Verifying a Lustre Compiler Part 2

Lélio Brun

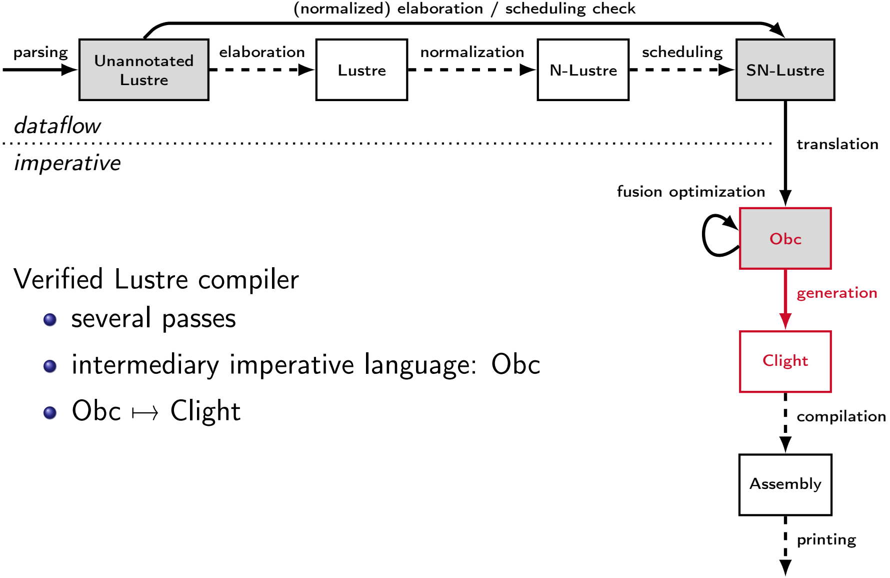
PARKAS (Inria - ENS)

Timothy Bourke, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, Lionel Rieg

SYNCHRON 2016

December 7, 2016

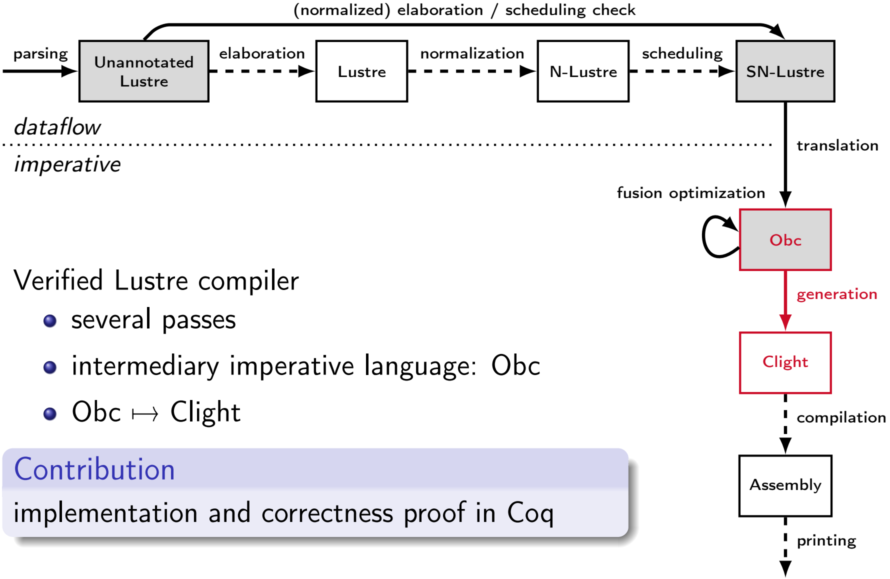
Context



Verified Lustre compiler

- several passes
- intermediary imperative language: Obc
- Obc $\mapsto$ Clight

Context



Verified Lustre compiler

- several passes
- intermediary imperative language: Obc
- Obc $\mapsto$ Clight

Contribution

implementation and correctness proof in Coq

Obc: Abstract Syntax

$e :=$		**expression**
$\mid$	x	(local variable)
$\mid$	$\text{state}(x)$	(state variable)
$\mid$	c	(constant)
$\mid$	$\diamond\ e$	(unary operator)
$\mid$	$e \oplus e$	(binary operator)

$s :=$		**statement**
$\mid$	$x := e$	(update)
$\mid$	$\text{state}(x) := e$	(state update)
$\mid$	$\text{if } e \text{ then } s \text{ else } s$	(conditional)
$\mid$	$\overrightarrow{x} := c(i).m(\overrightarrow{e})$	(method call)
$\mid$	$s;\ s$	(composition)
$\mid$	skip	(do nothing)

$cls :=$ **declaration**

$\mid$ class c { (class)

memory $\overrightarrow{x^{ty}}$

instance $\overrightarrow{i^{c}}$

$m(\overrightarrow{x^{ty}})$ returns $(\overrightarrow{x^{ty}})$ [var $\overrightarrow{x^{ty}}$] { s }

}

Example

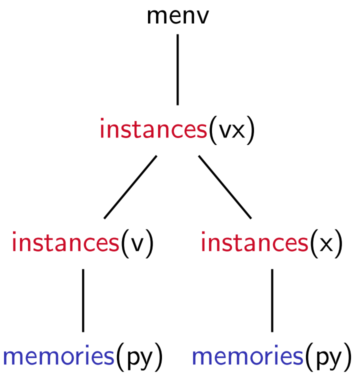
```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel

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 = v > max;
tel
```

```
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 := v > max
  }
}
```

State and memory model

$$venv \triangleq ident \rightarrow val$$

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



Statements rules

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

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

$$\ldots$$

Clight

- CompCert's frontend language
- block memory model
- 2 types of variables: local and temporaries
- 2 semantics variants: parameters as local variables or as temporaries
- 2 semantics: small and big step
  - small step: continuations
  - big step: state (e, le, m)

e local variables environment : $ident \longrightarrow block * int$

le temporaries environment : $ident \longrightarrow val$

m memory : $block \longrightarrow int \longrightarrow byte$

Generation function

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

Example

```
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
  }
}
```

```
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 = sx_step.x;
  (*out).e = v > max;
}
```

Semantics preservation

Obc : (me, ve) ; Clight : (e, le, m)

$$\begin{array}{ccc} me_1, ve_1 & \vdash_{\text{st}} s \Downarrow & me_2, ve_2 \\ \textit{match\_states} \wr & & \\ e_1, le_1, m_1 & & \end{array}$$

Semantics preservation

Obc : (me, ve) ; Clight : (e, le, m)

$$
\begin{array}{ccc}
me_1, ve_1 & \vdash_{st} s \Downarrow & me_2, ve_2 \\
\text{match\_states} \wr & & \wr\ \text{match\_states} \\
e_1, le_1, m_1 & \vdash_{Clight} |s|_s \Downarrow & e_1, le_2, m_2
\end{array}
$$

Separation logic

Consequences of CompCert's memory model:

- aliasing (overlapping)
- alignment
- permissions
- sizes

Separation logic

Consequences of CompCert's memory model:

- aliasing (overlapping)
- alignment
- permissions
- sizes

Solution

use a separation logic formalism

Separation logic in CompCert

predicate $\mathrm{P} : \begin{cases} \mathrm{mfoot} : \mathit{block} \rightarrow \mathit{int} \rightarrow \mathbb{P} \\ \mathrm{mpred} : \mathit{memory} \rightarrow \mathbb{P} \end{cases}$ $, m \vDash P \equiv (\mathrm{mpred}\ P)\ m$

conjonction $m \vDash P * Q$

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

Separation logic in CompCert

predicate $\mathrm{P} : \begin{cases} \mathrm{mfoot} : \mathit{block} \to \mathit{int} \to \mathbb{P} \\ \mathrm{mpred} : \mathit{memory} \to \mathbb{P} \end{cases}, m \vDash P \equiv (\mathrm{mpred}\ P)\ m$

conjonction $m \vDash P * Q$

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

$$P * Q = \begin{cases} \mathrm{mfoot} = \lambda\, b\, \mathit{ofs}.\ \mathrm{mfoot}\ P\ b\ \mathit{ofs} \vee \mathrm{mfoot}\ Q\ b\ \mathit{ofs} \\ \mathrm{mpred} = \lambda\, m.\ \mathrm{mpred}\ P\ m \wedge \mathrm{mpred}\ Q\ m \\ \qquad\qquad \wedge\, \mathrm{disjoint}(\mathrm{mfoot}\ P)\ (\mathrm{mfoot}\ Q) \end{cases}$$

States correspondence

Obc : (me, ve) ; Clight : (e, le, m)

$$\text{match\_states} \; =$$

States correspondence

Obc : (me, ve) ; Clight : (e, le, m)

$$\begin{aligned}\text{match\_states} = \; & \text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\ & * \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\ & * \text{pure}\,(\text{ge}(\text{f\_c}) = co_{out})\end{aligned}$$

self pointer

out pointer

output structure

States correspondence

Obc : (me, ve) ; Clight : (e, le, m)

$$
\begin{aligned}
\text{match\_states} \; = \; & \\
& \text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
& * \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
& * \text{pure}\,(\text{ge}(\text{f\_c}) = co_{out}) \\
& * \text{pure}\,(\text{wt\_env}\ ve\ m) \\
& * \text{pure}\,(\text{wt\_mem}\ me\ p\ c)
\end{aligned}
$$

States correspondence

Obc : (me, ve) ; Clight : (e, le, m)

$$
\begin{aligned}
\text{match\_states} \; = \;& \\
& \text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
& * \,\text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
& * \,\text{pure}\,(\text{ge}(\text{f\_c}) = co_{out}) \\
& * \,\text{pure}\,(\text{wt\_env}\ ve\ m) \\
& * \,\text{pure}\,(\text{wt\_mem}\ me\ p\ c) \\
& * \,\text{staterep}\ p\ c\ me\ b_s\ ofs
\end{aligned}
$$

memory *me* ≈ structure pointed by *self*

States correspondence

Obc : (me, ve) ; Clight : (e, le, m)

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

output variables of m $\approx$ fields of co_{out} pointed by *out*

States correspondence

Obc : (me, ve) ; Clight : (e, le, m)

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

parameters and local variables ≈ temporaries

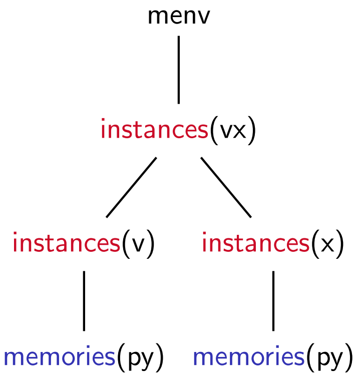
States correspondence

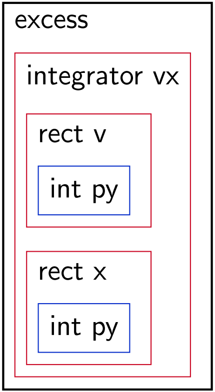
Obc : (me, ve) ; Clight : (e, le, m)

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

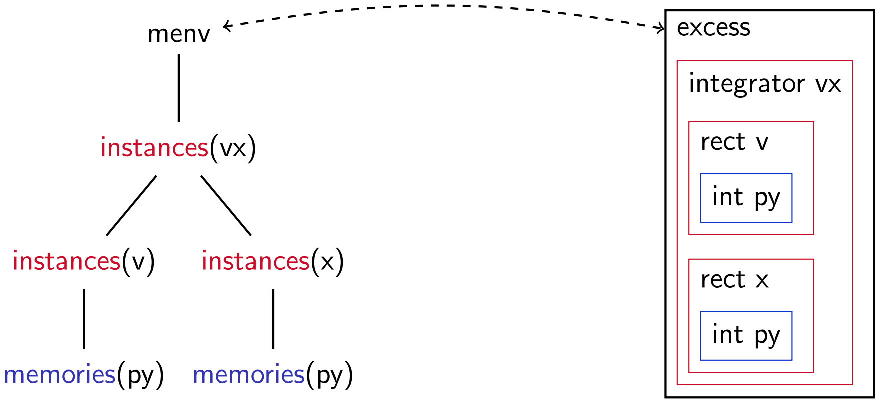
subcalls output
structures allocation

Staterep on the example



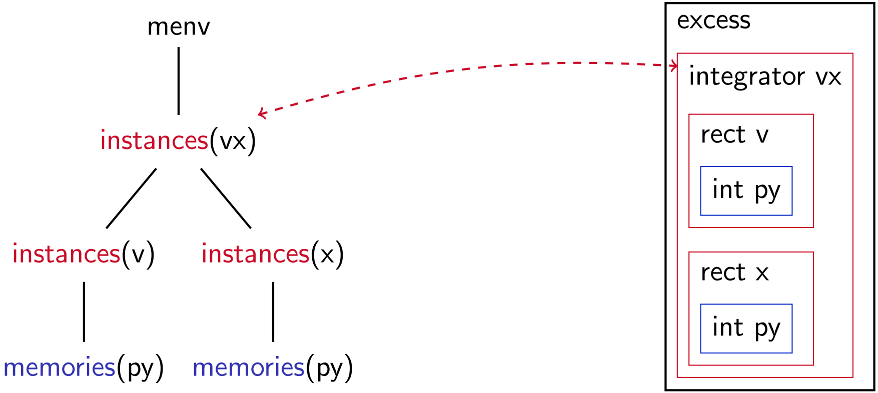


Staterep on the example



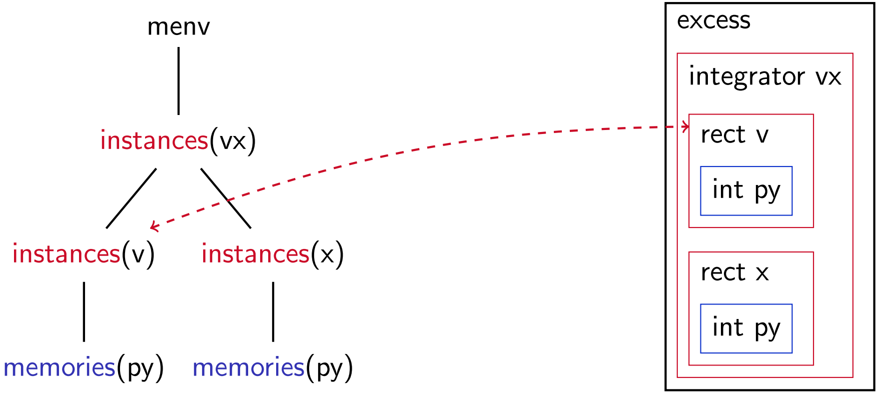
stateref *excess menv b_s ofs*

Staterep on the example



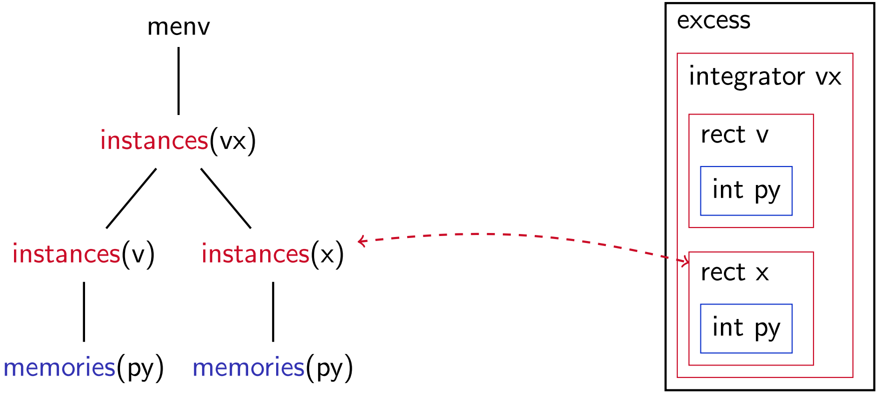
$$\text{staterep } \mathit{integrator}\ \mathit{menv}.\text{instances}(\mathit{vx})\ b_s\ (\mathit{ofs} + \delta_{vx}^{excess})$$

Staterep on the example



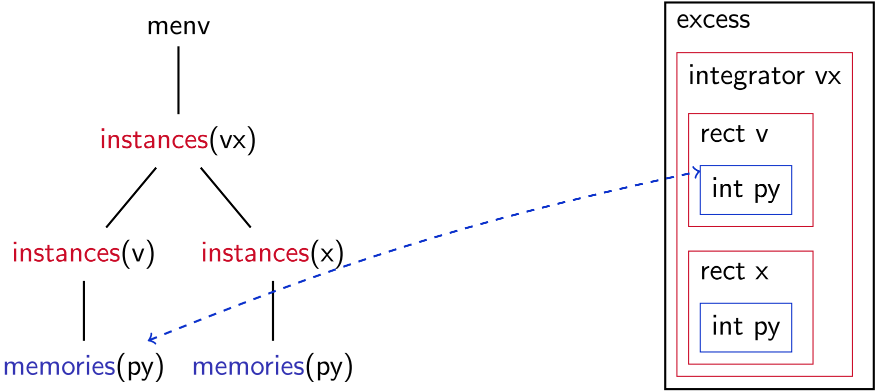
staterep *rect* *menv*.instances(vx).instances(v) b_s $(ofs + \delta_{vx}^{excess} + \delta_{v}^{integrator})$

Staterep on the example



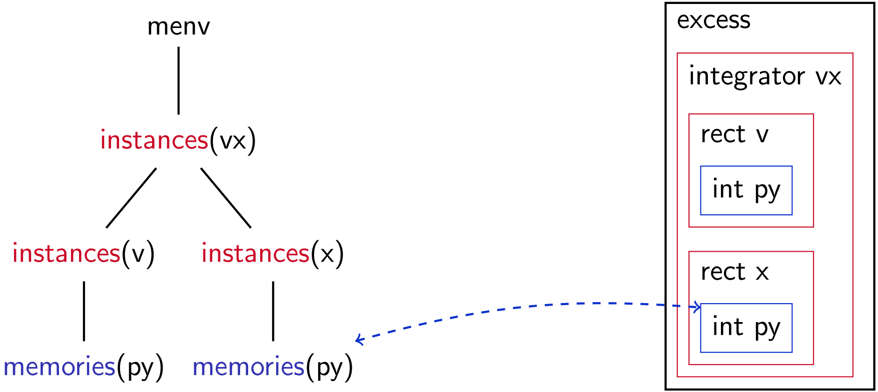
staterep $rect$ $menv$.instances(vx).instances(x) b_s $(ofs + \delta_{vx}^{excess} + \delta_{x}^{integrator})$

Staterep on the example



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

Staterep on the example



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

Staterep

$$\text{staterep}\ []\ c\ me\ b_s\ ofs = \bot^*$$

$$\text{staterep}\ (\text{class}\ k\{\dots\} :: p)\ c\ me\ b_s\ ofs = \text{staterep}\ p\ c\ me\ b_s\ ofs \qquad \text{if } k \neq c$$

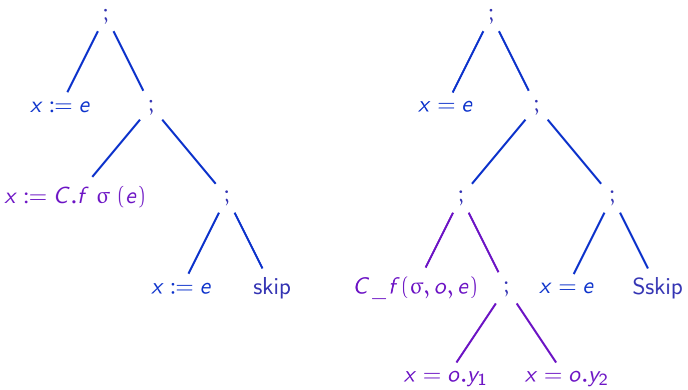
$$\text{staterep}\ (\text{class}\ c\{\overrightarrow{x^{ty}}\ \ \overrightarrow{i^{k}}\ \dots\} :: p)\ c\ me\ b_s\ ofs =$$

$$\mathop{\Huge *}_{\overrightarrow{x^{ty}}} \text{contains}\ ty\ b_s\ (ofs + \text{field\_offset}(x, \overrightarrow{x^{ty}} \cdot \overrightarrow{i^{k}}))\ \lceil me.\text{memories}(x) \rceil$$

$$\mathop{\Huge *}_{\overrightarrow{i^{k}}} \text{staterep}\ p\ l\ me.\text{instances}(k)\ b_s\ (ofs + \text{field\_offset}(i, \overrightarrow{x^{ty}} \cdot \overrightarrow{i^{k}}))$$

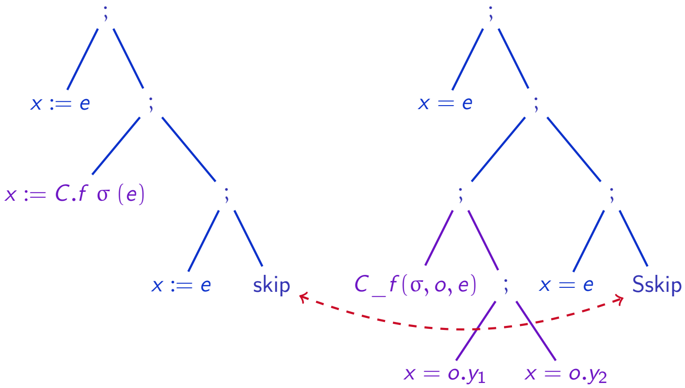
Invariant preservation

- frame rule "emulation":
  $m \vDash P * F \longrightarrow \text{hypotheses} \longrightarrow \exists m', \text{properties} \wedge m' \vDash P' * F$
- proof structure : simultaneous inductions (function calls, function bodies)



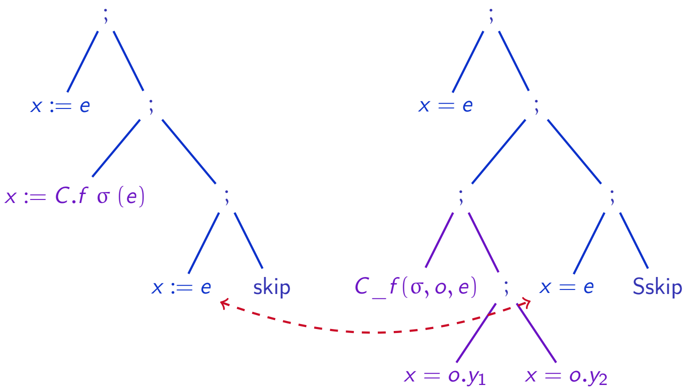
Invariant preservation

- frame rule "emulation":
  $m \vDash P * F \rightarrow \text{hypotheses} \rightarrow \exists m', \text{properties} \wedge m' \vDash P' * F$
- proof structure : simultaneous inductions (function calls, function bodies)



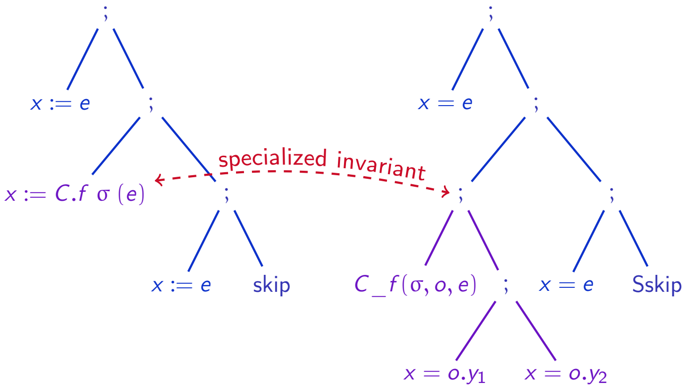
Invariant preservation

- frame rule "emulation":
  $m \vDash P * F \rightarrow \text{hypotheses} \rightarrow \exists m', \text{properties} \wedge m' \vDash P' * F$
- proof structure : simultaneous inductions (function calls, function bodies)



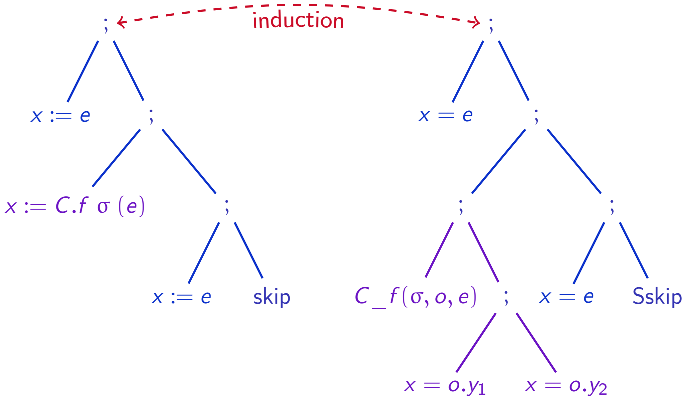
Invariant preservation

- frame rule "emulation":
  $m \vDash P * F \rightarrow \text{hypotheses} \rightarrow \exists m', \text{properties} \wedge m' \vDash P' * F$
- proof structure : simultaneous inductions (function calls, function bodies)



Invariant preservation

- frame rule "emulation":
  $m \models P * F \rightarrow \text{hypotheses} \rightarrow \exists m', \text{properties} \wedge m' \models P' * F$
- proof structure : simultaneous inductions (function calls, function bodies)



Proof case

$$
\begin{array}{ccc}
me, ve & \vdash_{\mathsf{st}} \mathsf{state}(x) := a \Downarrow & me', ve \\
\text{match\_states} \wr & & \wr \text{match\_states} \\
e, le, m & \vdash_{\mathsf{s}} (^*\mathsf{self}).x = |a|_{\mathsf{e}} \Downarrow & e, le, m'
\end{array}
$$

Proof case

$$\begin{array}{ccc} me, ve & \vdash_{st} \mathsf{state}(x) := a \Downarrow & me', ve \\ \text{match\_states} \wr & & \wr \text{match\_states} \\ e, le, m & \vdash_s (^*\mathsf{self}).x = |a|_e \Downarrow\!\Downarrow & e, le, m' \end{array}$$

- $$\dfrac{\dfrac{\dfrac{le(\mathsf{self}) = (b_s,\ ofs)}{\ldots}}{e, le, m \vdash_e\ ^*\mathsf{self} \Downarrow\!\Downarrow (b_s,\ ofs)} \qquad \dfrac{\text{match\_states}}{\mathsf{field\_offset}(x, \ldots) = \delta_x}\ \text{stateerep\_field\_offset}}{e, le, m \vdash_{lv} (^*\mathsf{self}).x \Downarrow\!\Downarrow b_s, ofs + \delta_x}$$

Proof case

$$\begin{array}{ccc} me, ve & \vdash_{st} \mathsf{state}(x) := a \Downarrow & me', ve \\ \text{match\_states} \wr & & \wr \text{match\_states} \\ e, le, m & \vdash_{s} (^*\mathsf{self}).x = |a|_{e} \Downarrow & e, le, m' \end{array}$$

-
$$\dfrac{\dfrac{\dfrac{le(\mathsf{self}) = (b_s,\ ofs)}{\ldots}}{e, le, m \vdash_{e}\ ^*\mathsf{self} \Downarrow (b_s,\ ofs)} \qquad \dfrac{\text{match\_states}}{\mathsf{field\_offset}(x, \ldots) = \delta_x}\ \text{stateerep\_field\_offset}}{e, le, m \vdash_{lv} (^*\mathsf{self}).x \Downarrow b_s, ofs + \delta_x}$$

-
$$\dfrac{me, ve \vdash_{exp} a \Downarrow v}{e, le, m \vdash_{e} |a|_{e} \Downarrow v}\ \text{expr\_eval\_simu}$$

Proof case

$$\begin{array}{ccc} me, ve & \vdash_{st} \text{state}(x) := a \Downarrow & me', ve \\ \text{match\_states} \wr & & \wr \text{match\_states} \\ e, le, m & \vdash_s (^*\text{self}).x = |a|_e \Downarrow & e, le, m' \end{array}$$

- $$\dfrac{\dfrac{\dfrac{le(\text{self}) = (b_s,\ ofs)}{\ldots}}{e, le, m \vdash_e {}^*\text{self} \Downarrow (b_s,\ ofs)} \quad \dfrac{\text{match\_states}}{\text{field\_offset}(x, \ldots) = \delta_x}\ \text{stateref\_field\_offset}}{e, le, m \vdash_{lv} (^*\text{self}).x \Downarrow b_s, ofs + \delta_x}$$

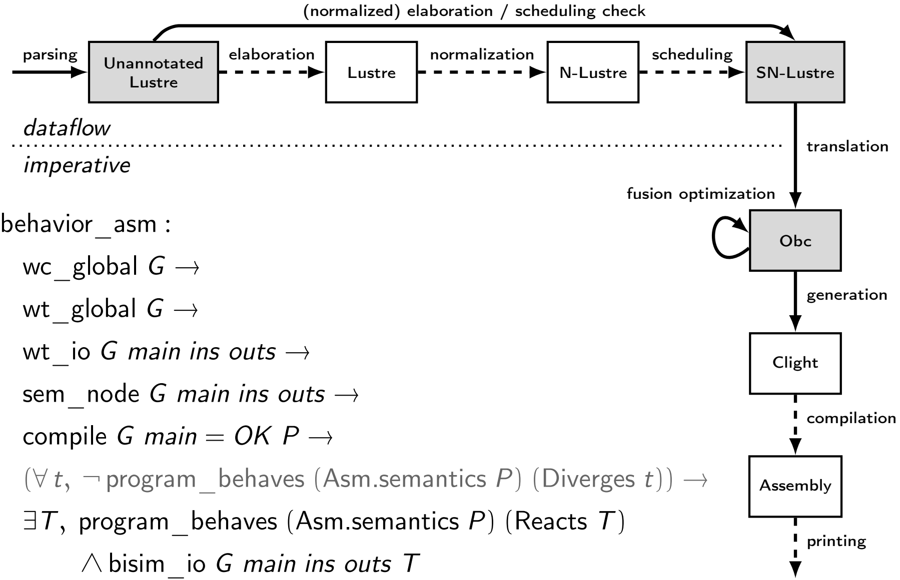
- $$\dfrac{me, ve \vdash_{exp} a \Downarrow v}{e, le, m \vdash_e |a|_e \Downarrow v}\ \text{expr\_eval\_simu}$$

- $$\dfrac{\text{match\_states}}{\text{store}(m, b_s, ofs + \delta_x, v) = m'}\ \text{match\_states\_assign\_state}$$

Summary

- size:
  - translation: 300 loc
  - separation: 2000 loc
  - correctness: 3300 loc
- memory models correspondence: separation logic
- other source languages
- certified Lustre compilation

Final lemma



behavior_asm :

wc_global $G \rightarrow$

wt_global $G \rightarrow$

wt_io G *main ins outs* $\rightarrow$

sem_node G *main ins outs* $\rightarrow$

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

$(\forall t, \neg$ program_behaves (Asm.semantics P) (Diverges t)) $\rightarrow$

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

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

Future Works

- complete the toolchain
- optimizations
- PhD: semantics, automata, reset

Other works

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

References I

[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.

[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.

[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.

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

[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.

References II

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

[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.

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

[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.

References III

[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.