Génération de code certifiée pour Lustre

Lélio Brun

Timothy Bourke & Marc Pouzet
PARKAS (INRIA - ENS)

28 novembre 2016

Contexte

Langages synchrones :

- applications critiques
- SCADE Suite (ANSYS / Esterel Tech.), Lustre (Caspi & Hallbachs, 87)
- normes de spécifications (DO-178B)

Contexte

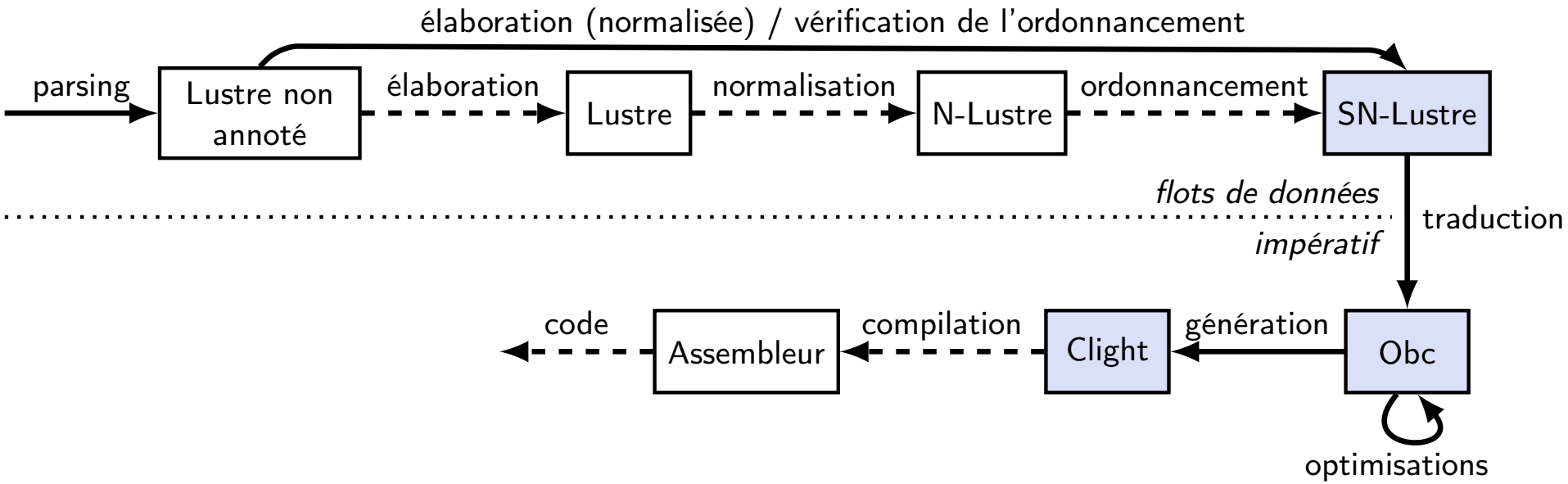
Langages synchrones :

- applications critiques
- SCADE Suite (ANSYS / Esterel Tech.), Lustre (Caspi & Hallbachs, 87)
- normes de spécifications (DO-178B)

Objectif

développement d'un générateur de code certifié formellement

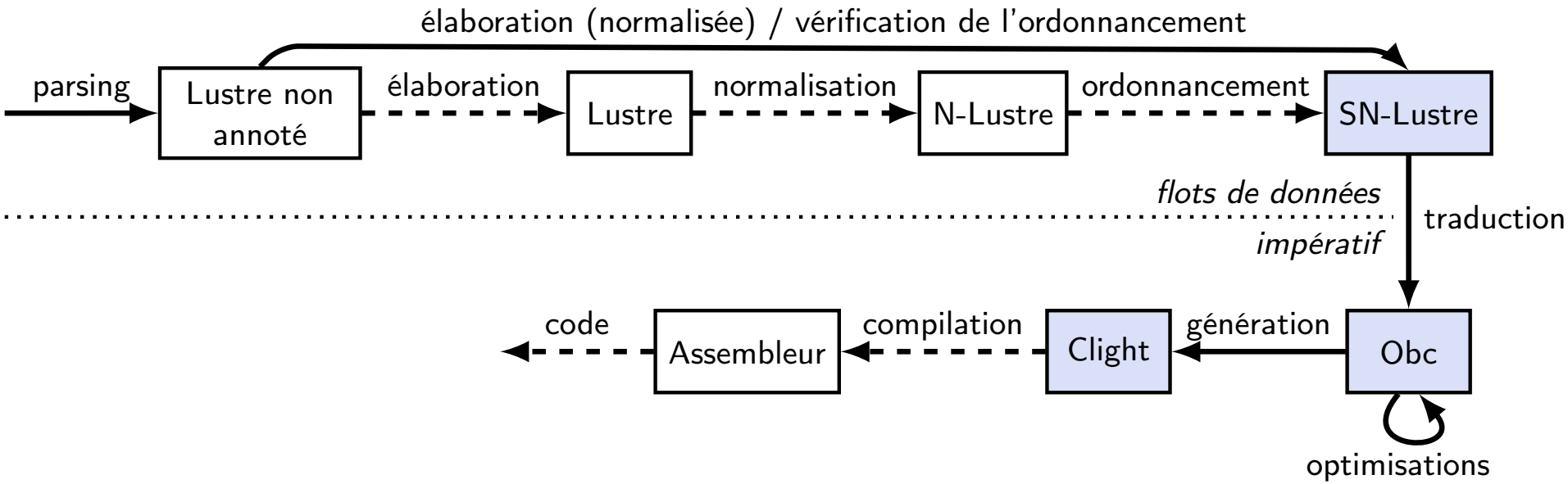
Problème



Compilateur vérifié pour Lustre

- plusieurs étapes de compilation
- langage impératif intermédiaire : Obc
- Obc $\mapsto$ Clight

Problème



Compilateur vérifié pour Lustre

- plusieurs étapes de compilation
- langage impératif intermédiaire : Obc
- Obc $\mapsto$ Clight

Contribution

implémentation et preuve de correction en Coq

Sommaire

Exemple

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



node integrator(a: int) returns (s, x: int)
let
  s = f_euler(a);
  x = f_euler(s);
tel









node excess(max, a: int)
  returns (e: bool, x:int)
  vars s: int;
let
  (s, x) = integrator(a);
  e = s > max;
tel
```

Traduction de l'exemple

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

```
class f_euler {
  mems py: int
  method reset { body state(py) := 0 }
  method step { in d: int; out y: int;
    body
      y := state(py) + d;
      state(py) := y }
}

class integrator {
```

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



node integrator(a: int) returns (s, x: int)
let
  s = f_euler(a);
  x = f_euler(s);
tel









node excess(max, a: int)
  returns (e: bool, x:int)
  vars s: int;
let
  (s, x) = integrator(a);
  e = s > max;
tel
```

```
class f_euler {
  mems py: int
  method reset { body state(py) := 0 }
  method step { in d: int; out y: int;
    body
      y := state(py) + d;
      state(py) := y }
}

class integrator {
  objs s, x: f_euler
  method reset {
    body
      (f_euler s).reset();
      (f_euler x).reset() }
  method step { in a: int; out s, x: int;
    body
      s := (f_euler s).step(a);
      x := (f_euler x).step(v) }
}

class excess {
  objs sx: integrator
  method reset {
```

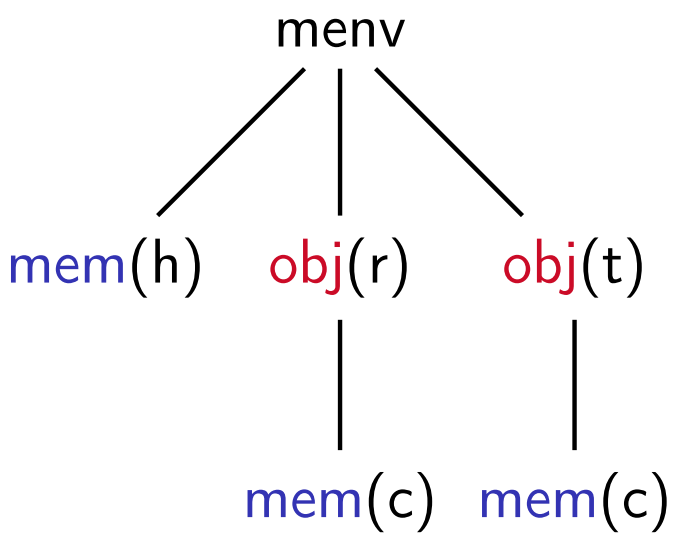
```
node f_euler(d: int) returns y: int
  vars py: int;
let
  y = py + d;
  py = 0 fby y;
tel



node integrator(a: int) returns (s, x: int)
let
  s = f_euler(a);
  x = f_euler(s);
tel









node excess(max, a: int)
  returns (e: bool, x:int)
  vars s: int;
let
  (s, x) = integrator(a);
  e = s > max;
tel
```

```
class f_euler {
  mems py: int
  method reset { body state(py) := 0 }
  method step { in d: int; out y: int;
    body
      y := state(py) + d;
      state(py) := y }
}

class integrator {
  objs s, x: f_euler
  method reset {
    body
      (f_euler s).reset();
      (f_euler x).reset() }
  method step { in a: int; out s, x: int;
    body
      s := (f_euler s).step(a);
      x := (f_euler x).step(v) }
}

class excess {
  objs sx: integrator
  method reset {
    body (integrator sx).reset() }
  method step { in max, a: int;
                out e: bool, x: int;
                vars s: int;
    body
      (s, x) := (integrator sx).step(a);
      e := s > max }
}
```

État et modèle mémoire

$$venv \triangleq ident \longrightarrow val$$

$$menv \triangleq \begin{cases} \text{mems} & : ident \longrightarrow val \\ \text{objs} & : ident \longrightarrow menv \end{cases}$$



Exemples de règles

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

$$\frac{me, ve \vdash_{\text{exp}} e \Downarrow v}{me, ve \vdash_{\text{exp}} op\ e \Downarrow [\![op]\!]_{\text{un}}(v, \text{typeof}\,(e))}$$

Sommaire

Opérateurs

Arité fixe :

$$[\![\cdot]\!]_{\mathsf{un}} : op \longrightarrow val \longrightarrow typ \longrightarrow val$$

$$[\![\cdot]\!]_{\mathsf{bin}} : bop \longrightarrow val \longrightarrow typ \longrightarrow val \longrightarrow typ \longrightarrow val$$

Opérateurs

Arité fixe :

$$[\![\cdot]\!]_{\text{un}} : op \longrightarrow val \longrightarrow typ \longrightarrow val$$
$$[\![\cdot]\!]_{\text{bin}} : bop \longrightarrow val \longrightarrow typ \longrightarrow val \longrightarrow typ \longrightarrow val$$

Objectif :

- paramétrer les modules du développement Coq
- instancier seulement lors de la traduction

Opérateurs

Arité fixe :

$$[\![\cdot]\!]_{\text{un}} : op \to val \to typ \to val$$
$$[\![\cdot]\!]_{\text{bin}} : bop \to val \to typ \to val \to typ \to val$$

Objectif :

- paramétrer les modules du développement Coq
- instancier seulement lors de la traduction

Utiliser des foncteurs Coq

les modules deviennent des foncteurs paramétrés par une interface commune

Sommaire

Clight

Présentation et choix de conception :

- un des langages de la partie avant de CompCert
- modèle mémoire par blocs
- 2 types de variables : locales et temporaires
- 2 variantes de sémantique : paramètres comme locales ou comme temporaires
- 2 sémantiques disponibles : petit pas et grand pas
  - petit pas : continuations
  - grand pas : état (e, le, m)

    e environnement des variables locales

    le environnement des variables temporaires

    m mémoire

Exemple

```
class A {
  ...
  method f {
    in x: int
    out y: bool; z: int
    body
      ... }
  ...
}
```

```
struct A { ... };
struct f_A {
  _Bool y;
  int z;
};
void f_A(struct A *self, struct f_A *out,
     int x) { ... }
```

```
class A {
  ...
  method f {
    in x: int
    out y: bool; z: int
    body
      ... }
  ...
}

class B {
  mems m1: int; m2: bool
  objs o: A
  method f {
    in x1: int, x2: bool
    out y1: int, y2: bool, y3: int
    vars v1: int, v2: bool
    body
      v1 := state(m1) + 4;
      y1 := 2 * v1;
      state(m1) := x1 + v1 + y1;
      v2, y3 := (A o).f(y1);
      y2 := v2 || (state(m2) && x2);
      state(m2) := not state(m2) }
}
```

```
struct A { ... };
struct f_A {
  _Bool y;
  int z;
};
void f_A(struct A *self, struct f_A *out,
     int x) { ... }

struct B {
  int m1;
  _Bool m2;
  struct A o;
};
struct f_B {
  int y1;
  _Bool y2;
  int y3;
};
void f_B(struct B *self, struct f_B *out,
     int x1, _Bool x2) {
  register int v1;
  register _Bool v2;
  struct f_A o;
  v1 = (*self).m1 + 4;
  (*out).y1 = 2 * v1;
  (*self).m1 = x1 + v1 + (*out).y1;
  f_A(&(*self).o, &o, (*out).y1);
    v2 = o.y;
    (*out).y3 = o.y2;
  (*out).y2 = v2 || ((*self).m2 && x2);
  (*self).m2 = not (*self).m2;
  return;
}
```

Conservation de sémantique

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

$$\begin{array}{ccc} me_1, ve_1 & \vdash_{\mathsf{st}} s \Downarrow & me_2, ve_2 \\ \wr & & \wr \\ e_1, le_1, m_1 & \vdash_{\mathsf{Clight}} |s|_{\mathsf{s}} \Downarrow\!\!\Downarrow & e_1, le_2, m_2 \end{array}$$

Logique de séparation

Conséquence du modèle mémoire de CompCert :

- *aliasing*
- recouvrement des champs de structures
- permissions (libération de mémoire)

Logique de séparation

Conséquence du modèle mémoire de CompCert :

- *aliasing*
- recouvrement des champs de structures
- permissions (libération de mémoire)

Solution

utiliser un formalisme de logique de séparation

Correspondance des états

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

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

Correspondance des états

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

match_states $=$

$$\text{pure}\left(le(\text{self}) = \overrightarrow{(b_{self},\ ofs_{self})}\right) \qquad \text{pointeur } \textit{self}$$

$$* \text{ pure}\left(le(\text{out}) = \overrightarrow{(b_{out},\ 0)}\right) \qquad \text{pointeur } \textit{out}$$

$$* \text{ pure}\,(\text{ge}(\text{f\_c}) = co_{out}) \qquad \text{structure de retour}$$

Correspondance des états

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

match_states =

$$\text{pure}\left(le(\text{self}) = \overrightarrow{(b_{self},\ ofs_{self})}\right)$$

$$* \text{ pure}\left(le(\text{out}) = \overrightarrow{(b_{out},\ 0)}\right)$$

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

$$* \text{ staterep } C\ me\ b_{self}\ ofs_{self}$$

mémoire $me \approx$ structure pointée par *self*

Correspondance des états

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

match_states $=$

$$\text{pure}\left(le(\text{self}) = \overrightarrow{(b_{self},\ ofs_{self})}\right)$$

$$* \ \text{pure}\left(le(\text{out}) = \overrightarrow{(b_{out},\ 0)}\right)$$

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

$$* \ \text{staterep}\ C\ me\ b_{self}\ ofs_{self}$$

$$* \ \text{blockrep}\ ve\ co_{out}\ b_{out}$$

variables de sorties de $F \approx$ champs de co_{out} pointée par out

Correspondance des états

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

match_states =

$$\text{pure}\left(le(\text{self}) = \overrightarrow{(b_{self},\ ofs_{self})}\right)$$

$$* \text{ pure}\left(le(\text{out}) = \overrightarrow{(b_{out},\ 0)}\right)$$

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

$$* \text{ staterep } C\ me\ b_{self}\ ofs_{self}$$

$$* \text{ blockrep } ve\ co_{out}\ b_{out}$$

$$* \text{ subrep } F\ e$$

allocation des structures de retour pour les appels

Correspondance des états

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

match_states =

$$\text{pure}\left(le(\text{self}) = \overrightarrow{(b_{self},\ ofs_{self})}\right)$$

$$* \text{ pure}\left(le(\text{out}) = \overrightarrow{(b_{out},\ 0)}\right)$$

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

$$* \text{ staterep } C\ me\ b_{self}\ ofs_{self}$$

$$* \text{ blockrep } ve\ co_{out}\ b_{out}$$

$$* \text{ subrep } F\ e$$

$$* \text{ varsrep } F\ ve\ le$$

paramètres et variables locales $\approx$ variables temporaires

Correspondance des états

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

match_states =

$$\text{pure}\left(le(\text{self}) = \overrightarrow{(b_{self},\ ofs_{self})}\right)$$

$$* \text{pure}\left(le(\text{out}) = \overrightarrow{(b_{out},\ 0)}\right)$$

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

$$* \text{staterep}\ C\ me\ b_{self}\ ofs_{self}$$

$$* \text{blockrep}\ ve\ co_{out}\ b_{out}$$

$$* \text{subrep}\ F\ e$$

$$* \text{varsrep}\ F\ ve\ le$$

$$* \text{subrep}\ F\ e \mathrel{-\!\!*} \text{subrep\_range}\ e$$

magic wand pour la libération de mémoire : $P * (P \mathrel{-\!\!*} Q) \mathrel{-\!\!\triangleright} Q$

Conservation de l'invariant

- souplesse de la conjonction séparante
- « émulation » de la règle *frame* :
  $m \vDash P * F \longrightarrow$ hypothèses $\longrightarrow \exists m', \text{propriétés} \wedge m' \vDash P' * F$
- correction des expressions : 3 lemmes principaux
  1. $(^*\text{self}).x$
  2. $(^*\text{out}).x$
  3. $\$x$
- correction des instructions : 6 lemmes principaux
  1. $(^*\text{self}).x = e$
  2. $(^*\text{out}).x = e$
  3. $\$x = e$
  4. entrée de fonction
  5. libération de mémoire après appel
  6. suite d'affectations après appel

Sommaire

Conclusion, travaux futurs

Bilan :

- taille :
  - traduction : 200 lignes
  - separation : 1800 lignes
  - correction : 1600 lignes
- correspondance entre les modèles mémoires : logique de séparation
- autres langages sources
- compilation certifiée de Lustre

Futur :

- finaliser l'intégration
- optimisations
- thèse : sémantique des parties plus haut niveau

Autres travaux

- langages synchrones, Lustre [Ben+03 ; Cas+87 ; Bie+08 ; Aug13 ; Aug+14 ; Bou+16]
- compilation cerifiée : CompCert [BDL06 ; Ler09a ; Ler09b]
- preuve automatique d'un compilateur [CG15]
- sémantique dénotationnelle [Chl07 ; BKV09 ; BH09]

Références I

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

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

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

[Aug13] Cédric Auger. « Compilation certifiée de SCADE/LUSTRE ». Thèse de doct. Orsay, France : Univ. Paris Sud 11, avr. 2013.

Références II

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

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

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

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

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

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

Références III

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

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

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