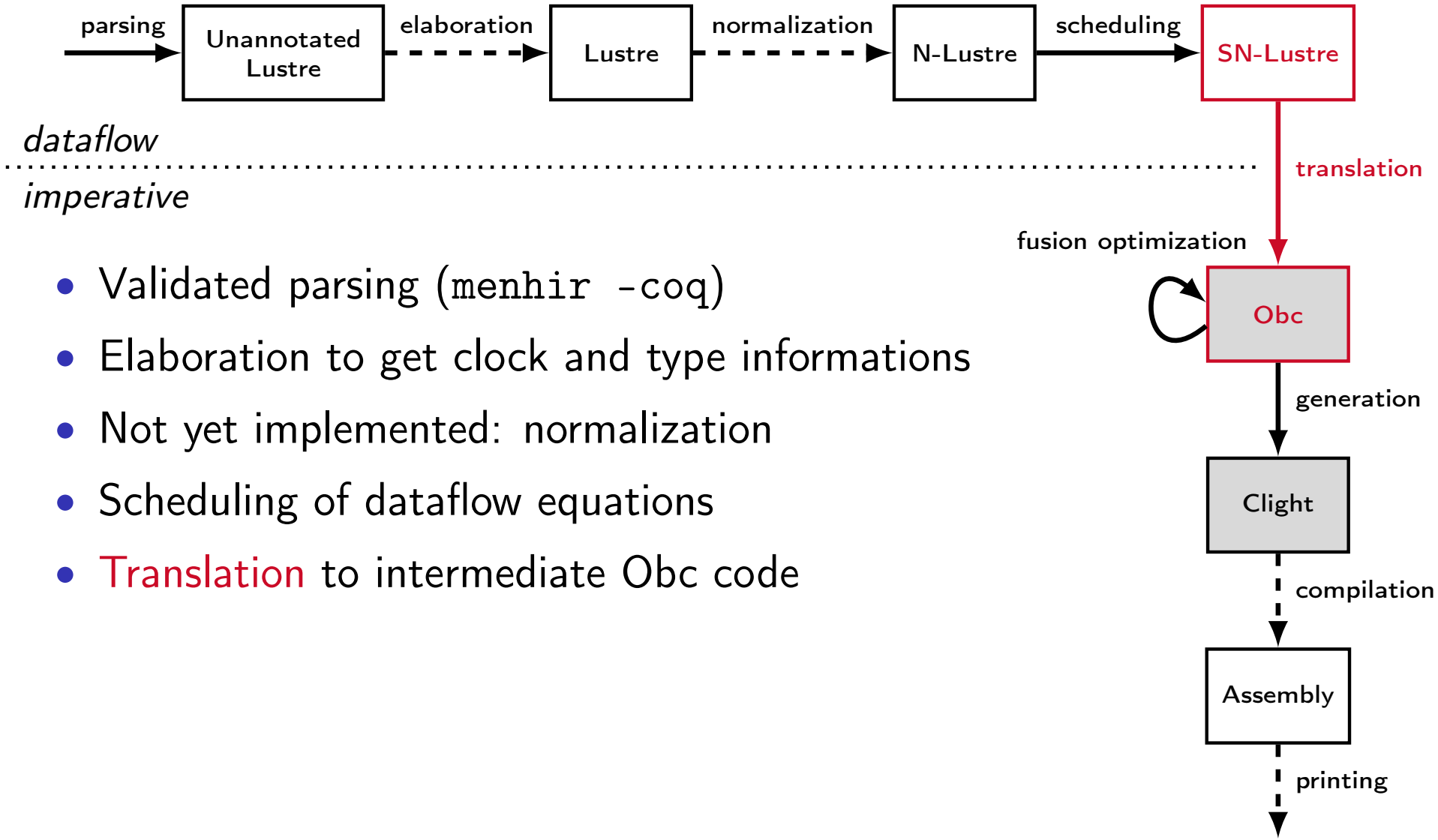
[a] Auger (2013): "Compilation certifiée de SCADE/LUSTRE"

Vélus: a verified compiler



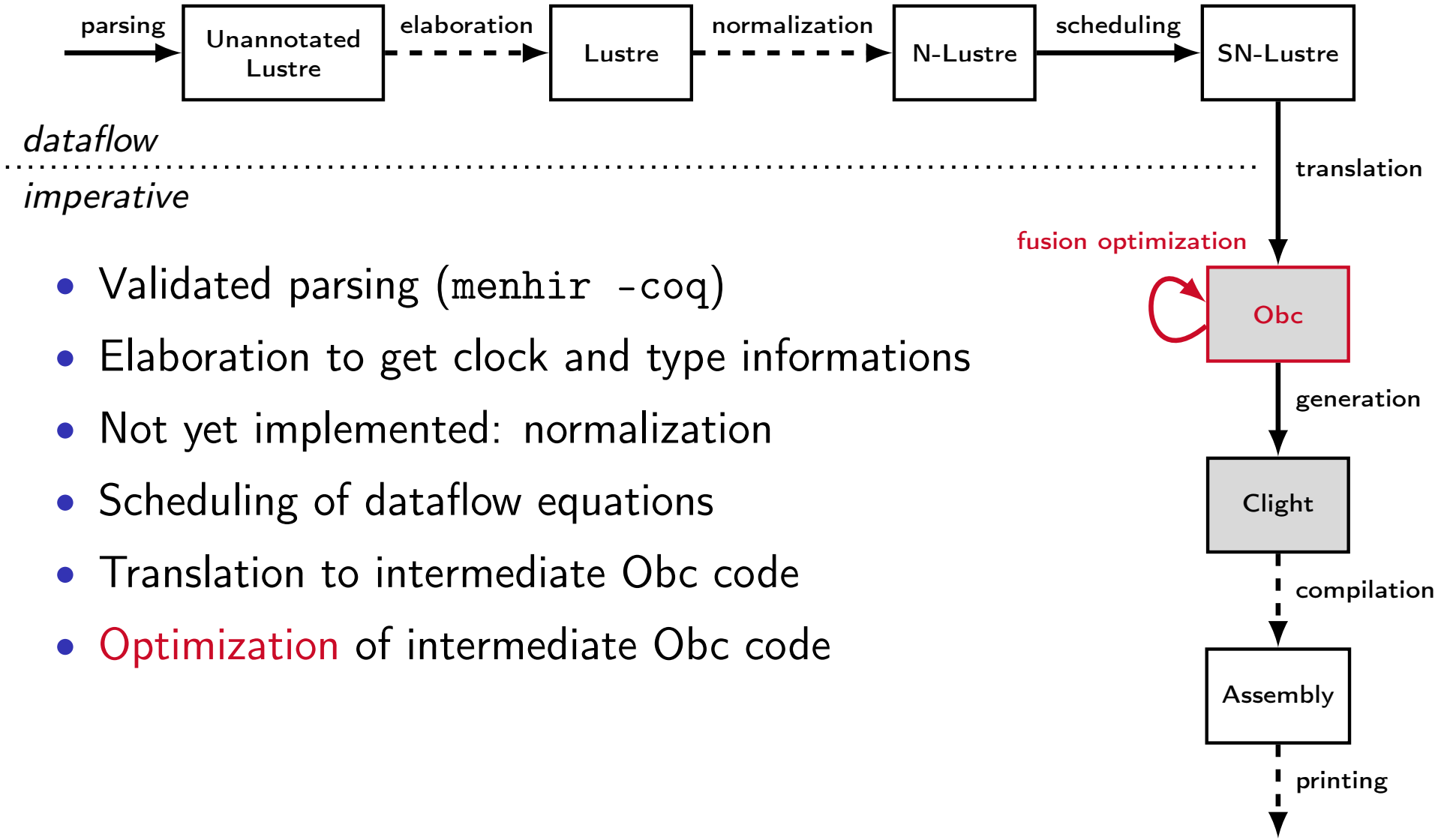
- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations

Vélus: a verified compiler



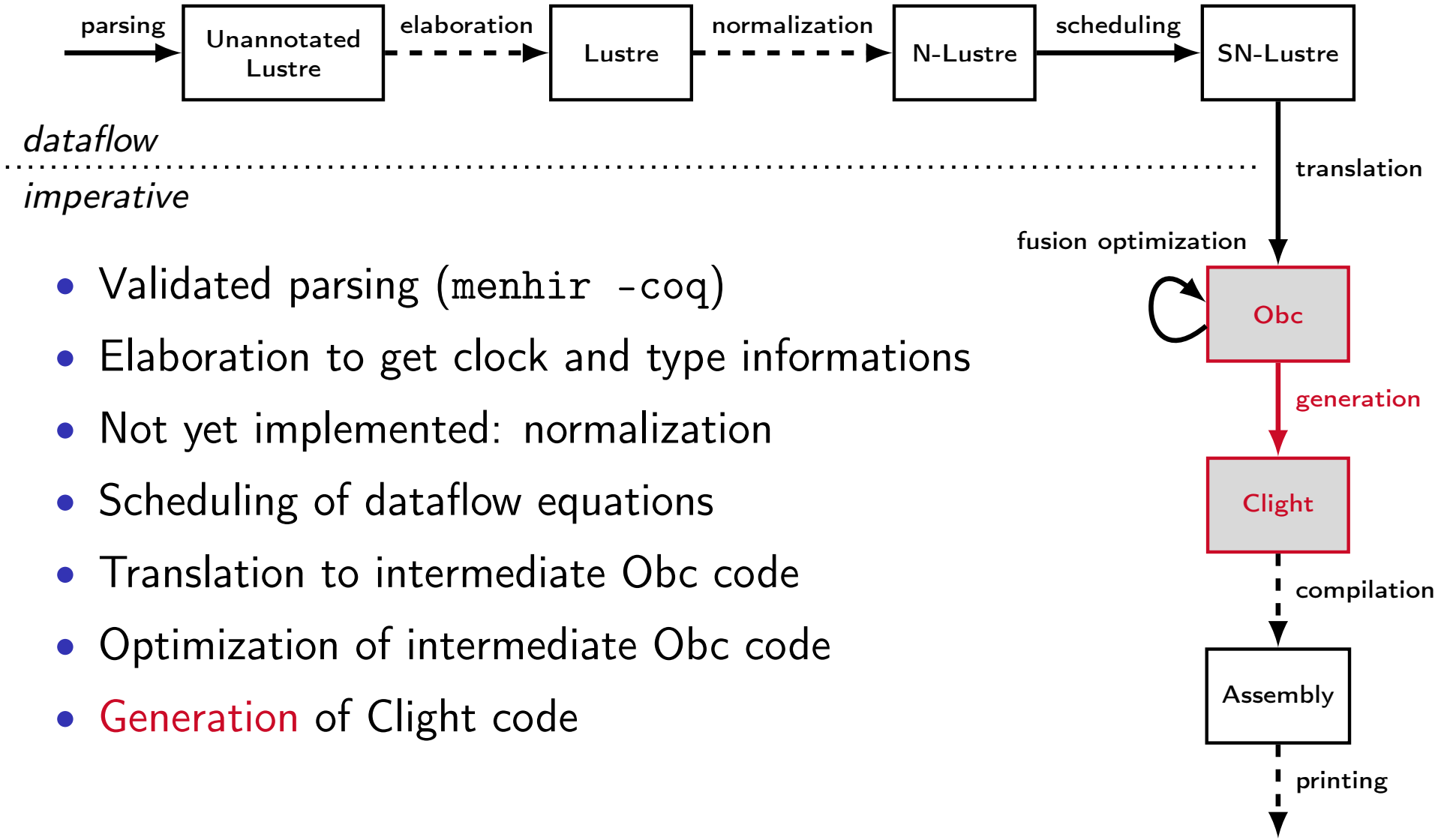
- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code

Vélus: a verified compiler



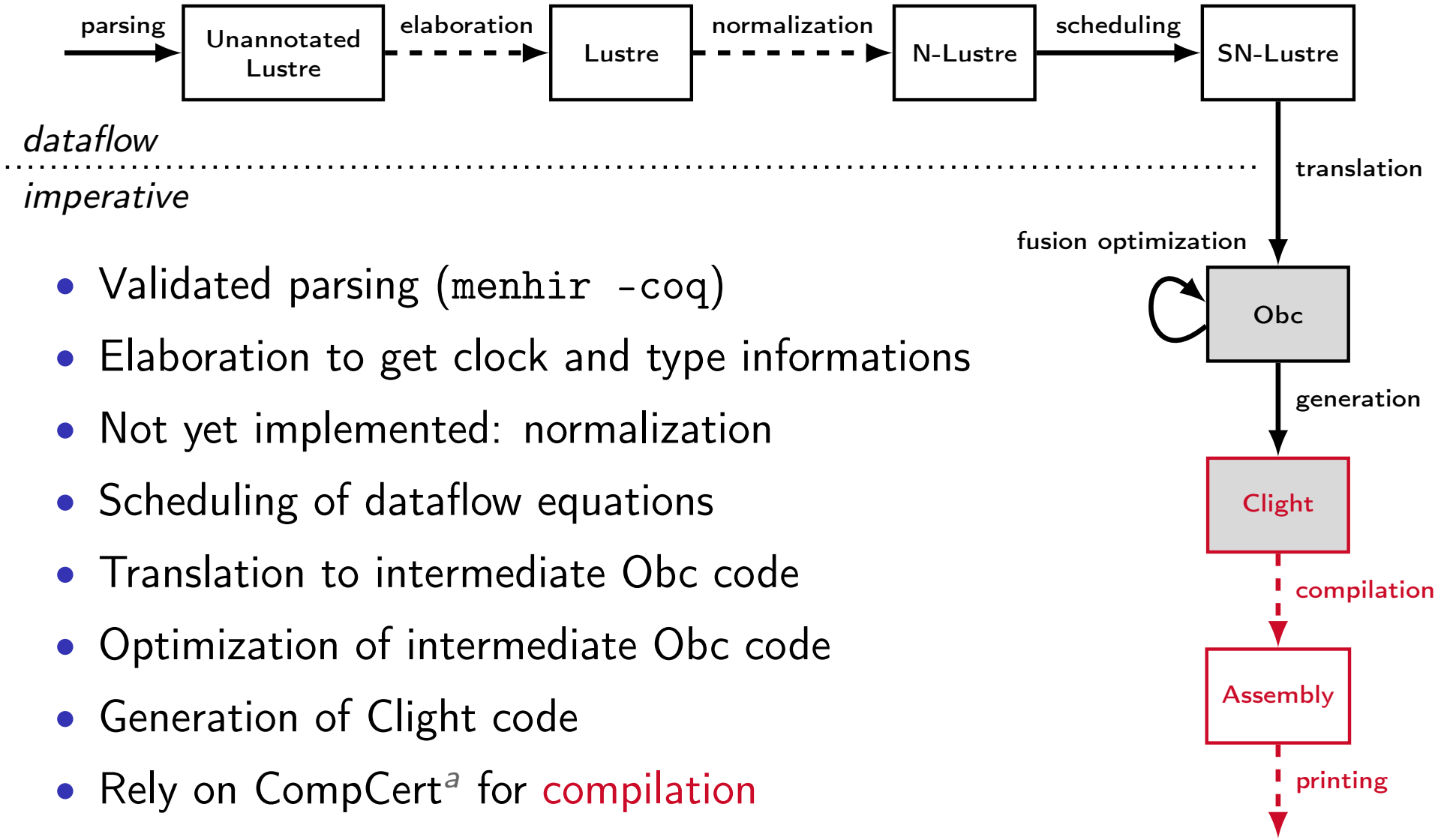
- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code
- Optimization of intermediate Obc code

Vélus: a verified compiler



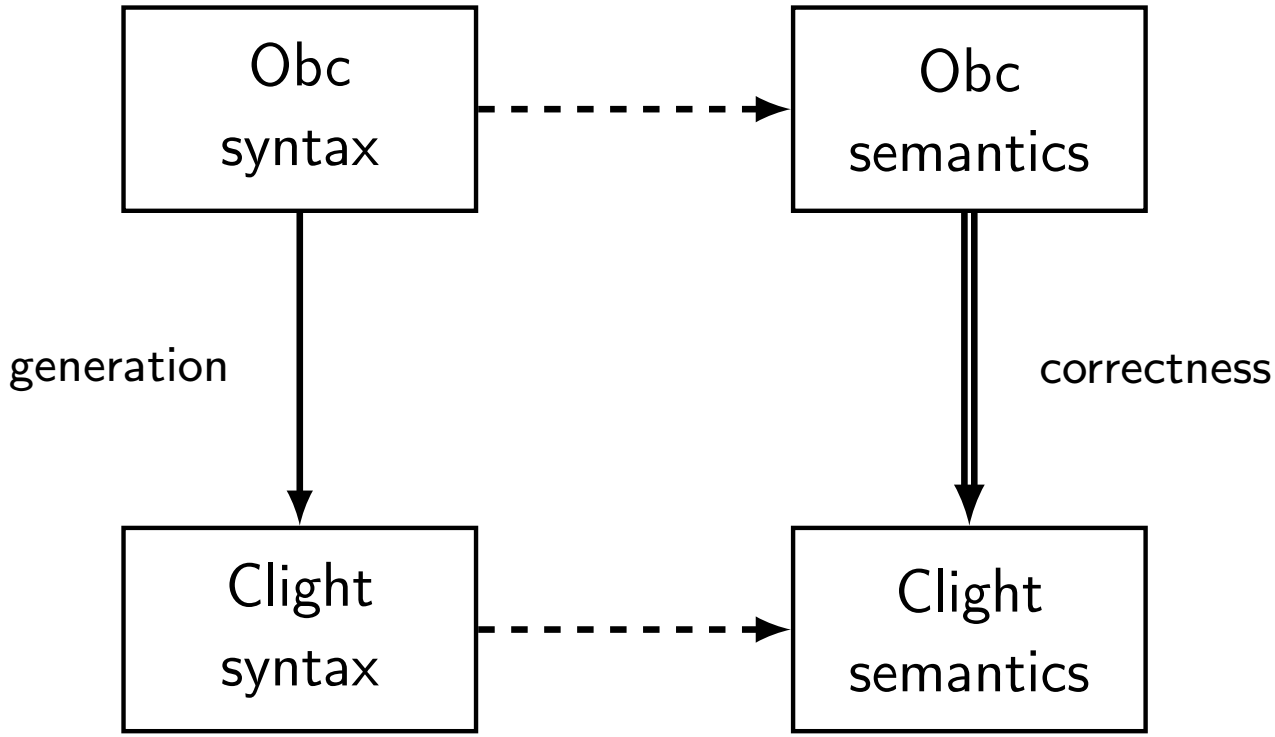
- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code
- Optimization of intermediate Obc code
- Generation of Clight code

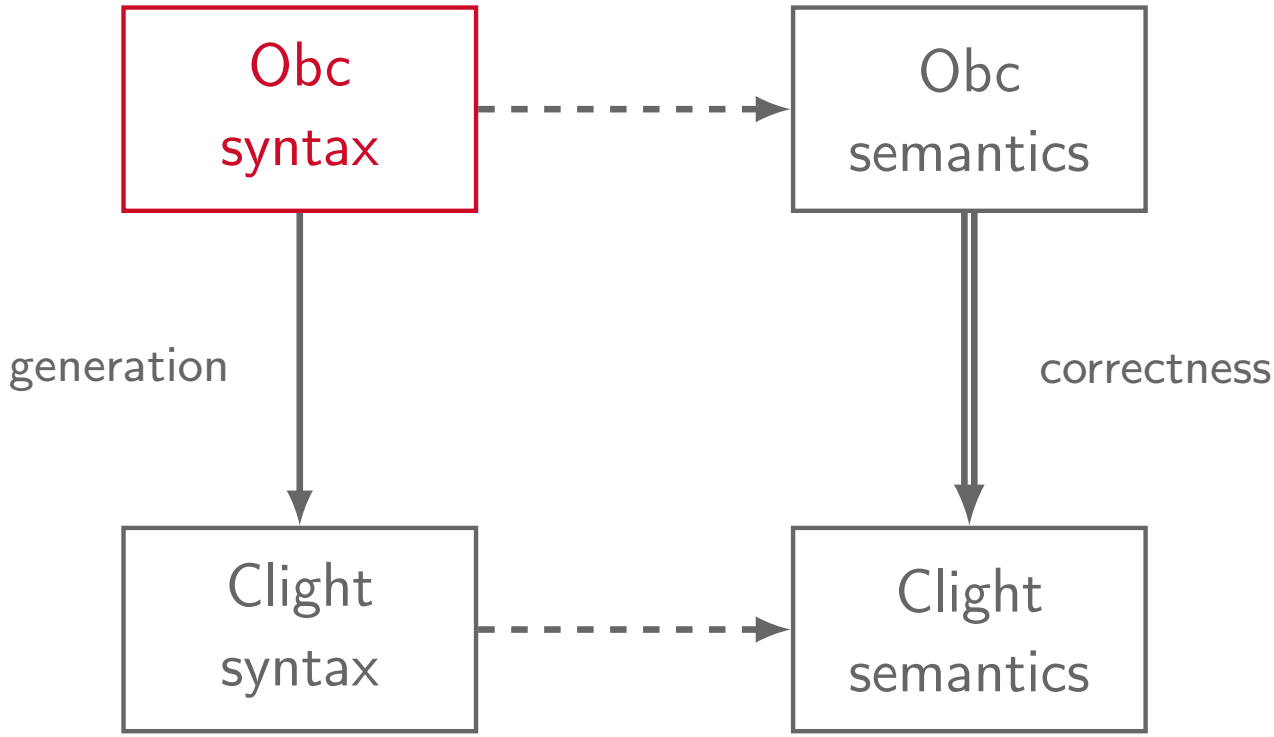
Vélus: a verified compiler



- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code
- Optimization of intermediate Obc code
- Generation of Clight code
- Rely on CompCert[a] for compilation

[a] Blazy, Dargaye, and Leroy (2006): "Formal verification of a C compiler front-end"


Obc
syntax
Obc
semantics
generation
correctness
Clight
syntax
Clight
semantics


Obc
syntax
Obc
semantics
generation
correctness
Clight
syntax
Clight
semantics

Lustre

```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

Obc

```
class rect {

  memory py: int;

  reset() { state(py) := 0 }

  step(d: int) returns (y: int) {
    y := state(py) + d;
    state(py) := y
  }
}
```

Lustre

```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

```
node integrator(a: int) returns (v, x: int)
let
  v = rect(a);
  x = rect(v);
tel
```

Obc

```
class rect {

  memory py: int;

  reset() { state(py) := 0 }

  step(d: int) returns (y: int) {
    y := state(py) + d;
    state(py) := y
  }
}
```

```
class integrator {

  instance v, x: rect;

  reset() {
    rect(v).reset();
    rect(x).reset()
  }

  step(a: int) returns (v, x: int) {
    v := rect(v).step(a);
    x := rect(x).step(v)
  }
}
```

Lustre

```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

```
node integrator(a: int) returns (v, x: int)
let
  v = rect(a);
  x = rect(v);
tel
```

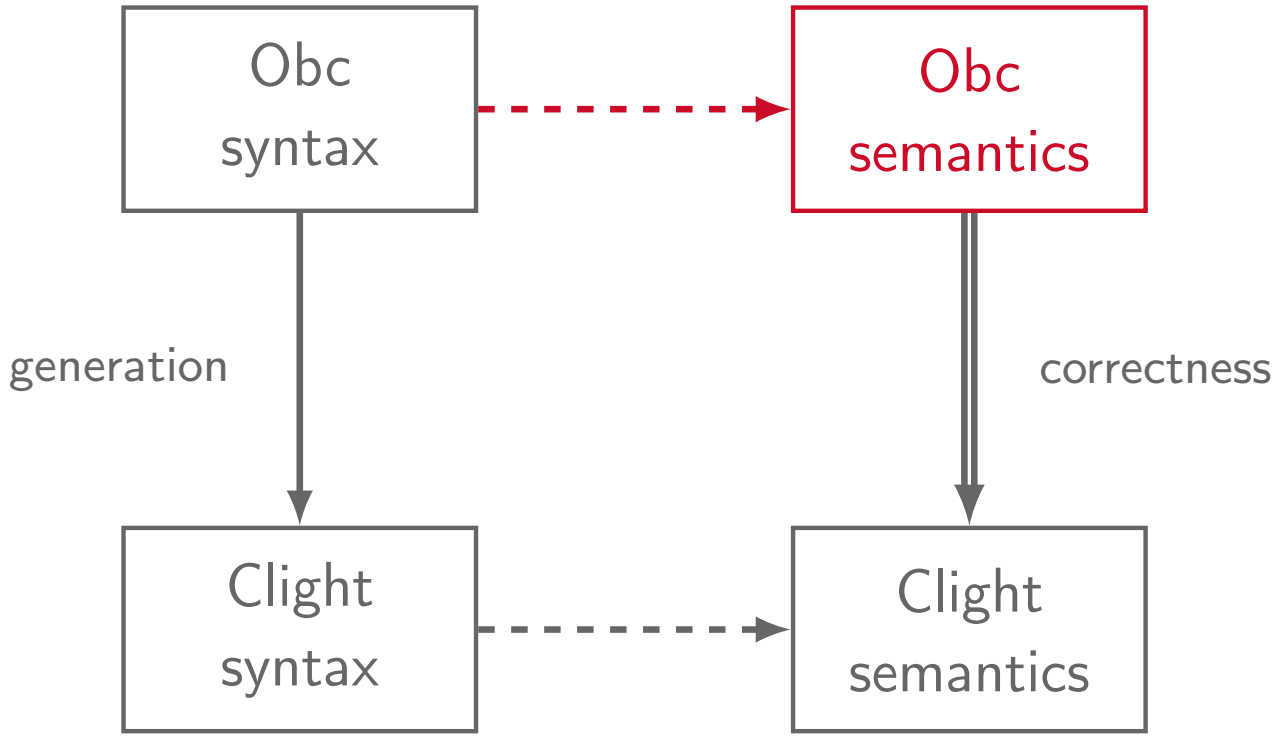
```
node excess(max, a: int)
  returns (e: bool; x: int)
  var v: int;
let
  (v, x) = integrator(a);
  e = max < v;
tel
```

Obc

```
class rect {

  memory py: int;

  reset() { state(py) := 0 }

  step(d: int) returns (y: int) {
    y := state(py) + d;
    state(py) := y
  }
}
```

```
class integrator {

  instance v, x: rect;

  reset() {
    rect(v).reset();
    rect(x).reset()
  }

  step(a: int) returns (v, x: int) {
    v := rect(v).step(a);
    x := rect(x).step(v)
  }
}
```

```
class excess {

  instance vx: integrator;

  reset() { integrator(vx).reset() }

  step(max, a: int)
    returns (e: bool, x: int)
    var v: int
  {
    v, x := integrator(vx).step(a);
    e := max < v
  }
}
```


Obc
syntax
Obc
semantics
generation
correctness
Clight
syntax
Clight
semantics

State and memory model

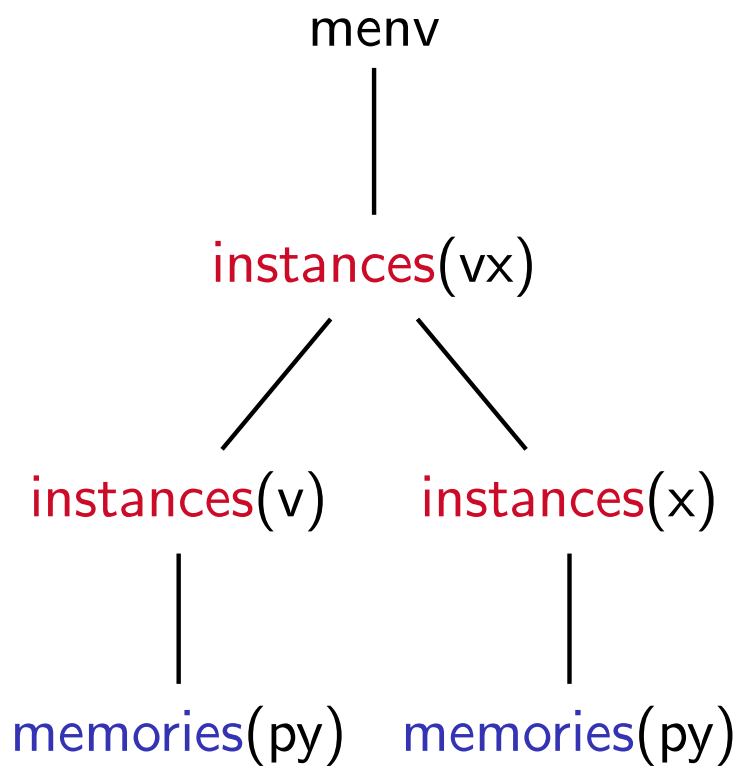
$$venv \triangleq ident \rightharpoonup val$$

$$menv \triangleq \begin{cases} \text{memories} & : ident \rightharpoonup val \\ \text{instances} & : ident \rightharpoonup menv \end{cases}$$

State and memory model

$$venv \triangleq ident \rightharpoonup val$$

$$menv \triangleq \begin{cases} \text{memories} & : ident \rightharpoonup val \\ \text{instances} & : ident \rightharpoonup menv \end{cases}$$



Some semantics rules

Expressions

$$\frac{}{me, ve \vdash_{\mathsf{exp}} x \Downarrow ve(x)}$$

$$\frac{}{me, ve \vdash_{\mathsf{exp}} \mathsf{state}(x) \Downarrow me.\mathsf{memories}(x)}$$

Some semantics rules

Expressions

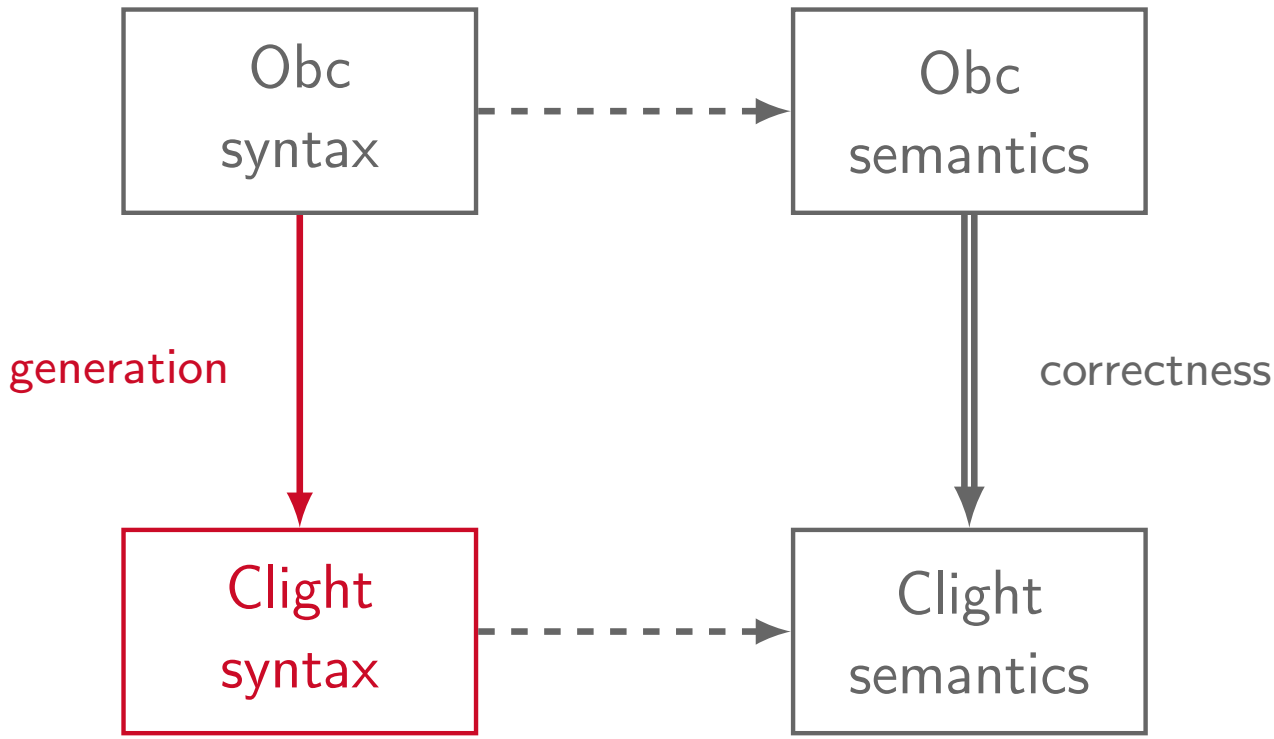
$$\frac{}{me, ve \vdash_{\text{exp}} x \Downarrow ve(x)}$$

$$\frac{}{me, ve \vdash_{\text{exp}} \text{state}(x) \Downarrow me.\text{memories}(x)}$$

Statements

$$\frac{me, ve \vdash_{\text{exp}} e \Downarrow v}{p, me, ve \vdash_{\text{st}} x := e \Downarrow me, ve\{x \mapsto v\}}$$

$$\frac{me, ve \vdash_{\text{exp}} e \Downarrow v}{p, me, ve \vdash_{\text{st}} \text{state}(x) := e \Downarrow \text{update\_mem}(me, x, v), ve}$$


Obc
syntax
Obc
semantics
generation
correctness
Clight
syntax
Clight
semantics

Clight

- CompCert's frontend language
- very similar to C
- low-level operation (addresses, offsets, structures,...)

Generation function

- Obc class $\mapsto$ Clight structure
- Obc method $\mapsto$ void-returning Clight function
  - state: pointer *self*
  - multiple outputs: pointer *out*

Obc

```
class rect {
  memory py : int;
  [...]
}
```

Clight

```
struct rect {
  int py;
};
```

Obc

```
class rect {
  memory py : int;
  [...]
}

class integrator {
  instance v, x: rect;
  [...]
}
```

Clight

```
struct rect {
  int py;
};

struct integrator {
  struct rect v;
  struct rect x;
};
```

Obc

```
class rect {
  memory py : int;
  [...]
}

class integrator {
  instance v, x: rect;
  [...]
}

class excess {
  instance vx: integrator;
  [...]
```

Clight

```
struct rect {
  int py;
};

struct integrator {
  struct rect v;
  struct rect x;
};

struct excess {
  struct integrator vx;
};
```

Obc

```
class rect {
  memory py : int;
  [...]
}

class integrator {
  instance v, x: rect;
  [...]
}

class excess {
  instance vx: integrator;
  [...]


  step(max, a: int)
    returns (e: bool, x: int)
    var v: int
  {
    v, x := integrator(vx).step(a);
    e := v > max
  }
}
```

Clight

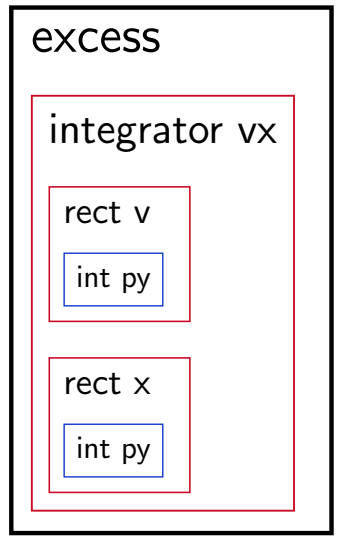
```
struct rect {
  int py;
};

struct integrator {
  struct rect v;
  struct rect x;
};

struct excess {
  struct integrator vx;
};


struct excess_step {
  _Bool e;
  int x;
};
void excess_step(struct excess *self,
                 struct excess_step *out,
                 int max, int a)
{
  struct integrator_step vx_step;
  register int v;
  integrator_step(&(*self).vx, &vx_step, a);
  v = vx_step.v;
  (*out).x = vx_step.x;
  (*out).e = v > max;
}
```

Obc

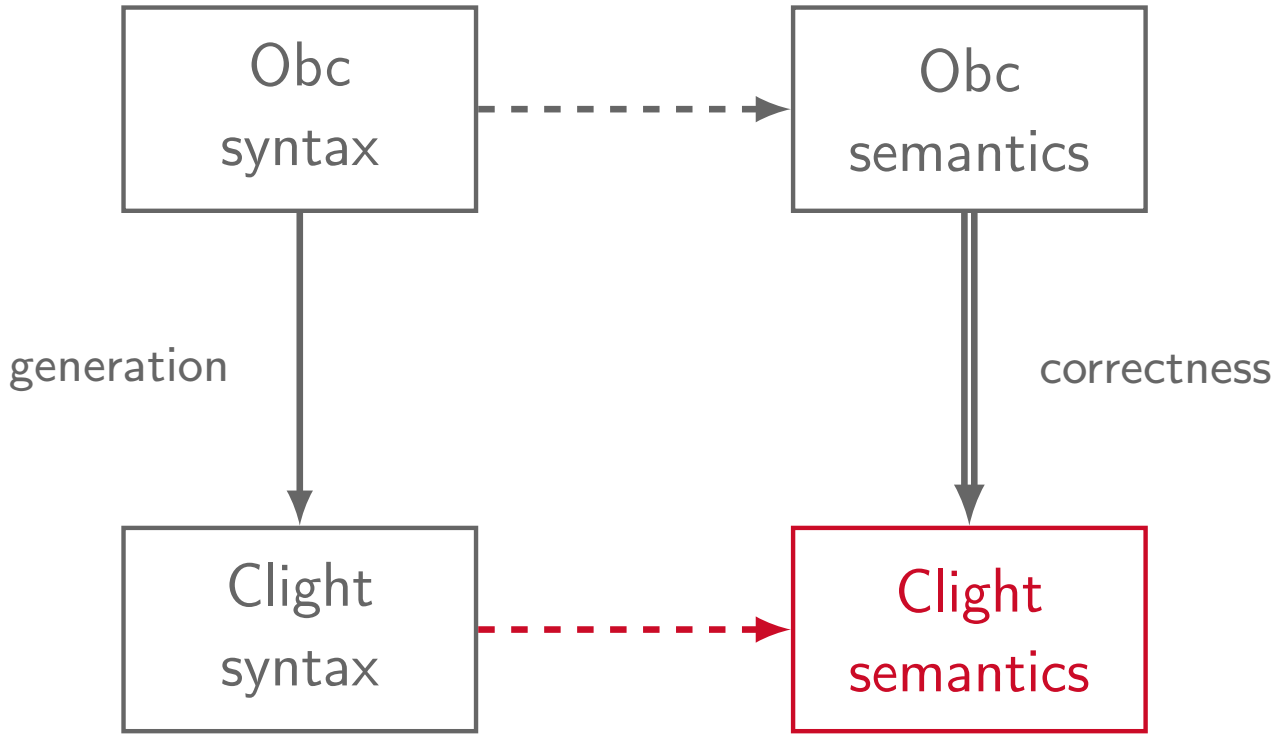
```
class rect {
  memory py : int;
  [...]
}

class integrator {
  instance v, x: rect;
  [...]
}

class excess {
  instance vx: integrator;
  [...]


  step(max, a: int)
    returns (e: bool, x: int)
    var v: int
  {
    v, x := integrator(vx).step(a);
    e := v > max
  }
}
```

Clight

```
struct rect {
  int py;
};

struct integrator {
  struct rect v;
  struct rect x;
};

struct excess {
  struct integrator vx;
};
```




```
struct excess_step {
  _Bool e;
  int x;
};
void excess_step(struct excess *self,
                 struct excess_step *out,
                 int max, int a)
{
  struct integrator_step vx_step;
  register int v;
  integrator_step(&(*self).vx, &vx_step, a);
  v = vx_step.v;
  (*out).x = vx_step.x;
  (*out).e = v > max;
}
```


Obc
syntax
Obc
semantics
generation
correctness
Clight
syntax
Clight
semantics

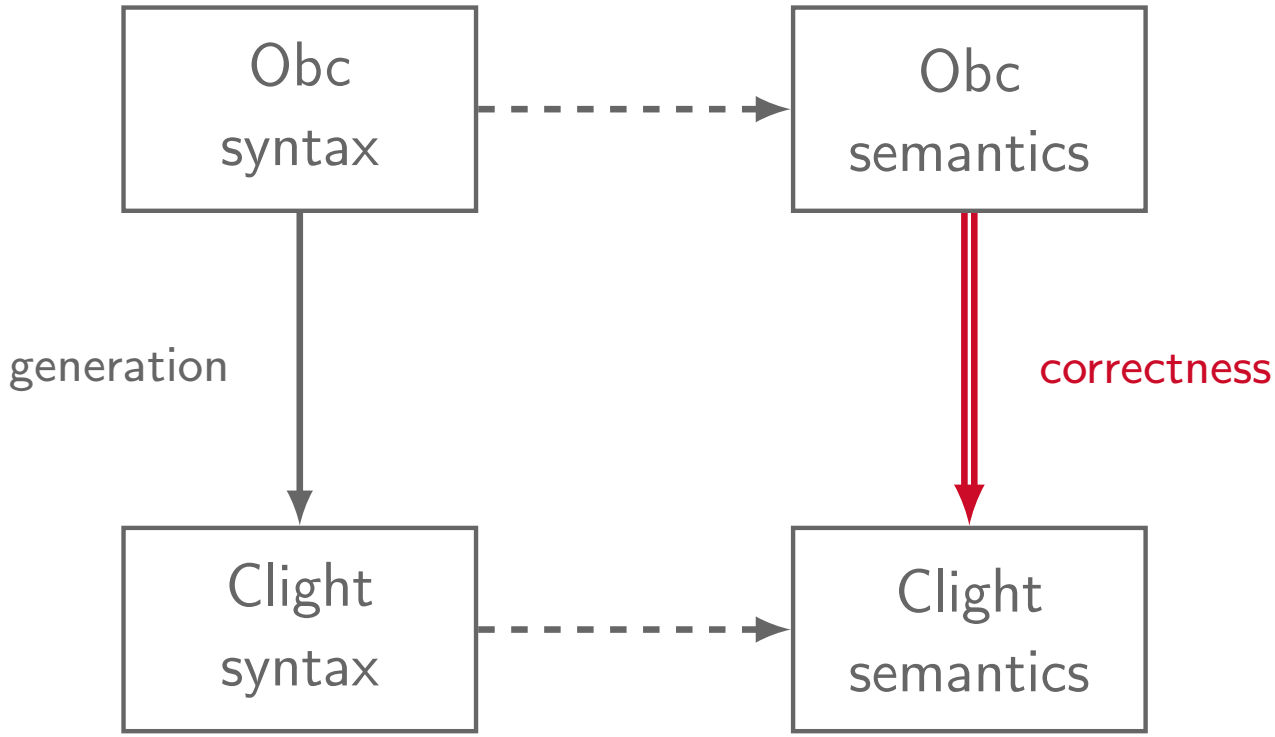
Clight semantics

- block memory model
- 2 types of variables: local and temporaries
- semantics: state (e, le, m)

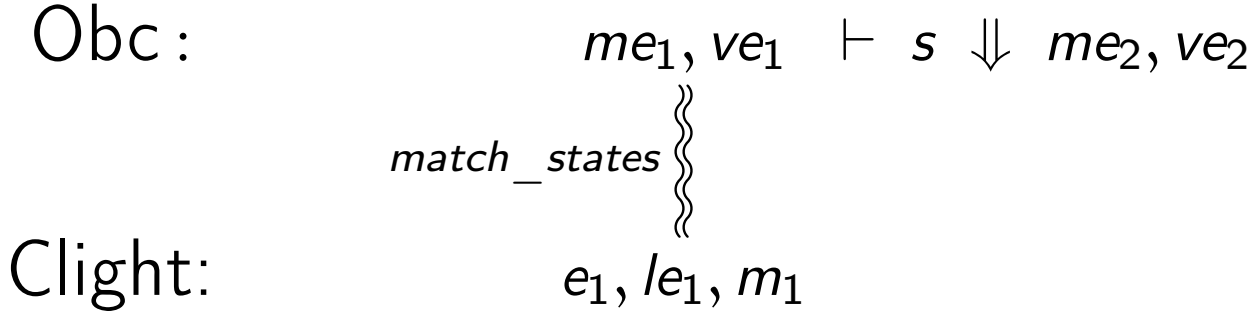
  e local variables environment: $ident \rightharpoonup block \times int$

  le temporaries environment: $ident \rightharpoonup val$

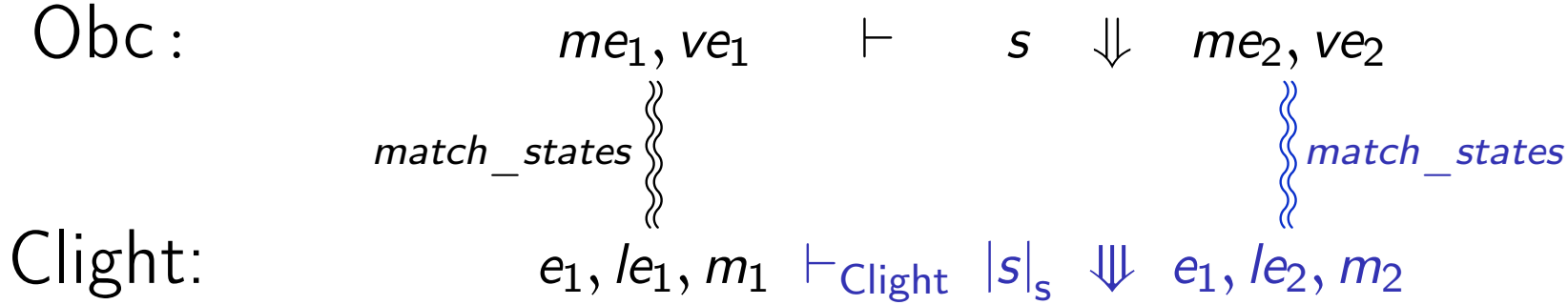
  m memory: $block \times int \rightharpoonup byte$


Obc
syntax
Obc
semantics
generation
correctness
Clight
syntax
Clight
semantics

Semantics preservation



Semantics preservation



Separation logic

Consequences of CompCert's memory model:

- aliasing (overlapping)
- alignment
- permissions (ownership)
- sizes

Separation logic

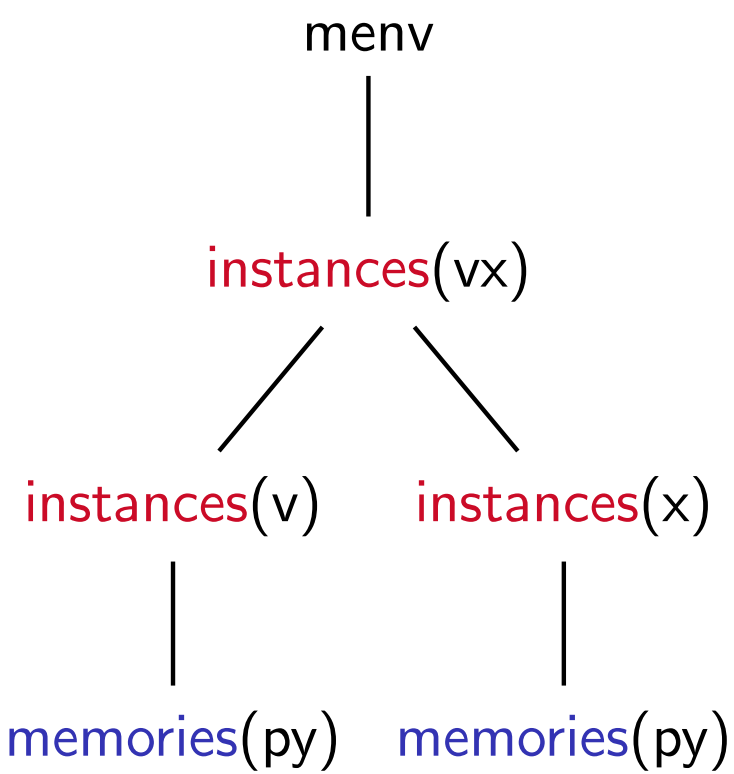
Consequences of CompCert's memory model:

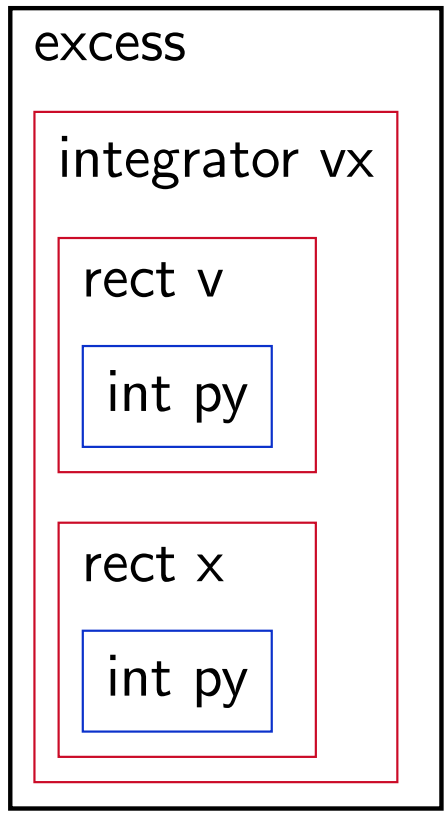
- aliasing (overlapping)
- alignment
- permissions (ownership)
- sizes

Solution

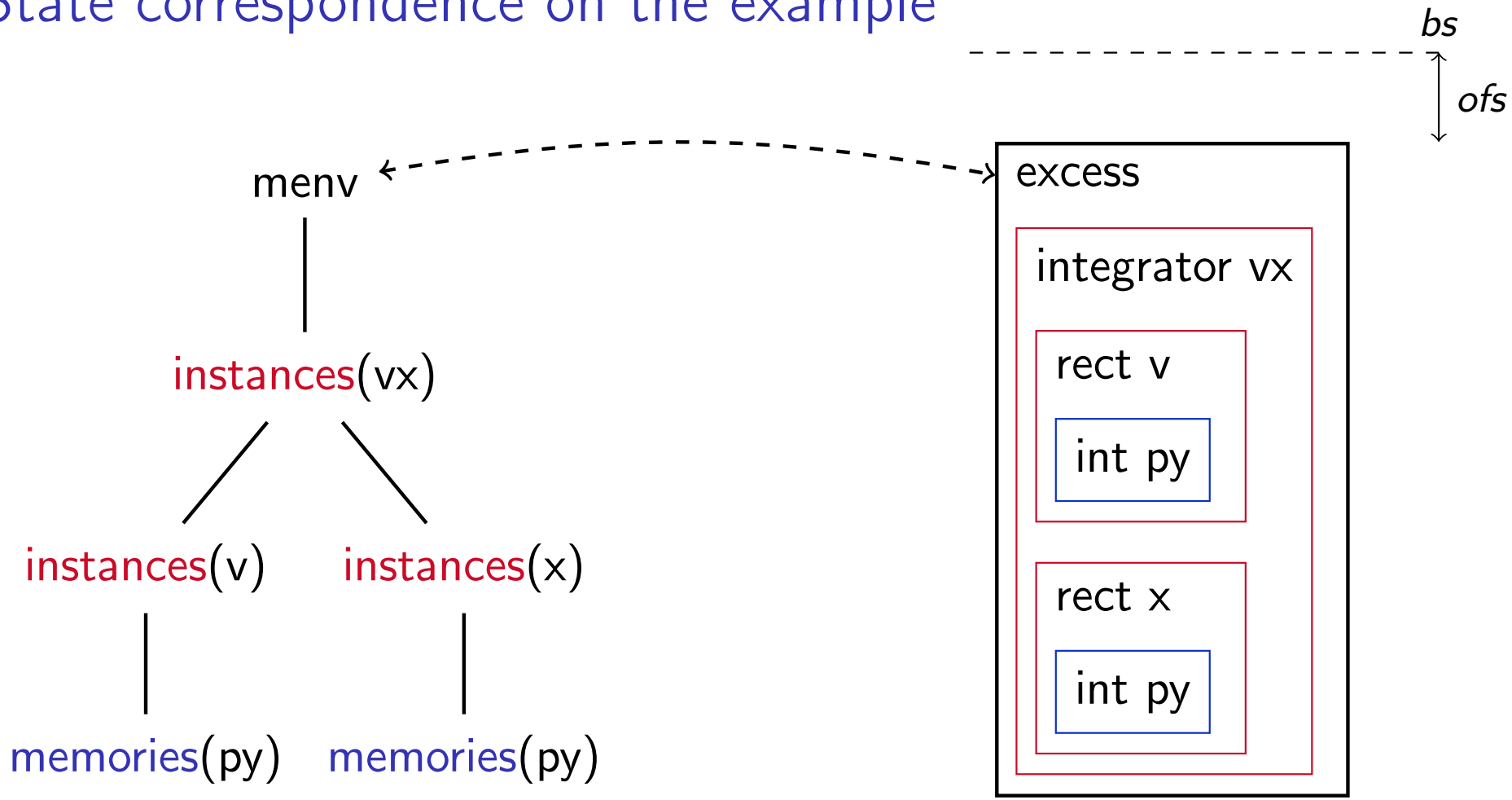
Use a separation logic formalism

State correspondence on the example



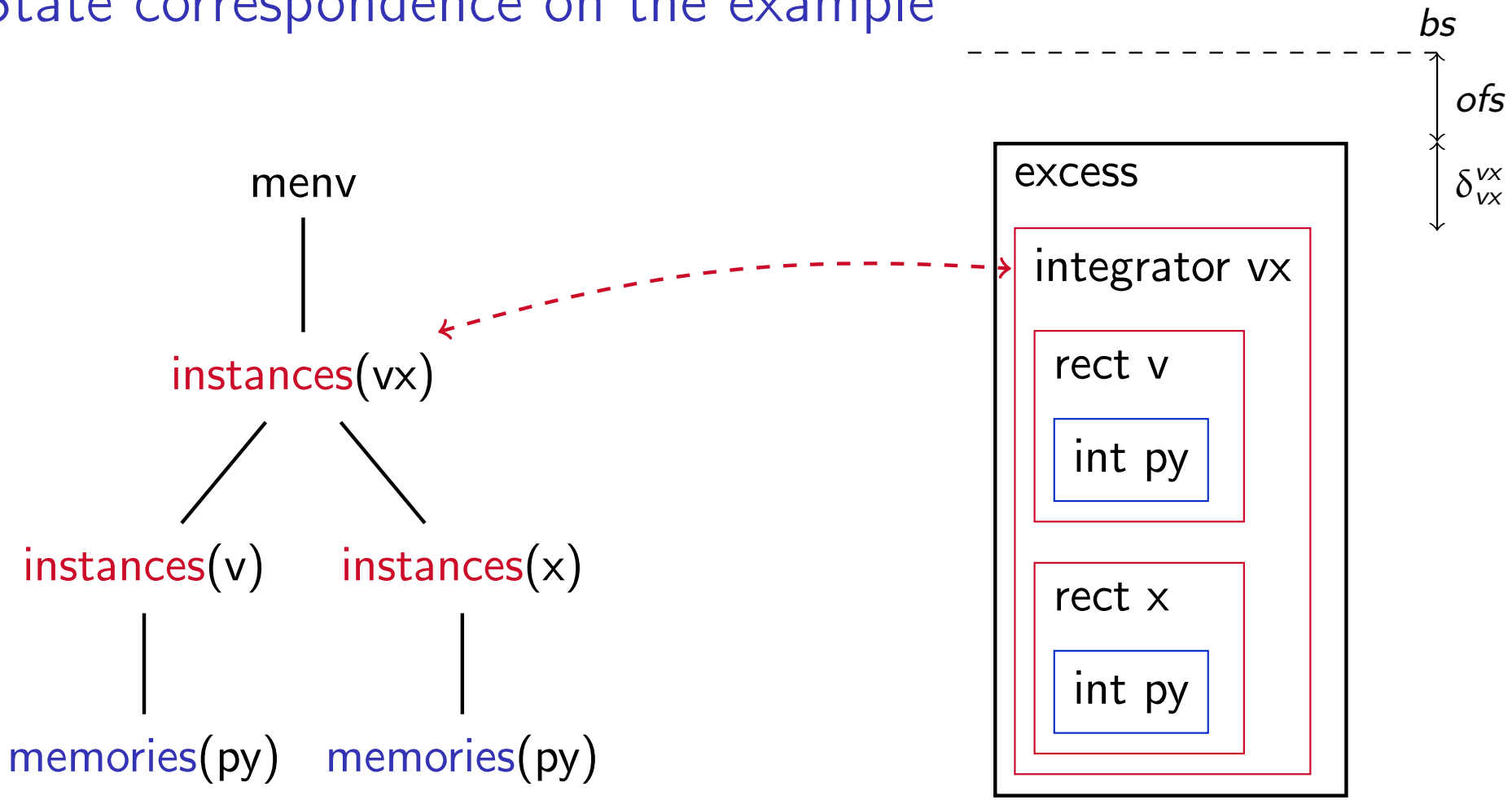


State correspondence on the example



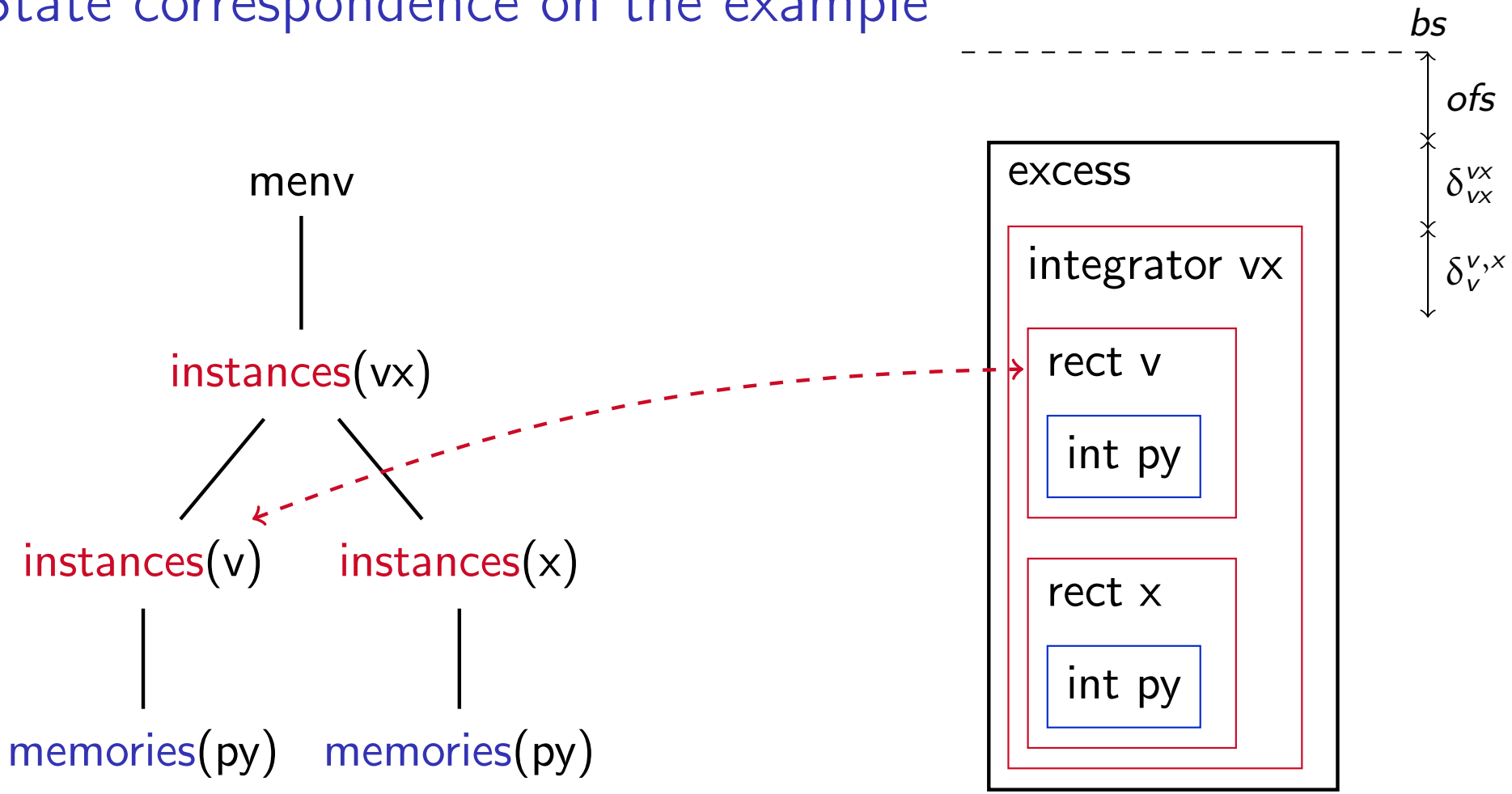
staterep *excess menv* b_s *ofs*

State correspondence on the example



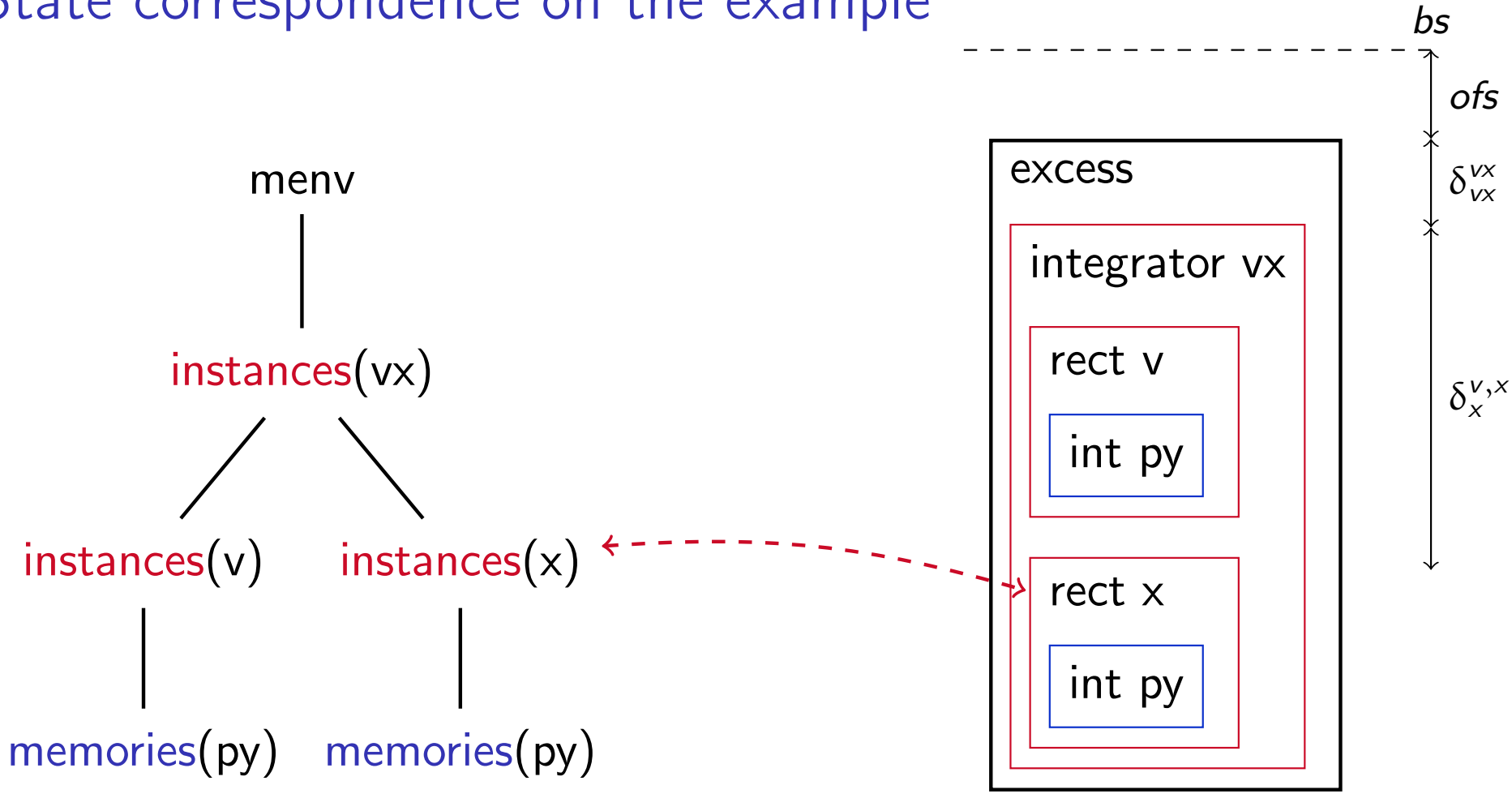
staterep *integrator menv*.instances(vx) b_s $(ofs + \delta^{vx}_{vx})$

State correspondence on the example



staterep *rect* *menv*.instances(*vx*).instances(*v*) b_s $(ofs + \delta_{vx}^{vx} + \delta_{v}^{v,x})$

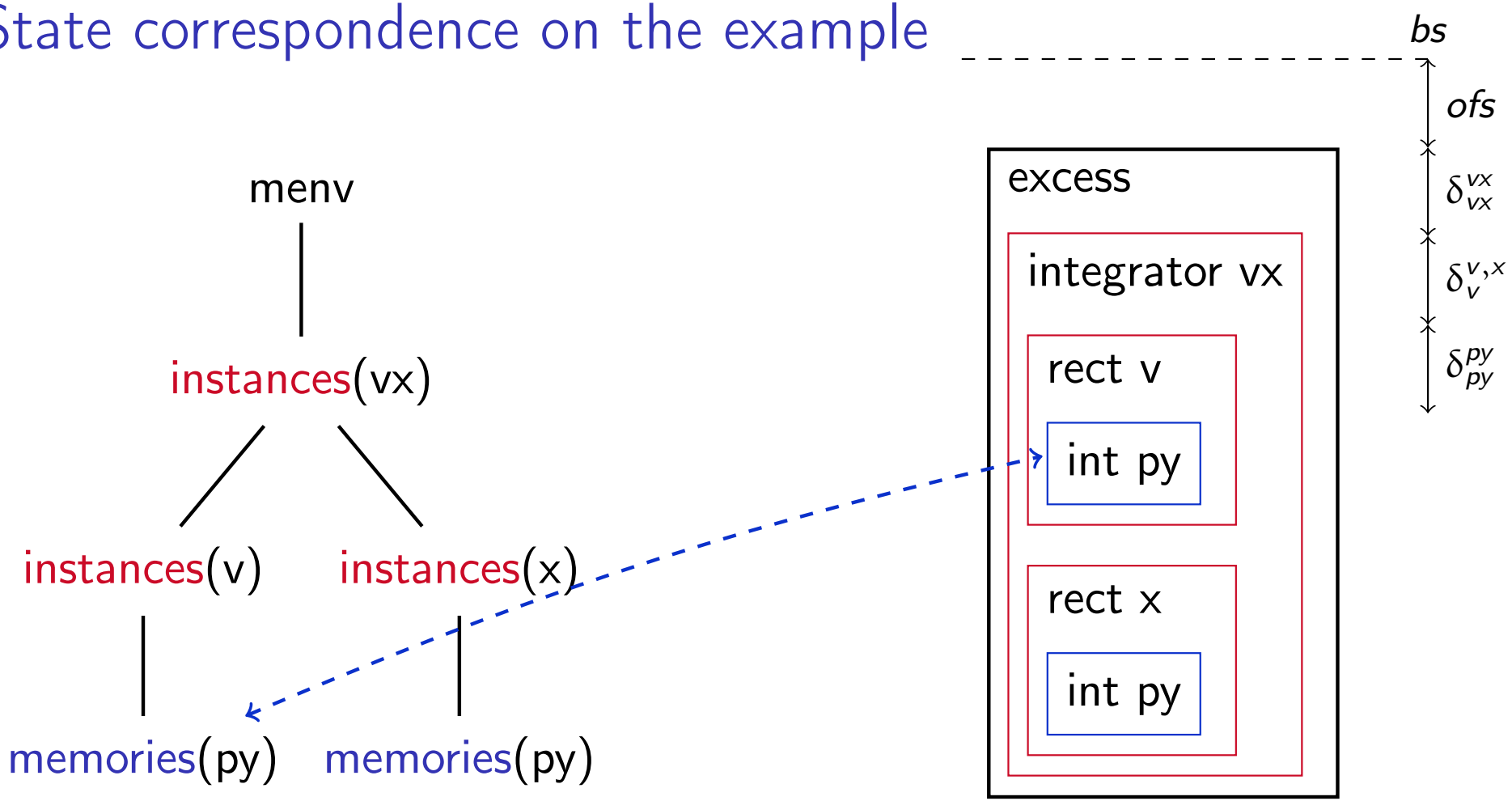
State correspondence on the example



$$\text{staterep } \mathit{rect}\ \mathit{menv}.\text{instances}(vx).\text{instances}(v)\ b_s\ (\mathit{ofs} + \delta^{vx}_{vx} + \delta^{v,x}_{v})$$

$$* \text{ staterep } \mathit{rect}\ \mathit{menv}.\text{instances}(vx).\text{instances}(x)\ b_s\ (\mathit{ofs} + \delta^{vx}_{vx} + \delta^{v,x}_{x})$$

State correspondence on the example

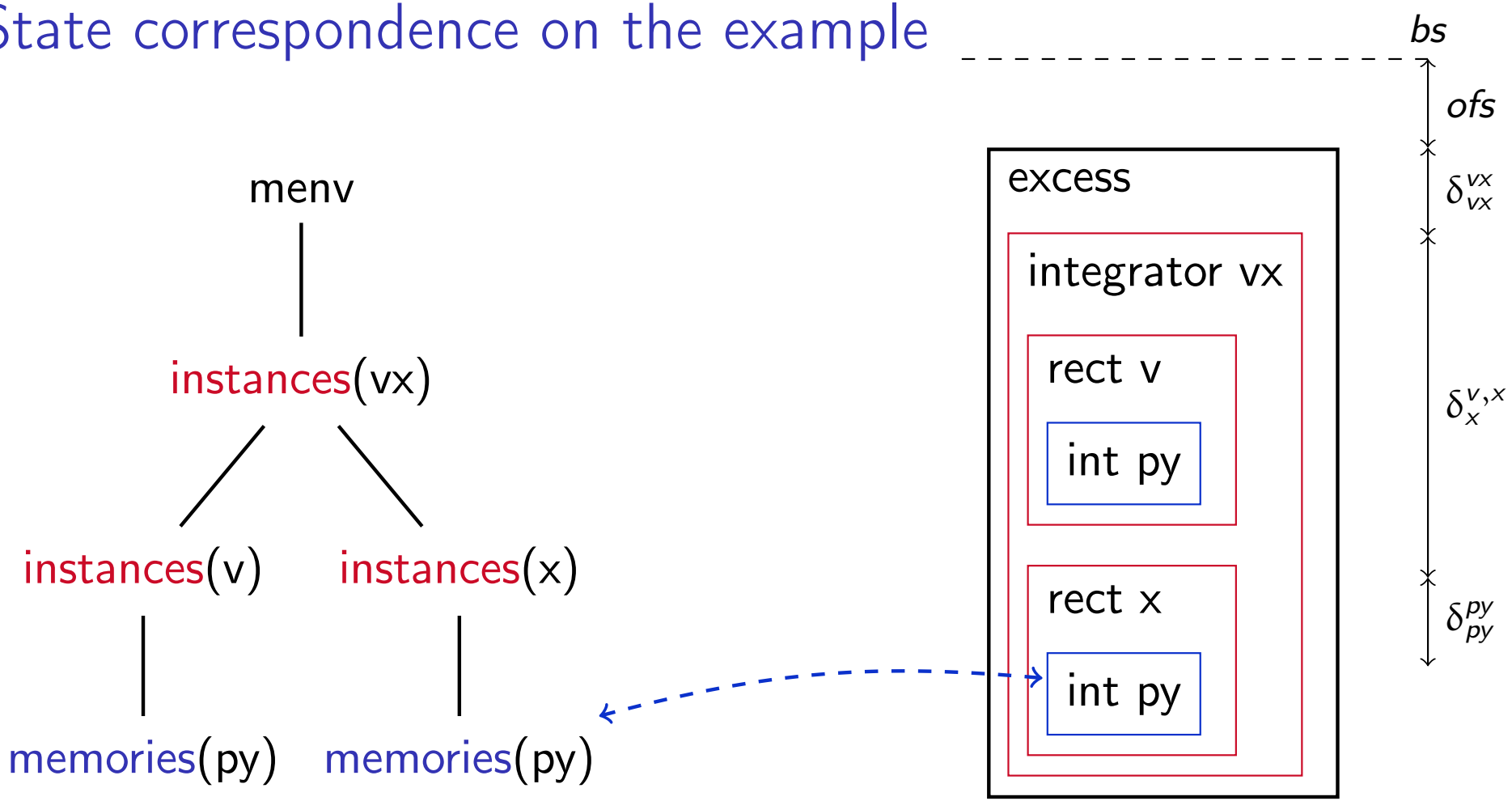


contains int32s b_s $(ofs + \delta^{vx}_{vx} + \delta^{v,x}_{v} + \delta^{py}_{py})$

$\lceil menv.\mathrm{instances}(vx).\mathrm{instances}(v).\mathrm{memories}(py) \rceil$

$*$ staterep *rect* $menv.\mathrm{instances}(vx).\mathrm{instances}(x)$ b_s $(ofs + \delta^{vx}_{vx} + \delta^{v,x}_{x})$

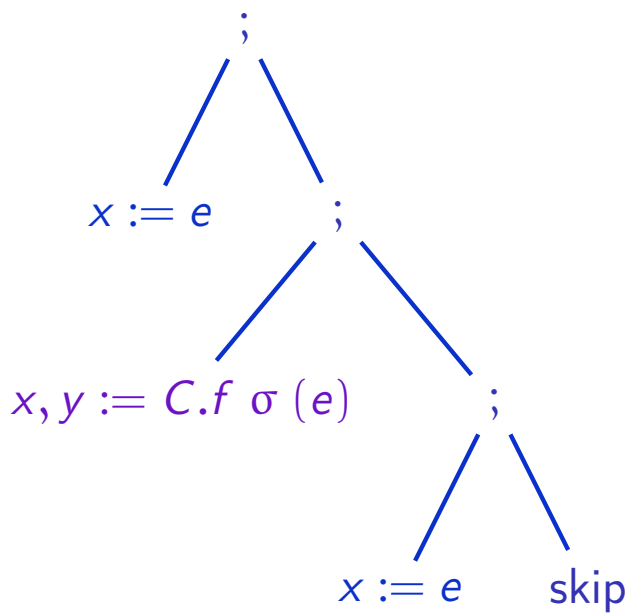
State correspondence on the example

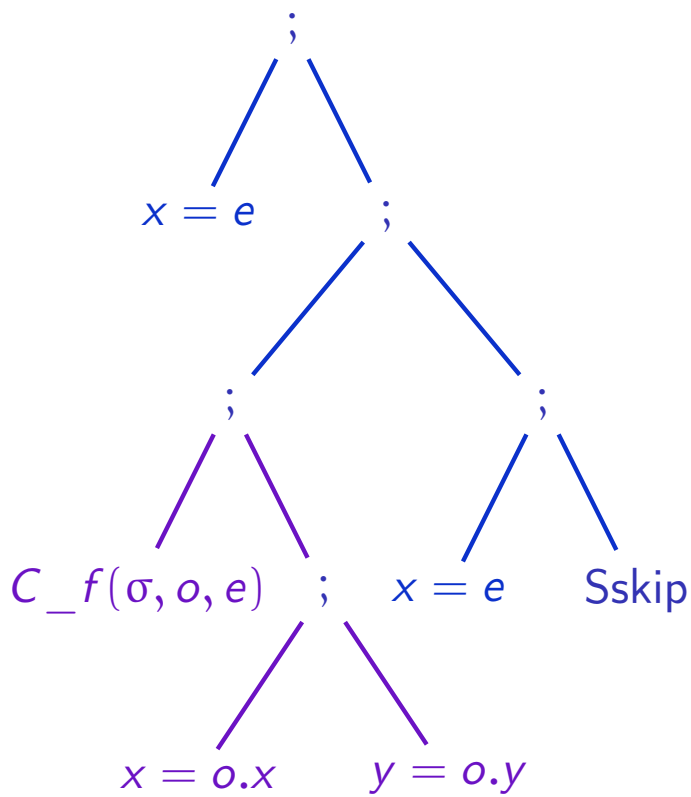


$$\text{contains int32s } b_s\ (ofs + \delta^{vx}_{vx} + \delta^{v,x}_{v} + \delta^{py}_{py})$$
$$\lceil menv.\text{instances}(vx).\text{instances}(v).\text{memories}(py) \rceil$$
$$* \ \text{contains int32s } b_s\ (ofs + \delta^{vx}_{vx} + \delta^{v,x}_{x} + \delta^{py}_{py})$$
$$\lceil menv.\text{instances}(vx).\text{instances}(x).\text{memories}(py) \rceil$$

Invariant preservation

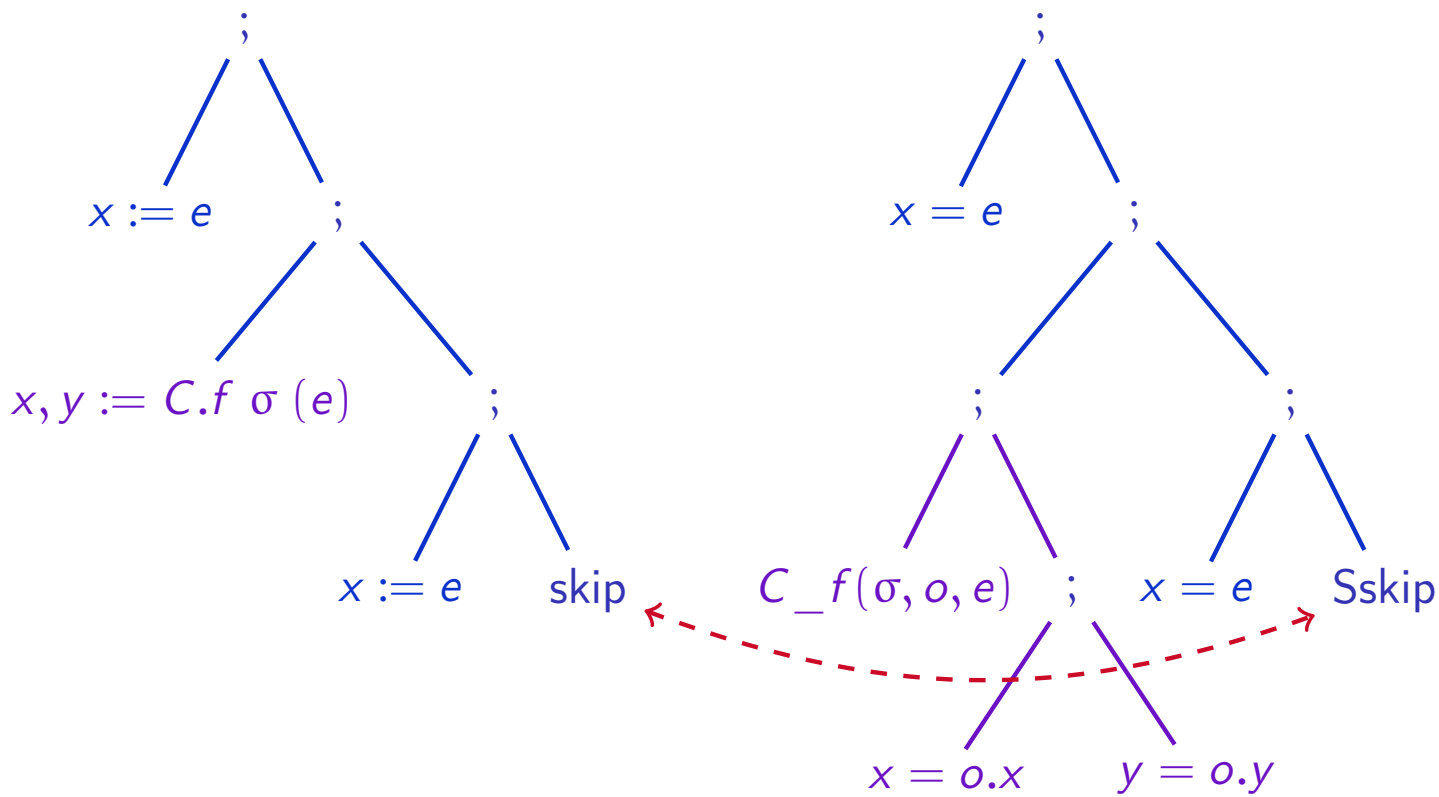
proof structure : simultaneous inductions (function calls, function bodies)





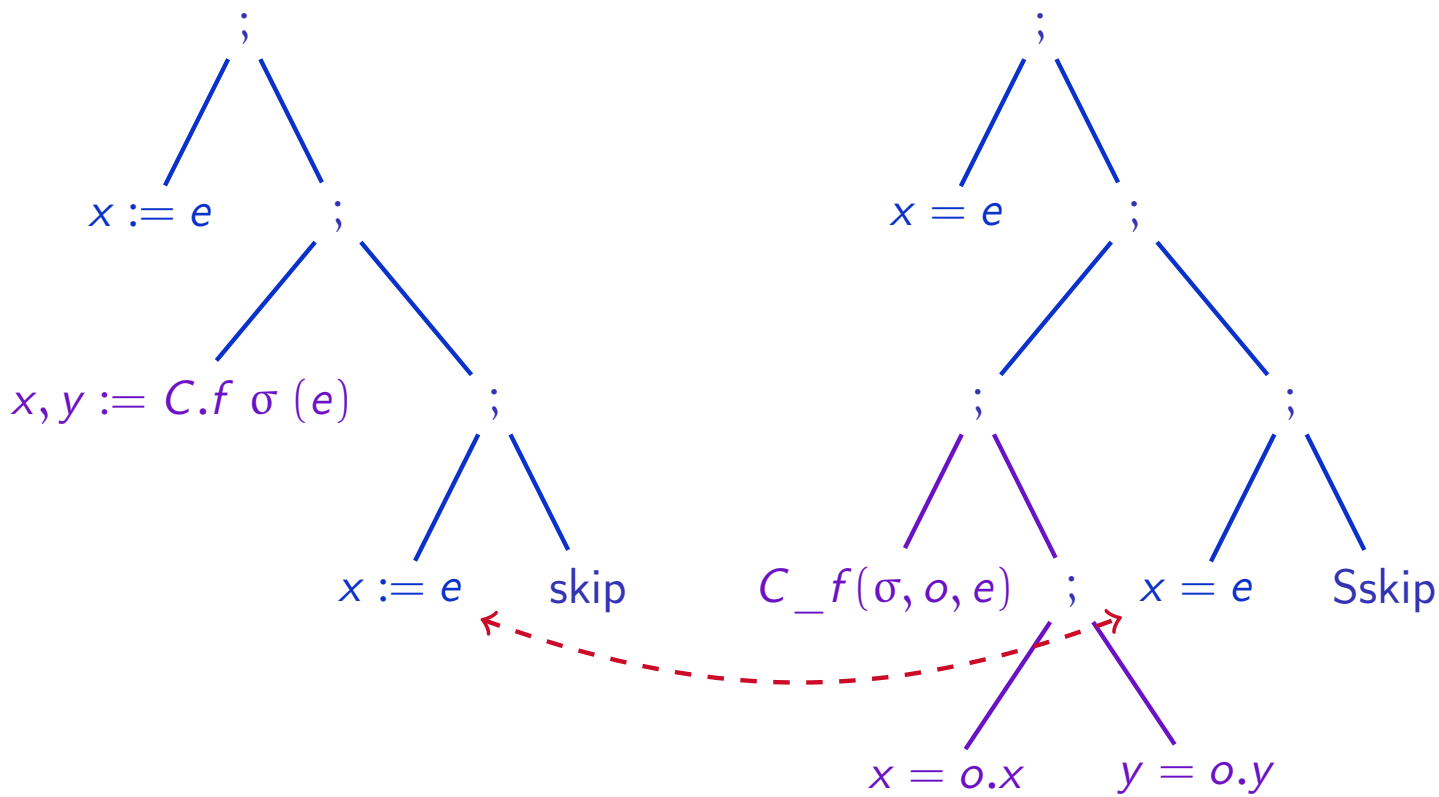
Invariant preservation

proof structure : simultaneous inductions (function calls, function bodies)



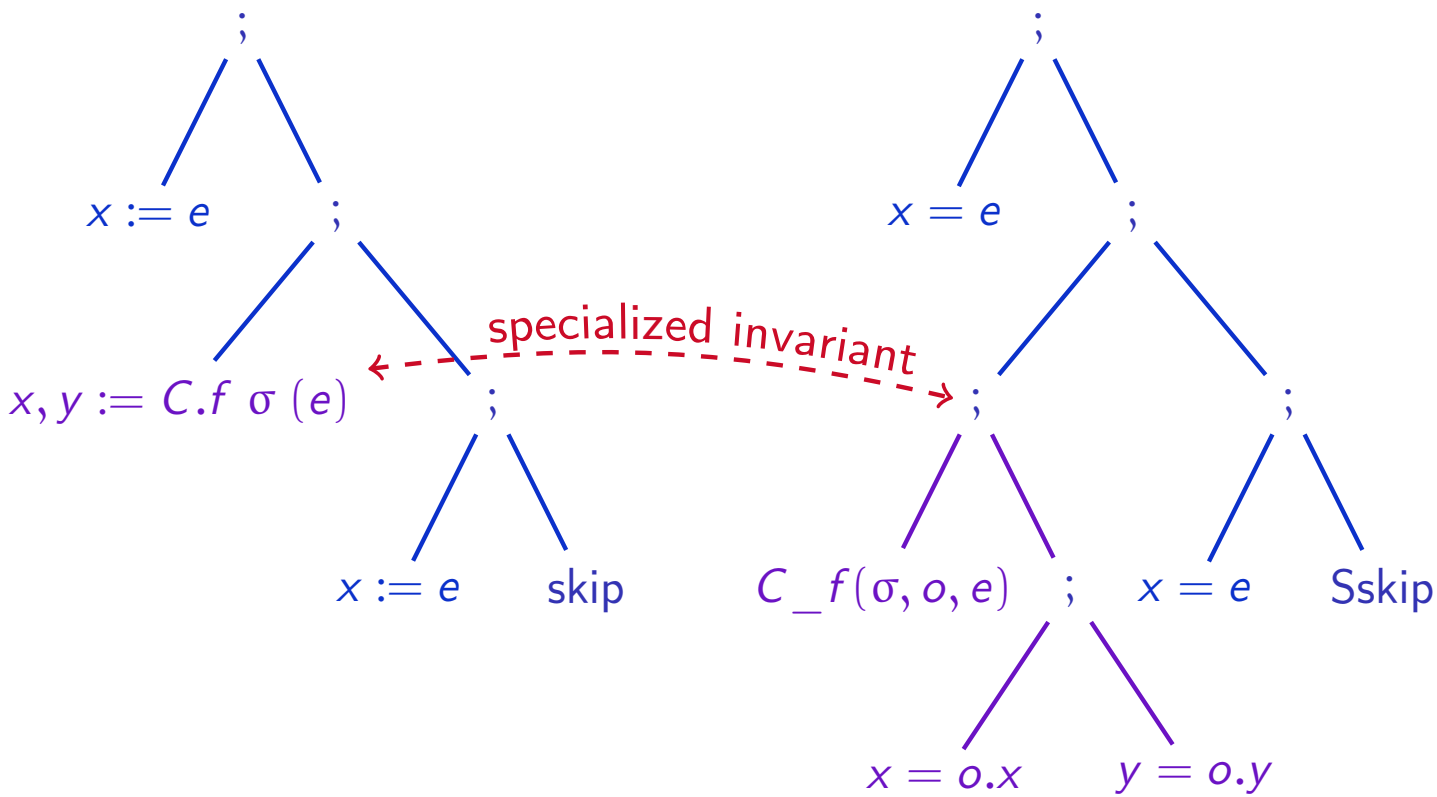
Invariant preservation

proof structure : simultaneous inductions (function calls, function bodies)



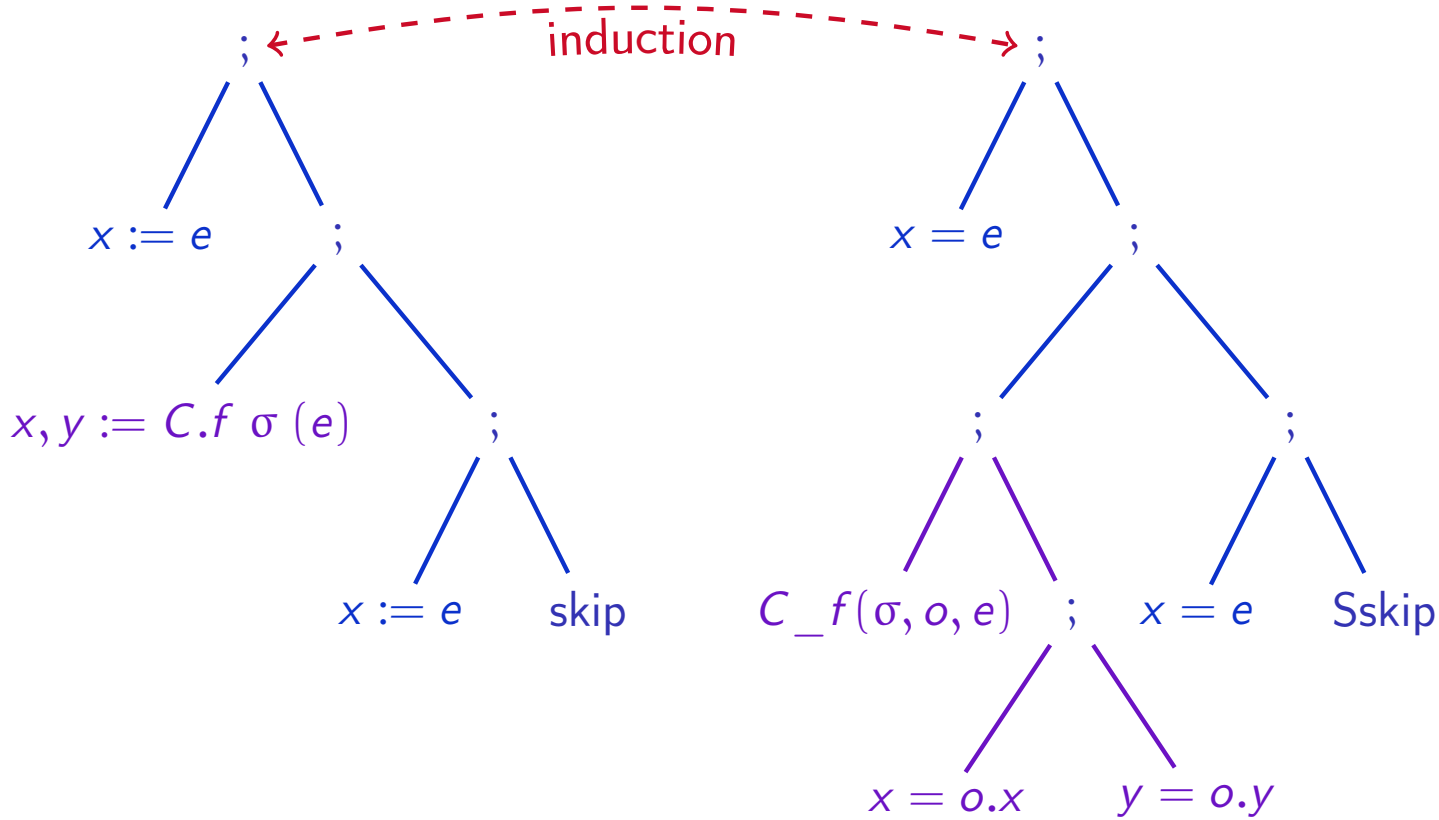
Invariant preservation

proof structure : simultaneous inductions (function calls, function bodies)



Invariant preservation

proof structure : simultaneous inductions (function calls, function bodies)

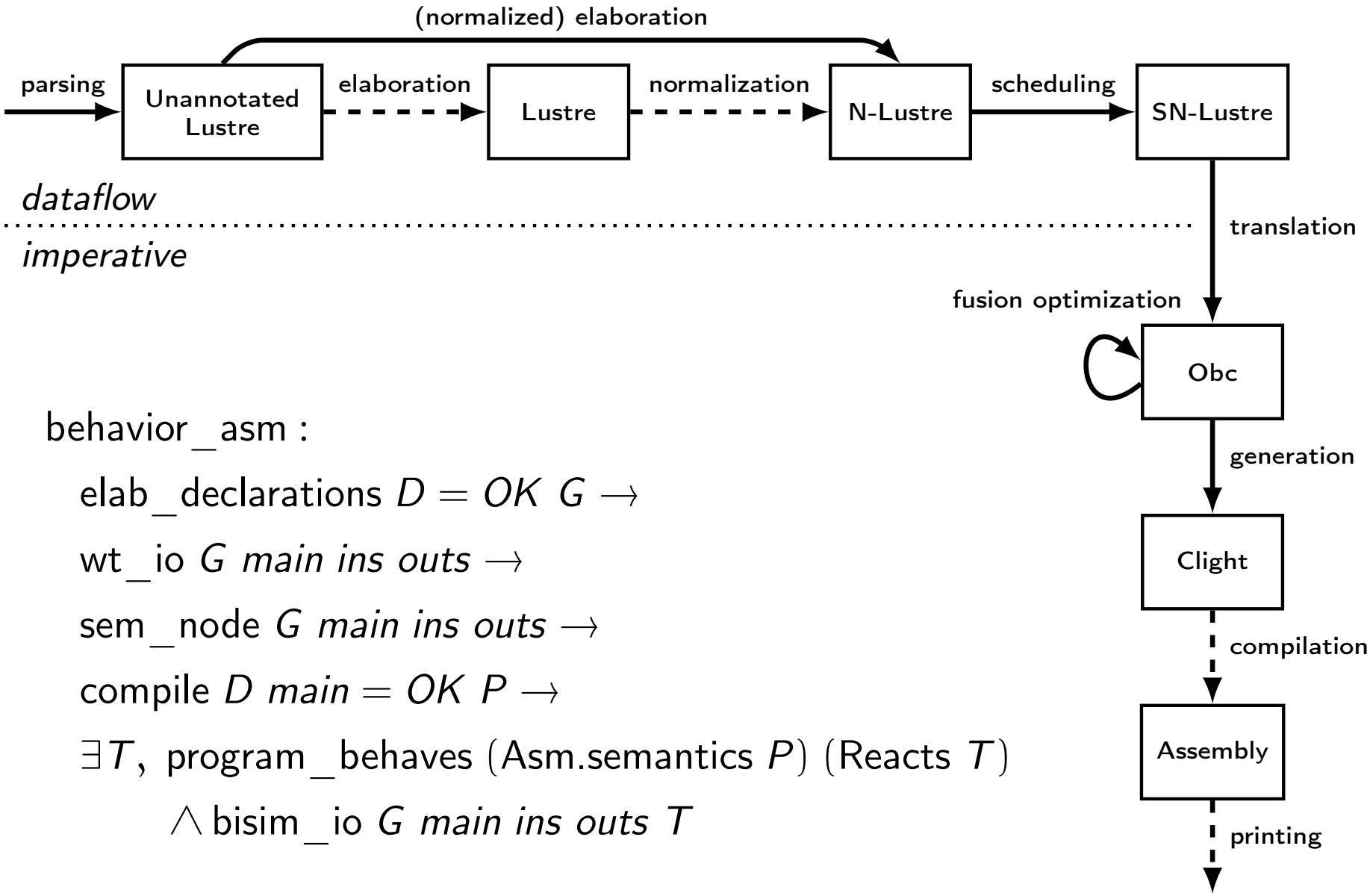


Summary

Obc $\mapsto$ Clight

- size:
  - translation: 400 loc
  - separation: 2100 loc
  - correctness: 4700 loc
- translation from tree-shaped memory towards contiguous blocks model
- memory models correspondence: separation logic
- verified Lustre compilation

Final lemma



behavior_asm :

elab_declarations $D = OK\ G \rightarrow$

wt_io G *main ins outs* $\rightarrow$

sem_node G *main ins outs* $\rightarrow$

compile D *main* $= OK\ P \rightarrow$

$\exists T,$ program_behaves (Asm.semantics P) (Reacts T)

$\wedge$ bisim_io G *main ins outs* T

Future work

- optimizations
- PhD: semantics, automata, reset
- enrich Obc (target for other source languages ?)

Other work

- synchronous languages, Lustre [Cas+87; Aug13; Ben+03; Bie+08; Aug+14; Bou+16]
- verified compilation: CompCert [BDL06; Ler09a; Ler09b]
- automatic proof of a compiler [CG15]
- denotational semantics [Chl07; BKV09; BH09]

References I

[Cas+87] Paul Caspi, Nicolas Halbwachs, Daniel Pilaud, and John Alexander Plaice. "LUSTRE: A declarative language for programming synchronous systems." In: *POPL'87*. ACM. Jan. 1987, pp. 178–188.

[The16] The Coq Development Team. *The Coq proof assistant reference manual*. Version 8.5. Inria. 2016. url: `http://coq.inria.fr`.

[JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. "Validating LR(1) parsers." In: *21st European Symposium on Programming (ESOP 2012), held as part of European Joint Conferences on Theory and Practice of Software (ETAPS 2012)*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Comp. Sci. Tallinn, Estonia: Springer, Mar. 2012, pp. 397–416.

[Aug13] Cédric Auger. "Compilation certifiée de SCADE/LUSTRE." PhD thesis. Orsay, France: Univ. Paris Sud 11, Apr. 2013.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal verification of a C compiler front-end." In: *FM 2006: Int. Symp. on Formal Methods* volume 4085 de LNCS (2006), pp. 460–475.

References II

[Ben+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Paul Le Guernic, Nicolas Halbwachs, and Robert De Simone. "The synchronous languages 12 years later." In: *proceedings of the IEEE*. Vol. 91. 1. Jan. 2003, pp. 178–188.

[Bie+08] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. "Clock-directed Modular Code Generation of Synchronous Data-flow Languages." In: *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. Tucson, Arizona, June 2008.

[Aug+14] Cédric Auger, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. "A Formalization and Proof of a Modular Lustre Code Generator." En préparation. 2014.

[Bou+16] Timothy Bourke, Pierre-Évariste Dagand, Marc Pouzet, and Lionel Rieg. "Verifying Clock-Directed Modular Code Generation for Lustre." En préparation. 2016.

[Ler09a] Xavier Leroy. "A formally verified compiler back-end." In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446.

References III

[Ler09b] Xavier Leroy. "Formal verification of a realistic compiler." In: *Comms. ACM* 52.7 (2009), pp. 107–115.

[CG15] Martin Clochard and Léon Gondelman. "Double WP : Vers une preuve automatique d'un compilateur." In: *Journées Francophones des Langages Applicatifs*. INRIA. Jan. 2015.

[Chl07] Adam Chlipala. "A certified type-preserving compiler from lambda calculus to assembly language." In: *Programming Language Design and Implementation*. ACM. 2007, pp. 54–65.

[BKV09] Nick Benton, Andrew Kennedy, and Carsten Varming. "Some domain theory and denotational semantics in Coq." In: *Theorem Proving in Higher Order Logics*. volume 5674 de LNCS. 2009, pp. 115–130.

[BH09] Nick Benton and Chung-Kil Hur. "Biorthogonality, step-indexing and compiler correctness." In: *International Conference on Functional Programming*. ACM. 2009, pp. 97–108.

Obc: Abstract Syntax

$e :=$ **expression**

$\mid$	x	(local variable)
$\mid$	$\mathsf{state}(x)$	(state variable)
$\mid$	c	(constant)
$\mid$	$\diamond\ e$	(unary operator)
$\mid$	$e \oplus e$	(binary operator)

$s :=$ **statement**

$\mid$	$x := e$	(update)
$\mid$	$\mathsf{state}(x) := e$	(state update)
$\mid$	if e then s else s	(conditional)
$\mid$	$\overrightarrow{x} := k(i).f(\overrightarrow{e})$	(method call)
$\mid$	$s;\ s$	(composition)
$\mid$	skip	(do nothing)

$cls :=$ **declaration**

$\mid$	class k {	(class)
	memory $\overrightarrow{x^{ty}}$	
	instance $\overrightarrow{i^{k}}$	
	$f(\overrightarrow{x^{ty}})$ returns $(\overrightarrow{x^{ty}})$ [var $\overrightarrow{x^{ty}}$] { s }	
	}	

Separation logic in CompCert

predicate

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

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

Separation logic in CompCert

predicate

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

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

conjonction

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

Separation logic in CompCert

predicate

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

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

conjonction

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

pure formula

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

States correspondence

Obc : (me, ve), $f \in c \in p$
Clight : (e, le, m)

$$\text{match\_states} \triangleq$$

States correspondence

Obc : $(me, ve),\ f \in c \in p$
Clight : (e, le, m)

$$\begin{aligned}
&\text{match_states} \triangleq \\
&\qquad \text{pure}\,(le(\text{self}) = (b_s,\ ofs)) && \textit{self} \text{ pointer} \\
&\qquad * \text{pure}\,(le(\text{out}) = (b_o,\ 0)) && \textit{out} \text{ pointer} \\
&\qquad * \text{pure}\,(\text{ge}(\text{f_c}) = co_{out}) && \text{output structure}
\end{aligned}$$

States correspondence

Obc : (me, ve), $f \in c \in p$
Clight : (e, le, m)

match_states $\triangleq$

$$\begin{aligned}
&\text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&* \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&* \text{pure}\,(\text{ge}(\text{f_c}) = co_{out}) \\
&* \text{pure}\,(\text{wt_env}\ ve\ (\text{all_vars_of}\ f)) \\
&* \text{pure}\,(\text{wt_mem}\ me\ p\ c)
\end{aligned}$$

the Obc state is well-typed wrt. the context

States correspondence

Obc : (me, ve), $f \in c \in p$
Clight : (e, le, m)

match_states $\triangleq$

$$\begin{aligned}
&\text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&* \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&* \text{pure}\,(\text{ge}(\text{f_c}) = co_{out}) \\
&* \text{pure}\,(\text{wt_env}\ ve\ (\text{all_vars_of}\ f)) \\
&* \text{pure}\,(\text{wt_mem}\ me\ p\ c) \\
&* \text{staterep}\ p\ c\ me\ b_s\ ofs
\end{aligned}$$

memory $me \approx$
structure pointed by *self*

States correspondence

Obc : (me, ve), $f \in c \in p$
Clight : (e, le, m)

match_states $\triangleq$

$$\begin{aligned}
&\text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&* \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&* \text{pure}\,(\text{ge}(\text{f_c}) = co_{out}) \\
&* \text{pure}\,(\text{wt_env}\ ve\ (\text{all_vars_of}\ f)) \\
&* \text{pure}\,(\text{wt_mem}\ me\ p\ c) \\
&* \text{staterep}\ p\ c\ me\ b_s\ ofs \\
&* \text{blockrep}\ ve\ co_{out}\ b_o
\end{aligned}$$

output of $f \approx$
co_{out} pointed by out

States correspondence

Obc : (me, ve), $f \in c \in p$
Clight : (e, le, m)

match_states $\triangleq$

$$
\begin{aligned}
&\text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&* \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&* \text{pure}\,(\text{ge}(\text{f\_c}) = co_{out}) \\
&* \text{pure}\,(\text{wt\_env}\ ve\ (\text{all\_vars\_of}\ f)) \\
&* \text{pure}\,(\text{wt\_mem}\ me\ p\ c) \\
&* \text{staterep}\ p\ c\ me\ b_s\ ofs \\
&* \text{blockrep}\ ve\ co_{out}\ b_o \\
&* \text{varsrep}\ f\ ve\ le
\end{aligned}
$$

parameters and local variables $\approx$ temporaries

States correspondence

Obc : (me, ve), $f \in c \in p$
Clight : (e, le, m)

match_states $\triangleq$

$\quad$ pure $(le(\text{self}) = (b_s,\ ofs))$

$\quad$ $*$ pure $(le(\text{out}) = (b_o,\ 0))$

$\quad$ $*$ pure $(\text{ge}(\text{f\_c}) = co_{out})$

$\quad$ $*$ pure $(\text{wt\_env}\ ve\ (\text{all\_vars\_of}\ f))$

$\quad$ $*$ pure $(\text{wt\_mem}\ me\ p\ c)$

$\quad$ $*$ staterep $p\ c\ me\ b_s\ ofs$

$\quad$ $*$ blockrep $ve\ co_{out}\ b_o$

$\quad$ $*$ varsrep $f\ ve\ le$

$\quad$ $*$ subrep_range e

subcalls output
structures allocation