La vérification formelle d'un compilateur Lustre

Timothy Bourke[1,2] Lélio Brun[1,2] Pierre-Évariste Dagand[4,3,1]

Xavier Leroy[1] Marc Pouzet[4,2,1] Lionel Rieg[5,6]

1. Inria Paris

2. DI, École normale supérieure

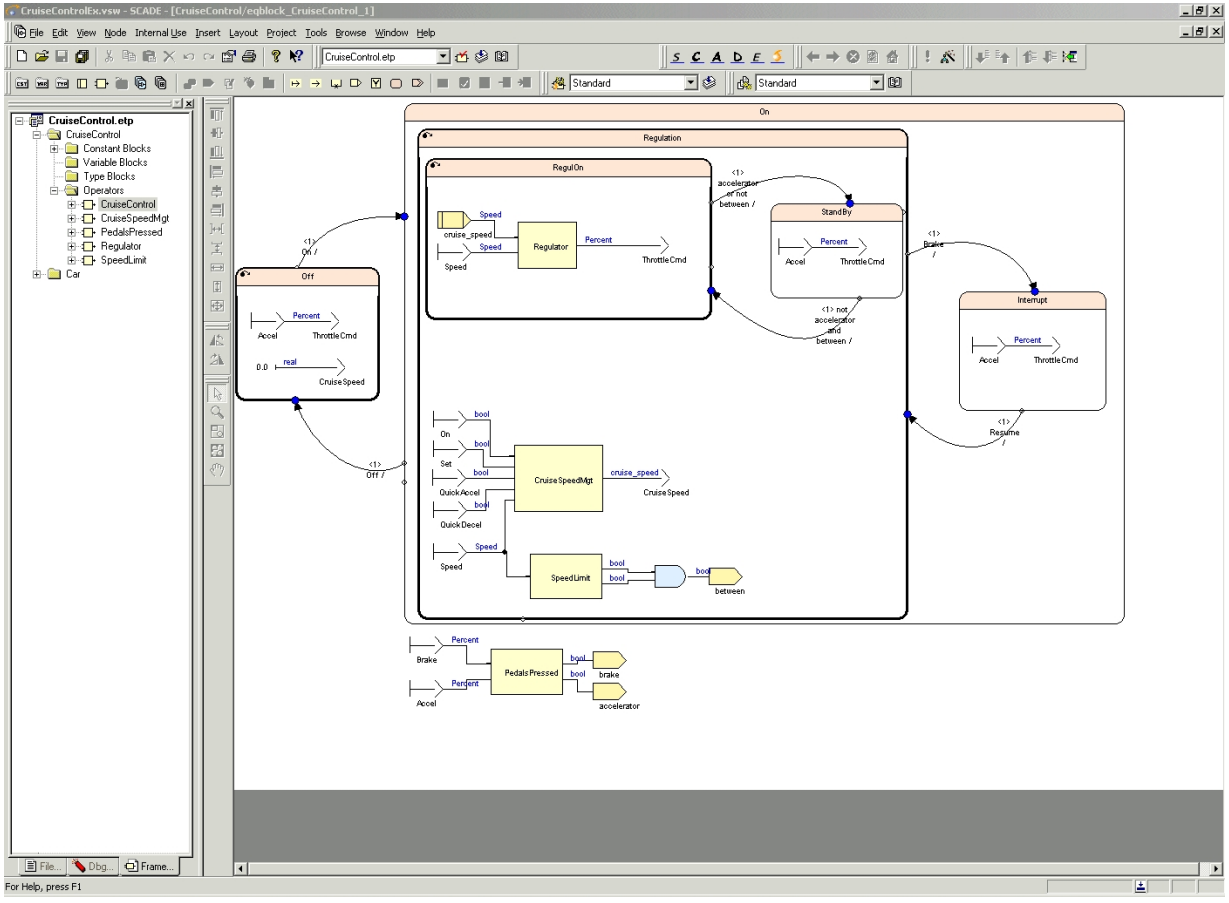
3. CNRS

4. Univ. Pierre et Marie Curie

5. Yale University

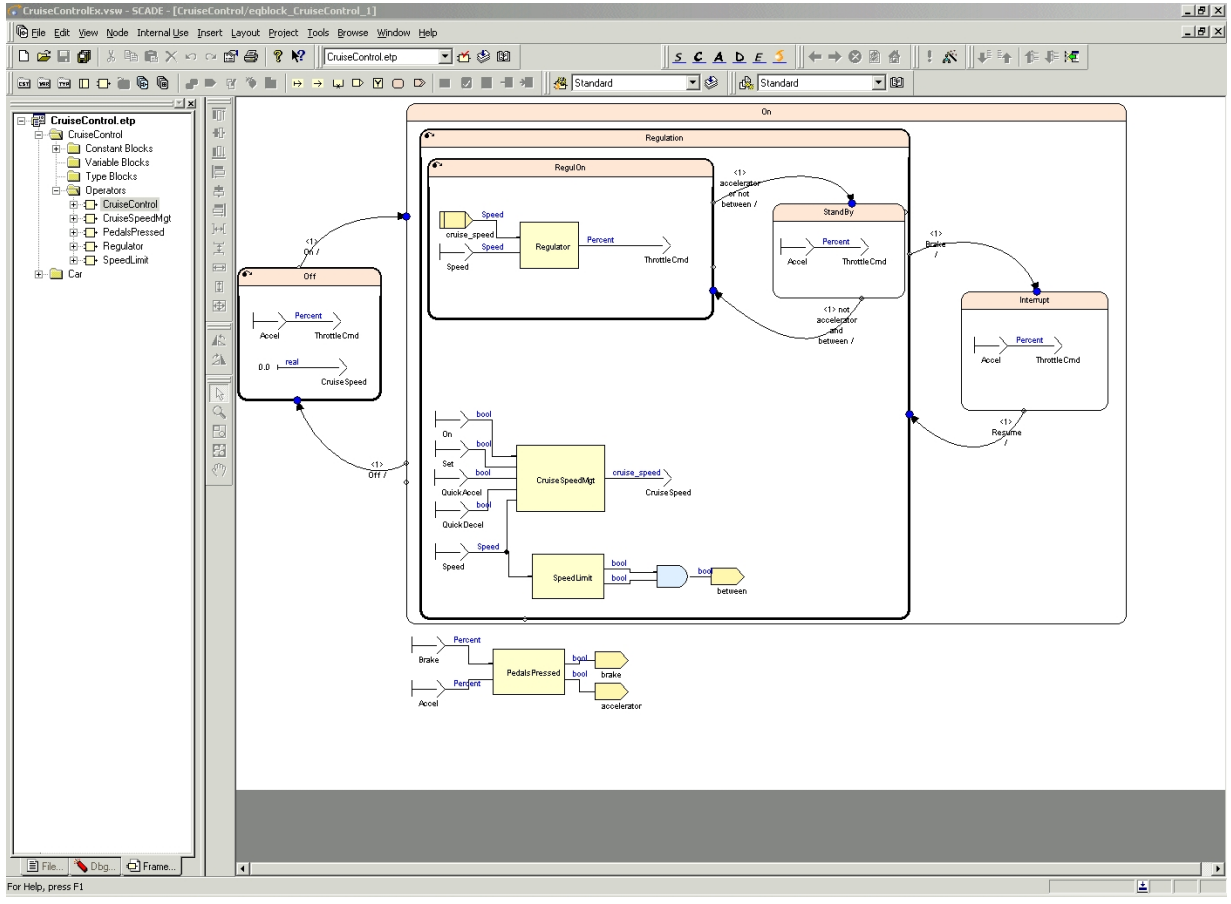
6. Collège de France

Collège de France—21 mars 2018



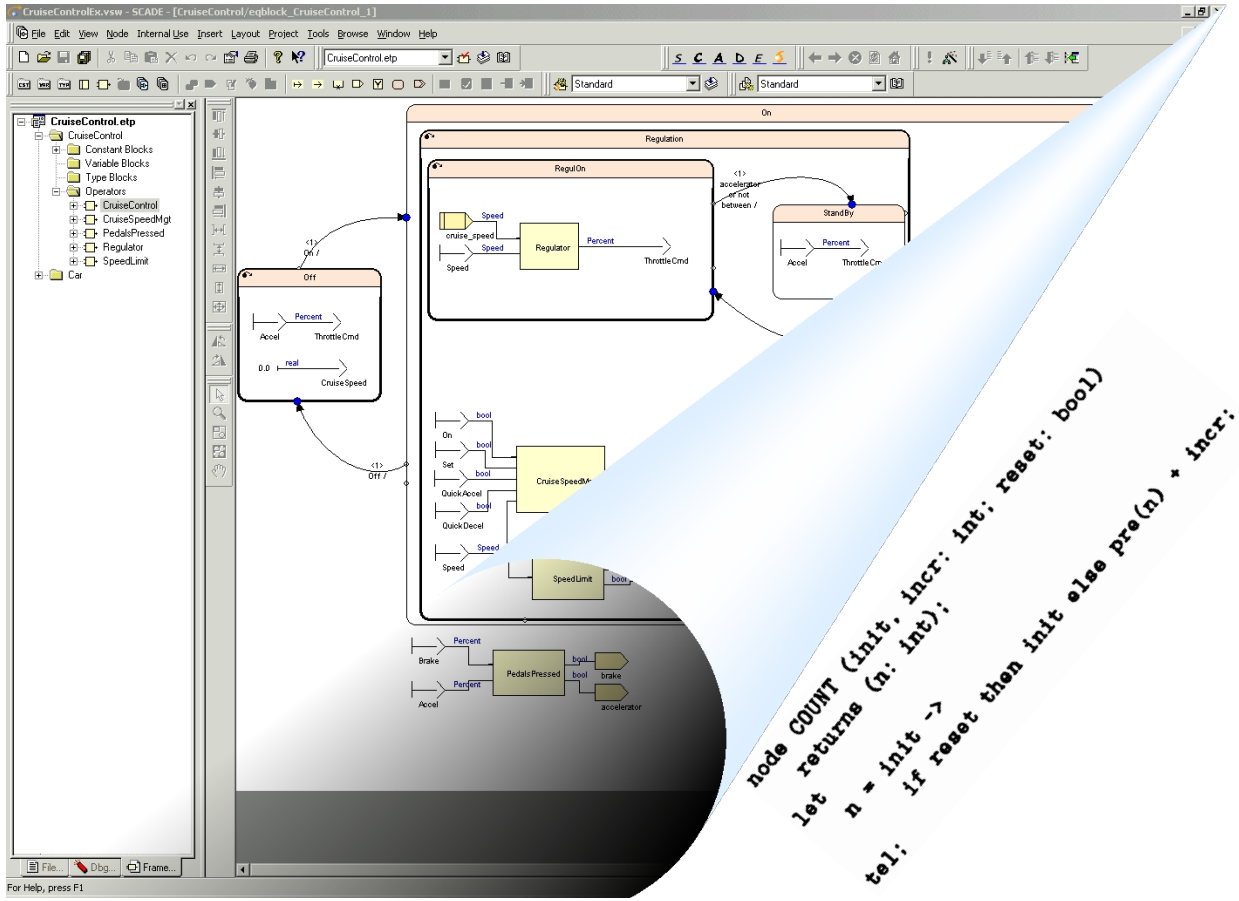
Capture d'écran de l'outil SCADE Suite d'ANSYS/Esterel Technologies

- La spécification par schéma-bloc : on connecte des unités fonctionnelles (*blocs*) avec des signaux (*lignes*) et on utilise les machines à état pour décrire les modes opérationnels.
- Le Model-Based Design : l'idée que ces schémas peuvent être exécutables et donc une base pour la simulation, le test et la vérification formelle, avec la génération automatique du code de bas niveau.



Capture d'écran de l'outil SCADE Suite d'ANSYS/Esterel Technologies

- Utilisation répandue pour la programmation des logiciels critiques dans l'aérospatial, la défense, le ferroviaire, l'énergie, le nucléaire, ...
- Qualifications industrielles d'un grand intérêt pratique :
  - La satisfaction des exigences peut être évaluée sur les modèles directement.
  - Les tests peuvent être réalisés sur les modèles.
  - On fait confiance au générateur du code.



Capture d'écran de l'outil SCADE Suite d'ANSYS/Esterel Technologies

- Derrière l'interface graphique, on trouve le langage de programmation synchrone Scade 6 [Colaço, Pagano et Pouzet (2017): "Scade 6 : A Formal Language for Embedded Critical Software Development"].
- Scade 6 est fondé sur le langage académique Lustre [Caspi, Pilaud, Halbwachs et Plaice (1987): "LUSTRE : A declarative language for programming synchronous systems"]. + Esterel + Lucid Synchrone
- Idées centrales : signal = flot de données (suite de valeurs)
  bloc = fonction entre suites

La compilation vérifiée

- Cet exposé : il ne s'agit pas de la certification, mais de la vérification.

La compilation vérifiée

- Cet exposé : il ne s'agit pas de la certification, mais de la vérification.
- Réussites récentes en employant les assistants de preuve :
  - CompCert : compilateur C vérifié dans Coq [Leroy (2009): "Formal verification of a realistic compiler"]
  - CakeML : compilateur SML vérifié dans HOL4 [Kumar, Myreen, Norrish et Owens (2014): "CakeML : A Verified Implementation of ML"]
- L'assistant de preuve Coq ? [The Coq Development Team (2017): *The Coq proof assistant reference manual*]
  - un langage de programmation fonctionnelle ;
  - avec 'extraction' vers le langage de programmation OCaml ;
  - un language de spécification (une logique d'ordre supérieure) ;
  - la preuve interactive basé sur des « tactiques ».

La compilation vérifiée

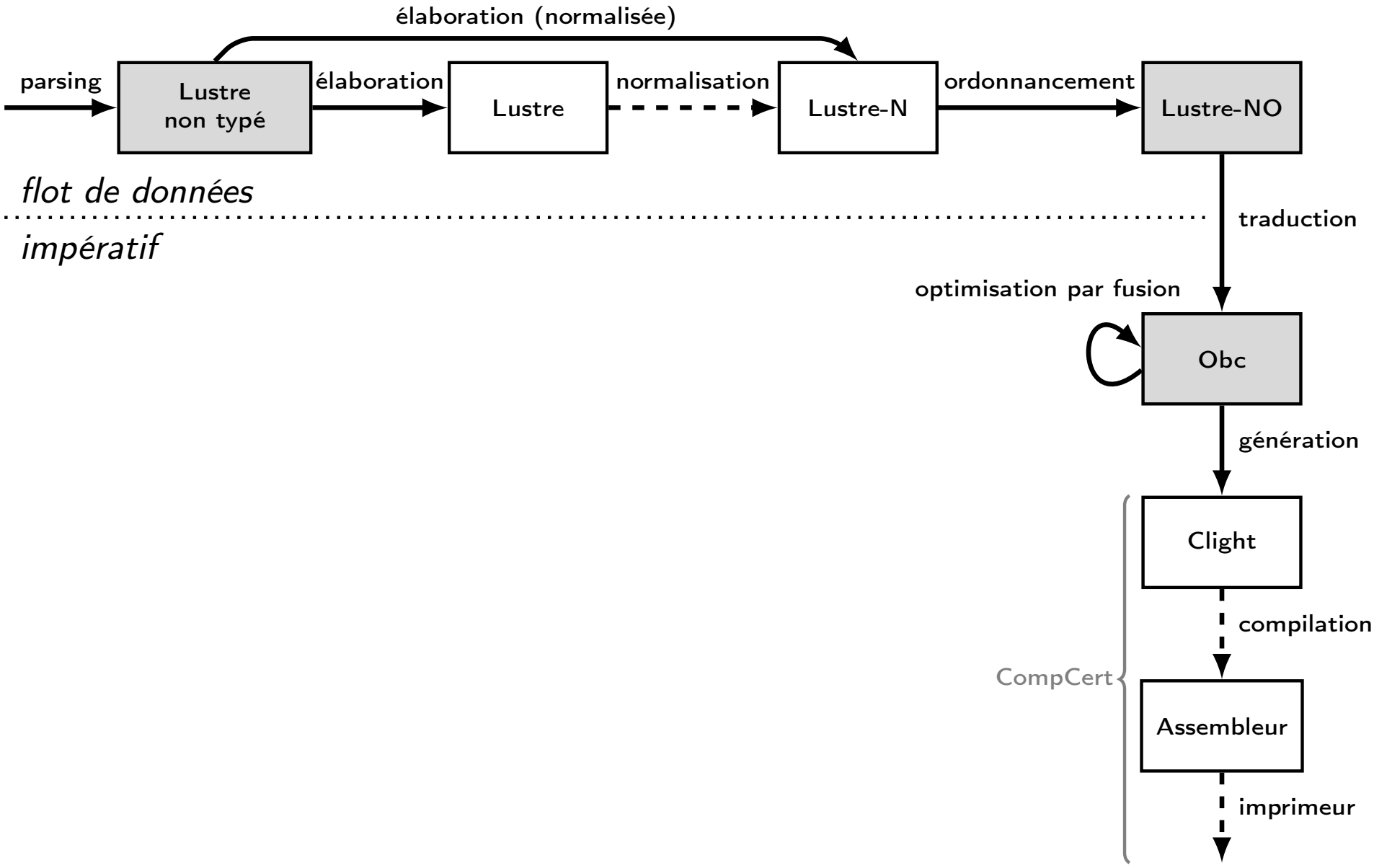
- Cet exposé : il ne s'agit pas de la certification, mais de la vérification.
- Réussites récentes en employant les assistants de preuve :
  - CompCert : compilateur C vérifié dans Coq [Leroy (2009): "Formal verification of a realistic compiler"]
  - CakeML : compilateur SML vérifié dans HOL4 [Kumar, Myreen, Norrish et Owens (2014): "CakeML : A Verified Implementation of ML"]
- L'assistant de preuve Coq ? [The Coq Development Team (2017): *The Coq proof assistant reference manual*]
  - un langage de programmation fonctionnelle ;
  - avec 'extraction' vers le langage de programmation OCaml ;
  - un language de spécification (une logique d'ordre supérieure) ;
  - la preuve interactive basé sur des « tactiques ».
- Nous suivons cette approche pour obtenir un compilateur Lustre vérifié.

  Vérifier des programmes Lustre pour assurer leurs propriétés sur le code (assembleur) qui s'exécute.

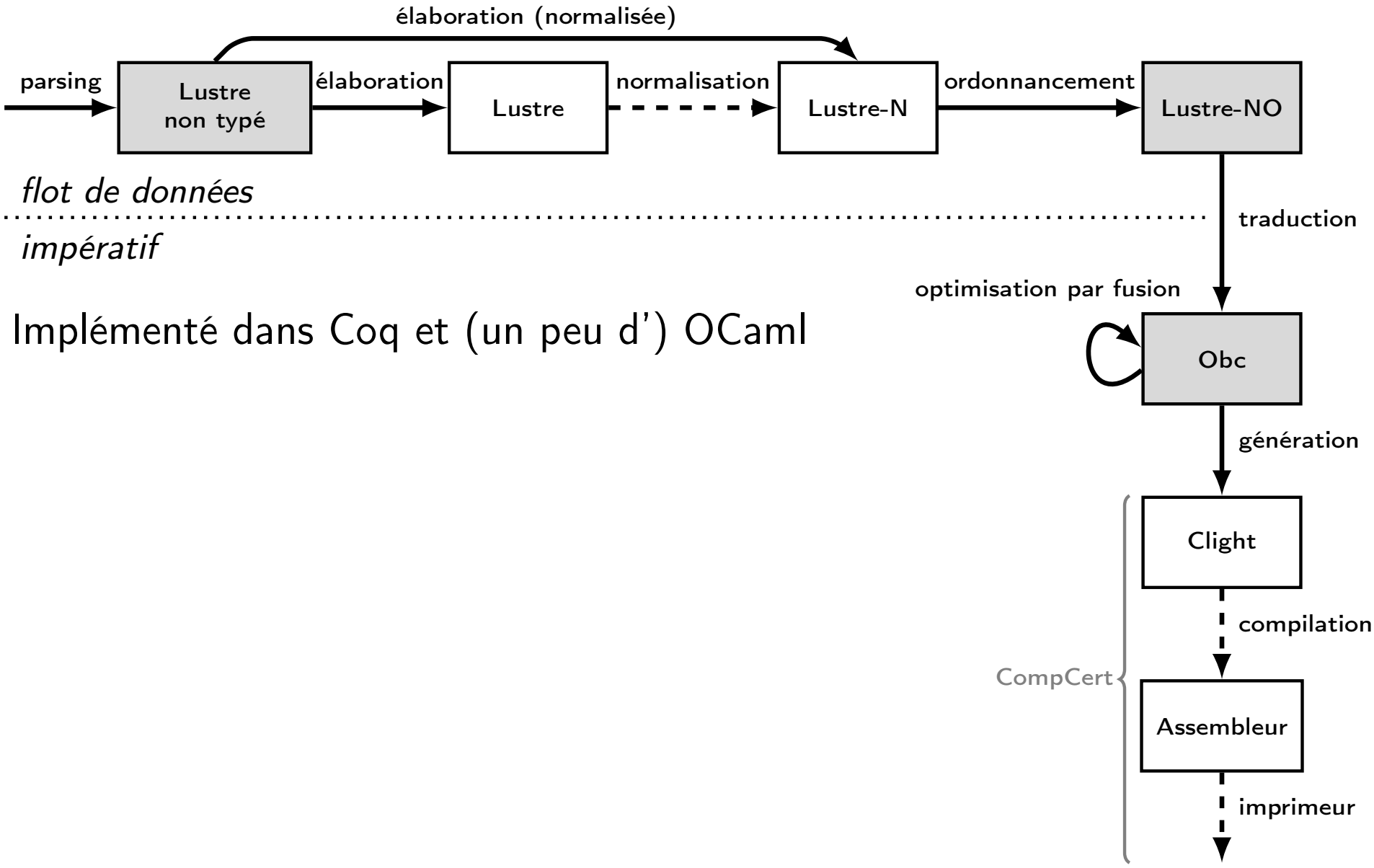
  Le principe de WYPIWYE, « What You Prove Is What You Execute »

  [Berry (1989): "Real Time Programming : Special Purpose or General Purpose Languages"]

Le compilateur Vélus

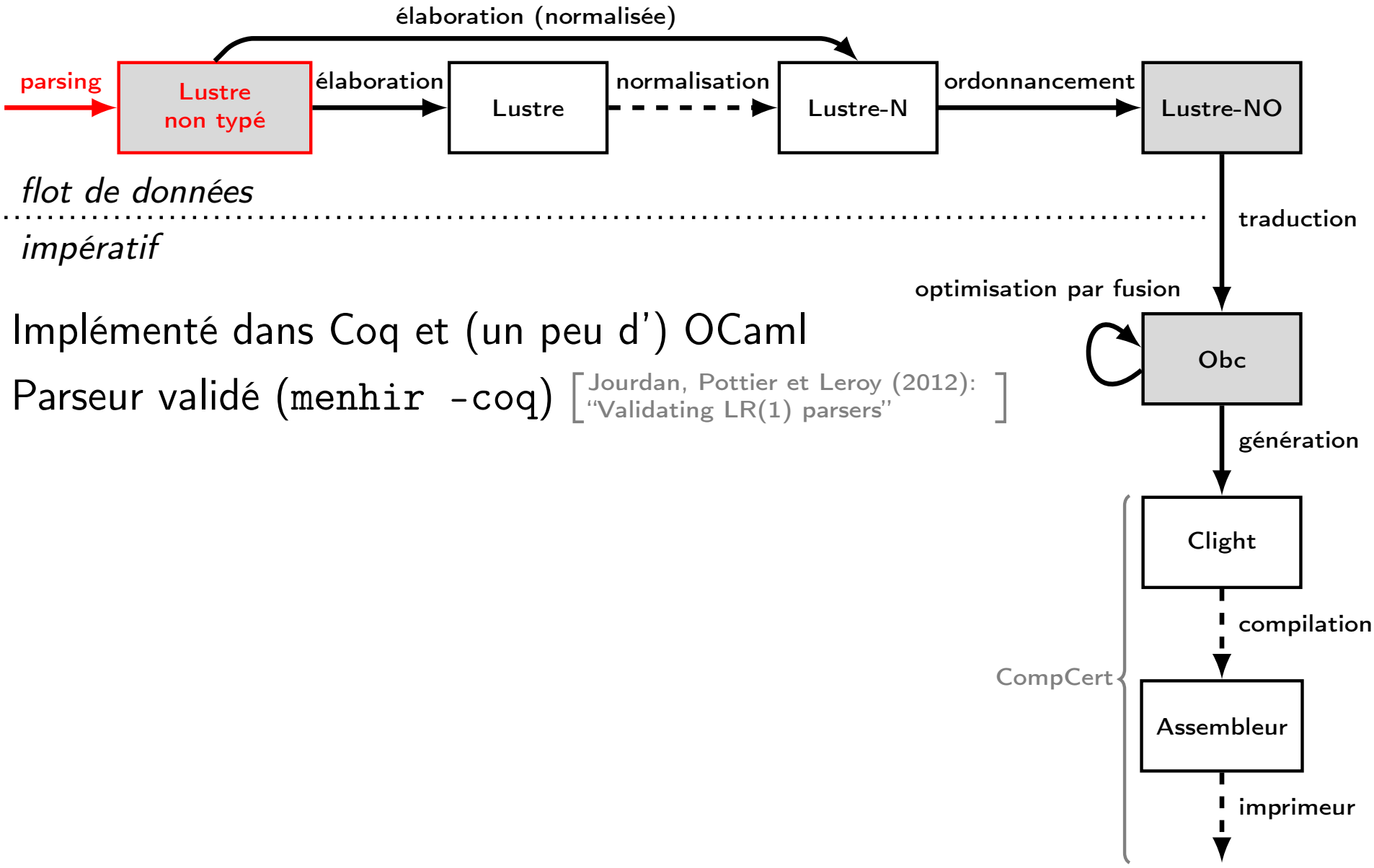


Le compilateur Vélus



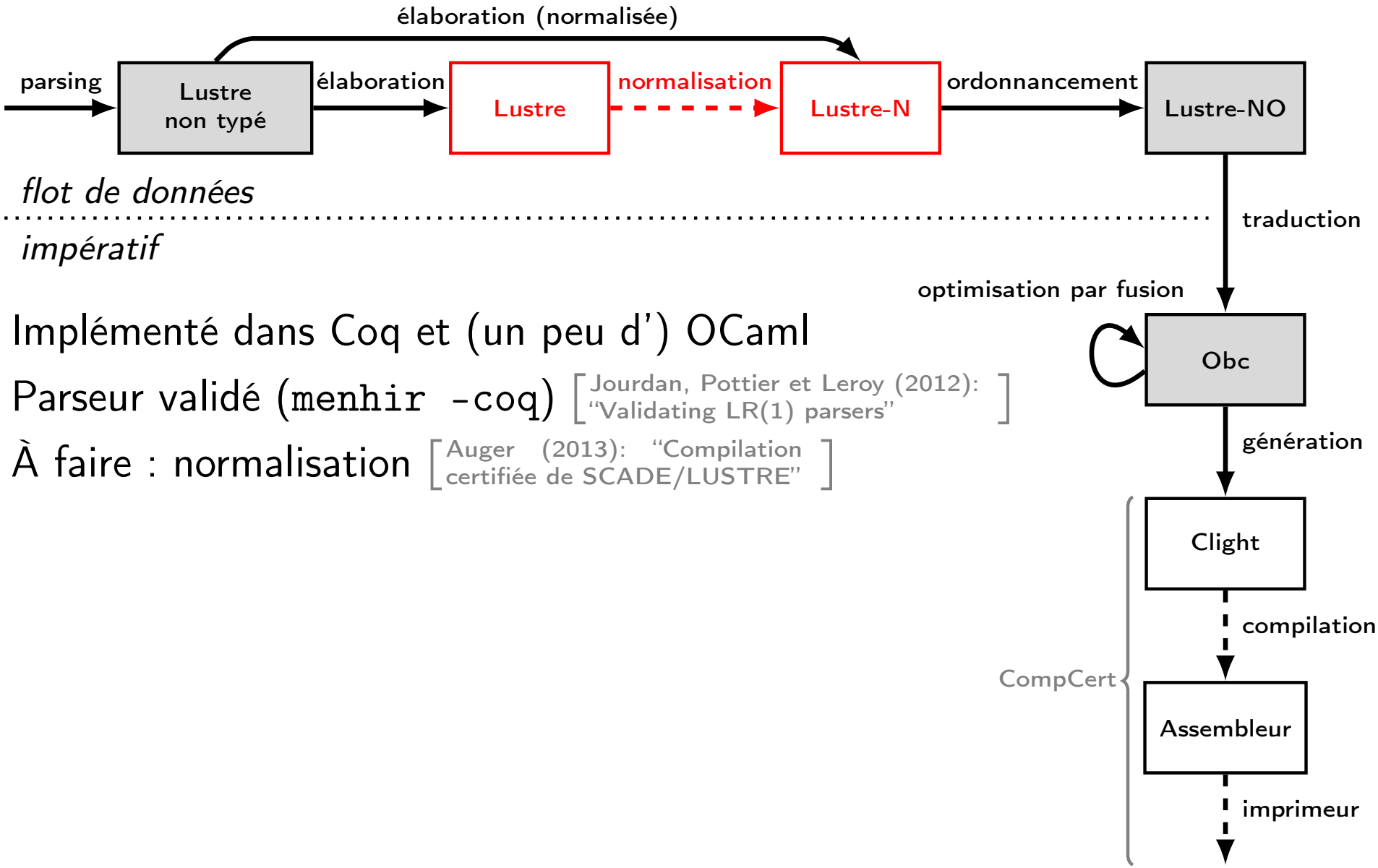
- Implémenté dans Coq et (un peu d') OCaml

Le compilateur Vélus



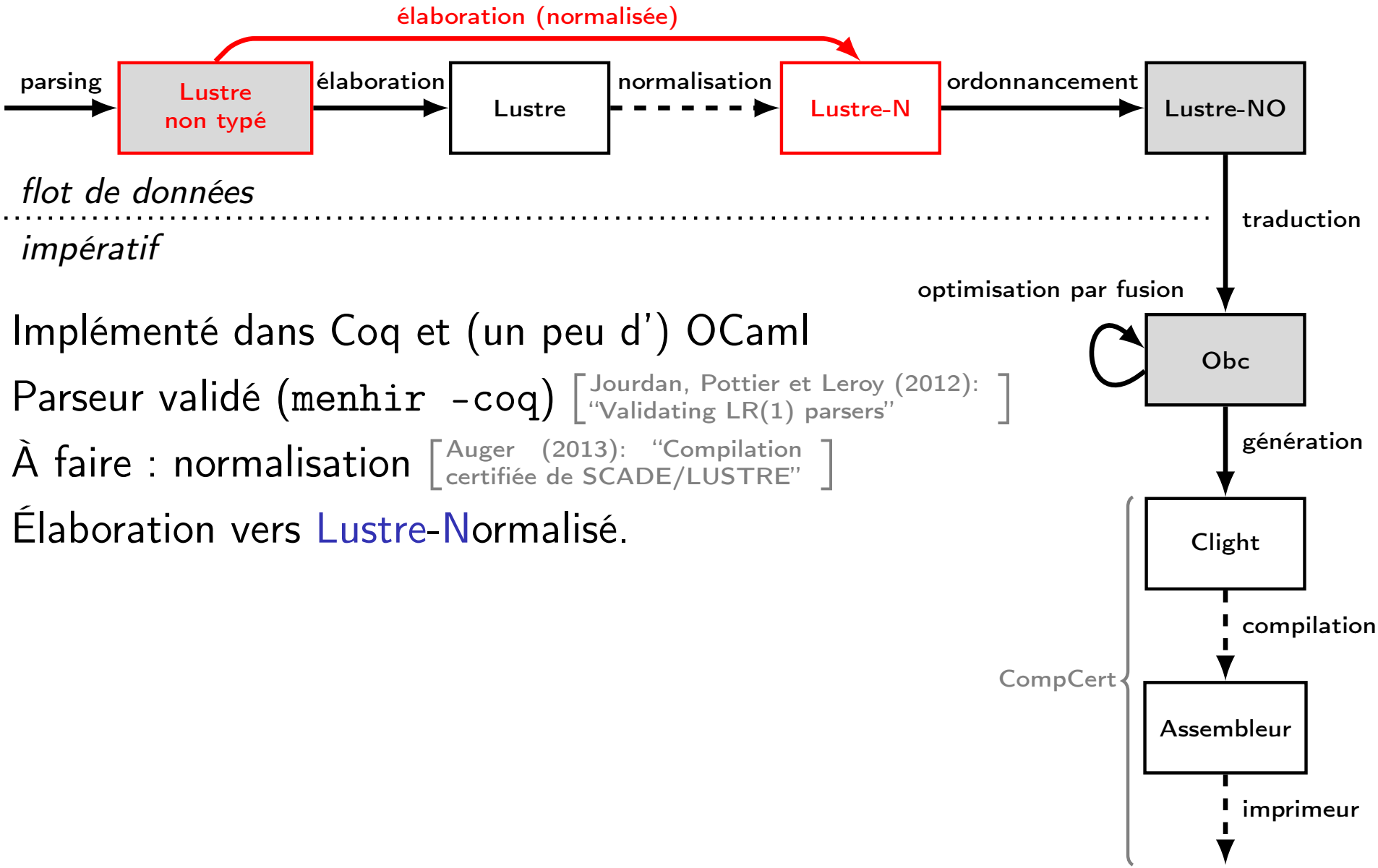
- Implémenté dans Coq et (un peu d') OCaml
- Parseur validé (`menhir -coq`) [Jourdan, Pottier et Leroy (2012): "Validating LR(1) parsers"]

Le compilateur Vélus



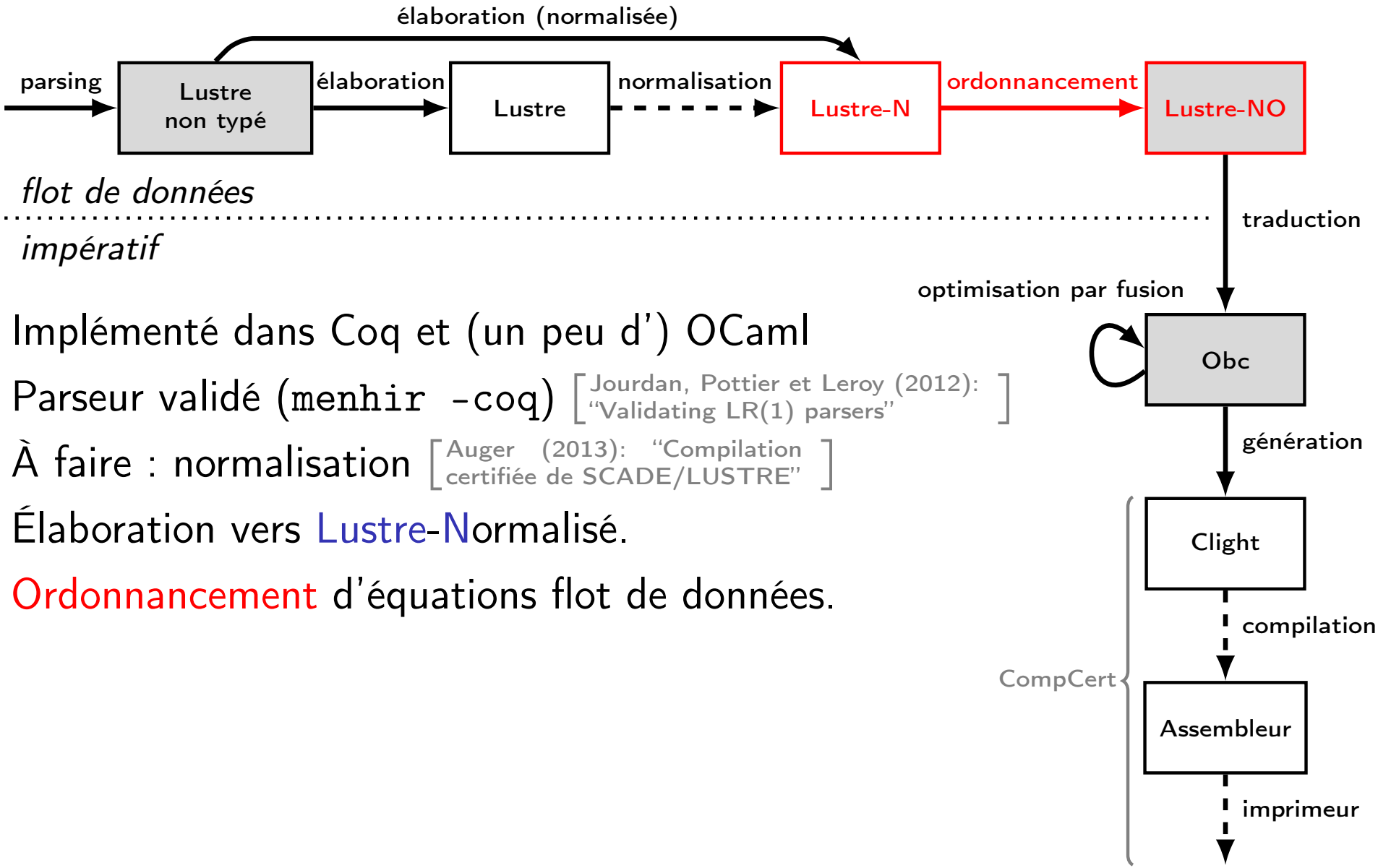
- Implémenté dans Coq et (un peu d') OCaml
- Parseur validé (`menhir -coq`) [Jourdan, Pottier et Leroy (2012): "Validating LR(1) parsers"]
- À faire : normalisation [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]

Le compilateur Vélus



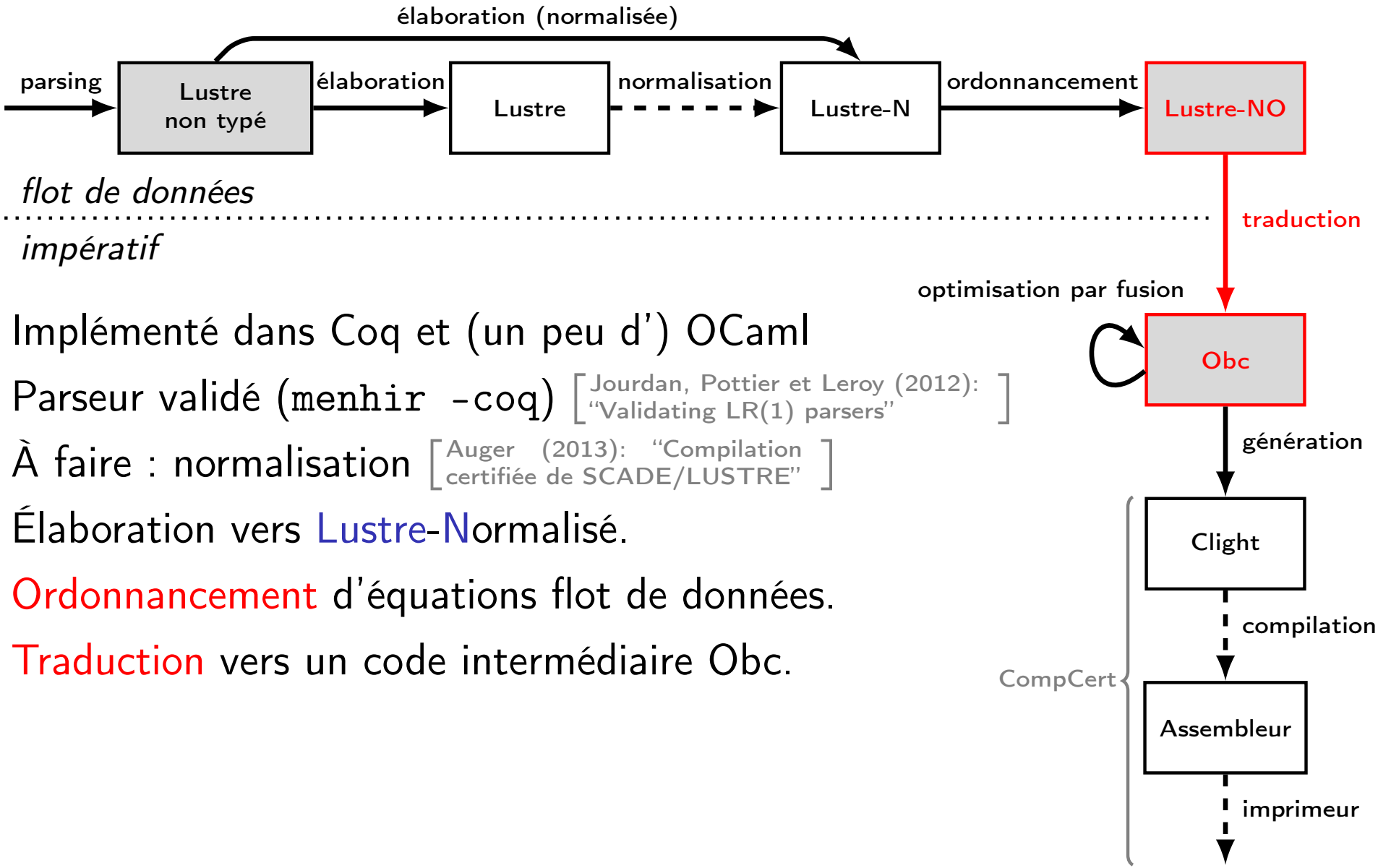
- Implémenté dans Coq et (un peu d') OCaml
- Parseur validé (`menhir -coq`) [Jourdan, Pottier et Leroy (2012): "Validating LR(1) parsers"]
- À faire : normalisation [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Élaboration vers Lustre-Normalisé.

Le compilateur Vélus



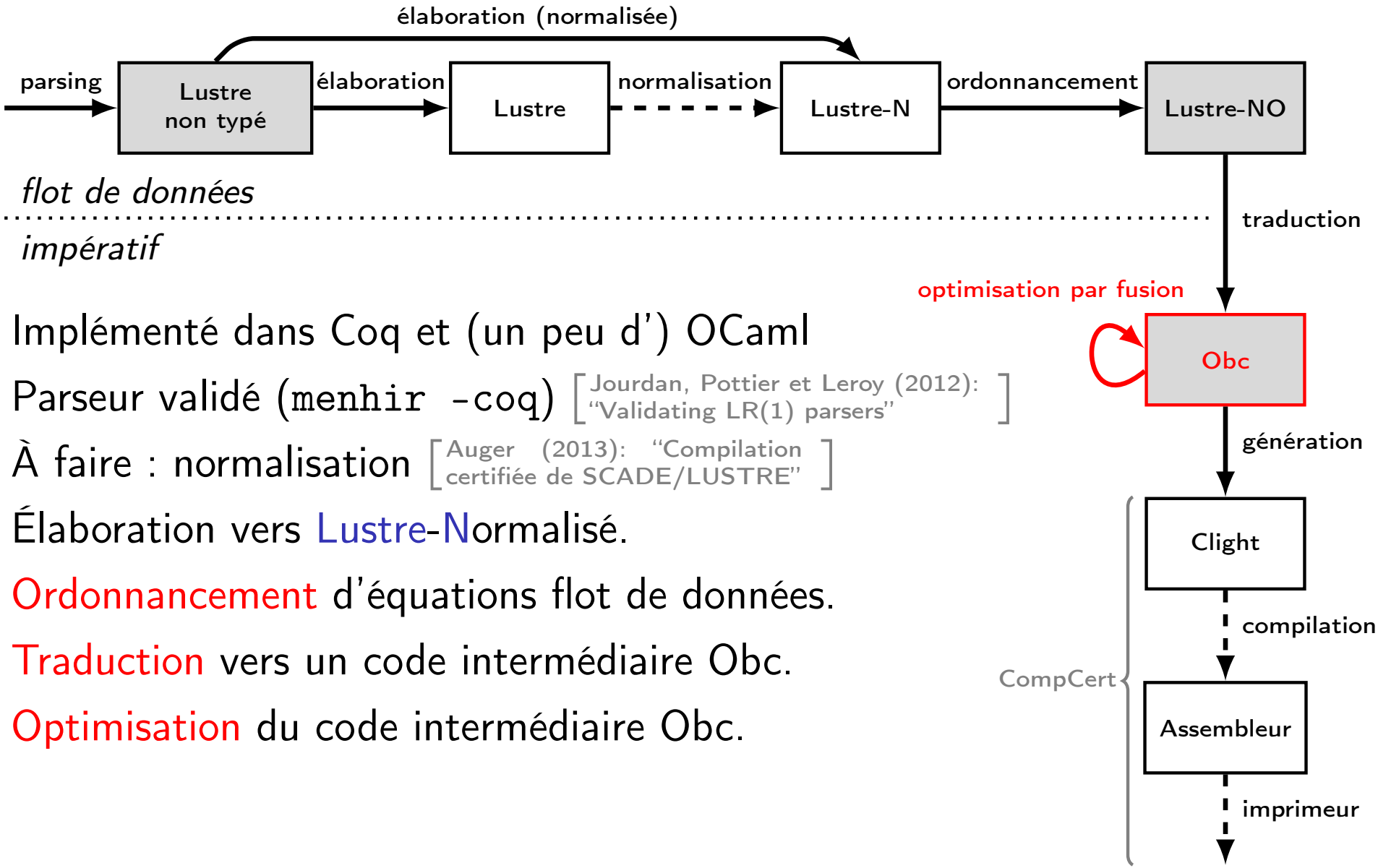
- Implémenté dans Coq et (un peu d') OCaml
- Parseur validé (`menhir -coq`) [Jourdan, Pottier et Leroy (2012): "Validating LR(1) parsers"]
- À faire : normalisation [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Élaboration vers Lustre-Normalisé.
- Ordonnancement d'équations flot de données.

Le compilateur Vélus



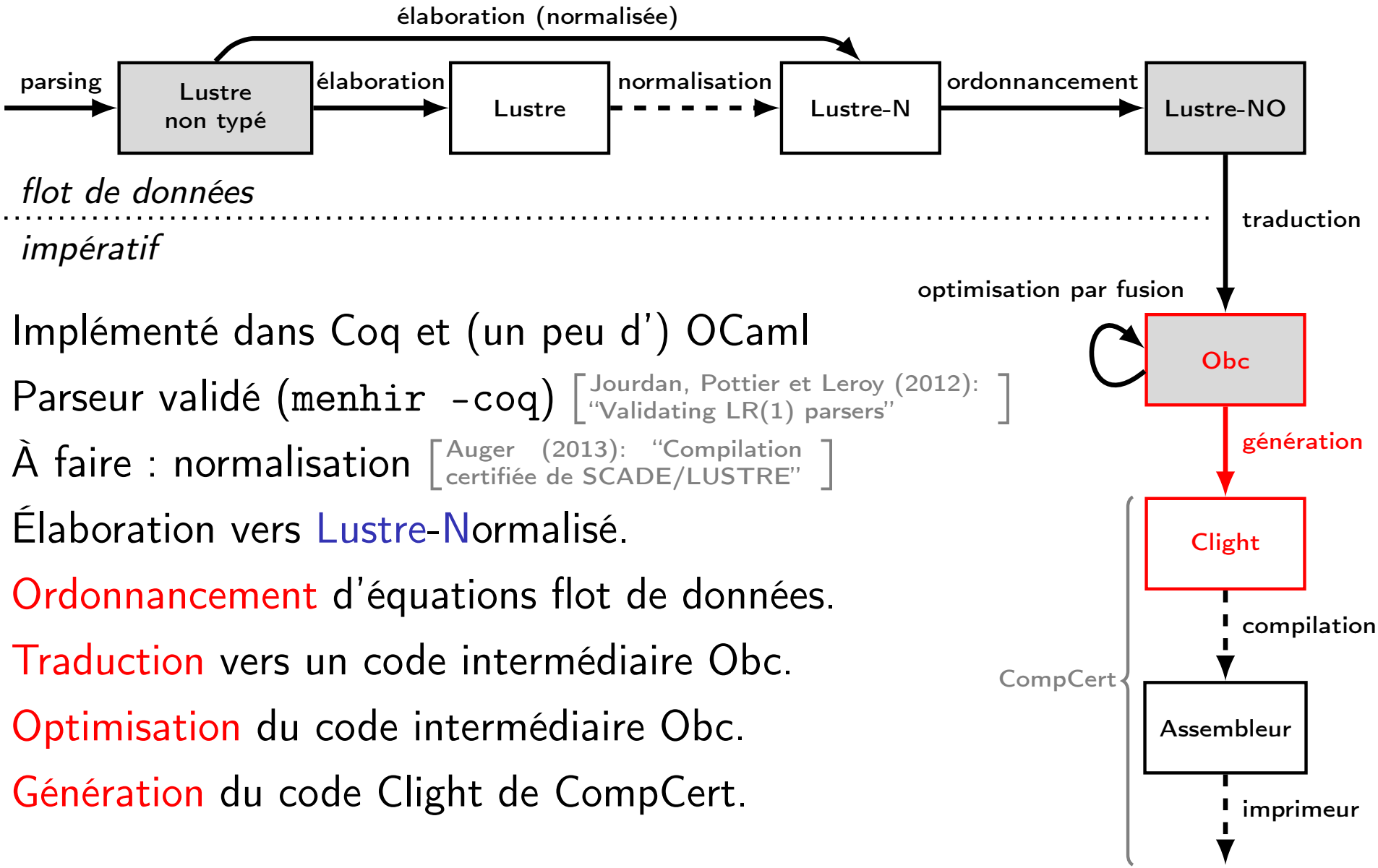
- Implémenté dans Coq et (un peu d') OCaml
- Parseur validé (`menhir -coq`) [Jourdan, Pottier et Leroy (2012): "Validating LR(1) parsers"]
- À faire : normalisation [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Élaboration vers Lustre-Normalisé.
- Ordonnancement d'équations flot de données.
- Traduction vers un code intermédiaire Obc.

Le compilateur Vélus



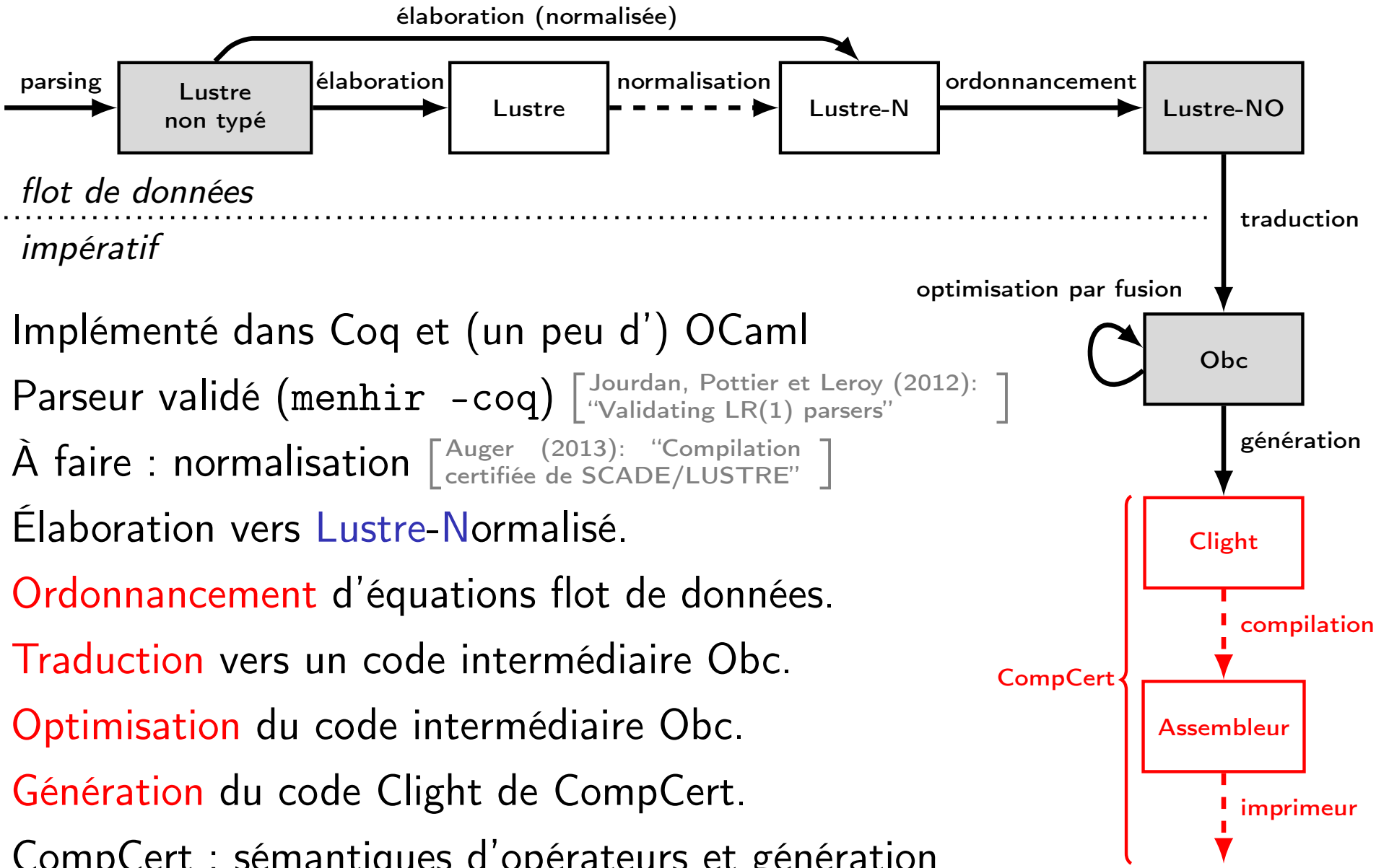
- Implémenté dans Coq et (un peu d') OCaml
- Parseur validé (`menhir -coq`) [Jourdan, Pottier et Leroy (2012): "Validating LR(1) parsers"]
- À faire : normalisation [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Élaboration vers Lustre-Normalisé.
- Ordonnancement d'équations flot de données.
- Traduction vers un code intermédiaire Obc.
- Optimisation du code intermédiaire Obc.

Le compilateur Vélus



- Implémenté dans Coq et (un peu d') OCaml
- Parseur validé (`menhir -coq`) [Jourdan, Pottier et Leroy (2012): "Validating LR(1) parsers"]
- À faire : normalisation [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Élaboration vers Lustre-Normalisé.
- Ordonnancement d'équations flot de données.
- Traduction vers un code intermédiaire Obc.
- Optimisation du code intermédiaire Obc.
- Génération du code Clight de CompCert.

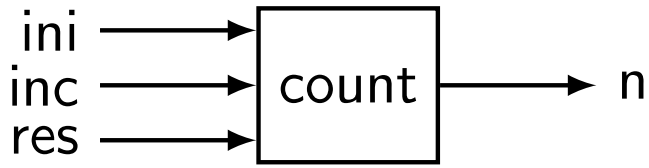
Le compilateur Vélus



- Implémenté dans Coq et (un peu d') OCaml
- Parseur validé (`menhir -coq`) [Jourdan, Pottier et Leroy (2012): "Validating LR(1) parsers"]
- À faire : normalisation [Auger (2013): "Compilation certifiée de SCADE/LUSTRE"]
- Élaboration vers Lustre-Normalisé.
- Ordonnancement d'équations flot de données.
- Traduction vers un code intermédiaire Obc.
- Optimisation du code intermédiaire Obc.
- Génération du code Clight de CompCert.
- CompCert : sémantiques d'opérateurs et génération d'assembleur.

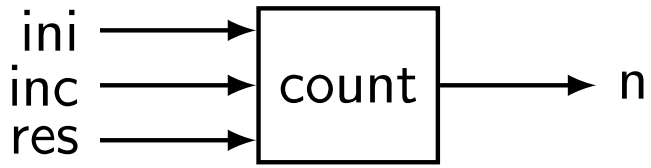
Lustre

[Caspi, Pilaud, Halbwachs et Plaice (1987): "LUSTRE : A declarative language for programming synchronous systems"]



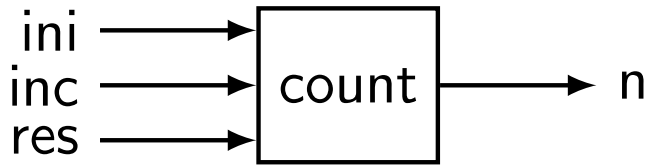
Lustre [Caspi, Pilaud, Halbwachs et Plaice (1987): "LUSTRE : A declarative language for programming synchronous systems"]

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```



Lustre [Caspi, Pilaud, Halbwachs et Plaice (1987): "LUSTRE : A declarative language for programming synchronous systems"]

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```



ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

- Nœud : ensemble d'équations causales (variables à gauche).
- Modèle sémantique : flots synchronisés de données (suites de valeurs).
- Un nœud définit une fonction entre des flots d'entrée et de sortie.

Compilation de Lustre : normalisation et ordonnancement

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

Compilation de Lustre : normalisation et ordonnancement

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```


```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  f = true fby false;
  c = 0 fby n;
  n = if f or res then ini else c + inc;
tel
```

Normalisation

- Réécrire pour ajouter une équation par fby.
- Introduire des variables fraîches en exploitant le principe de substitution.

Compilation de Lustre : normalisation et ordonnancement

```
node count (ini, inc: int; res: bool)
returns (n: int)
let
  n = if (true fby false) or res then ini
      else (0 fby n) + inc;
tel
```

normalisation

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  f = true fby false;
  c = 0 fby n;
  n = if f or res then ini else c + inc;
tel
```

ordonnancement

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

Ordonnancement

- La sémantique des équations est indépendante de l'ordre de définition ; mais la correction de la compilation en dépend.
- Réécrire pour que
  - les variables 'normales' soient écrites avant d'être lues, . . . et que
  - les variables 'fby' soient lues avant d'être écrites.

Lustre : syntaxe et sémantique

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  f = true fby false;
  c = 0 fby n;
  n = if f or res then ini else c + inc;
tel
```

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

Lustre : syntaxe et sémantique

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  f = true fby false;
  c = 0 fby n;
  n = if f or res then ini else c + inc;
tel
```

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

```
Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

Inductive cexp : Type :=
| Emerge : ident → cexp → cexp → cexp
| Eite   : lexp → cexp → cexp → cexp
| Eexp   : lexp → cexp.

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;
  ... }.
```

Lustre : syntaxe et sémantique

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  f = true fby false;
  c = 0 fby n;
  n = if f or res then ini else c + inc;
tel
```

ini	0	0	0	0	0	0	0	···
inc	0	1	2	1	2	3	0	···
res	F	F	F	F	T	F	F	···
true fby false	T	F	F	F	F	F	F	···
0 fby n	0	0	1	3	4	0	3	···
n	0	1	3	4	0	3	3	···

```
Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

Inductive cexp : Type :=
| Emerge : ident → cexp → cexp → cexp
| Eite   : lexp → cexp → cexp → cexp
| Eexp   : lexp → cexp.

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;
  ... }.
```

```
Inductive sem_equation (G: global) : stream bool → history
                                   → equation → Prop :=
| SEqFby: sem_laexp bk H ck le ls →
          sem_var bk H x xs →
          (∀ n, xs n = fby (sem_const c0) ls n) →
          sem_equation G bk H (EqFby x ck c0 le)

| SEqDef: ...  | SEqApp: ...

with sem_node (G: global) : ident → stream (list value)
                          → stream (list value) → Prop :=
| SNode: find_node f G = Some n →
         clock_of xss bk →
         sem_vars bk H (map fst n.(n_in)) xss →
         sem_vars bk H (map fst n.(n_out)) yss →
         sem_clocked_vars bk H (idck n.(n_in)) →
         Forall (sem_equation G bk H) n.(n_eqs) →
         sem_node G f xss yss.
```

Lustre : syntaxe et sémantique

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  f = true fby false;
  c = 0 fby n;
  n = if f or res then ini else c + inc;
tel
```

```
Inductive lexp : Type :=
| Econst : const → lexp
| Evar   : ident → type → lexp
| Ewhen  : lexp → ident → bool → lexp
| Eunop  : unop → lexp → type → lexp
| Ebinop : binop → lexp → lexp → type → lexp.

Inductive cexp : Type :=
| Emerge : ident → cexp → cexp → cexp
| Eite   : lexp → cexp → cexp → cexp
| Eexp   : lexp → cexp.

Inductive equation : Type :=
| EqDef : ident → clock → cexp → equation
| EqApp : idents → clock → ident → lexps → equation
| EqFby : ident → clock → const → lexp → equation.

Record node : Type := mk_node {
  n_name : ident;
  n_in   : list (ident * (type * clock));
  n_out  : list (ident * (type * clock));
  n_vars : list (ident * (type * clock));
  n_eqs  : list equation;
  ... }.
```

ini	0	0	0	0	0	0	0	⋯
inc	0	1	2	1	2	3	0	⋯
res	F	F	F	F	T	F	F	⋯
true fby false	T	F	F	F	F	F	F	⋯
0 fby n	0	0	1	3	4	0	3	⋯
n	0	1	3	4	0	3	3	⋯

```
Inductive sem_equation (G: global) : stream bool → history
                                   → equation → Prop :=
| SEqFby: sem_laexp bk H ck le ls →
          sem_var bk H x xs →
          (∀ n, xs n = fby (sem_const c0) ls n) →
          sem_equation G bk H (EqFby x ck c0 le)

| SEqDef: ... | SEqApp: ...

with sem_node (G: global) : ident → stream (list value)
                          → stream (list value) → Prop :=
| SNode: find_node f G = Some n →
         clock_of xss bk →
         sem_vars bk H (map fst n.(n_in)) xss →
         sem_vars bk H (map fst n.(n_out)) yss →
         sem_clocked_vars bk H (idck n.(n_in)) →
         Forall (sem_equation G bk H) n.(n_eqs) →
         sem_node G f xss yss.
```

sem_node G f xss yss

$f : \text{stream}(T^+) \to \text{stream}(T^+)$

Compilation de Lustre : traduction vers du code impératif

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

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Compilation de Lustre : traduction vers du code impératif

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

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Compilation de Lustre : traduction vers du code impératif

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

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Compilation de Lustre : traduction vers du code impératif

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

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Compilation de Lustre : traduction vers du code impératif

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

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

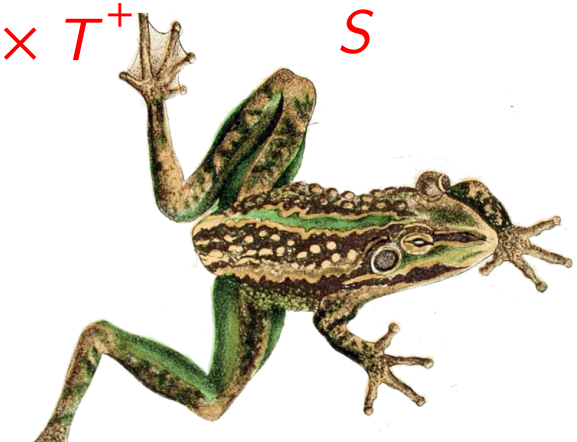
Compilation de Lustre : traduction vers du code impératif

Biernacki, Colaço, Hamon et Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"

```
node count (ini, inc: int; res: bool)
returns (n: int)
var f : bool; c : int;
let
  n = if f or res then ini else c + inc;
  f = true fby false;
  c = 0 fby n;
tel
```

(f_t, s_0)

$S \times T^+ \to S \times T^+$ $\qquad S$



```
class count {
  memory f : bool;
  memory c : int;

  reset() {
    state(f) := true;
    state(c) := 0
  }

  step(ini: int, inc: int, res: bool)
  returns (n: int) {
    if (state(f) | restart)
      then n := ini
      else  n := state(c) + inc;
    state(f) := false;
    state(c) := n
  }
}
```

Lustre : instanciation et échantillonnage

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

Lustre : instanciation et échantillonnage

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯

Lustre : instanciation et échantillonnage

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯

Lustre : instanciation et échantillonnage

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯

Lustre : instanciation et échantillonnage

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯

Lustre : instanciation et échantillonnage

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯
0 fby v	0	0	0	0	4	4	4	3	⋯
(0 fby v) when not sec	0	0	0		4			3	⋯

Lustre : instanciation et échantillonnage

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯
0 fby v	0	0	0	0	4	4	4	3	⋯
(0 fby v) when not sec	0	0	0		4			3	⋯
v	0	0	0	4	4	4	3	3	⋯

Lustre : instanciation et échantillonnage

Modèle sémantique

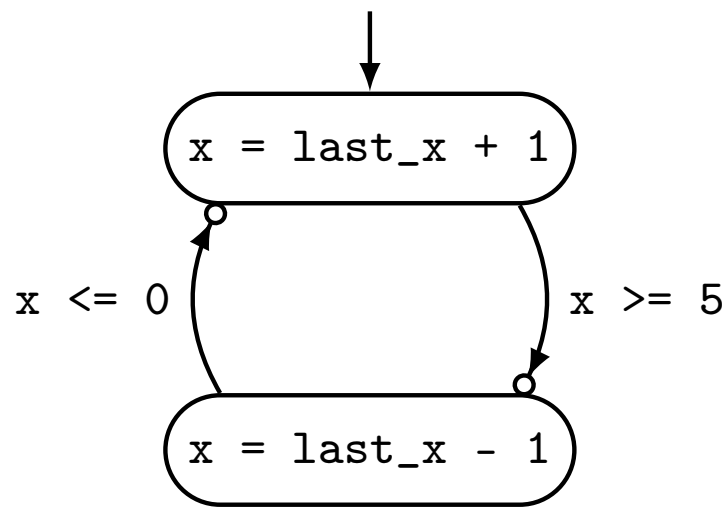
- Une dictionnaire associe les identifiants avec leurs flots de données.
- Flot : une application sur les entiers naturels `Notation stream A := nat → A`
- Avec modélisation de l'absence : `Inductive value := absent | present v`

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯
0 fby v	0	0	0	0	4	4	4	3	⋯
(0 fby v) when not sec	0	0	0		4			3	⋯
v	0	0	0	4	4	4	3	3	⋯

Pourquoi les constructions when et merge ?

- Elles permettent l'activation conditionnelle.
- Mais il n'est pas facile de les utiliser.
- Plutôt une forme interne pour des constructions plus structurées.

```
node main (go : bool)
    returns (x : int)
  var last_x : int;
let
  last_x = 0 fby x;

  automaton
  state Up
    do x = last_x + 1
    until x >= 5 then Down

  state Down
    do x = last_x - 1
    until x <= 0 then Up
  end;
tel
```



Pourquoi les constructions when et merge ?

- Elles permettent l'activation conditionnelle.
- Mais il n'est pas facile de les utiliser.
- Plutôt une forme interne pour des constructions plus structurées.

```
node main (go : bool)
    returns (x : int)
  var last_x : int;
let
  last_x = 0 fby x;

  automaton
  state Up
    do x = last_x + 1
    until x >= 5 then Down

  state Down
    do x = last_x - 1
    until x <= 0 then Up
  end;
tel
```

```
type st = St_Up | St_Down

(* ... *)

last_x = 0 fby x

x_St_Down = (last_x when St_Down(ck)) - 1
x_St_Up = (last_x when St_Up(ck)) + 1
x = merge ck (St_Down: x_St_Down)
             (St_Up: x_St_Up);

ck = St_Up fby ns
ns = ...
```

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

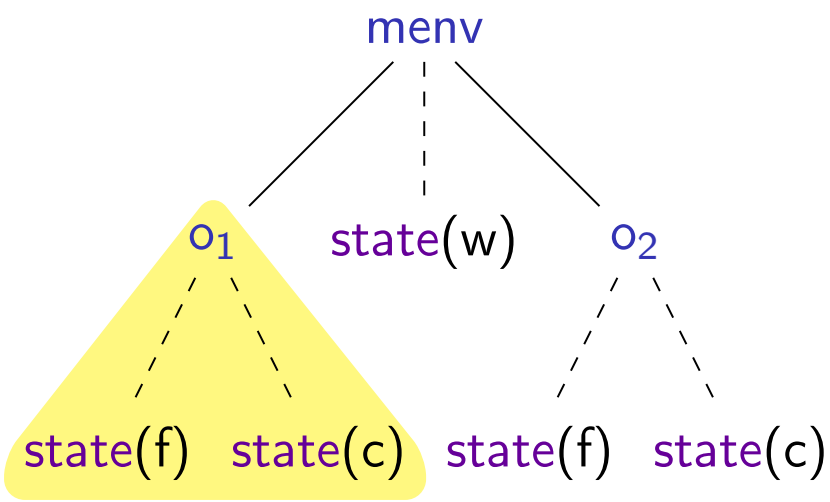
Compilation de Lustre : traduction vers du code impératif bis

```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```

```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

Compilation de Lustre : traduction vers du code impératif bis

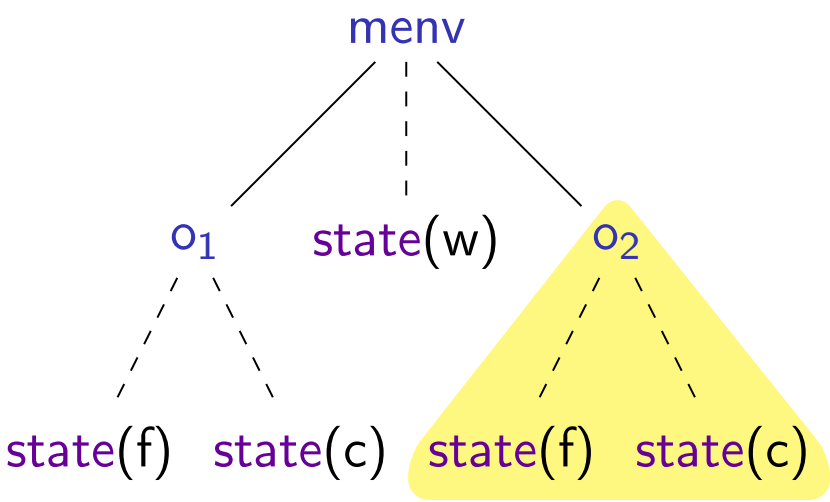
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

Compilation de Lustre : traduction vers du code impératif bis

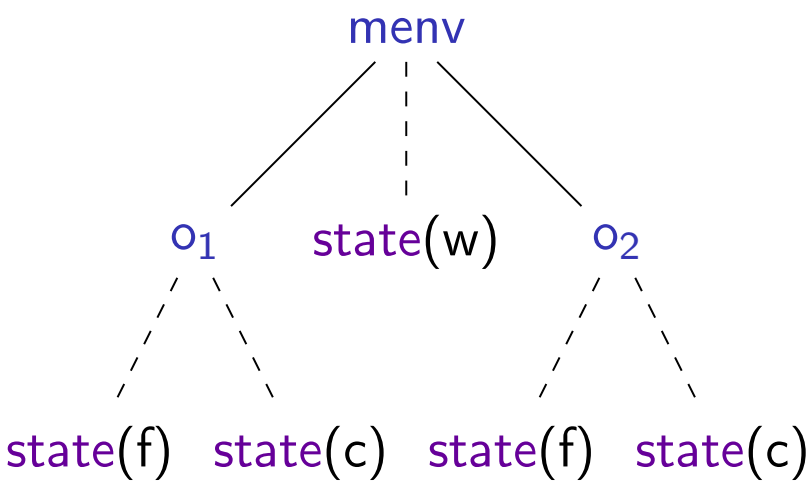
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

Compilation de Lustre : traduction vers du code impératif bis

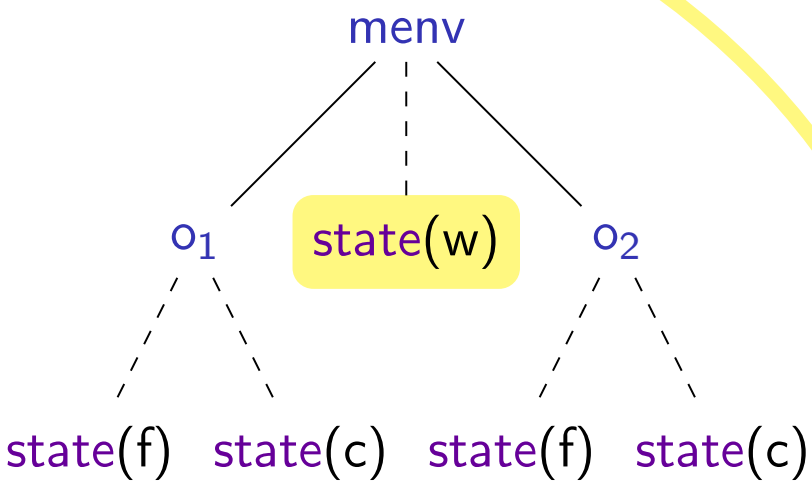
```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

Compilation de Lustre : traduction vers du code impératif bis

```
node avgvelocity(delta: int; sec: bool)
returns (r, v: int)
var t, w: int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t)
                (w when not sec);
  w = 0 fby v;
tel
```




```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  { var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then t := count.step o2 (1, 1, false);
    if sec
      then v := r / t else v := state(w);
    state(w) := v
  }
}
```

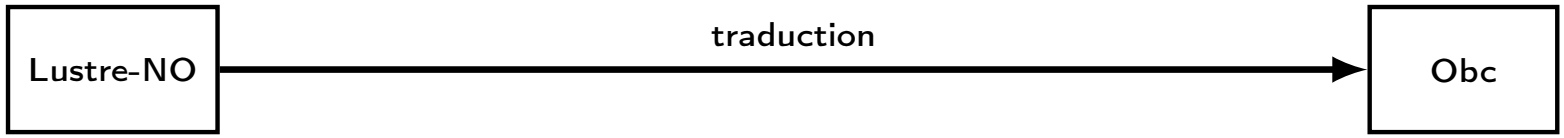
Implémentation de la traduction vers du code impératif

- Passe de traduction : quelques fonctions sur la syntaxe abstraite.
- Le défi : justifier le passage d'un modèle sémantique à un autre.

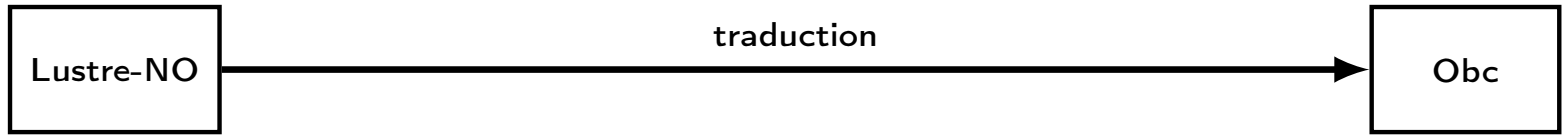
```
Definition tovar (x: ident) : exp :=
  if PS.mem x memories then State x else Var x.

Fixpoint Control (ck: clock) (s: stmt) : stmt :=
  match ck with
  | Cbase ⇒ s
  | Con ck x true  ⇒ Control ck (Ifte (tovar x) s Skip)
  | Con ck x false ⇒ Control ck (Ifte (tovar x) Skip s)
  end.

Fixpoint translate_lexp (e : lexp) : exp :=
  match e with
  | Econst c ⇒ Const c
  | Evar x ⇒ tovar x
  | Ewhen e c x ⇒ translate_lexp e
  | Eop op es ⇒ Op op (map translate_lexp es)
  end.

Fixpoint translate_cexp (x: ident) (e: cexp) : stmt :=
  match e with
  | Emerge y t f ⇒ Ifte (tovar y) (translate_cexp x t)
                                  (translate_cexp x f)
  | Eexp l ⇒ Assign x (translate_lexp l)
  end.

Definition translate_eqn (eqn: equation) : stmt :=
  match eqn with
  | EqDef x ck ce   ⇒ Control ck (translate_cexp x ce)
  | EqApp x ck f les ⇒ Control ck (Step_ap x f x (map translate_lexp les))
  | EqFby x ck v le ⇒ Control ck (AssignSt x (translate_lexp le))
  end.
```

```
Definition translate_eqns (eqns: list equation) : stmt :=
  fold_left (fun i eq ⇒ Comp (translate_eqn eq) i) eqns Skip.

Definition translate_reset_eqn (s: stmt) (eqn: equation) : stmt :=
  match eqn with
  | EqDef _ _ _ ⇒ s
  | EqFby x _ v0 _ ⇒ Comp (AssignSt x (Const v0)) s
  | EqApp x _ f _ ⇒ Comp (Reset_ap f x) s
  end.

Definition translate_reset_eqns (eqns: list equation): stmt :=
  fold_left translate_reset_eqn eqns Skip.

Definition ps_from_list (l: list ident) : PS.t :=
  fold_left (fun s i⇒PS.add i s) l PS.empty.

Definition translate_node (n: node): class :=
  let names := gather_eqs n.(n_eqs) in
  let mems := ps_from_list (fst names) in
  mk_class n.(n_name) n.(n_input) n.(n_output)
           (fst names) (snd names)
           (translate_eqns mems n.(n_eqs))
           (translate_reset_eqns n.(n_eqs)).

Definition translate (G: global) : program := map translate_node G.
```

Correction de la traduction

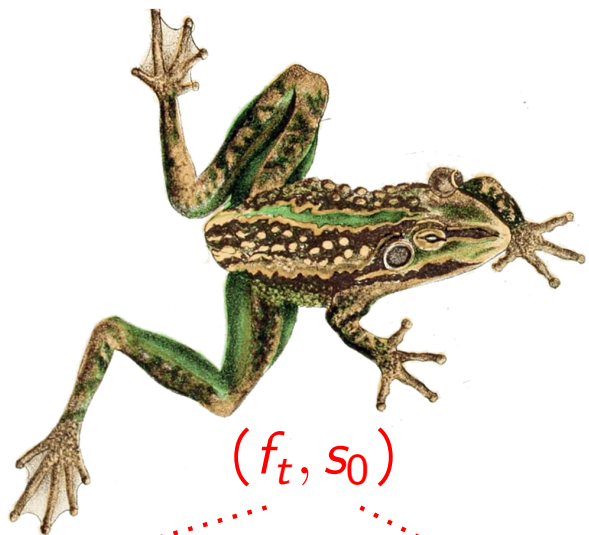


Correction de la traduction



`sem_node G f xss yss`

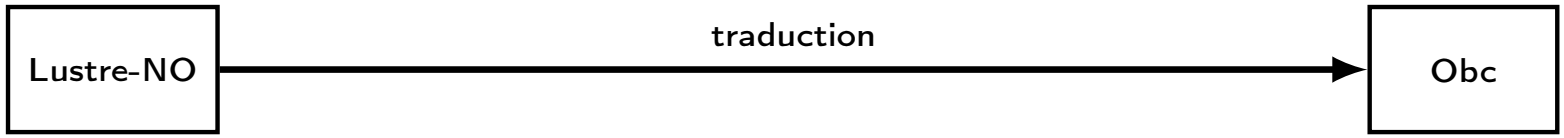
$\mathsf{stream}(T^+) \rightarrow \mathsf{stream}(T^+)$



(f_t, s_0)

$S \times T^+ \rightarrow T^+ \times S$ $\qquad$ S

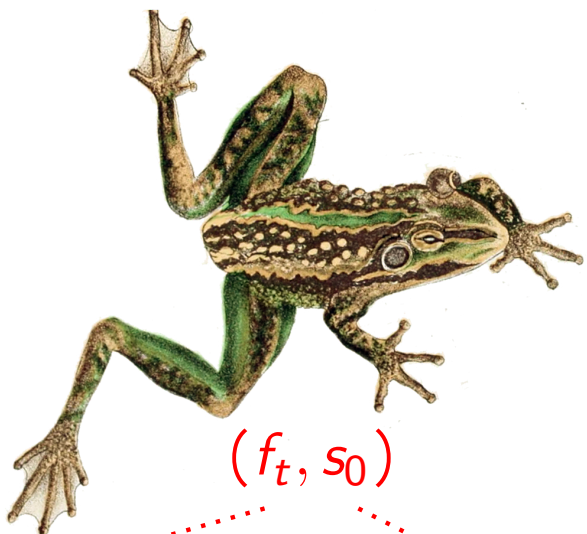
Correction de la traduction



L'énoncé est trop faible pour une preuve directe par récurrence ✗

`sem_node G f xss yss`

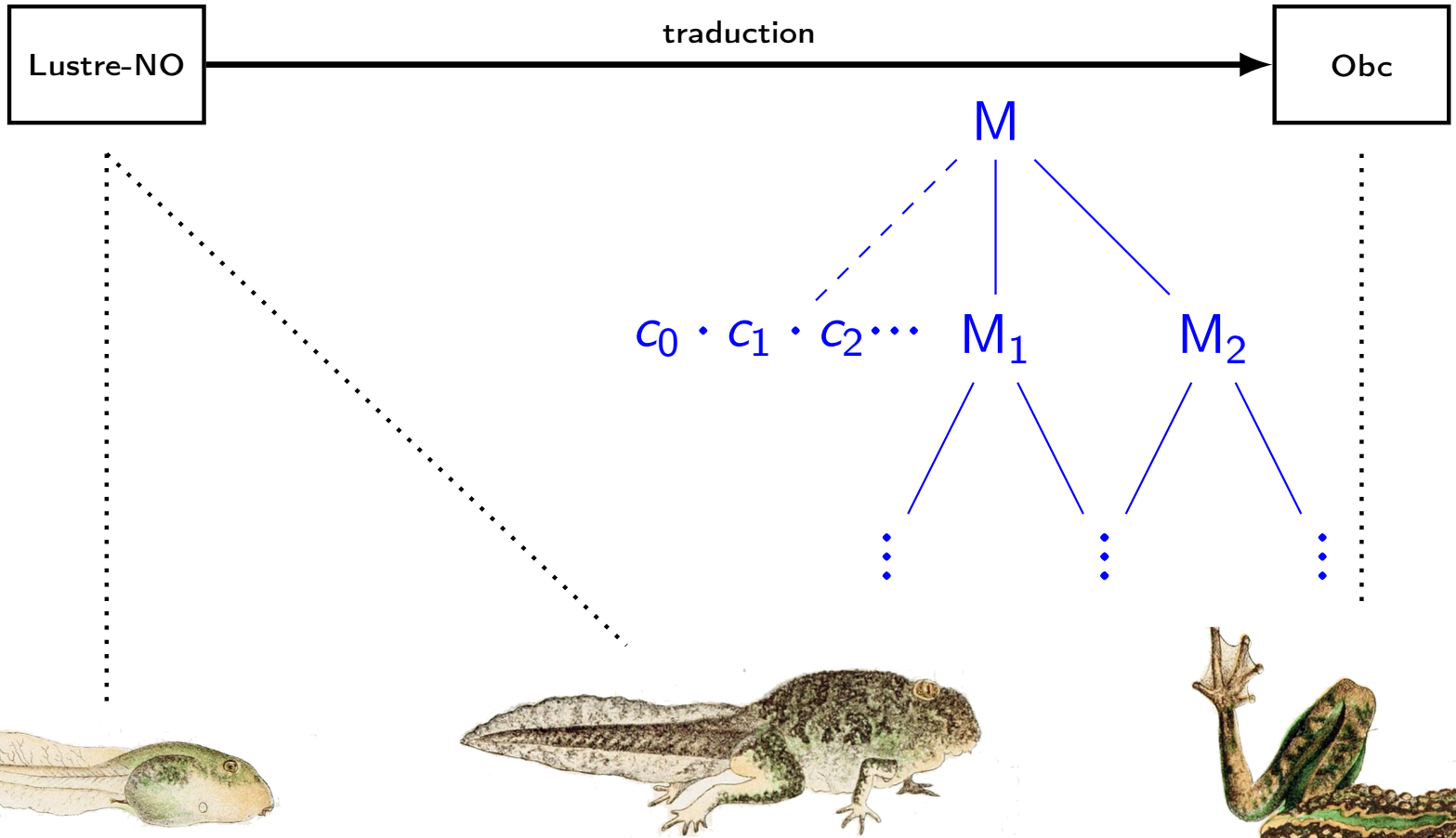
$\mathsf{stream}(T^+) \to \mathsf{stream}(T^+)$

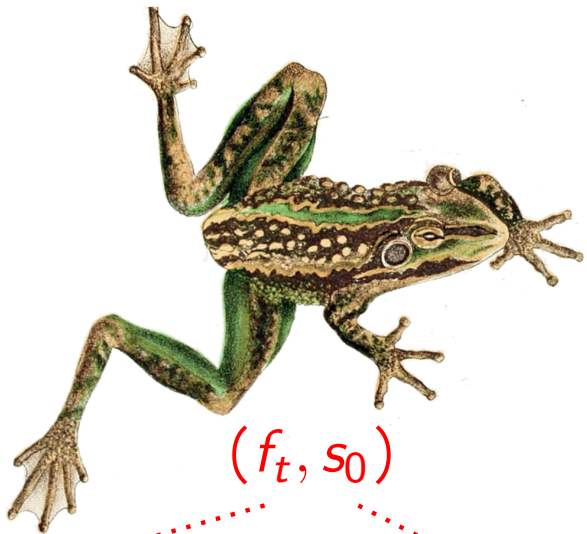


(f_t, s_0)

$S \times T^+ \to T^+ \times S$ $\qquad$ S

Correction de la traduction





`sem_node G f xss yss`

`msem_node G f xss M yss`

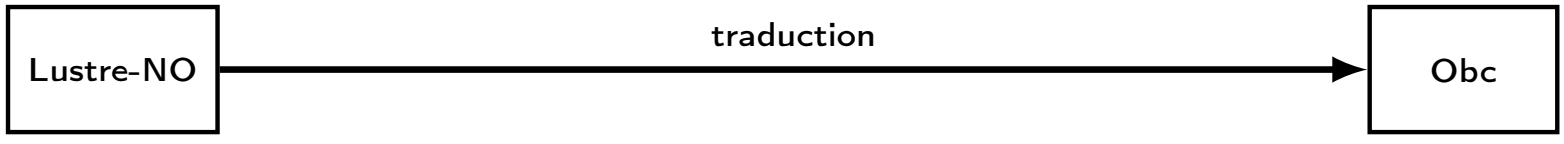
$\text{stream}(T^+) \rightarrow \text{stream}(T^+)$

(f_t, s_0)

$S \times T^+ \rightarrow T^+ \times S$

S

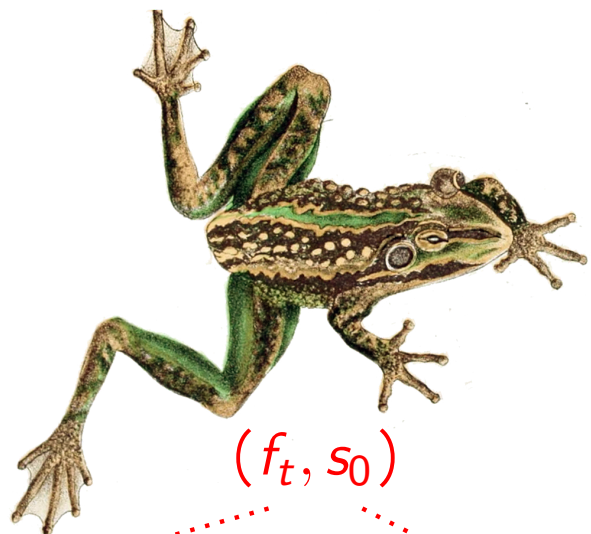
Correction de la traduction



preuve facile : $\exists M$

`sem_node G f xss yss` `msem_node G f xss M yss`

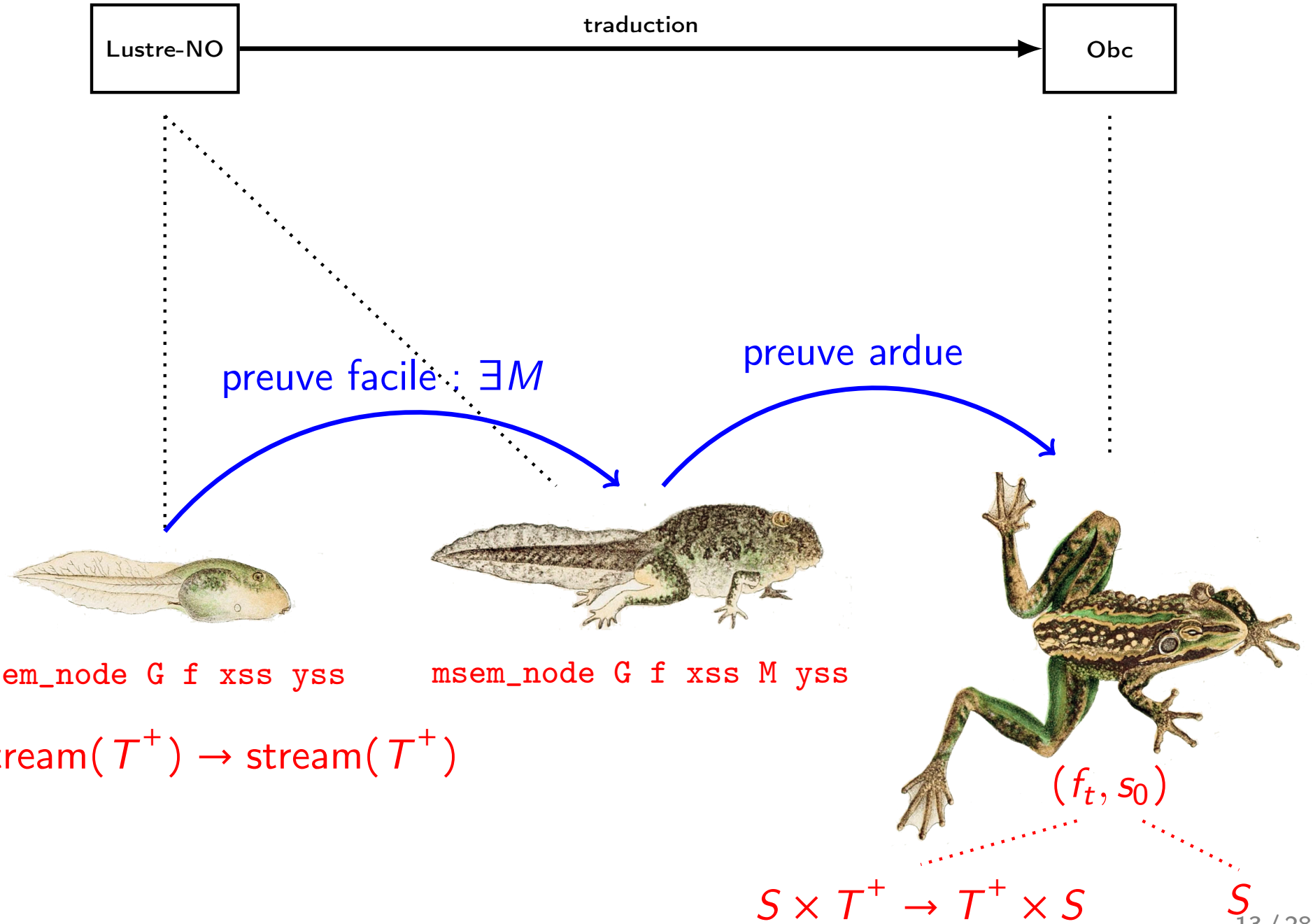
$\text{stream}(T^+) \to \text{stream}(T^+)$



(f_t, s_0)

$S \times T^+ \to T^+ \times S$ S

Correction de la traduction

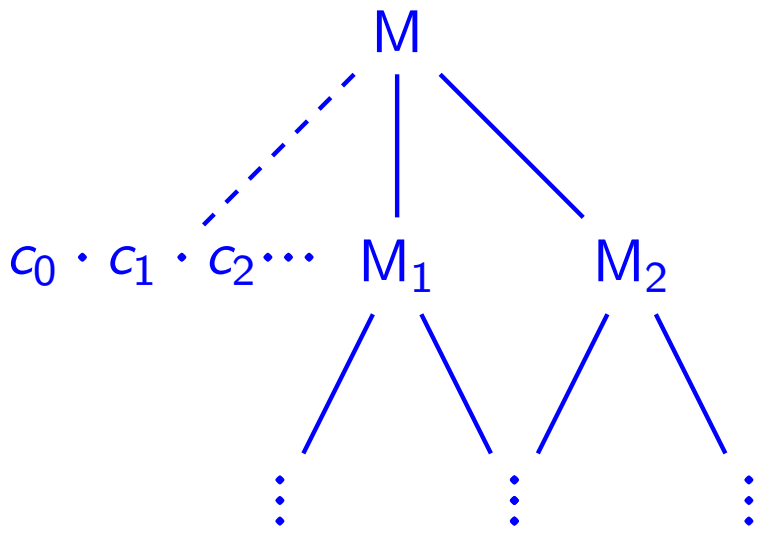


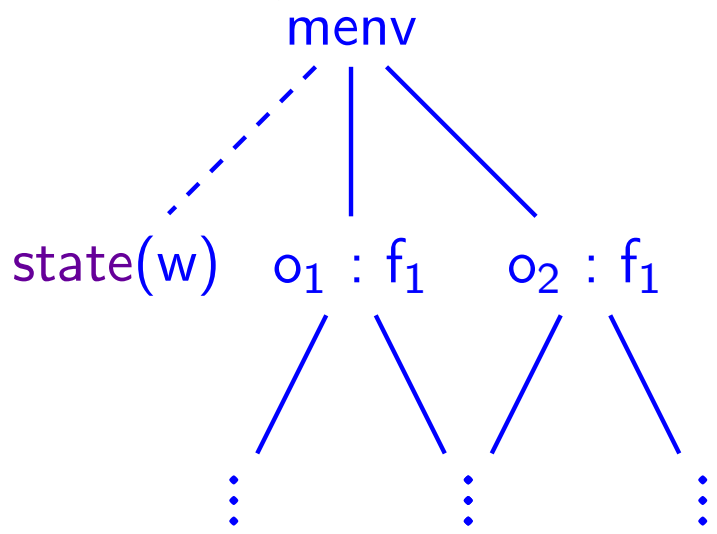
Correction de la traduction

- récurrence n
  - récurrence G
    - récurrence eqs
      - cas : $x = (ce)^{ck}$
        - cas : present
        - cas : absent
      - cas : $x = (f\ e)^{ck}$
        - cas : present
        - cas : absent
      - cas : $x = (k\ \texttt{fby}\ e)^{ck}$
        - cas : present
        - cas : absent

- Preuve ardue pleine de détails techniques.
- ≈100 lemmes
- Plusieurs itérations pour trouver les bonnes définitions.
- Le modèle intermédiaire joue un rôle clé.

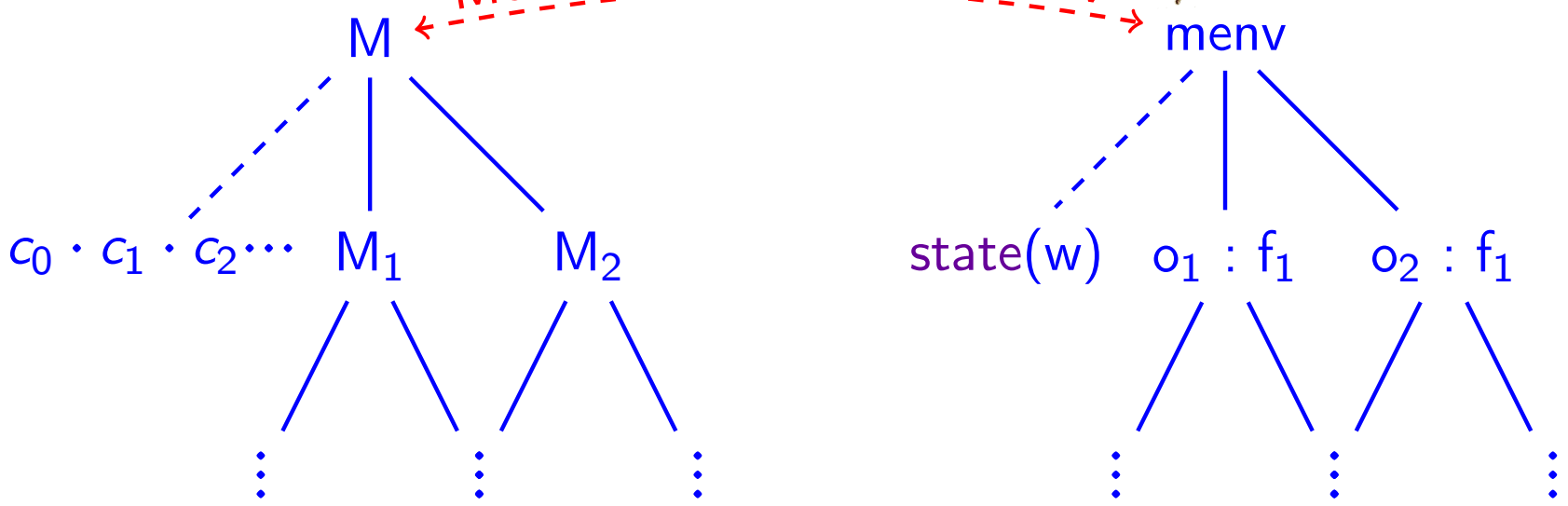
Lustre-NO à Obc : correspondance entre mémoires





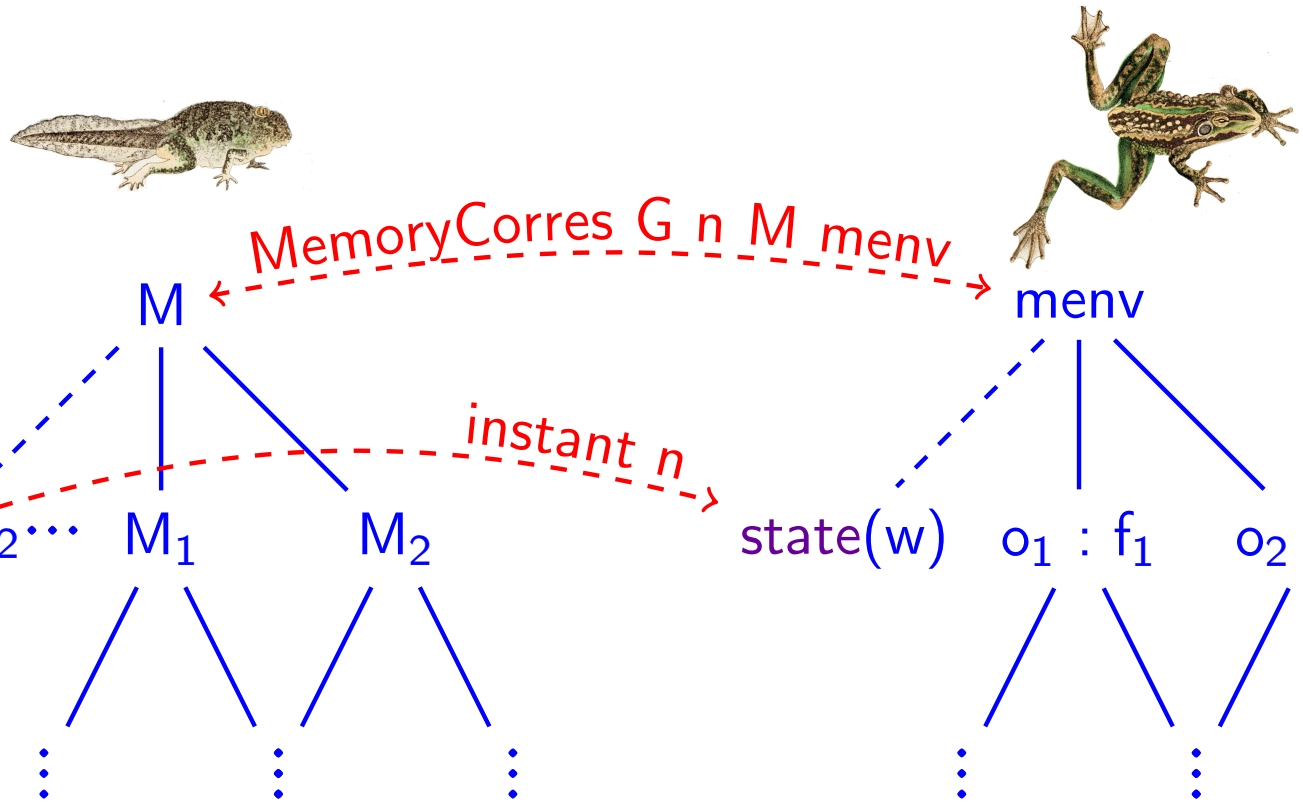
- La représentation de la mémoire ne change pas beaucoup entre Lustre-NO et Obc : il y a une correspondance directe à chaque instant.
- La difficulté est plutôt dans le basculement entre modèles sémantiques : on passe des flots de données vers des séquences d'affectations.

Lustre-NO à Obc : correspondance entre mémoires



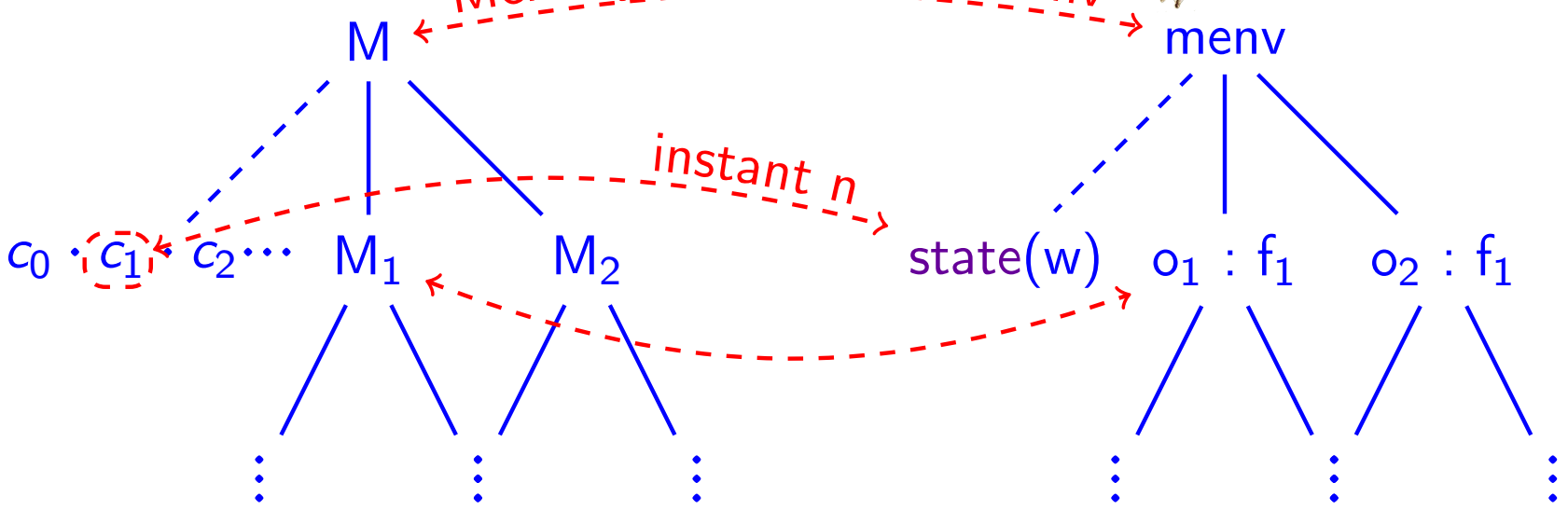
- La représentation de la mémoire ne change pas beaucoup entre Lustre-NO et Obc : il y a une correspondance directe à chaque instant.
- La difficulté est plutôt dans le basculement entre modèles sémantiques : on passe des flots de données vers des séquences d'affectations.

Lustre-NO à Obc : correspondance entre mémoires



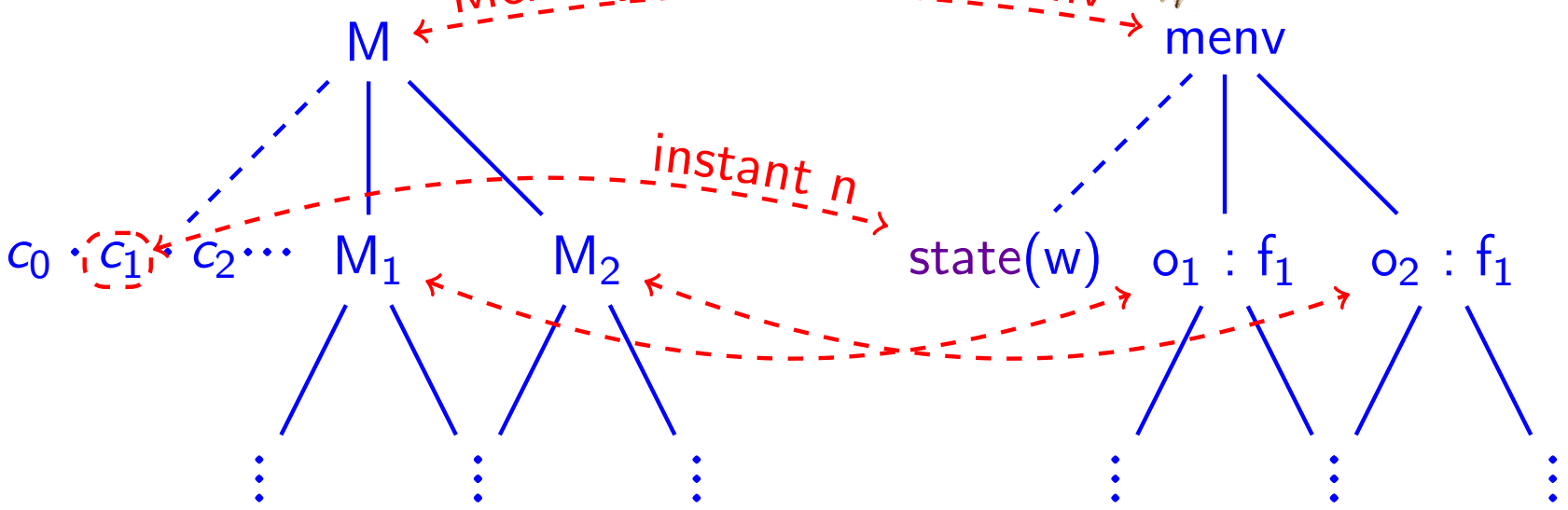
- La représentation de la mémoire ne change pas beaucoup entre Lustre-NO et Obc : il y a une correspondance directe à chaque instant.
- La difficulté est plutôt dans le basculement entre modèles sémantiques : on passe des flots de données vers des séquences d'affectations.

Lustre-NO à Obc : correspondance entre mémoires



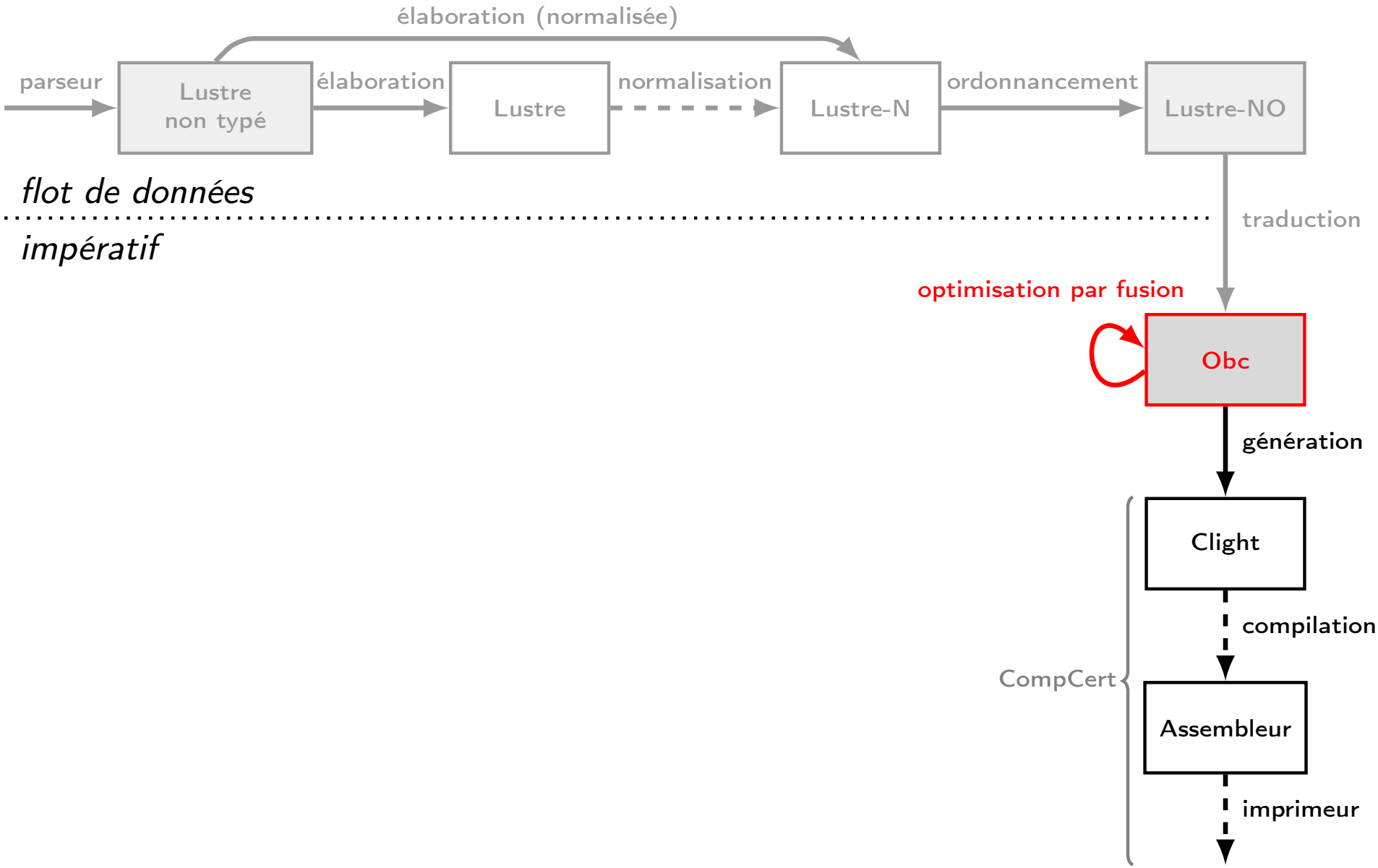
- La représentation de la mémoire ne change pas beaucoup entre Lustre-NO et Obc : il y a une correspondance directe à chaque instant.
- La difficulté est plutôt dans le basculement entre modèles sémantiques : on passe des flots de données vers des séquences d'affectations.

Lustre-NO à Obc : correspondance entre mémoires



- La représentation de la mémoire ne change pas beaucoup entre Lustre-NO et Obc : il y a une correspondance directe à chaque instant.
- La difficulté est plutôt dans le basculement entre modèles sémantiques : on passe des flots de données vers des séquences d'affectations.

Optimisation : fusion des structures de contrôle (Obc à Obc)



Fusion des structures de contrôle

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

```
step(delta: int, sec: bool)
    returns (v: int) {
  var r, t : int;

  r := count.step o1 (0, delta, false);
  if sec then {
    t := count.step o2 (1, 1, false)
  };
  if sec then {
    v := r / t
  } else {
    v := state(w)
  };
  state(w) := v
}
```

```
step(delta: int, sec: bool)
    returns (v: int) {
  var r, t : int;

  r := count.step o1 (0, delta, false);
  if sec then {
    t := count.step o2 (1, 1, false);
    v := r / t
  } else {
    v := state(w)
  };

  state(w) := v
}
```

- Génération des conditionnelles équation par équation et fusion par la suite : répartition des obligations de preuve.
- L'ordonnanceur essaie de regrouper les équations par horloge.
- On utilise tout le développement pour justifier l'invariant requis.

We also define the function $Join(.,.)$ which merges two control structures gathered by the same guards:

$$
\begin{aligned}
&Join(\ \texttt{case}\ (x)\ \{C_1 : S_1; ...; C_n : S_n\},\\
&\qquad \texttt{case}\ (x)\ \{C_1 : S'_1; ...; C_n : S'_n\})\\
&\quad = \texttt{case}\ (x)\ \{C_1 : Join(S_1, S'_1); ...; C_n : Join(S_n, S'_n)\}\\
&Join(S_1, S_2) = S_1; S_2
\end{aligned}
$$

$$
\begin{aligned}
&JoinList(S) = S\\
&JoinList(S_1, ..., S_n) = Join(S_1, JoinList(S_2, ..., S_n))
\end{aligned}
$$

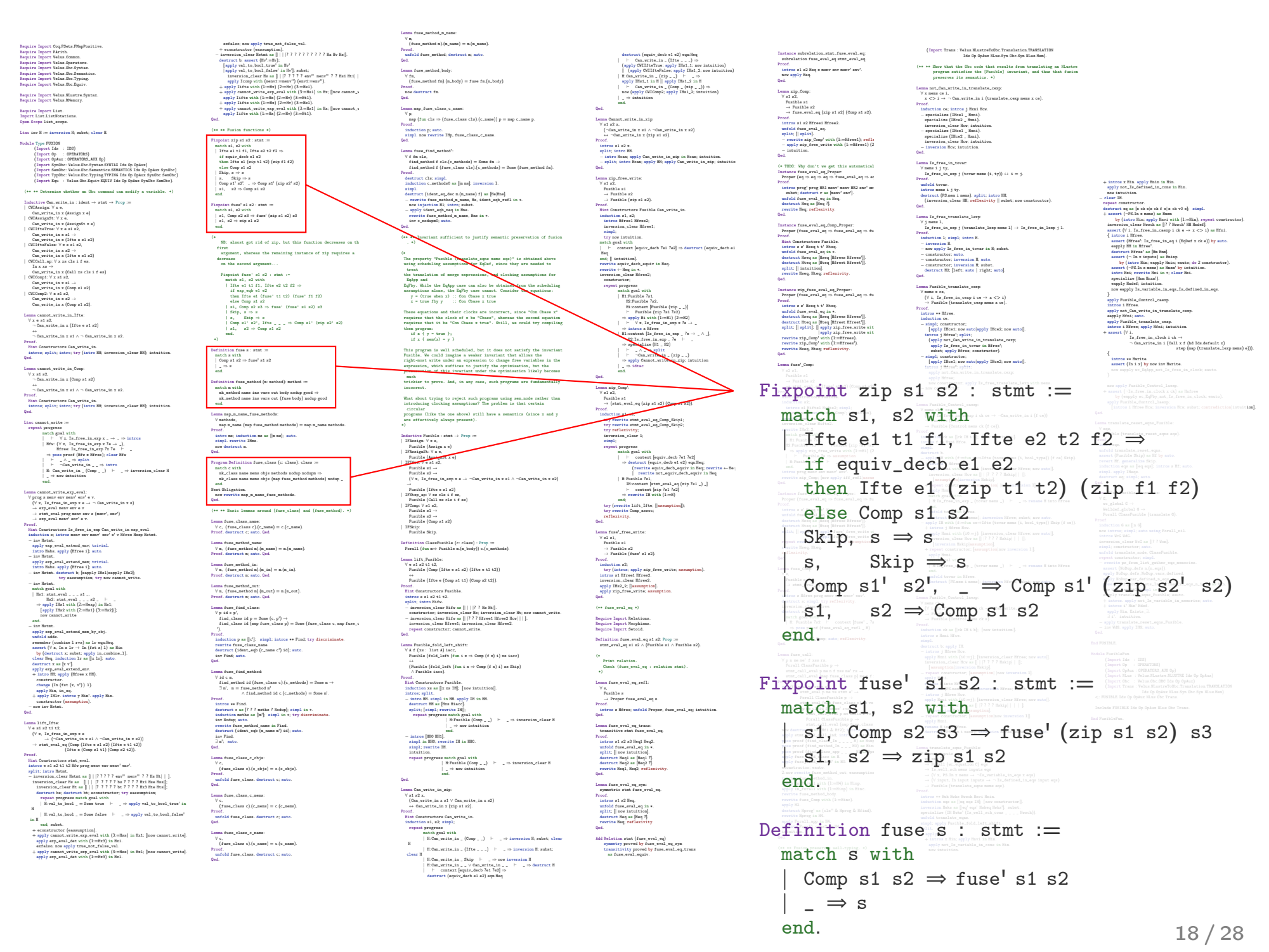
[Biernacki, Colaço, Hamon et Pouzet (2008): "Clock-directed modular code generation for synchronous data-flow languages"]

We also define the function $Join(.,.)$ which merges two control structures gathered by the same guards:

$$
\begin{aligned}
&Join(\texttt{case}\ (x)\ \{C_1 : S_1; ...; C_n : S_n\},\\
&\qquad \texttt{case}\ (x)\ \{C_1 : S'_1; ...; C_n : S'_n\})\\
&\quad = \texttt{case}\ (x)\ \{C_1 : Join(S_1, S'_1); ...; C_n : Join(S_n, S'_n)\}\\
&Join(S_1, S_2) = S_1; S_2
\end{aligned}
$$

$$
\begin{aligned}
&JoinList(S) = S\\
&JoinList(S_1, ..., S_n) = Join(S_1, JoinList(S_2, ..., S_n))
\end{aligned}
$$

```
Fixpoint zip s1 s2 : stmt :=
  match s1, s2 with
  | Ifte e1 t1 f1, Ifte e2 t2 f2 ⇒
    if equiv_decb e1 e2
    then Ifte e1 (zip t1 t2) (zip f1 f2)
    else Comp s1 s2
  | Skip, s ⇒ s
  | s,    Skip ⇒ s
  | Comp s1' s2', _ ⇒ Comp s1' (zip s2' s2)
  | s1,   s2 ⇒ Comp s1 s2
  end.

Fixpoint fuse' s1 s2 : stmt :=
  match s1, s2 with
  | s1, Comp s2 s3 ⇒ fuse' (zip s1 s2) s3
  | s1, s2 ⇒ zip s1 s2
  end.

Definition fuse s : stmt :=
  match s with
  | Comp s1 s2 ⇒ fuse' s1 s2
  | _ ⇒ s
  end.
```



```
Fixpoint zip s1 s2 : stmt :=
  match s1, s2 with
  | Ifte e1 t1 f1, Ifte e2 t2 f2 ⇒
    if equiv_decb e1 e2
    then Ifte e1 (zip t1 t2) (zip f1 f2)
    else Comp s1 s2
  | Skip, s ⇒ s
  | s,    Skip ⇒ s
  | Comp s1' s2', _ ⇒ Comp s1' (zip s2' s2)
  | s1,   s2 ⇒ Comp s1 s2
  end.

Fixpoint fuse' s1 s2 : stmt :=
  match s1, s2 with
  | Ifte e1 t1 f1, Ifte e2 t2 f2 ⇒
    if equiv_decb e1 e2
    then Ifte e1 (fuse' t1 t2) (fuse' f1 f2)
    else Comp s1 s2
  | s1, Comp s2 s3 ⇒ fuse' (fuse' s1 s2) s3
  | s1, s2 ⇒ zip s1 s2
  end.

Definition fuse s : stmt :=
  match s with
  | Comp s1 s2 ⇒ fuse' s1 s2
  | _ ⇒ s
  end.
```

```
Require Import Coq.PArith.PArithPositive.
Require Import FAvInth.
Require Import Velus.Common.
Require Import Velus.Operators.
Require Import Velus.Obc.Syntax.
Require Import Velus.Obc.Semantics.
Require Import Velus.Obc.Typing
Require Import Velus.Obc.Equiv.

Require Import Velus.NLustre.Syntax.
Require Import Velus.NMemory.

Require Import List.
Import List.ListNotations.
Open Scope list_scope.

Ltac inv H := inversion H; subst; clear H.

Module Type FUSION
     (Import Ids   : IDS)
     (Import Op    : OPERATORS)
     (Import OpAux : OPERATORS_AUX Op)
     (Import SynObc: Velus.Obc.Syntax.SYNTAX Ids Op OpAux)
     (Import SemObc: Velus.Obc.Semantics.SEMANTICS Ids Op OpAux SynObc)
     (Import TypObc: Velus.Obc.Typing.TYPING Ids Op OpAux SynObc SemObc)
     (Import Equiv : Velus.Obc.Equiv.EQUIV Ids Op OpAux SynObc SemObc).

  (** ** Determine whether an Obc command can modify a variable. *)

  Inductive Can_write_in : ident → stmt → Prop :=
  | CWAssign: ∀ x e,
      Can_write_in x (Assign x e)
  | CWAssignSt: ∀ x e,
      Can_write_in x (AssignSt x e)
  | CWIfteTrue: ∀ x e s1 s2,
      Can_write_in x s1 →
      Can_write_in x (Ifte e s1 s2)
  | CWIfteFalse: ∀ x e s1 s2,
      Can_write_in x s2 →
      Can_write_in x (Ifte e s1 s2)
  | CWCall_sp: ∀ x xs cls i f es,
      In x xs →
      Can_write_in x (Call xs cls i f es)
  | CWComp1: ∀ x s1 s2,
      Can_write_in x s1 →
      Can_write_in x (Comp s1 s2)
  | CWComp2: ∀ x s1 s2,
      Can_write_in x s2 →
      Can_write_in x (Comp s1 s2).

  Lemma cannot_write_in_Ifte:
    ∀ x e s1 s2,
      ¬Can_write_in x (Ifte e s1 s2)
      ↔
      ¬Can_write_in x s1 ∧ ¬Can_write_in x s2.
  Proof.
    Hint Constructors Can_write_in.
    intros; split; intro; try (intros HH; inversion_clear HH); intuition.
  Qed.

  Lemma cannot_write_in_Comp:
    ∀ x s1 s2,
      ¬Can_write_in x (Comp s1 s2)
      ↔
      ¬Can_write_in x s1 ∧ ¬Can_write_in x s2.
  Proof.
    Hint Constructors Can_write_in.
    intros; split; intro; try (intros HH; inversion_clear HH); intuition.
  Qed.

  Ltac cannot_write :=
    repeat progress
           match goal with
           | ⊢ ∀ x, Is_free_in_exp x _ → _ ⇒ intros
           | Hfw: ∀ x, Is_free_in_exp x ?e → _,
                  Hfree: Is_free_in_exp ?e ?n ⊢ _
             ⇒ pose proof (Hfw n Hfree); clear Hfw
           | ⊢ _ ∧ _ ⇒ split
           | ⊢ ¬Can_write_in _ _ ⇒ intro
           | H: Can_write_in _ (Comp _ _) ⊢ _ ⇒ inversion_clear H
           | _ ⇒ now intuition
           end.

  Lemma cannot_write_exp_eval:
    ∀ prog menv menv' env' e v,
      (∀ x, Is_free_in_exp e x → ¬ Can_write_in x s)
      → exp_eval menv env e v
      → stmt_eval prog menv env s (menv', env')
      → exp_eval menv' env' e v.
  Proof.
    Hint Constructors Is_free_in_exp Can_write_in exp_eval.
    induction e; intros menv menv' env' v' vfree Hexp Heval.
    - inv Heval.
      apply exp_eval_extend_env; trivial.
      intro Habs. apply (Hfree i) ; auto.
    - inv Heval.
      apply exp_eval_extend_mem; trivial.
      intro Habs. apply (Hfree i) ; auto.
    - inv Heval. destruct b; [apply IHe1|apply IHe2];
                   try eassumption; try now cannot_write.
    - inv Heval.
      match goal with
      | Hs1: stmt_eval _ _ _ s1 _,
             Hs2: stmt_eval _ _ _ s2 _ ⊢ _
        ⇒ apply IHs1 with (2:=Hexp) in Hs1;
          [|apply IHe2 with (2:=Hs1) (3:=Hs2)];
          now cannot_write
      end.
    - inv Heval.
      apply exp_eval_extend_mem_by_obj.
      unfold adds.
      remember (combine l vs) as lr eqn:Heq.
      assert (∀ x, In x lr ↔ In (fst x) l) as Hin
        by (destruct x; subst; apply in_combine_l).
      clear Heq. induction lr as [|x lr]; auto.
      destruct x as [a v]; simpl in *.
      apply exp_eval_extend_env.
      + intro HI; apply (Hfree a HI).
        constructor.
        change (In (fst (x, v')) l).
        apply Hin. in_eq.
      + apply IHlr. intros y Hiny'. apply Hin.
        constructor (assumption).
    - now inv Heval.
  Qed.

  Lemma lift_Ifte:
    ∀ e s1 s2 v1 t2,
      (∀ x, Is_free_in_exp x e
            → (¬Can_write_in x s1 ∧ ¬Can_write_in x s2))
      → stmt_eval prog (Ifte e s1 s2) (Ifte e v1 t2))
                  (Ifte e (Comp s1 v1) (Comp s2 t2)).
  Proof.
    Hint Constructors stmt_eval.
    intros e s1 s2 t1 t2 HFv prog menv env menv' env'.
    split; intro Heval.
    - inversion_clear Heval as [| | |? ? ? ? ? env'' menv'' ? ? Hs Ht| | ].
      inversion_clear Hs as [| | |? ? ? ? ? bs ? ? ? ? Hs1 Hss Hss'|];
        inversion_clear Hss as [| | |? ? ? ? ? bs ? ? ? ? Hs2 Hss Hss'|];
        destruct bs; destruct bt; econstructor; try eassumption;
          repeat progress match goal with
                  | H: val_to_bool _ = Some true ⊢ _ ⇒ apply val_to_bool_true' in H
                  | H: val_to_bool _ = Some false ⊢ _ ⇒ apply val_to_bool_false' in H
                  end; subst.
      + econstructor (eassumption).
      + apply cannot_write_exp_eval with (3:=Hss) in Hs1; [now cannot_write|].
        apply exp_eval_det with (1:=Hs1) in Hs1.
        exfalso; now apply true_not_false_val.
      + apply cannot_write_exp_eval with (3:=Hss') in Hs1; [now cannot_write|].
        apply exp_eval_det with (1:=Hs1) in Hs1.
        exfalso; now apply true_not_false_val.
      + econstructor (eassumption).
    - inversion_clear Heval as [| | |? ? ? ? ? ? ? ? Hs Ht Ha|].
      destruct b; assert (Hv':=Hv);
        [apply val_to_bool_true' in Hv'
        |apply val_to_bool_false' in Hv']; subst;
          inversion_clear Ha as [| | |? ? ? ? ? env''' menv''' ? ? Hs1 Ht1| |
                                 apply Icomp with (menv1:=menv'') (env1:=env'')].
      + apply IHe1s with (1:=Hs) (2:=Hv) (3:=Hs1).
      + apply cannot_write_exp_eval with (3:=Hs1) in Hx; [now cannot_write|].
        apply IHe1s with (1:=Hx) (2:=Hv) (3:=Ht1).
      + apply cannot_write_exp_eval with (3:=Hs1) in Hx; [now cannot_write|].
        apply IHe1s with (1:=Hx) (2:=Hv) (3:=Ht1).
  Qed.

  (** ** Fusion functions *)

  Fixpoint zip s1 s2 : stmt :=
    match s1, s2 with
    | Ifte e1 t1 f1, Ifte e2 t2 f2 ⇒
      if equiv_decb e1 e2
      then Ifte e1 (zip t1 t2) (zip f1 f2)
      else Comp s1 s2
    | Skip, s ⇒ s
    | s, Skip ⇒ s
    | Comp s1' s2', _ ⇒ Comp s1' (zip s2' s2)
    | s1, s2 ⇒ Comp s1 s2
    end.

  Fixpoint fuse' s1 s2 : stmt :=
    match s1, s2 with
    | s1, Comp s2 s3 ⇒ fuse' (zip s1 s2) s3
    | s1, s2 ⇒ zip s1 s2
    end.

  (*
      NB: almost got rid of zip, but this function decreases on th
      first
      argument, whereas the remaining instance of zip requires a
      decrease
      on the second argument...

      Fixpoint fuse' s1 s2 : stmt :=
        match s1, s2 with
        | Ifte e1 t1 f1, Ifte e2 t2 f2 ⇒
          if equiv_decb e1 e2
          then Ifte e1 (fuse' t1 t2) (fuse' f1 f2)
          else Comp s1 s2
        | s1, Comp s2 s3 ⇒ fuse' (fuse' s1 s2) s3
        | Skip, s ⇒ s
        | s, _ ⇒ s
        | Comp s1' s2', Ifte _ _ _ ⇒ Comp s1' (zip s2' s2)
        | s1, _ ⇒ Comp s1 s2
        end.
  *)

  Definition fuse s : stmt :=
    match s with
    | Comp s1 s2 ⇒ fuse' s1 s2
    | _ ⇒ s
    end.

  Definition fuse_method (m: method): method :=
    match m with
      mk_method name ins vars out body nodup good ⇒
      mk_method name ins vars out (fuse body) nodup good
    end.

  Lemma map_m_name_fuse_methods:
    ∀ methods,
      map m_name (map fuse_method methods) = map m_name methods.
  Proof.
    intro ms; induction ms as [|m ms]; auto.
    simpl; rewrite IHms.
    now destruct m.
  Qed.

  Program Definition fuse_class (c: class): class :=
    match c with
      mk_class name mems objs methods nodup nodupm ⇒
      mk_class name mems objs (map fuse_method methods) nodup _
    end.
  Next Obligation.
    now rewrite map_m_name_fuse_methods.
  Qed.

  (** ** Basic lemmas around [fuse_class] and [fuse_method]. *)

  Lemma fuse_class_name:
    ∀ c, (fuse_class c).(c_name) = c.(c_name).
  Proof. destruct c; auto. Qed.

  Lemma fuse_method_name:
    ∀ m, (fuse_method m).(m_name) = m.(m_name).
  Proof. destruct m; auto. Qed.

  Lemma fuse_method_in:
    ∀ m, (fuse_method m).(m_in) = m.(m_in).
  Proof. destruct m; auto. Qed.

  Lemma fuse_method_out:
    ∀ m, (fuse_method m).(m_out) = m.(m_out).
  Proof. destruct m; auto. Qed.

  Lemma fuse_find_class:
    ∀ p id c p',
      find_class id p = Some (c, p') →
      find_class id (map fuse_class p) = Some (fuse_class c, map fuse_c
    ).
  Proof.
    induction p as [|c']; simpl; intros ** Find; try discriminate.
    rewrite fuse_class_name.
    destruct (ident_eqb (c_name c') id); auto.
    inv Find; auto.
  Qed.

  Lemma fuse_find_method:
    ∀ id c m,
      find_method id (fuse_class c).(c_methods) = Some m →
      ∃ m', m = fuse_method m'
            ∧ find_method id c.(c_methods) = Some m'.
  Proof.
    intros ** Find.
    destruct c as [? ? ? methx ? Nodup]; simpl in *.
    induction methx as [|m']; simpl in *; try discriminate.
    inv Nodup; auto.
    rewrite fuse_method_name in Find.
    destruct (ident_eqb (m_name m') id); auto.
    inv Find.
    ∃ m'; auto.
  Qed.

  Lemma fuse_class_c_objs:
    ∀ c,
      (fuse_class c).(c_objs) = c.(c_objs).
  Proof.
    unfold fuse_class. destruct c; auto.
  Qed.

  Lemma fuse_class_c_mems:
    ∀ c,
      (fuse_class c).(c_mems) = c.(c_mems).
  Proof.
    unfold fuse_class. destruct c; auto.
  Qed.

  Lemma fuse_class_c_name:
    ∀ c,
      (fuse_class c).(c_name) = c.(c_name).
  Proof.
    unfold fuse_class. destruct c; auto.
  Qed.

  Lemma fuse_method_m_name:
    ∀ m,
      (fuse_method m).(m_name) = m.(m_name).
  Proof.
    unfold fuse_method; destruct m; auto.
  Qed.

  Lemma fuse_method_body:
    ∀ fm,
      (fuse_method fm).(m_body) = fuse fm.(m_body).
  Proof.
    now destruct fm.
  Qed.

  Lemma map_fuse_class_c_name:
    ∀ p,
      map (fun cls ⇒ (fuse_class cls).(c_name)) p = map c_name p.
  Proof.
    induction p; auto.
    simpl. now rewrite IHp, fuse_class_c_name.
  Qed.

  Lemma fuse_find_method':
    ∀ f fm cls,
      find_method f cls.(c_methods) = Some fm →
      find_method f (fuse_class cls).(c_methods) = Some (fuse_method fm).
  Proof.
    destruct cls; simpl.
    induction c_methods0 as [m m]; inversion 1.
    simpl.
    destruct (ident_eq_dec m.(m_name) f) as [Hm|Hm].
    - rewrite fuse_method_m_name, Hm, ident_eqb_refl in *.
      now injection H1; intro; subst.
    - apply ident_eqb_neq in Hm.
      rewrite fuse_method_m_name, Hm in *.
      inv c_methods0; auto.
  Qed.

  (** ** Invariant sufficient to justify semantic preservation of fusion
      . *)

  (*
    The property "Fusible (translate_eqns nems eqs)" is obtained above
    using scheduling assumptions for EqSwf, since they are needed to
    treat
    the translation of merge expressions, and clocking assumptions for
    EqApp and
    EqFby. While the EqApp case can also be obtained from the scheduling
    assumptions alone, the EqFby case cannot. Consider the equations:

        y = (true when x) ;: Con Cbase x true
        x = true fby y    ;: Con Cbase x true

    These equations and their clocks are incorrect, since "Con Cbase x"
    requires that the clock of x be "Cbase", whereas the second equation
    requires that it be "Con Cbase x true". Still, we could try compiling
    them program:

        if x { y = true };
        if x { mem(x) = y }

    This program is well scheduled, but it does not satisfy the invariant
    Fusible. We could imagine a weaker invariant that allows the
    right-most write under an expression to read free variables in the
    expression, which suffices to justify the optimisation, but the
    preservation of this invariant under the optimisation likely becomes
      much
    trickier to prove. And, in any case, such programs are fundamentally
    incorrect.

    What about trying to reject such programs using sem_node rather than
    introducing clocking assumptions? The problem is that certain
      circular
    programs (like the one above) still have a semantics (since x and y
    are effectively always present).
  *)

  Inductive Fusible : stmt → Prop :=
  | FAssign: ∀ x e,
      Fusible (Assign x e)
  | FAssignSt: ∀ x e,
      Fusible (AssignSt x e)
  | FIfte: ∀ x e1 s2,
      Fusible s1 →
      Fusible s2 →
      (∀ x, Is_free_in_exp x e → ¬Can_write_in x s1 ∧ ¬Can_write_in x s2)
      →
      Fusible (Ifte e s1 s2)
  | FIfteSp: ∀ x ck i f es,
      Fusible (Call xs cls i f es)
  | FComp: ∀ s1 s2,
      Fusible s1 →
      Fusible s2 →
      Fusible (Comp s1 s2)
  | FSkip:
      Fusible Skip.

  Definition ClassFusible (c: class) : Prop :=
    Forall Fusible m.(m_body)) c.(c_methods).

  Lemma lift_Fusible:
    ∀ e s1 s2 t1 t2.
      Fusible (Comp (Ifte e s1 s2) (Ifte e t1 t2))
      ↔
      Fusible (Ifte e (Comp s1 t1) (Comp s2 t2)).
  Proof.
    Hint Constructors Fusible.
    intros e s1 s2 t1 t2.
    split; intro Hifte.
    - inversion_clear Hifte as [| | | | | | | | | Hs Ht|].
      constructor; inversion_clear Hs; inversion_clear Ht; now cannot_write.
    - inversion_clear Hifte as [| |? ? ? Hfree1 Hfree2 Hcw| | |].
      inversion_clear Hfree1; inversion_clear Hfree2.
      repeat constructor; cannot_write.
  Qed.

  Lemma Fusible_fold_left_shift:
    ∀ A f (xs : list A) iacc,
      Fusible (fold_left (fun i x ⇒ Comp (f x i) x) xs iacc)
      ↔
      (Fusible (fold_left (fun i x ⇒ Comp (f x i) x) xs Skip)
       ∧ Fusible iacc).
  Proof.
    Hint Constructors Fusible.
    induction xs as [|x xs IH]. [now intuition].
    intros; split.
    - intro HH. simpl in HH. apply IH in HH.
      destruct HH as [Hxs Hiacc].
      split; [simpl; rewrite IH|];
        repeat progress match goal with
                        | H:Fusible (Comp _ _) ⊢ _ ⇒ inversion_clear H
                        | _ ⇒ now intuition
                        end.
    - intros [HH0 HH1].
      simpl in HH0; rewrite IH in HH0.
      simpl; rewrite IH.
      intuition.
      repeat progress match goal with
                      | H:Fusible (Comp _ _) ⊢ _ ⇒ inversion_clear H
                      | _ ⇒ now intuition
                      end.
  Qed.

  Lemma Can_write_in_zip:
    ∀ s1 s2 x,
      (Can_write_in x s1 ∨ Can_write_in x s2)
      ↔ Can_write_in x (zip s1 s2).
  Proof.
    Hint Constructors Can_write_in.
    induction s1, s2; simpl;
      repeat progress
             match goal with
             | H: Can_write_in _ (Comp _ _) ⊢ _ ⇒ inversion H; subst; clear H
             | H: Can_write_in _ (Ifte _ _ _) ⊢ _ ⇒ inversion H; subst;
           clear H
             | H: Can_write_in _ Skip ⊢ _ ⇒ now inversion H
             | H: Can_write_in _ ∨ Can_write_in _ _ ⊢ _ ⇒ destruct H
             | ⊢ context [equiv_decb ?e1 ?e2] ⇒
               destruct (equiv_decb e1 e2) eqn:Heq
             | ⊢ Can_write_in _ (Ifte _ _ _) ⇒
               (apply CWIfteTrue; apply IHs1_1; now intuition)
               || (apply CWIfteFalse; apply IHs1_2; now intuition)
             | H: Can_write_in _ (Comp _ (zip _ _)) ⇒
               apply IHs1_1 in H || apply IHs1_2 in H
             | ⊢ Can_write_in _ (Comp _ (zip _ _)) ⇒
               now (apply CWComp2; apply IHs1_2; intuition)
             | _ ⇒ intuition
             end.
  Qed.

  Lemma zip_Free_write:
    ∀ s1 s2,
      Fusible s1
      → Fusible s2
      → Fusible (zip s1 s2).
  Proof.
    Hint Constructors Fusible Can_write_in.
    induction s1, s2;
      try rewrite stmt_eval_eq_Comp_Skip1;
      try rewrite stmt_eval_eq_Comp_Skip2;
      try reflexivity;
      inversion_clear 1;
      simpl;
      repeat progress
             match goal with
             | ⊢ context [equiv_decb ?e1 ?e2]
               ⇒ destruct (equiv_decb e1 e2) eqn:Heq;
                 [ rewrite equiv_decb_equiv in Heq; rewrite ← Heq
                 | rewrite not_equiv_decb_equiv in Heq ]
             | H: Fusible ?s1,
                  IH:context [stmt_eval_eq (zip ?s1 _) _]
               ⊢ context [zip ?s1 ?s2]
               ⇒ rewrite IH with (1:=H)
             end;
      try (rewrite lift_Ifte; [eassumption]);
      try rewrite Comp_assoc;
      reflexivity.
  Qed.

  Lemma fuse'_free_write:
    ∀ s2 s1,
      Fusible s1
      → Fusible s2
      → Fusible (fuse' s1 s2).
  Proof.
    induction s2;
      try (intros; apply zip_Free_write; assumption).
    intros s1 Hfree1 Hfree2.
    inversion_clear Hfree2.
    apply IHs2_2; [|assumption].
    apply zip_Free_write; assumption.
  Qed.

  (** ** fuse_eval_eq *)

  Require Import Relations.
  Require Import Morphisms.
  Require Import Setoid.

  Definition fuse_eval_eq s1 s2: Prop :=
    stmt_eval_eq s1 s2 ∧ (Fusible s1 ∧ Fusible s2).

  (*
      Print relation.
      Check (fuse_eval_eq : relation stmt).
    *)

  Lemma fuse_eval_eq_refl:
    ∀ s,
      Fusible s
      → Proper fuse_eval_eq s.
  Proof.
    intros s Hfree; unfold Proper, fuse_eval_eq; intuition.
  Qed.

  Lemma fuse_eval_eq_trans:
    transitive stmt fuse_eval_eq.
  Proof.
    intros s1 s2 s3 Heq1 Heq2.
    unfold fuse_eval_eq in *.
    split; [| now intuition].
    destruct Heq1 as [Heq1 ?].
    destruct Heq2 as [Heq2 ?].
    rewrite Heq1, Heq2; reflexivity.
  Qed.

  Lemma fuse_eval_eq_sym:
    symmetric stmt fuse_eval_eq.
  Proof.
    intros s1 s2 Heq.
    unfold fuse_eval_eq in *.
    split; [| now intuition].
    destruct Heq as [Heq ?].
    rewrite Heq; reflexivity.
  Qed.

  Add Relation stmt (fuse_eval_eq)
      symmetry proved by fuse_eval_eq_sym
      transitivity proved by fuse_eval_eq_trans
      as fuse_eval_equiv.

  Instance subrelation_stmt_fuse_eval_eq
    : subrelation fuse_eval_eq stmt_eval_eq.
  Proof.
    intros s1 s2 Heq x menv env menv' env'.
    now apply Heq.
  Qed.

  Lemma zip_Comp:
    ∀ s1 s2,
      Fusible s1
      → Fusible s2
      → fuse_eval_eq (zip s1 s2) (Comp s1 s2).
  Proof.
    intros s1 s2 Hfree1 Hfree2.
    unfold fuse_eval_eq.
    split; [| split].
    - rewrite zip_Comp' with (1:=Hfree1); reflexivity.
    - apply zip_Free_write with (1:=Hfree1) (2:=Hfree2).
    - intro; apply Comp_assoc.
  Qed.

  (* TODO: Why don't we get this automatical
  Instance fuse_eval_eq_Proper:
    Proper (eq ⇒ eq ⇒ eq ⇒ fuse_eval_eq ⇒ eq ⇒ fuse_eval_eq ⇒ eq ⇒ eq ⇒
```

Fusion des structures de contrôle : invariant

```
if e then {s1} else {s2};
if e then {t1} else {t2}
```

```
if e then {s1; t1} else {s2; t2};
```

Fusion des structures de contrôle : invariant

```
if e then {s1} else {s2};
if e then {t1} else {t2}
```

```
if e then {s1; t1} else {s2; t2};
```

```
if x then {x := false} else {x := true};
if x then {t1} else {t2}
```

Fusion des structures de contrôle : invariant

```
if e then {s1} else {s2};
if e then {t1} else {t2}
```

```
if e then {s1; t1} else {s2; t2};
```

```
if x then {x := false} else {x := true};
if x then {t1} else {t2}
```

$$\frac{\text{fusionnable}(s_1) \qquad \text{fusionnable}(s_2) \qquad \forall x \in \text{libres}(e), \neg\text{peut-écrire } x\ s_1 \wedge \neg\text{peut-écrire } x\ s_2}{\text{fusionnable}(\text{if } e\ \{s_1\}\ \text{else}\ \{s_2\})}$$

$$\frac{\text{fusionnable}(s_1) \quad \text{fusionnable}(s_2)}{\text{fusionnable}(s_1; s_2)} \qquad \ldots$$

Fusion des structures de contrôle : correction

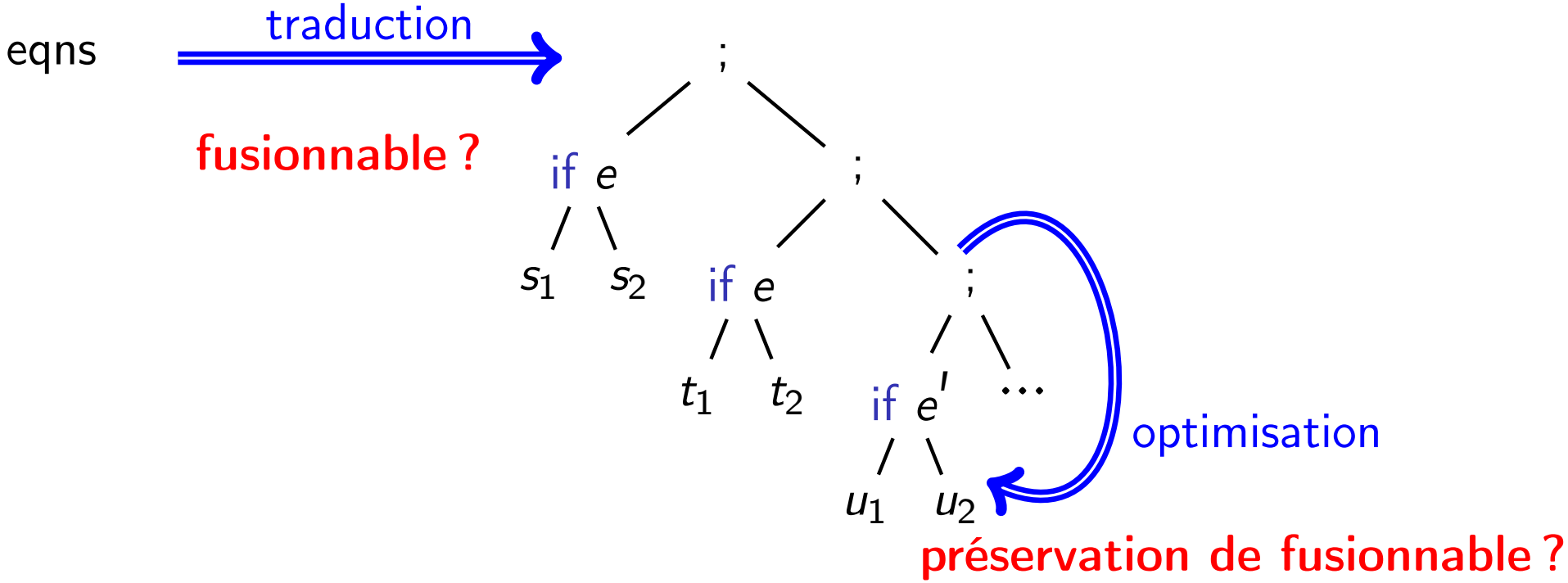
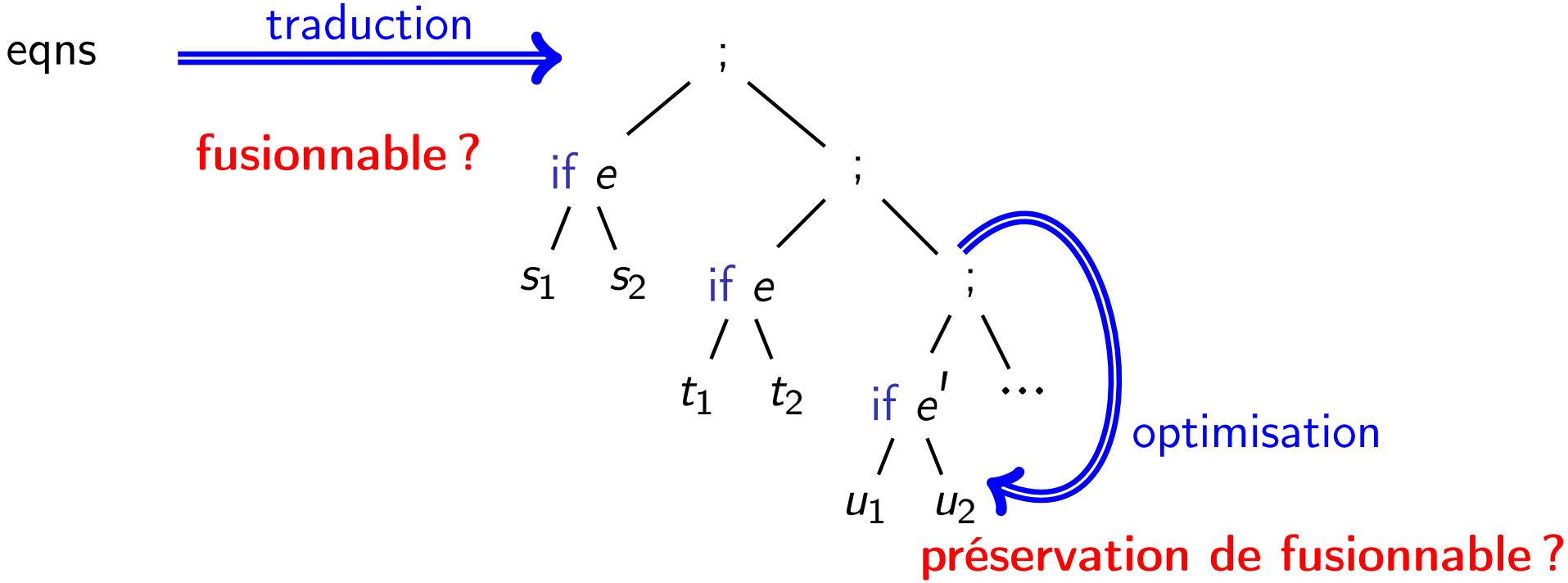


Schéma général

- Implémenter l'optimisation comme une fonction sur la syntaxe abstraite.
- Imaginer un invariant sous lequel la sémantique est préservée :
  - Il doit être satisfait par le code généré en amont.
  - Il doit être préservé par la fonction d'optimisation.

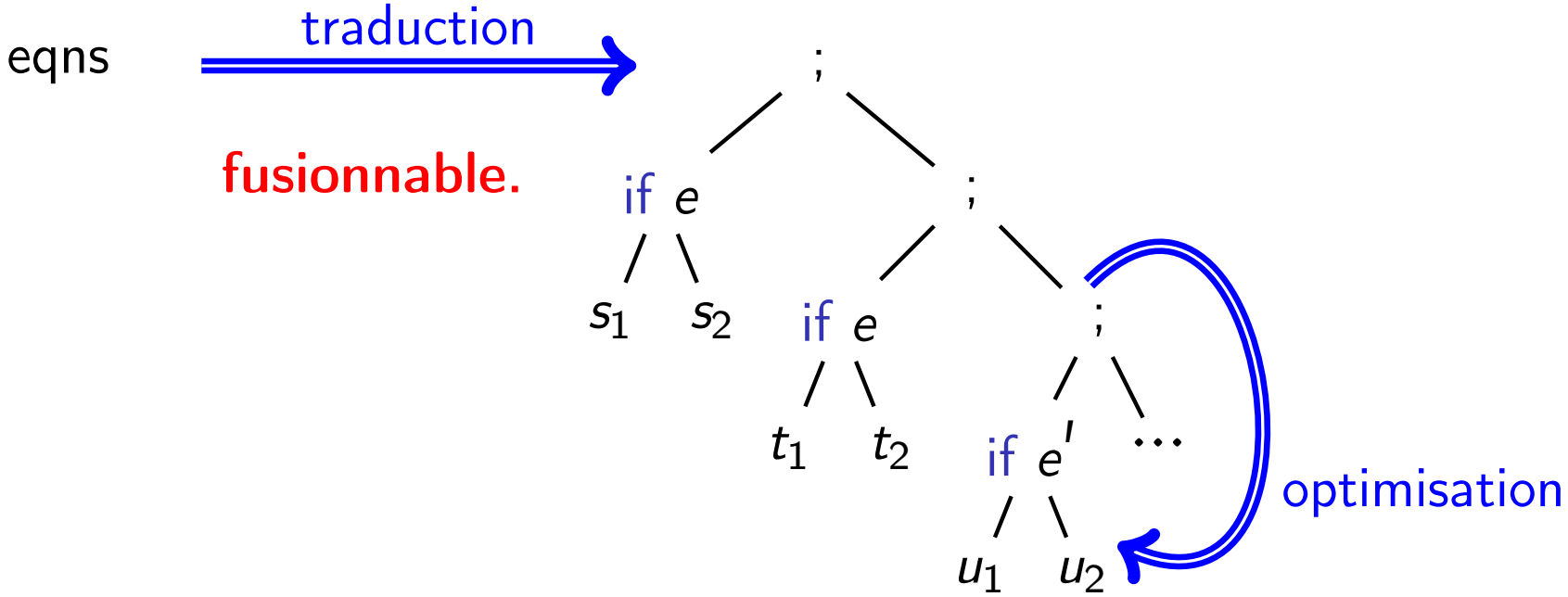
Fusion des structures de contrôle : correction



- Définir $s_1 \approx_{eval} s_2$

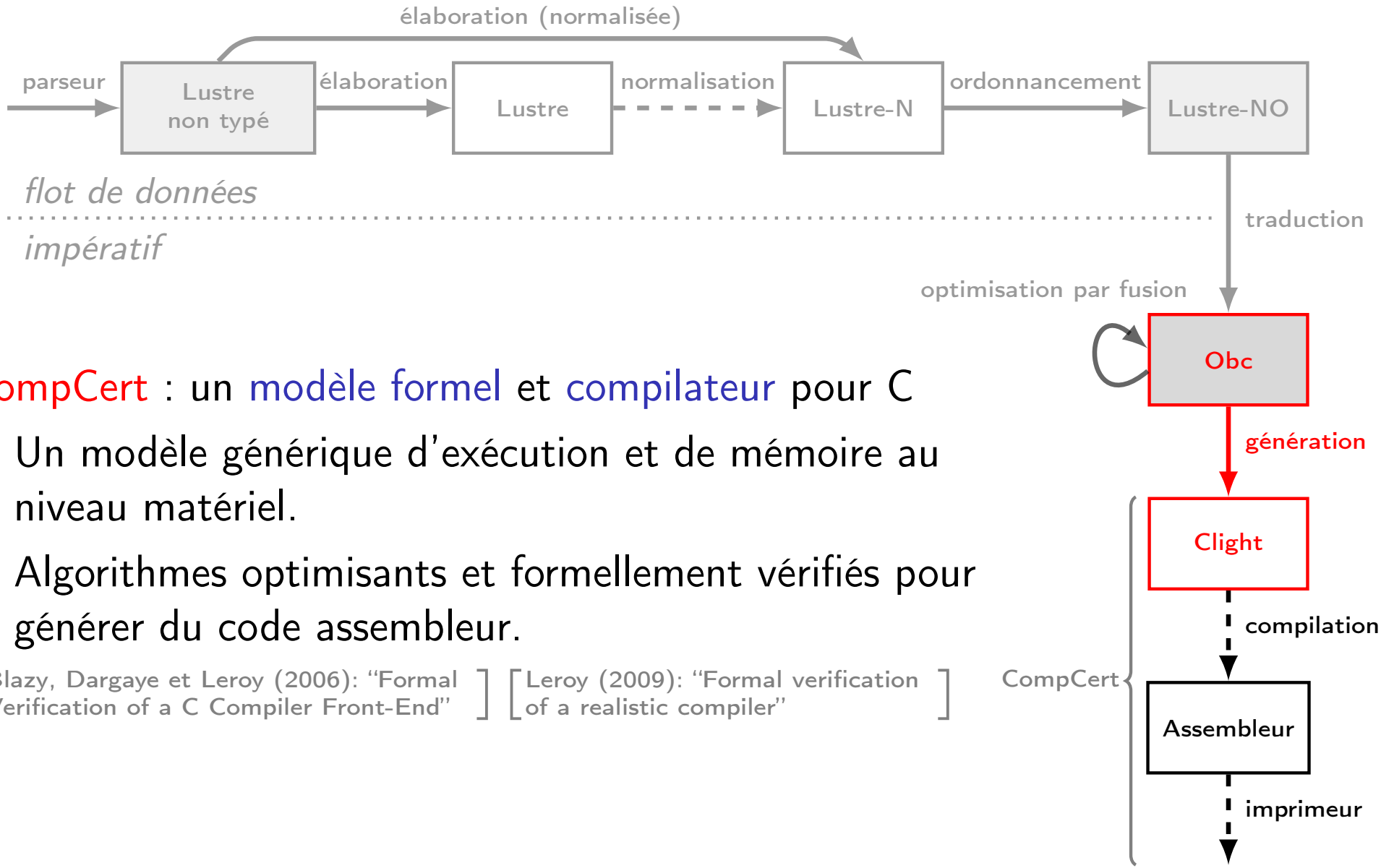
```
Definition stmt_eval_eq s1 s2: Prop :=
  ∀ prog menv env menv' env',
    stmt_eval prog menv env s1 (menv', env')
    ↔
    stmt_eval prog menv env s2 (menv', env').
```

Fusion des structures de contrôle : correction



- Définir $s_1 \approx_{eval} s_2$
- Définir $s_1 \approx_{fuse} s_2$ par $s_1 \approx_{eval} s_2 \wedge \text{fusionnable}(s_1) \wedge \text{fusionnable}(s_2)$
- Démontrer sa congruence pour ' ;', fuse, fuse' et zip.
- Preuves par réécriture pour obtenir : $\dfrac{\text{fusionnable}(s)}{\text{fuse}(s) \approx_{eval} s}$

Génération : d'Obc vers Clight

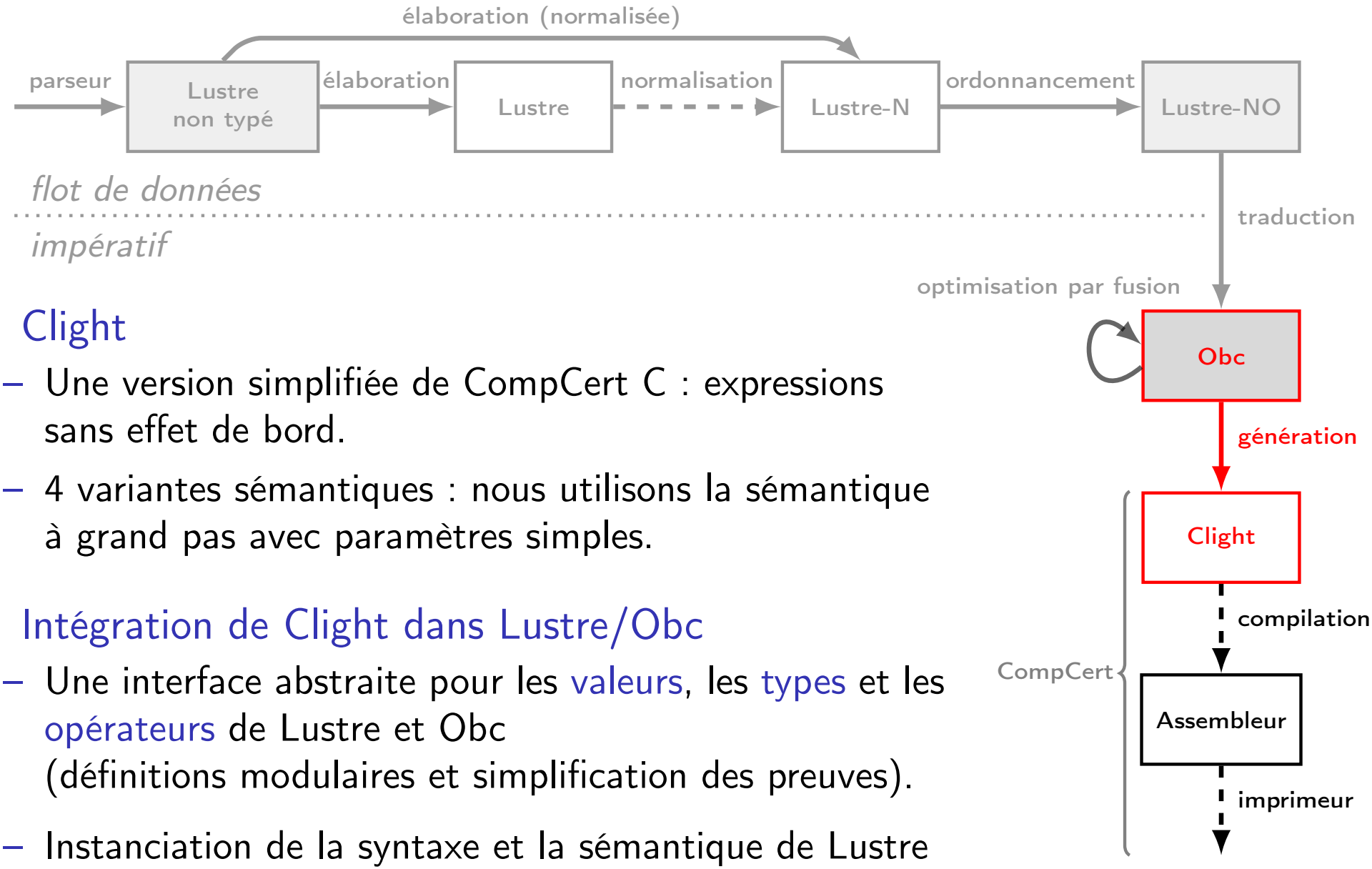


CompCert : un modèle formel et compilateur pour C

- Un modèle générique d'exécution et de mémoire au niveau matériel.
- Algorithmes optimisants et formellement vérifiés pour générer du code assembleur.

[Blazy, Dargaye et Leroy (2006): "Formal Verification of a C Compiler Front-End"] [Leroy (2009): "Formal verification of a realistic compiler"]

Génération : d'Obc vers Clight



- Clight
  - Une version simplifiée de CompCert C : expressions sans effet de bord.
  - 4 variantes sémantiques : nous utilisons la sémantique à grand pas avec paramètres simples.
- Intégration de Clight dans Lustre/Obc
  - Une interface abstraite pour les valeurs, les types et les opérateurs de Lustre et Obc (définitions modulaires et simplification des preuves).
  - Instanciation de la syntaxe et la sémantique de Lustre et Obc avec des définitions de CompCert.

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  {
    var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then (t := count.step o2 (1, 1, false);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

- Technique standard pour l'encapsulation d'état.
- Chaque détail engendre des complications dans la preuve de correction.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

```
class count { ... }
```

```
class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  {
    var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then (t := count.step o2 (1, 1, false);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }
```

```
struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

- Technique standard pour l'encapsulation d'état.
- Chaque détail engendre des complications dans la preuve de correction.

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  {
    var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then (t := count.step o2 (1, 1, false);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

- Technique standard pour l'encapsulation d'état.
- Chaque détail engendre des complications dans la preuve de correction.

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  {
    var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then (t := count.step o2 (1, 1, false);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

- Technique standard pour l'encapsulation d'état.
- Chaque détail engendre des complications dans la preuve de correction.

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  {
    var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then (t := count.step o2 (1, 1, false);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

- Technique standard pour l'encapsulation d'état.
- Chaque détail engendre des complications dans la preuve de correction.

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  {
    var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then (t := count.step o2 (1, 1, false);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

- Technique standard pour l'encapsulation d'état.
- Chaque détail engendre des complications dans la preuve de correction.

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  {
    var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then (t := count.step o2 (1, 1, false);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

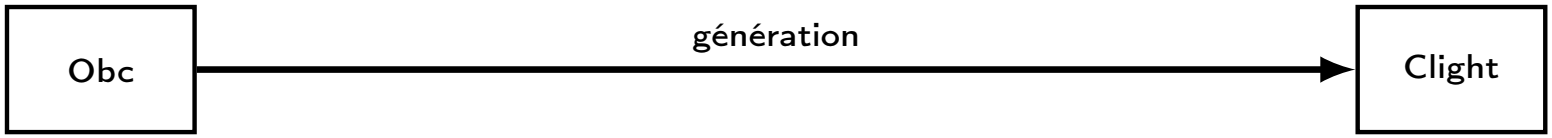
- Technique standard pour l'encapsulation d'état.
- Chaque détail engendre des complications dans la preuve de correction.

```
class count { ... }

class avgvelocity {
  memory w : int;
  class count o1, o2;

  reset() {
    count.reset o1;
    count.reset o2;
    state(w) := 0
  }

  step(delta: int, sec: bool) returns (r, v: int)
  {
    var t : int;

    r := count.step o1 (0, delta, false);
    if sec
      then (t := count.step o2 (1, 1, false);
            v := r / t)
      else v := state(w);
    state(w) := v
  }
}
```

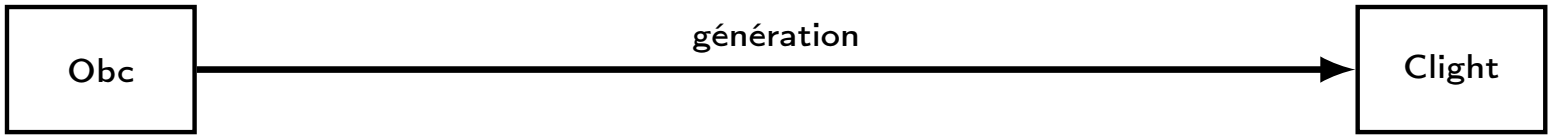
- Technique standard pour l'encapsulation d'état.
- Chaque détail engendre des complications dans la preuve de correction.

```
struct count { _Bool f; int c; };
void count$reset(struct count *self) { ... }
int count$step(struct count *self, int ini, int inc, _Bool res) { ... }

struct avgvelocity {
  int w;
  struct count o1;
  struct count o2;
};

struct avgvelocity$step {
  int r;
  int v;
};

void avgvelocity$reset(struct avgvelocity *self)
{
  count$reset(&(self→o1));
  count$reset(&(self→o2));
  self→w = 0;
}

void avgvelocity$step(struct avgvelocity *self,
                      struct avgvelocity$step *out, int delta, _Bool sec)
{
  register int t, step$n;

  step$n = count$step(&(self→o1), 0, delta, 0);
  out→r = step$n;
  if (sec) {
    step$n = count$step(&(self→o2), 1, 1, 0);
    t = step$n;
    out→v = out→r / t;
  } else {
    out→v = self→w;
  }
  self→w = out→v;
}
```

Correction de la génération de Clight



Correction de la génération de Clight



$me, ve \vdash s \Downarrow (me', ve')$

$e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow\!\Downarrow (e', le', m')$

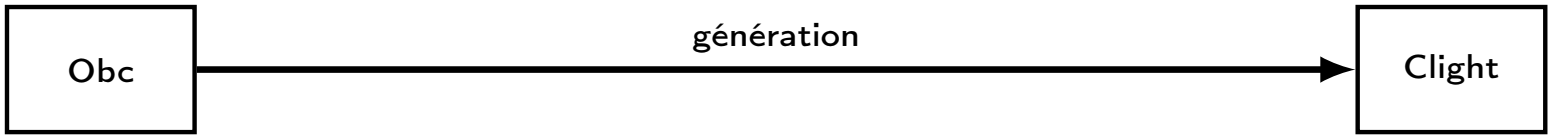
Correction de la génération de Clight

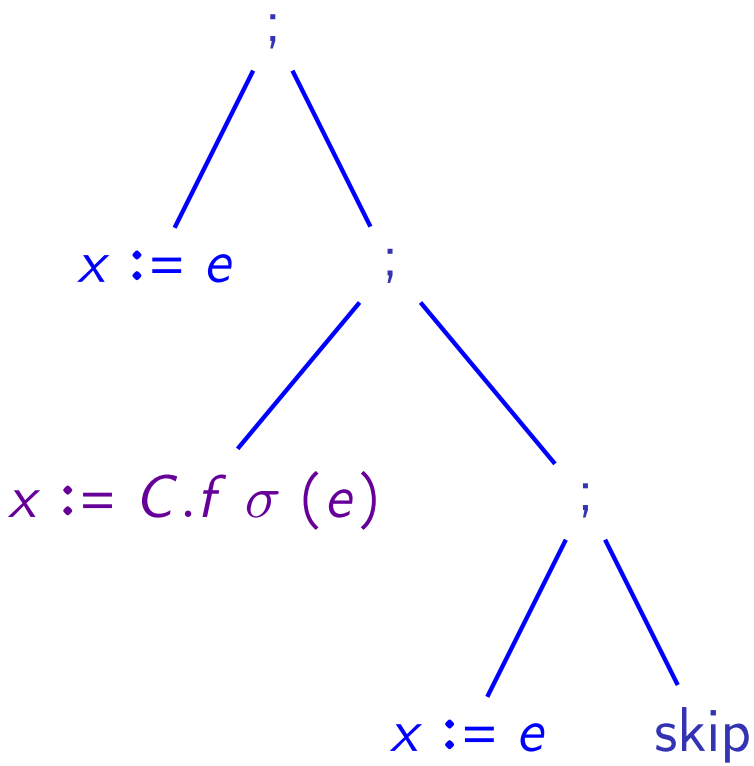
preuve par récurrence sur la sémantique à grand pas

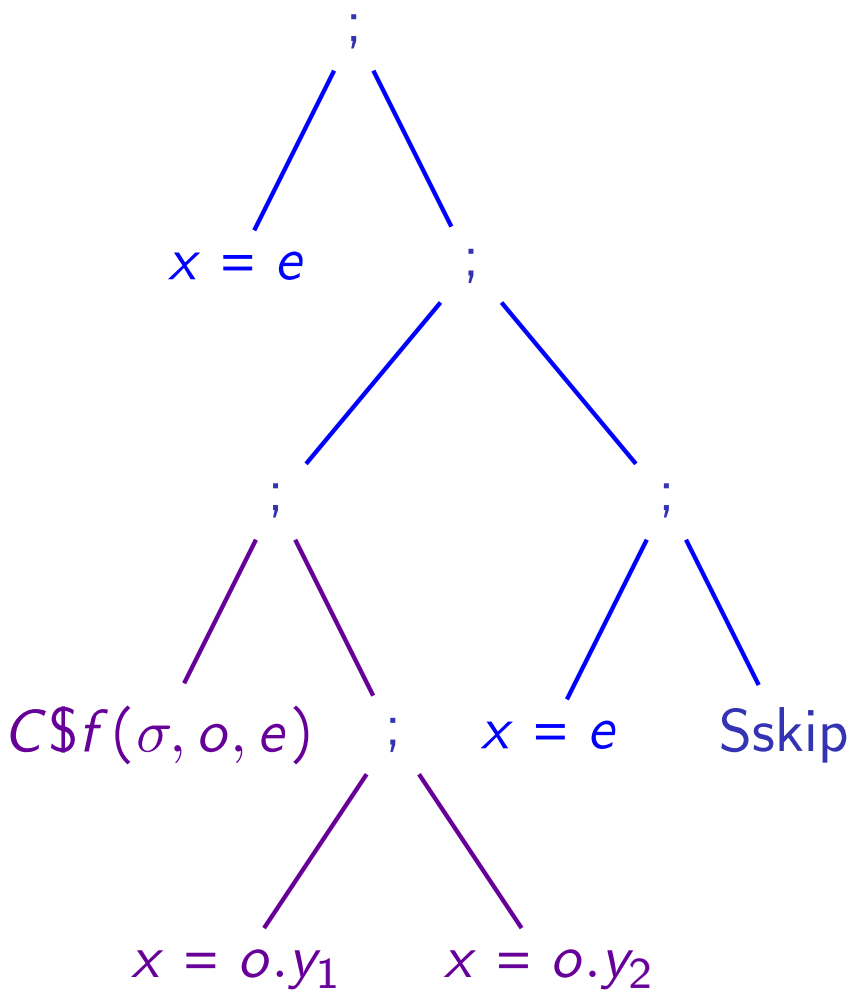
$me, ve \vdash s \Downarrow (me', ve')$

$e, le, m \vdash_{\mathsf{Clight}} \mathsf{generate}(s) \Downarrow (e', le', m')$

Correction de la génération de Clight



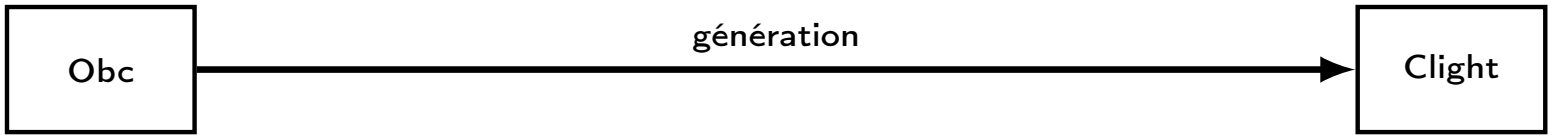


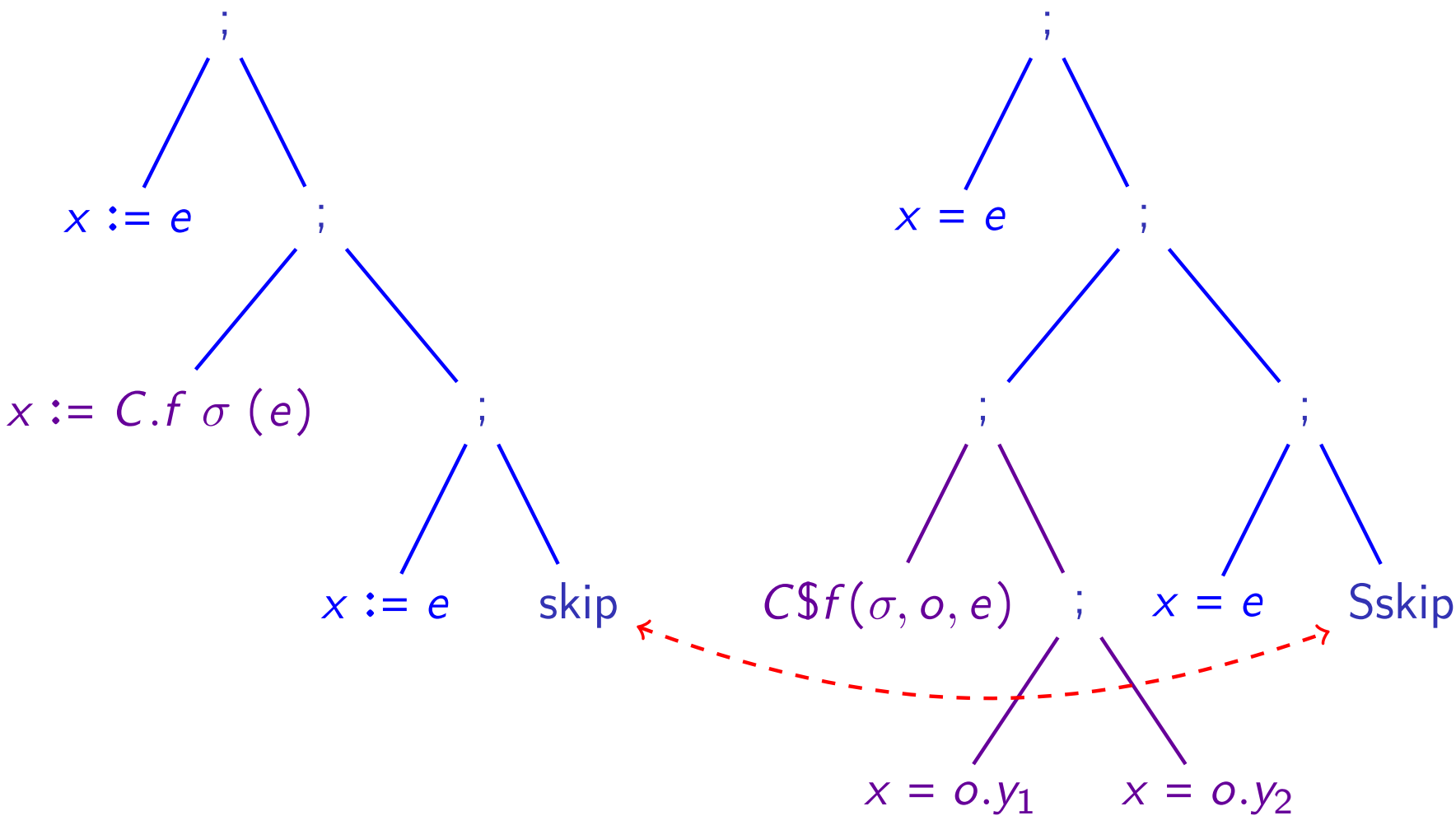


$me, ve \vdash s \Downarrow (me', ve')$

$e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correction de la génération de Clight

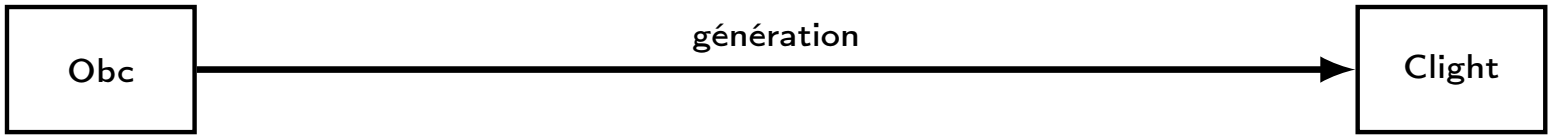


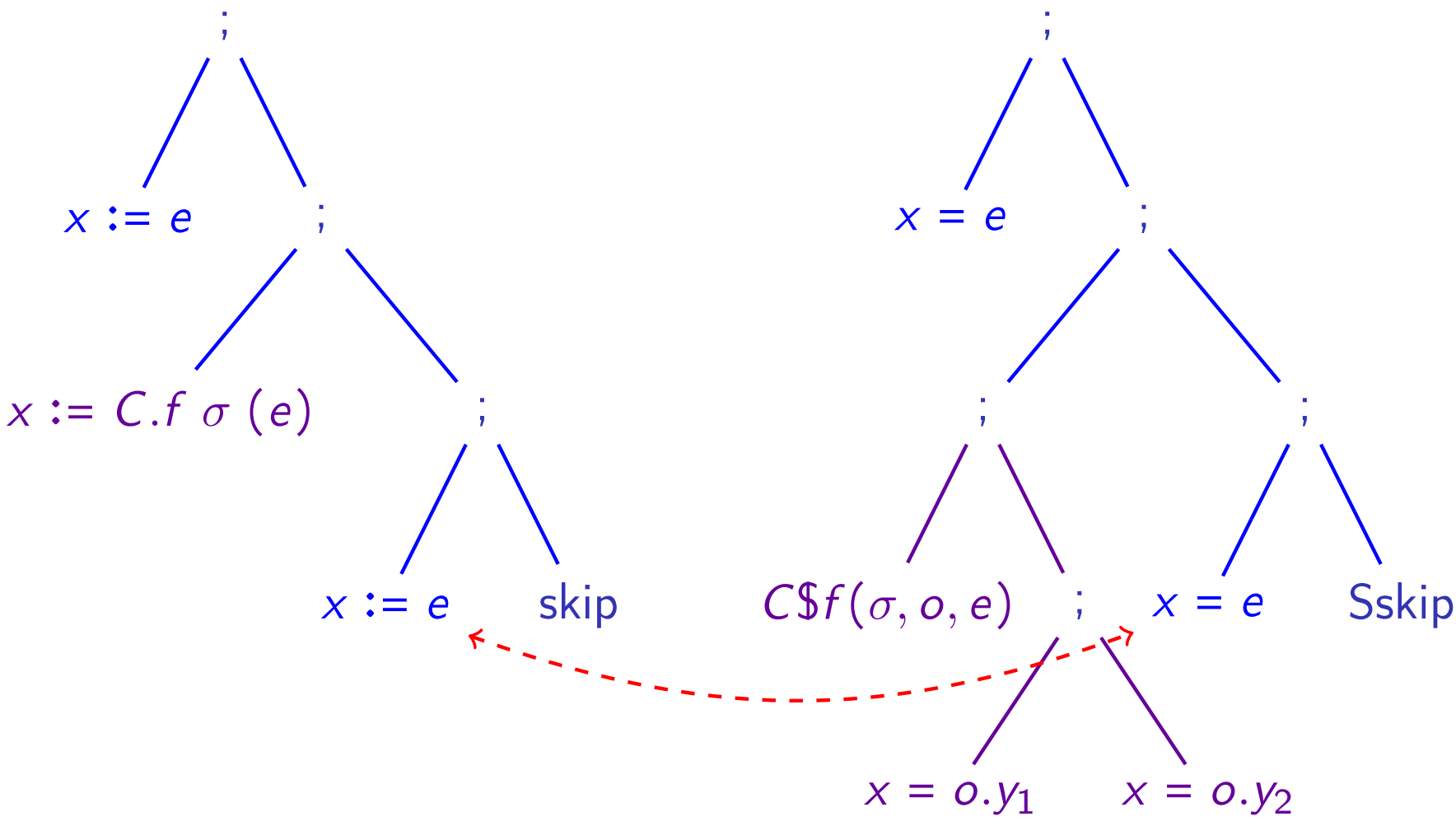


$me, ve \vdash s \Downarrow (me', ve')$

$e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correction de la génération de Clight

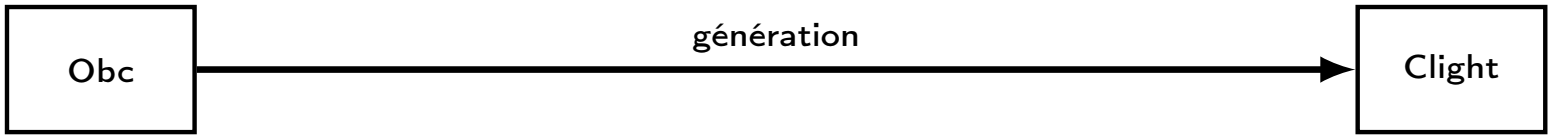


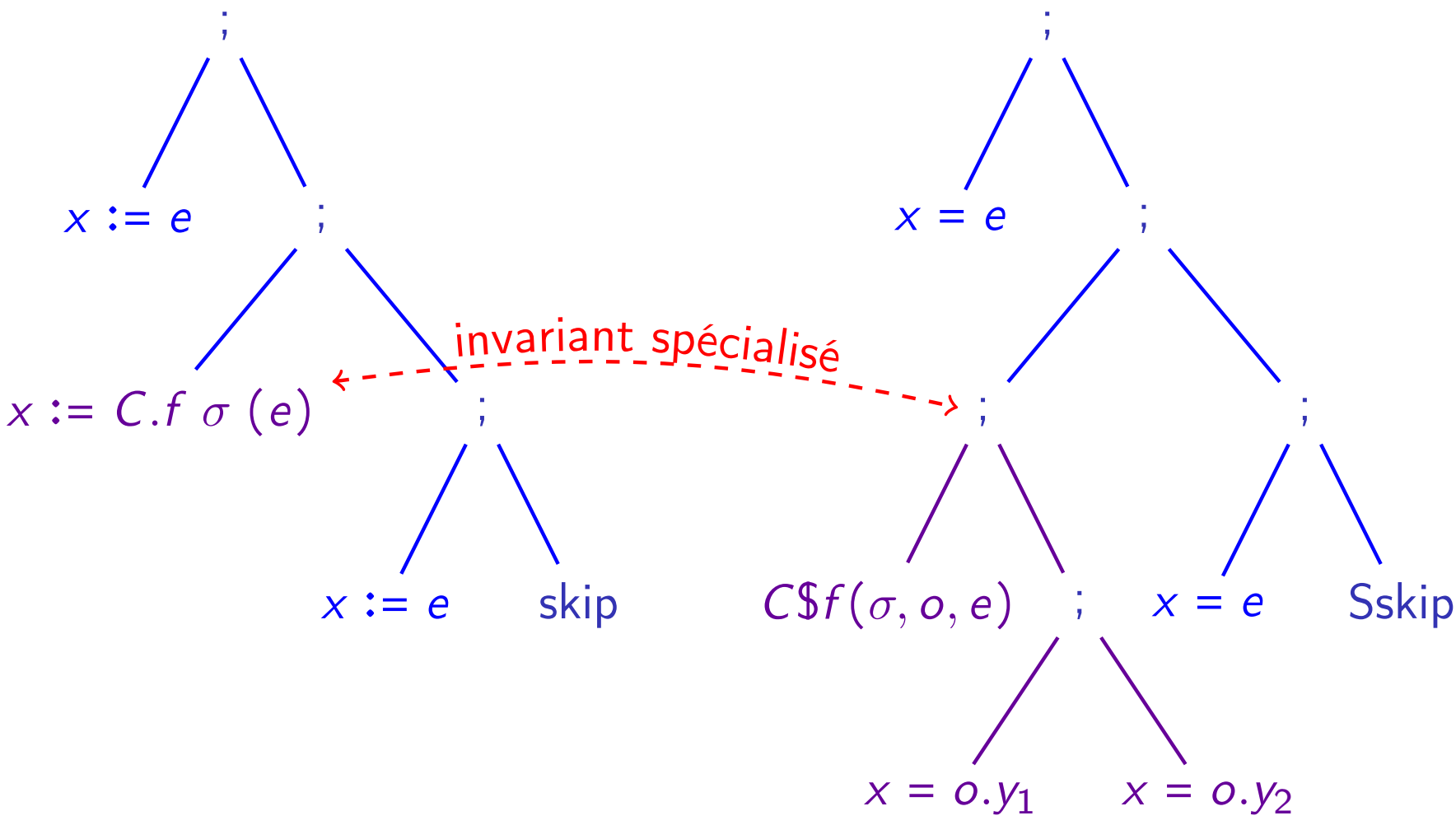


$me, ve \vdash s \Downarrow (me', ve')$

$e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correction de la génération de Clight



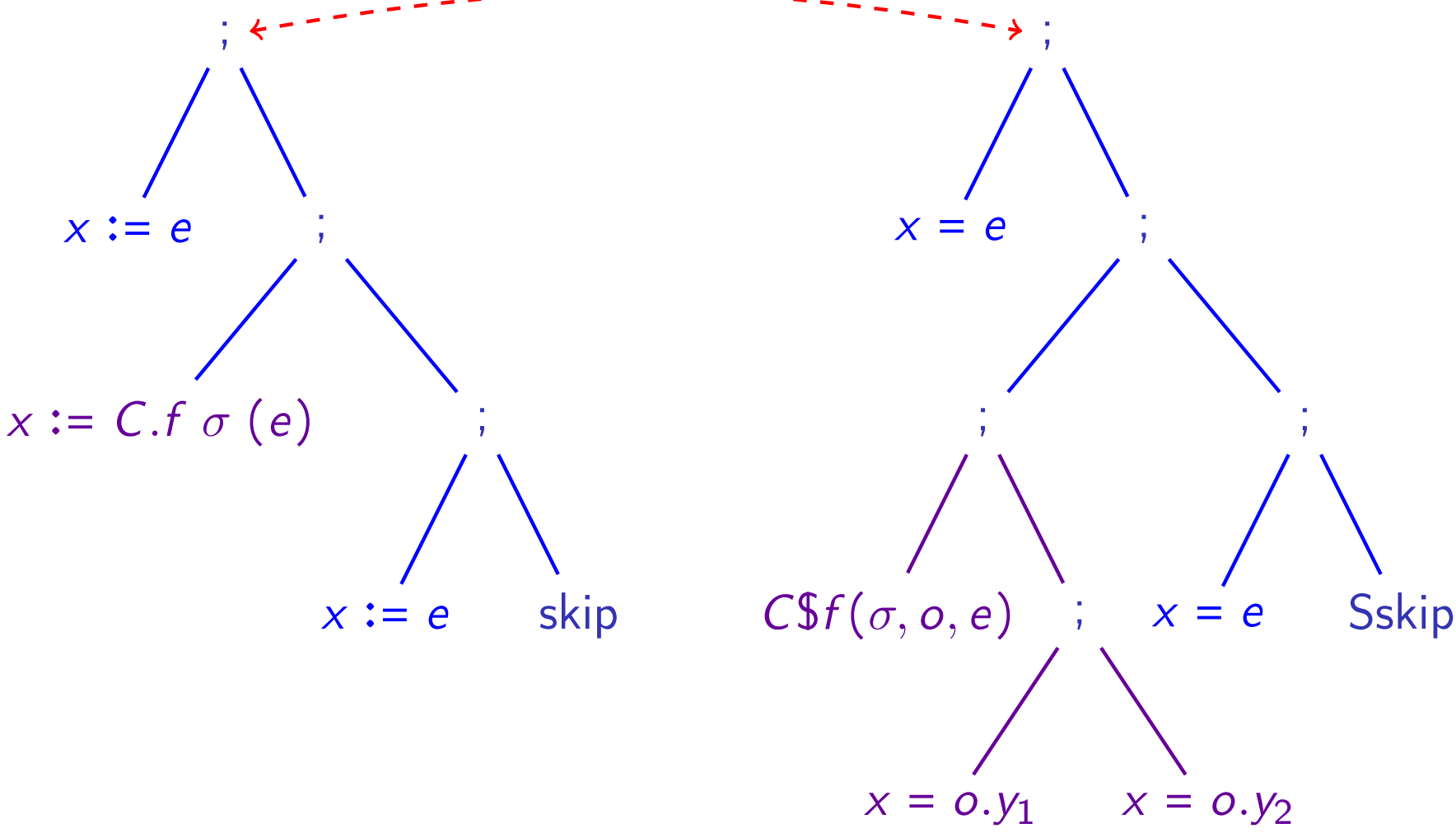


$me, ve \vdash s \Downarrow (me', ve')$

$e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Correction de la génération de Clight

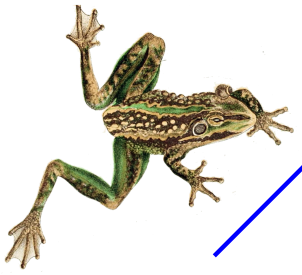
par hérédité

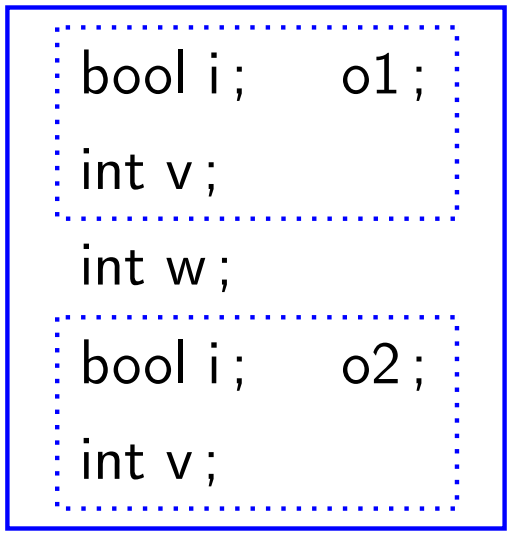


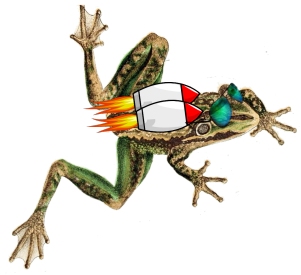
$me, ve \vdash s \Downarrow (me', ve')$

$e, le, m \vdash_{\text{Clight}} \text{generate}(s) \Downarrow (e', le', m')$

Obc à Clight : correspondance entre mémoires





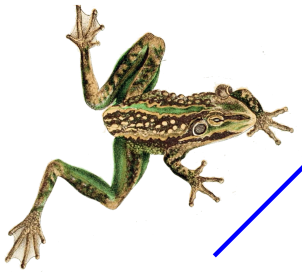


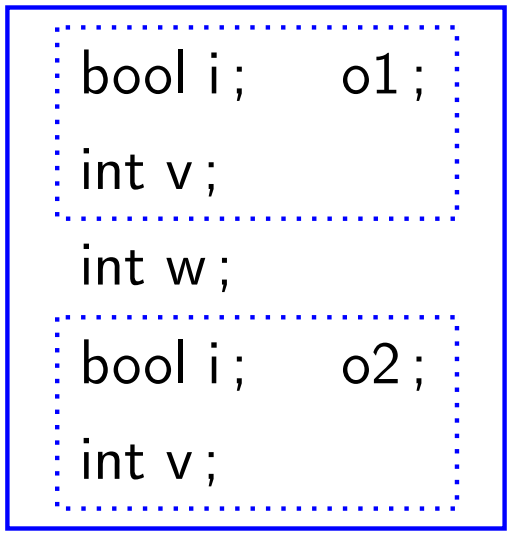
- Cette fois les modèles sémantiques d'exécution sont similaires.
- La difficulté est de rapprocher les représentations de la mémoire.
  - Obc : structure arborescente, la séparation entre variables est manifeste.
  - Clight : blocs d'octets, il faut considérer l'aliasing, l'alignement et la taille.

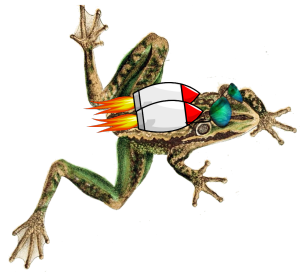
- Étendre une bibliothèque CompCert d'assertions de séparation :
  `https://github.com/AbsInt/CompCert/common/Separation.v`.

[Ishtiaq et O'Hearn (2001): "BI as an Assertion Language for Mutable Data Structures"] [Reynolds (2002): "Separation Logic : A Logic for Shared Mutable Data Structures"]

Obc à Clight : correspondance entre mémoires



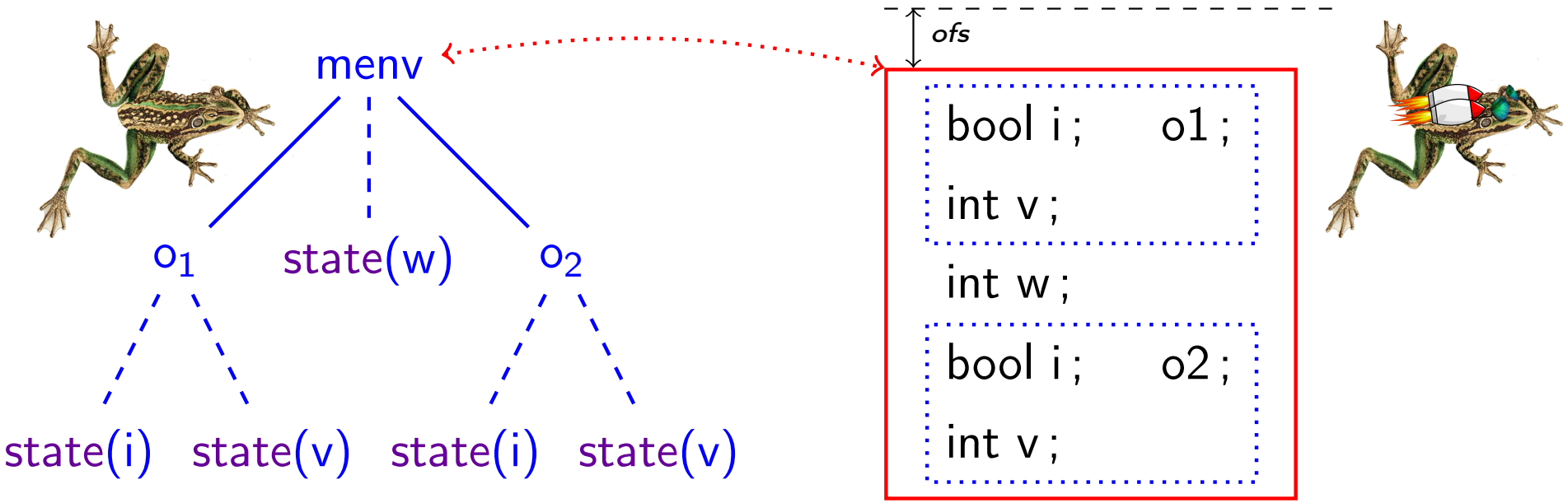




- Cette fois les modèles sémantiques d'exécution sont similaires.
- La difficulté est de rapprocher les représentations de la mémoire.
  - Obc : structure arborescente, la séparation entre variables est manifeste.
  - Clight : blocs d'octets, il faut considérer l'aliasing, l'alignement et la taille.

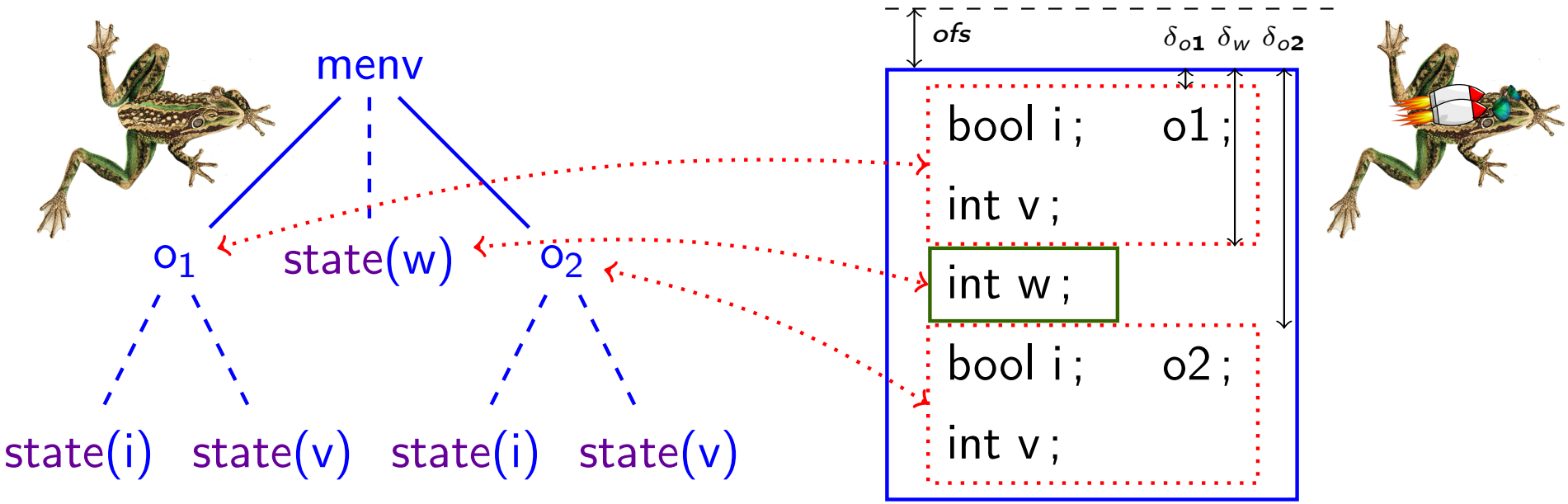
- Étendre une bibliothèque CompCert d'assertions de séparation : `https://github.com/AbsInt/CompCert/common/Separation.v`.
- Encoder la simplicité du modèle source dans le modèle riche du cible.
- Technique générale (et bien commode) pour interfacer avec CompCert.

Obc à Clight : correspondance entre mémoires



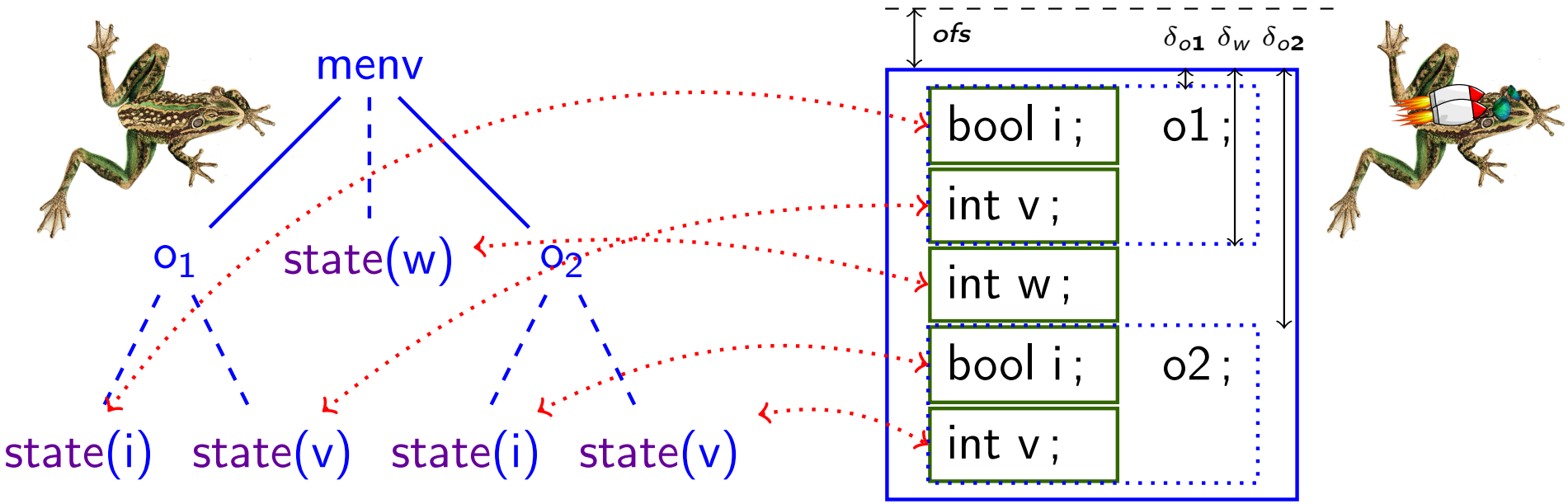
$m \vDash \text{staterep}\ \ \text{avgvelocity}\ \ \mathit{menv}\ \ (b_s, \mathit{ofs})$

Obc à Clight : correspondance entre mémoires

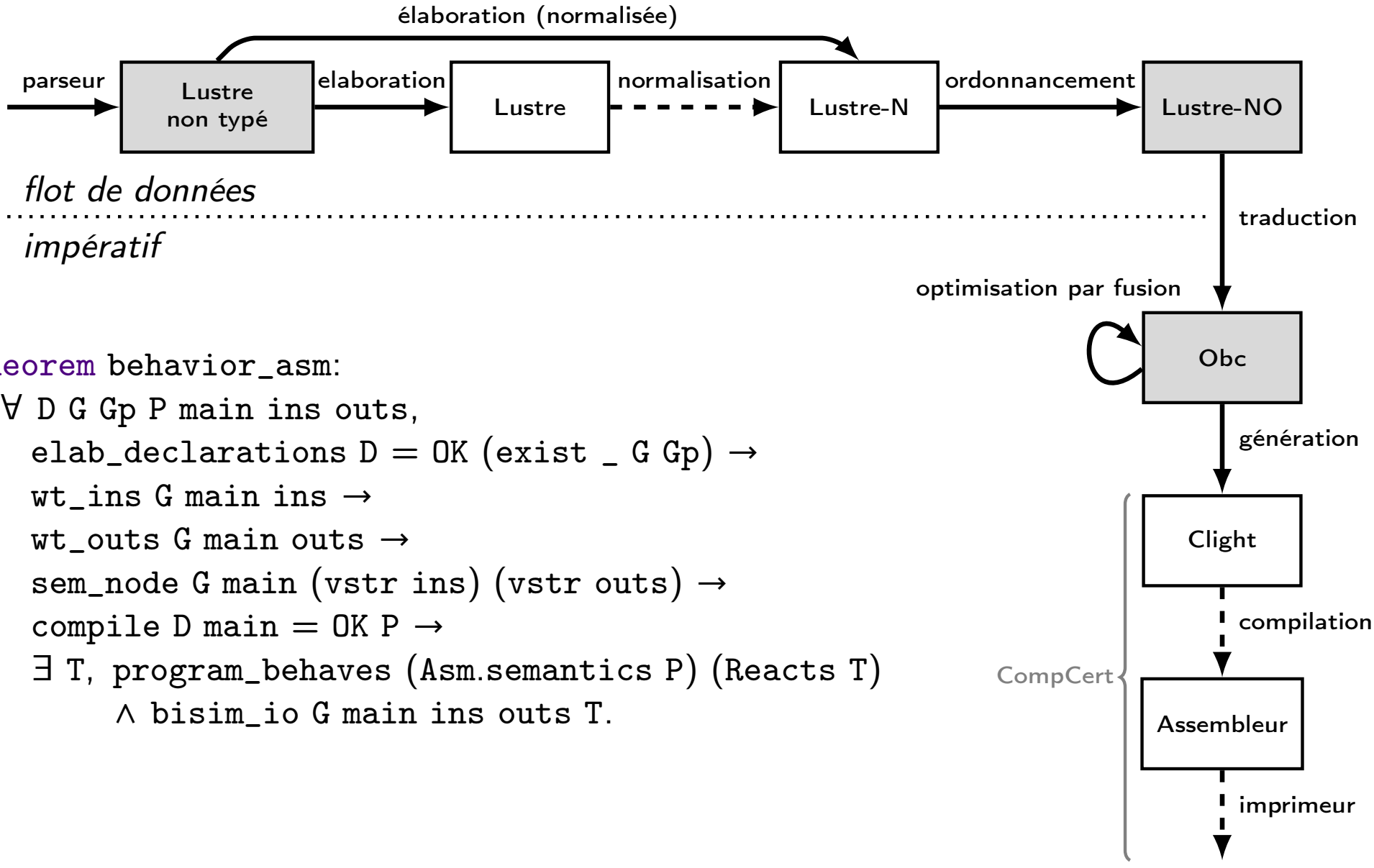


$$
\begin{aligned}
m \vDash \quad & \text{staterep count } menv(o1)\ (b_s, ofs + \delta_{o1}) \\
& * \text{ contains int32s } (b_s, ofs + \delta_w)\ [menv.\text{state}(w)] \\
& * \text{ staterep count } menv(o2)\ (b_s, ofs + \delta_{o2})
\end{aligned}
$$

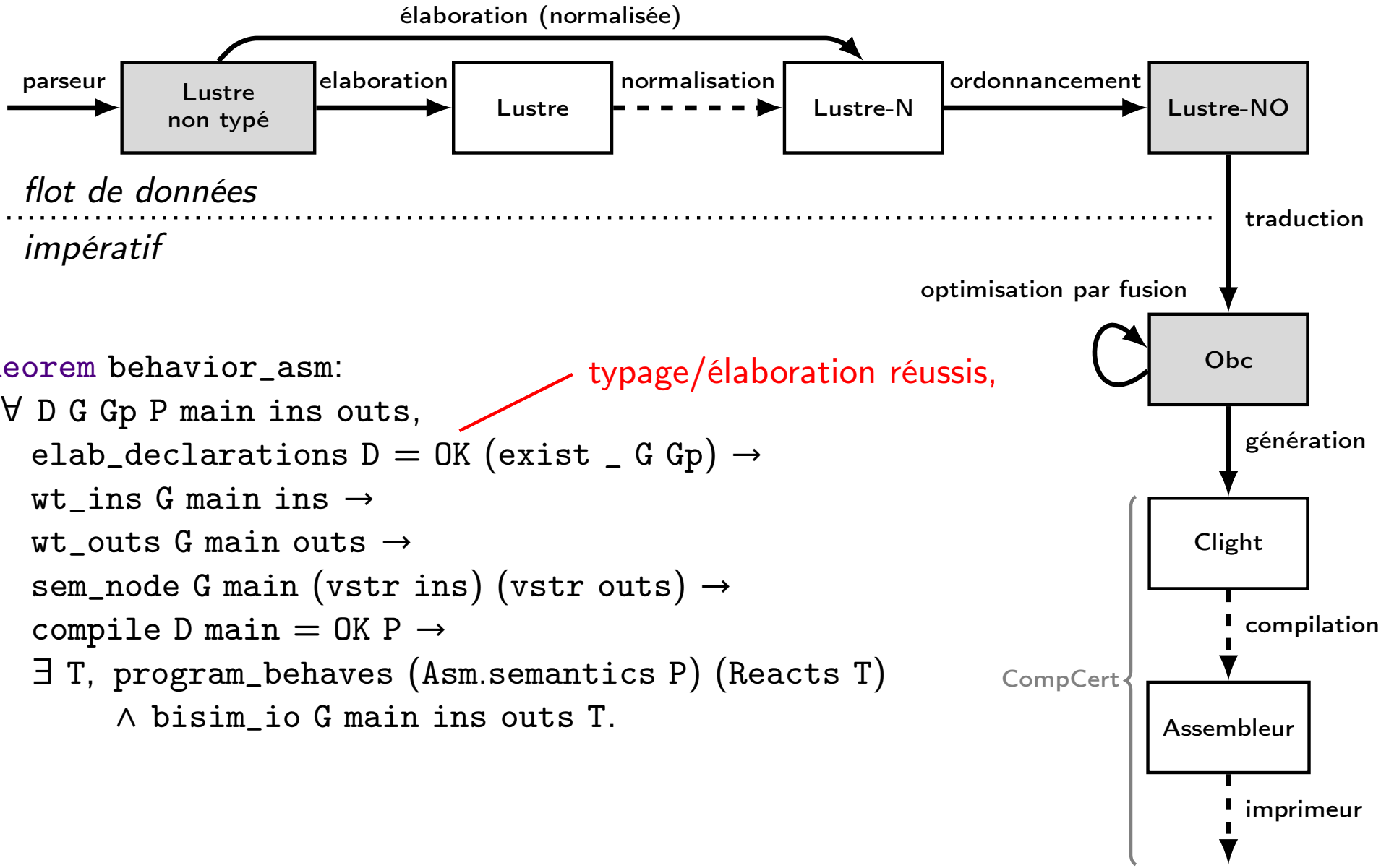
Obc à Clight : correspondance entre mémoires



$$
\begin{array}{rlll}
m \vDash & \text{contains bool} & (b_s, ofs + \delta_{o1} + \delta_i) & \lceil menv.o_1.\text{state}(i) \rceil \\
* & \text{contains int32s} & (b_s, ofs + \delta_{o1} + \delta_v) & \lceil menv.o_1.\text{state}(v) \rceil \\
* & \text{contains int32s} & (b_s, ofs + \delta_w) & \lceil menv.\text{state}(w) \rceil \\
* & \text{contains bool} & (b_s, ofs + \delta_{o2} + \delta_i) & \lceil menv.o_2.\text{state}(i) \rceil \\
* & \text{contains int32s} & (b_s, ofs + \delta_{o2} + \delta_v) & \lceil menv.o_2.\text{state}(v) \rceil
\end{array}
$$

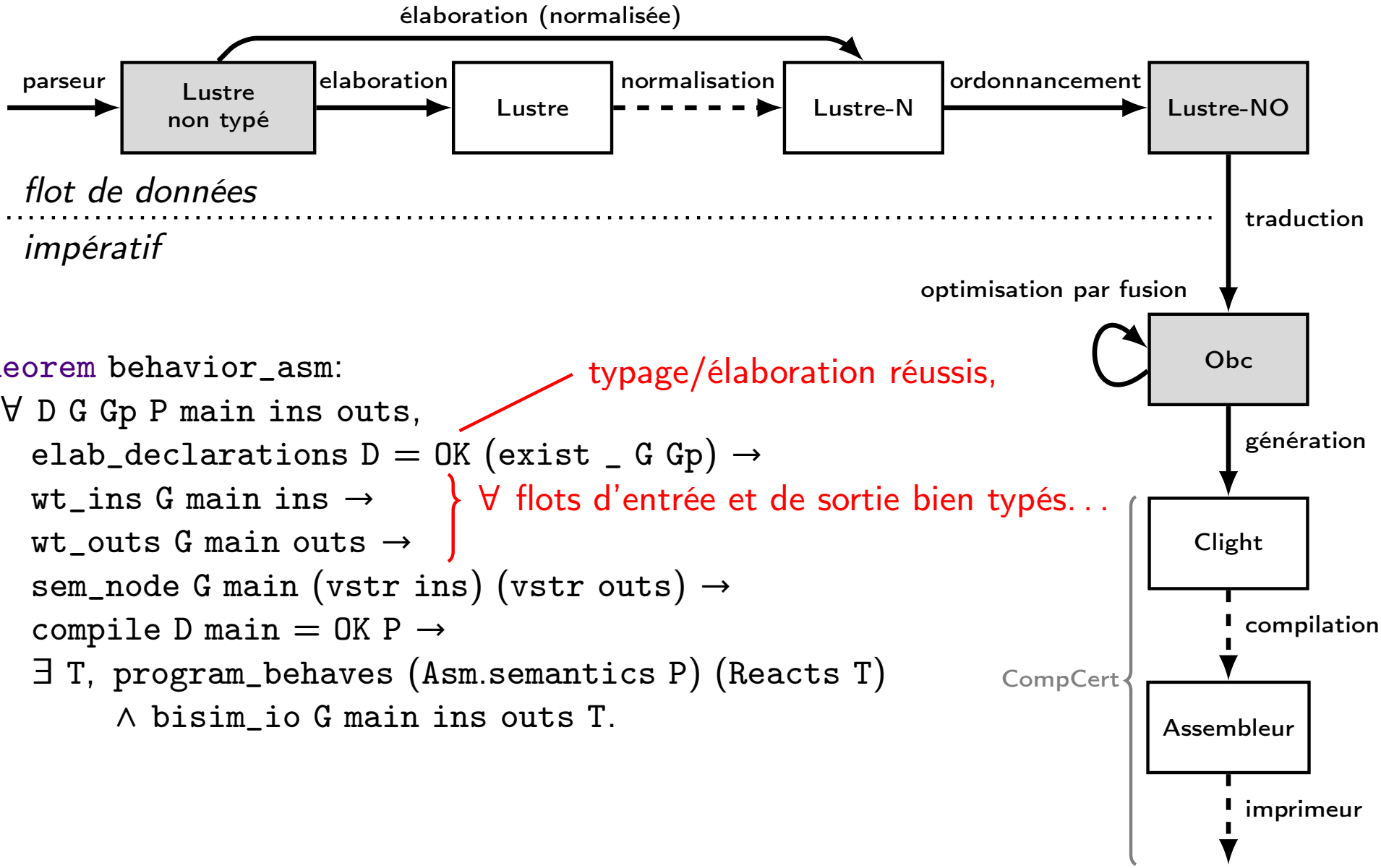



```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main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



typage/élaboration réussis,

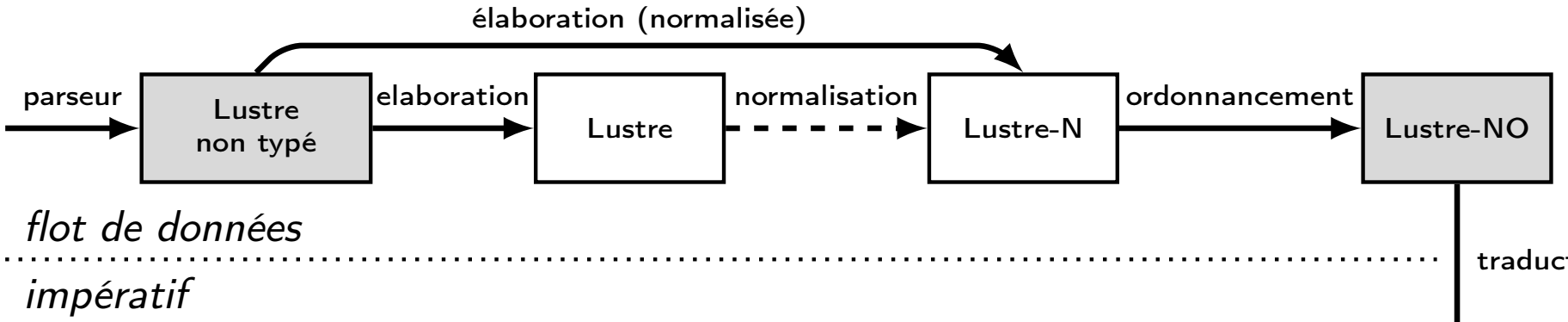
```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main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



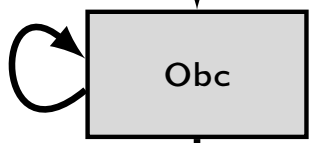

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main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

typage/élaboration réussis,

∀ flots d'entrée et de sortie bien typés. . .



optimisation par fusion

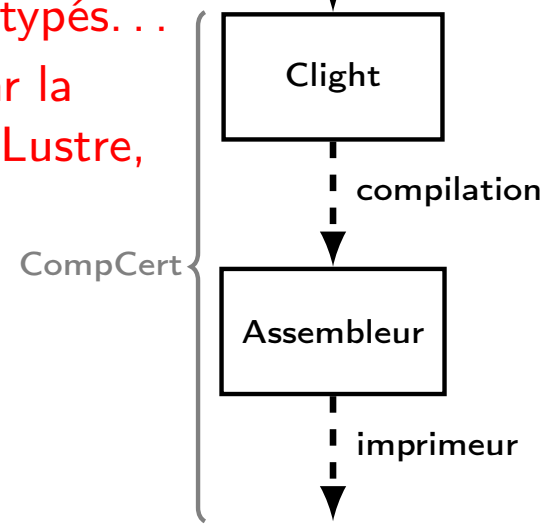


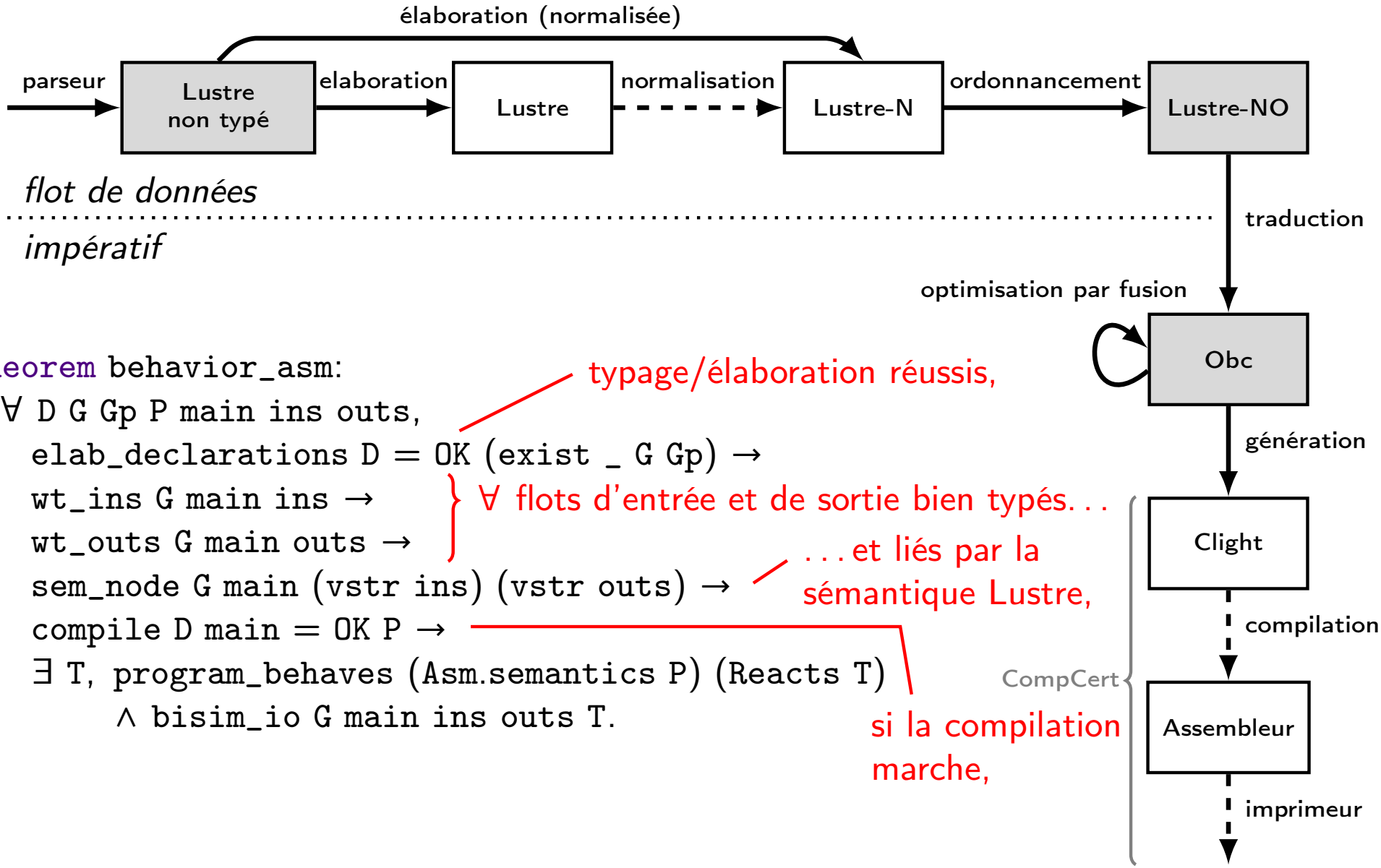

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main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

typage/élaboration réussis,

∀ flots d'entrée et de sortie bien typés...

...et liés par la sémantique Lustre,





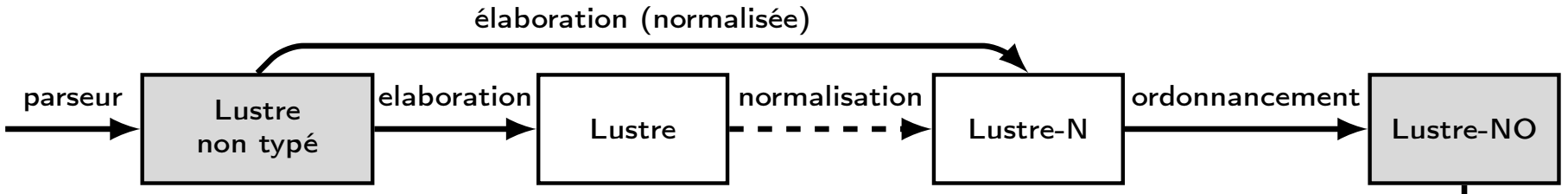

```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    wt_outs G main outs →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main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

typage/élaboration réussis,

∀ flots d'entrée et de sortie bien typés...

...et liés par la sémantique Lustre,

si la compilation marche,

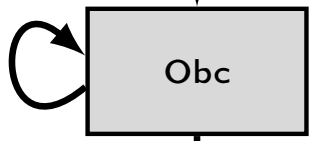


flot de données

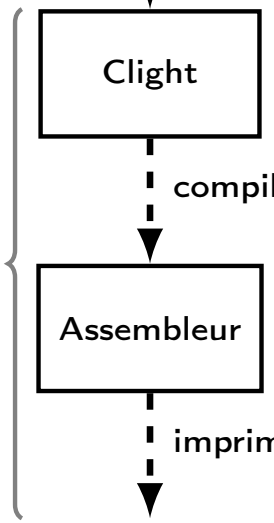
impératif

traduction

optimisation par fusion



génération




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

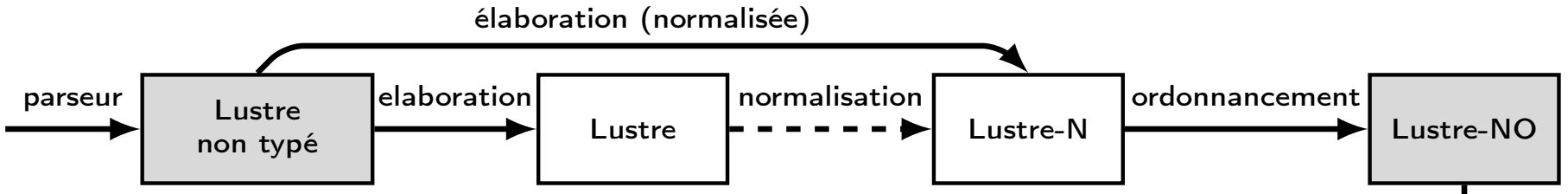
typage/élaboration réussis,

∀ flots d'entrée et de sortie bien typés. . .

. . . et liés par la sémantique Lustre,

si la compilation marche,

alors, l'assembleur généré produit une trace infinie. . .

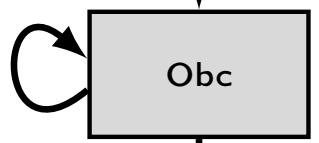


flot de données

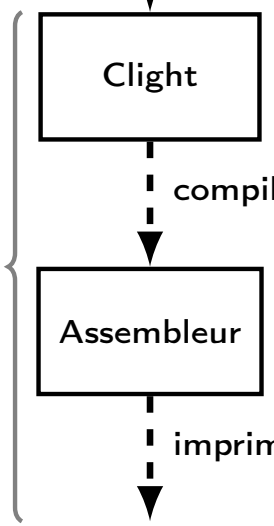
impératif

traduction

optimisation par fusion



génération




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

typage/élaboration réussis,

∀ flots d'entrée et de sortie bien typés...

...et liés par la sémantique Lustre,

si la compilation marche,

alors, l'assembleur généré produit une trace infinie...

...en correspondance avec le modèle flot de données.

Résumé

Lustre :

Un langage dédié d'automaticien pour écrire des spécifications exécutables.

Question fondamentale :

Sont-elles fidèlement retranscrites dans le code généré ?

Notre réponse :

L'implémentation d'un prototype d'un compilateur dans Coq avec la preuve d'un lien formel entre le modèle flot de données et le code impératif généré.

Deux problèmes techniques principaux

- Le changement de modèle entre flot de données et impératif.
- Le changement de représentation de mémoire entre arbre et blocs imbriqués.

Perspectives

Défis scientifiques

- Mécaniser la sémantique d'un langage flot de données dans un assistant de preuve.
  La suite : traiter les structures des plus en plus sophistiquées.
- Développer un composant essentiel de la vision WYPIWYE.
  La suite : la vérification interactive de programmes Lustre.

Questions pratiques

- Les assistants de preuve, peuvent-ils alléger le processus de certification industrielle qui est effectif mais coûteux et difficile à maintenir ?
- Peut-on exploiter les spécifications mécanisées dans le développement des compilateurs et des applications, même si on n'applique pas la vérification bout à bout ?

References I


Auger, C. (2013). “Compilation certifiée de SCADE/LUSTRE”. Thèse de doct. Orsay, France : Univ. Paris Sud 11.


Ballabriga, C., H. Cassé, C. Rochange et P. Sainrat (2010). “OTAWA : An Open Toolbox for Adaptive WCET Analysis”. In : *8th IFIP WG 10.2 Int. Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*. T. 6399. LNCS. Waidhofen an der Ybbs, Austria : Springer, p. 35–46.


Berry, G. (1989). “Real Time Programming : Special Purpose or General Purpose Languages”. In : *Proc. 11th Int. Federation for Information Processing (IFIP) World Computer Congress*. Sous la dir. de G. Ritter. San Francisco, USA, p. 11–17.


Biernacki, D., J.-L. Colaço, G. Hamon et M. Pouzet (2008). “Clock-directed modular code generation for synchronous data-flow languages”. In : *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. Tucson, AZ, USA : ACM Press, p. 121–130.


Blazy, S., Z. Dargaye et X. Leroy (2006). “Formal Verification of a C Compiler Front-End”. In : *Proc. 14th Int. Symp. Formal Methods (FM 2006)*. T. 4085. LNCS. Hamilton, Canada : Springer, p. 460–475.

References II


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


Colaço, J.-L., B. Pagano et M. Pouzet (2005). “A Conservative Extension of Synchronous Data-flow with State Machines”. In : *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*. Sous la dir. de W. Wolf. Jersey City, USA : ACM Press, p. 173–182.


Colaço, J.-L., B. Pagano et M. Pouzet (2017). “Scade 6 : A Formal Language for Embedded Critical Software Development”. In : *Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE 2017)*. Nice, France : IEEE Computer Society.


Ishtiaq, S. et P. W. O’Hearn (2001). “BI as an Assertion Language for Mutable Data Structures”. In : *book title*. London, UK : ACM Press, p. 14–26.


Jourdan, J.-H., F. Pottier et X. Leroy (2012). “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)*. Sous la dir. de H. Seidl. T. 7211. LNCS. Tallinn, Estonia : Springer, p. 397–416.

References III


Kumar, R., M. O. Myreen, M. Norrish et S. Owens (2014). “CakeML : A Verified Implementation of ML”. In : *Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2014)*. San Diego, CA, USA : ACM Press, p. 179–191.


Leroy, X. (2009). “Formal verification of a realistic compiler”. In : *Comms. ACM* 52.7, p. 107–115.


McCoy, F. (1885). *Natural history of Victoria : Prodromus of the Zoology of Victoria*. Frog images.


Reynolds, J. C. (2002). “Separation Logic : A Logic for Shared Mutable Data Structures”. In : *Proc.17th Annual IEEE Symp. Logic in Computer Science (LICS 2002)*. Copenhagen, Denmark : IEEE Computer Society, p. 55–74.


The Coq Development Team (2017). *The Coq proof assistant reference manual*. v. 8.7. Inria.

Résultats expérimentaux 1/2

Application industrielle

- ≈6 000 nœuds
- ≈162 000 équations
- ≈12 MB fichier source
  (sans commentaires)
- Modifications :
  - Suppression des tables de correspondance.
  - Remplacement d'appels vers du code assembleur.
- Compilation par Vélus : ≈1 min 40 s

Résultats expérimentaux 2/2

	Vélus	*Hept+CC*	*Hept+gcc*	*Hept+gcci*	*Lus6+CC*	*Lus6+gcc*	*Lus6+gcci*
avgvelocity	315	385 (22%)	265 (-15%)	70 (-77%)	1 150 (265%)	625 (98%)	350 (11%)
count	55	55 (0%)	25 (-54%)	25 (-54%)	300 (445%)	160 (190%)	50 (-9%)
tracker	680	790 (16%)	530 (-22%)	500 (-26%)	2 610 (283%)	1 515 (122%)	735 (8%)
pip_ex	4 415	4 065 (-7%)	2 565 (-41%)	2 040 (-53%)	10 845 (145%)	6 245 (41%)	2 905 (-34%)
mp_longitudinal [16]	5 525	6 465 (17%)	3 465 (-37%)	2 835 (-48%)	11 675 (111%)	6 785 (22%)	3 135 (-43%)
cruise [54]	1 760	1 875 (6%)	1 230 (-30%)	1 230 (-30%)	5 855 (232%)	3 595 (104%)	1 965 (11%)
risingedgeretrigger [19]	285	300 (5%)	190 (-33%)	190 (-33%)	1 440 (405%)	820 (187%)	335 (17%)
chrono [20]	410	425 (3%)	305 (-25%)	305 (-25%)	2 490 (507%)	1 500 (265%)	670 (63%)
watchdog3 [26]	610	575 (-5%)	355 (-41%)	310 (-49%)	2 015 (230%)	1 135 (86%)	530 (-13%)
functionalchain [17]	11 550	13 535 (17%)	8 545 (-26%)	7 525 (-34%)	23 085 (99%)	14 280 (23%)	8 240 (-28%)
landing_gear [11]	9 660	8 475 (-12%)	5 880 (-39%)	5 810 (-39%)	25 470 (163%)	15 055 (55%)	8 025 (-16%)
minus [57]	890	900 (1%)	580 (-34%)	580 (-34%)	2 825 (217%)	1 620 (82%)	800 (-10%)
prodcell [32]	1 020	990 (-2%)	620 (-39%)	410 (-59%)	3 615 (254%)	2 050 (100%)	1 070 (4%)
ums_verif [57]	2 590	2 285 (-11%)	1 380 (-46%)	920 (-64%)	11 725 (352%)	6 730 (159%)	3 420 (32%)

Figure 12. WCET estimates in cycles [4] for step functions compiled for an armv7-a/vfpv3-d16 target with CompCert 2.6 (CC) and GCC 4.4.8 `-O1` without inlining (gcc) and with inlining (gcci). Percentages indicate the difference relative to the first column.

It performs loads and stores of volatile variables to model, respectively, input consumption and output production. The coinductive predicate presented in Section 1 is introduced to relate the trace of these events to input and output streams.

Finally, we exploit an existing CompCert lemma to transfer our results from the big-step model to the small-step one, from whence they can be extended to the generated assembly code to give the property stated at the beginning of the paper. The transfer lemma requires showing that a program does not diverge. This is possible because the body of the main loop always produces observable events.

5. Experimental Results

Our prototype compiler, Vélus, generates code for the platforms supported by CompCert (PowerPC, ARM, and x86). The code can be executed in a 'test mode' that `scanfs` inputs and `printfs` outputs using an alternative (unverified) entry point. The *verified* integration of generated code into a complete system where it would be triggered by interrupts and interact with hardware is the subject of ongoing work.

As there is no standard benchmark suite for Lustre, we adapted examples from the literature and the Lustre v4 distribution [57]. The resulting test suite comprises 14 programs, totaling about 160 nodes and 960 equations. We compared the code generated by Vélus with that produced by the Heptagon 1.03 [23] and Lustre v6 [35, 57] academic compilers. For the example with the deepest nesting of clocks (3 levels), both Heptagon and our prototype found the same optimal schedule. Otherwise, we follow the approach of [23, §6.2] and estimate the Worst-Case Execution Time (WCET) of the generated code using the open-source OTAWA v5 framework [4] with the 'trivial' script and default parameters.[10] For the targeted domain, an over-approximation to the WCET is usually more valuable than raw performance numbers. We compiled with CompCert 2.6 and GCC 4.8.4 (`-O1`) for the `arm-none-eabi` target (armv7-a) with a hardware floating-point unit (vfpv3-d16).

The results of our experiments are presented in Figure 12. The first column shows the worst-case estimates in cycles for the step functions produced by Vélus. These estimates compare favorably with those for generation with either Heptagon or Lustre v6 and then compilation with CompCert. Both Heptagon and Lustre (automatically) re-normalize the code to have one operator per equation, which can be costly for nested conditional statements, whereas our prototype simply maintains the (manually) normalized form. This re-normalization is unsurprising: both compilers must treat a richer input language, including arrays and automata, and both expect the generated code to be post-optimized by a C compiler. Compiling the generated code with GCC but still without any inlining greatly reduces the estimated WCETs, and the Heptagon code then outperforms the Vélus code. GCC applies 'if-conversions' to exploit predicated ARM instructions which avoids branching and thereby improves WCET estimates. The estimated WCETs for the Lustre v6 generated code only become competitive when inlining is enabled because Lustre v6 implements operators, like **pre** and –>, using separate functions. CompCert can perform inlining, but the default heuristic has not yet been adapted for this particular case. We note also that we use the modular compilation scheme of Lustre v6, while the code generator also provides more aggressive schemes like clock enumeration and automaton minimization [29, 56].

Finally, we tested our prototype on a large industrial application (≈6 000 nodes, ≈162 000 equations, ≈12 MB source file without comments). The source code was already normalized since it was generated with a graphical interface,

[10] This configuration is quite pessimistic but suffices for the present analysis.

12

- Comparaison de WCET du code généré avec deux compilateurs académiques sur de petits exemples.

  [Ballabriga, Cassé, Rochange et Sainrat (2010): "OTAWA : An Open Toolbox for Adaptive WCET Analysis"]

- Les résultats dépendent sur le compilateur C :
  - CompCert : Vélus code égal/mieux
  - gcc -O1 no-inlining : Vélus code plus lent
  - gcc -O1 : Vélus code beaucoup plus lent

- [TODO] : Régler l'heuristique d'extension inline de CompCert.