CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage Lotos

Hubert Garavel **Philippe Turlier**

VERIMAG[1]
Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE

[1]VERIMAG est une Unité Mixte de Recherche du Centre National de la Recherche Scientifique (CNRS), de l'Institut National Polytechnique de Grenoble (INPG), de l'Université Joseph Fourier (UJF, Grenoble-I) et de VERILOG SA. Le laboratoire VERIMAG est associé à l'Institut Informatique et Mathématiques Appliquées de Grenoble (IMAG) et constitue le projet SPECTRE de l'Institut National de Recherche en Informatique et Automatique (INRIA, unité Rhône-

Introduction

Les types abstraits existent en LOTOS et en SDL.

Comment les traiter ?

par le mépris : restriction à "basic LOTOS"

par la subversion : importation de types externes ou remplacement par des formalismes de plus bas niveau

par l'interprétation : réécriture de termes, souvent couplée avec une approche symbolique (résolution, surréduction)

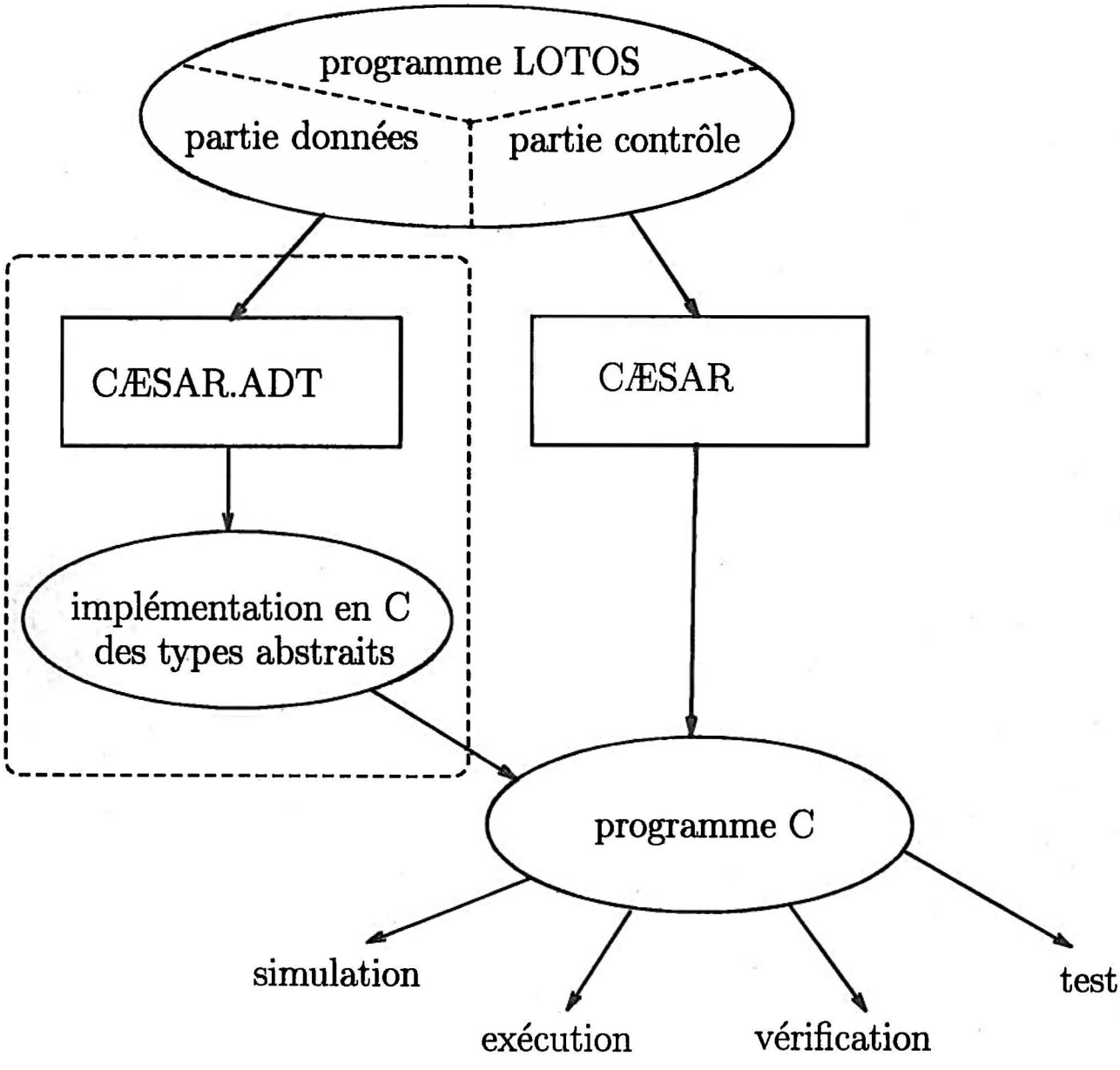
par la compilation : traduction vers un langage impératif (LISP, C, Ada...)

Le compilateur CÆSAR.ADT

Traduction types abstraits LOTOS → C

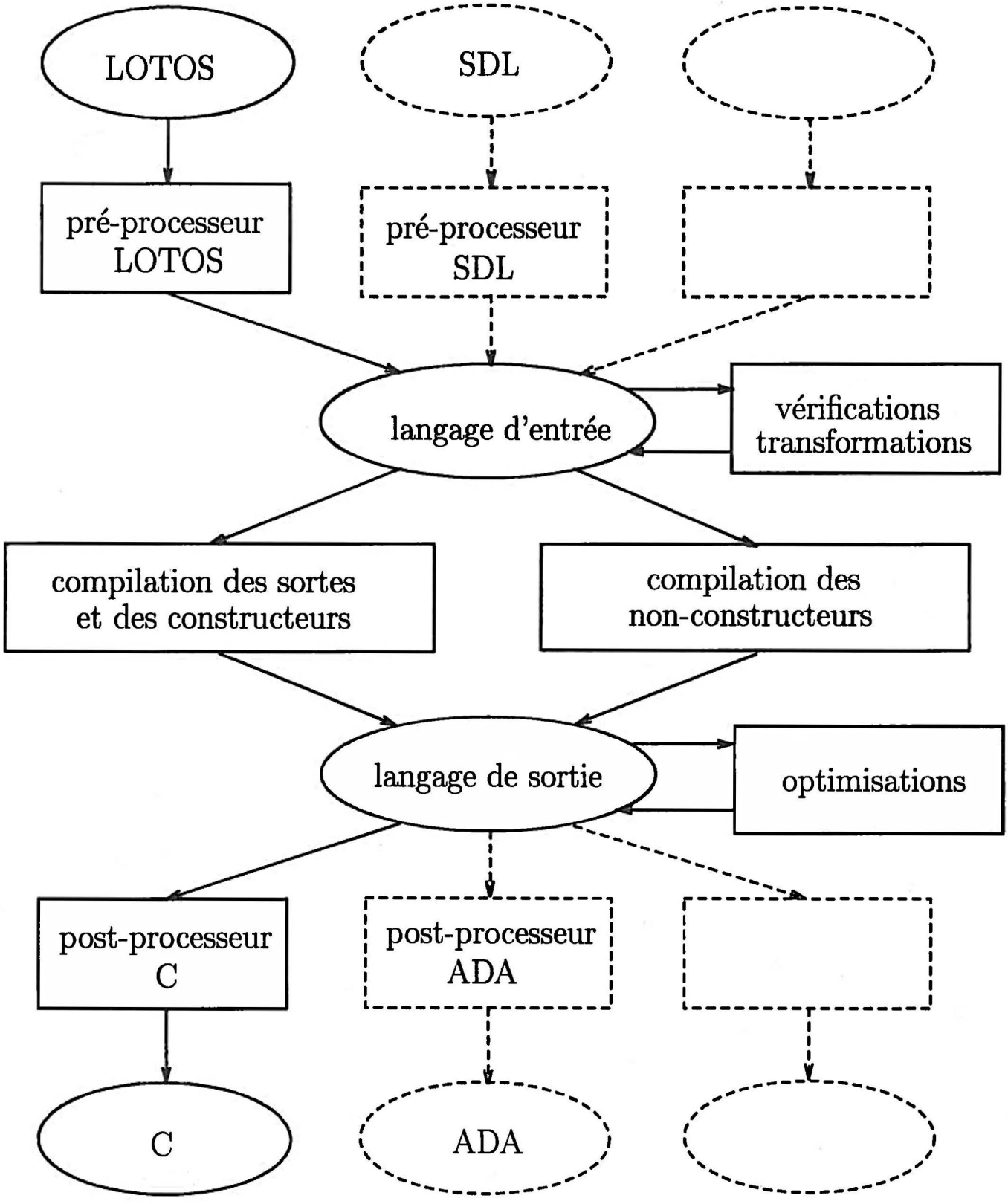
- 1988 : premier prototype
- 1989 : première version diffusable (V1.0, Forte'89)
- 1992 : réécriture complète de l'outil (V4.1)

L'environnement OPEN/CÆSAR



Nécessité d'un compilateur "types abstraits $\longrightarrow$ C"

Architecture de CÆSAR.ADT



Sous-ensemble de Lotos accepté

Spécifications exécutables

- indiquer les constructeurs
- orienter les équations de gauche à droite
- éliminer les équation entre constructeurs

Limitations

- restrictions sur le renommage
- restrictions sur les sortes et opérations externes
- pas d'itérations sur les sortes complexes
- pas de code engendré pour les types génériques et instanciés

Stratégie de réécriture

- appel par valeur (évaluation fonctionnelle)
- priorité décroissante des équations

uniquement en cas de non-confluence ou de non-terminaison

Vérifications sémantiques

- détection des sortes sans constructeur
- détection des non-constructeurs sans équation
- détection des sortes improductives
- détection des équations superflues
- incitation à la modularité

Compilation des sortes

L'implémentation d'une sorte dépend uniquement du profil de ses constructeurs

Cas particulier 1 :

sorte **ADDR**
constructeur n° 1 : **FIRST** :⟶ **ADDR**
constructeur n° 2 : **NEXT** : **ADDR** ⟶ **ADDR**

$\Longrightarrow$ implémentée par un type entier

Cas particulier 2 :

sorte **SIGNAL**
constructeur n° 1 : **SIGHUP** :⟶ **SIGNAL**
constructeur n° 2 : **SIGINT** :⟶ **SIGNAL**
constructeur n° 3 : **SIGQUIT** :⟶ **SIGNAL**
constructeur n° 4 : **SIGILL** :⟶ **SIGNAL**

$\Longrightarrow$ implémentée par un type énuméré (sur 2 bits)

Cas particulier 3 :

sorte **TIMEVAL**
constructeur : **TIME** : **SEC**, **USEC** ⟶ **TIMEVAL**

$\Longrightarrow$ implémentée par un enregistrement

Cas général :

$\Longrightarrow$ structures de données récursives, constituées de pointeurs et d'enregistrements à champs variants

Compilation des non-constructeurs

Principe : compiler et non pas interpréter
formalisme déclaratif → formalisme impératif

Différents algorithmes pour compiler le filtrage :
[Augustsson-85] [Wadler-87] [Kaplan-87]
[Schnoebelen-88] [Pettersson-92] [Puel-Suarez-93]

Algorithme utilisé : [Schnoebelen-88]

- il est orthogonal à l'implémentation des sortes
- il considère le domaine de la fonction à compiler
- il traite les équations conditionnelles
- il traite les constructeurs non libres

eq : nat × nat ⟶ bool
0 eq 0 = true 0 eq succ (N) = false succ (M) eq 0 = false succ (M) eq succ (N) = M eq N

```
BOOL EQ (M, N)
NAT M;
NAT N;
{
if ((M == 0) ?(N > 0) :(N == 0)) return false;
else if (N == 0) return true;
else return EQ (M - 1, N - 1);
}
```

Optimisations

Catalogue de règles de transformations
pour améliorer l'efficacité du code engendré

Détection des constantes

utilisation de variables (au lieu de fonctions)

```
nat n3 = succ (succ (succ (0)));
```

Détection des fonctions "simples"

utilisation de macros (au lieu de fonctions)

```
#define implies(P, Q) (not (P) or (Q))
```

Simplification des tests

utilisation de "court-circuits" : `&&`, `||`, `?:`

```
if P then if Q then X else Y else Y
-----------------------------------
      if (P && Q) then X else Y
```

Factorisation de code

utilisation de "`goto`" pour partager du code

Réorganisation des structures

permutation des champs (contraintes de cadrage)

```
struct { char X; short Y; char Z; }
-----------------------------------
struct { char X; char Z; short Y; }
```

Applications

ATP : compilateur algèbre de processus temporisée

MAA : algorithme cryptographique [ISO 8731]

VTT : routeur de messages

OTS : service transport [ISO 8072]

XTL : compilateur logique temporelle étendue

FWC : calculateur d'alarmes embarqué Airbus A320

Taille des programmes LOTOS source

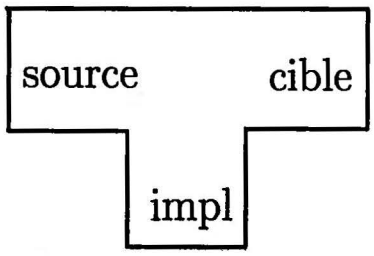
application	L_L	T_L	N_T	N_S	N_C	N_F	N_E
ATP	376	12.638	12	12	30	73	81
MAA	1126	34.178	10	10	16	184	167
VTT	1130	39.444	17	16	39	111	345
OTS	1802	72.170	40	32	87	245	338
XTL	3100	147.345	24	15	135	271	761
FWC	13524	580.119	145	139	515	786	1432

Code C engendré par CÆSAR.ADT 4.1 sur SUN 4/40

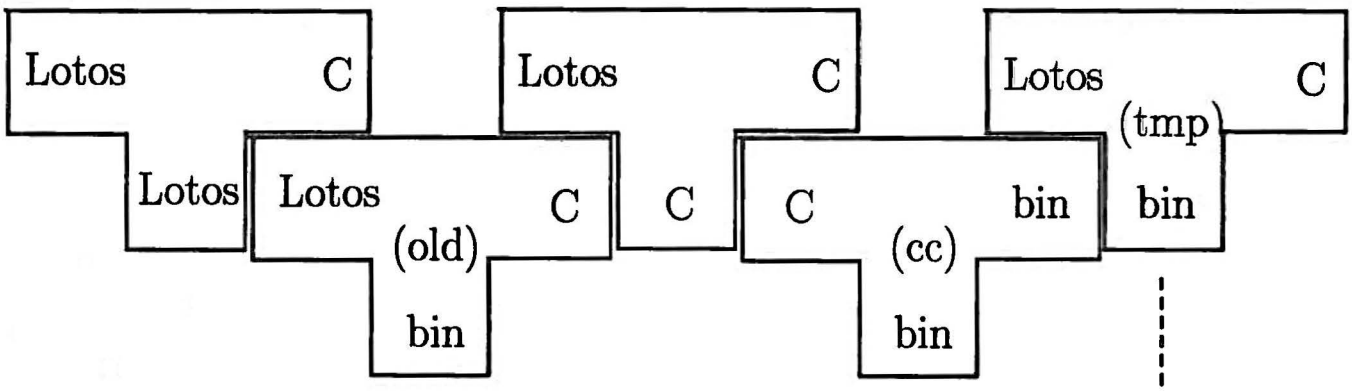
application	t	L_C	T_C	T_O
ATP	4.5	2294	90.165	49.992
MAA	9.1	2041	110.878	40.960
VTT	7.7	3202	161.159	57.940
OTS	19	5252	251.407	72.144
XTL	32	8386	463.823	221.944
FWC	79	21502	1074.158	320.468

Mécanisme d'auto-compilation

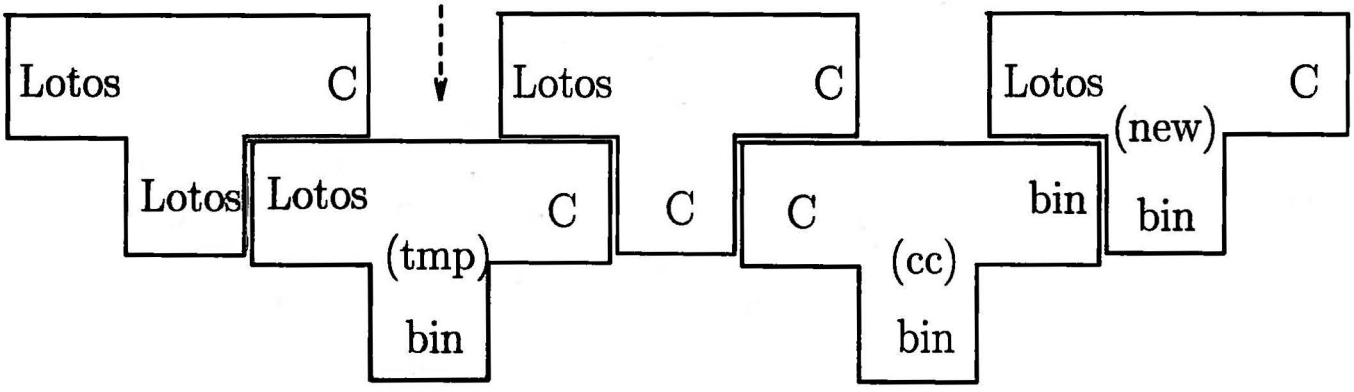
Diagrammes en "T" de Bratman [1961]



Première auto-compilation



Seconde auto-compilation



Bilan

- restrictions pragmatiques sur le langage source :
  - équations orientées
  - constructeurs explicites
  - pas d'équations entre constructeurs

- implémentation efficace des sortes et constructeurs :
  - méthode générale d'implémentation
  - reconnaissance des cas particuliers

- compilation efficace des non-constructeurs :
  - algorithme de compilation du filtrage
  - nombreuses optimisations

- réalisation d'un outil utilisable :
  - robustesse et messages d'erreurs soignés
  - rapidité de compilation
  - vérifications sémantiques poussées
  - plusieurs modes de mise au point (debug, trace)
  - possibilité d'avoir des sortes/opérations externes

- application à des exemples de taille significative

Perspectives

Evolution selon 2 critères essentiels :

- conformité avec la norme LOTOS
- réponse à de "vrais" problèmes

Extension du langage accepté :

- constructeurs non libres
- types paramétrés et instanciés

Ajout d'autres pré- et post-processeurs

Ajout de nouvelles vérifications :

- conditions suffisantes de terminaison
- exclusion mutuelle entre les équations

Optimisations concernant les types :

- reconnaissance des listes, arbres binaires, etc.
- couplage avec un ramasse-miettes

Optimisations concernant les fonctions :

- détection des tests imbriqués
- élimination de la récursion

Le projet EUCALYPTUS

Consortium euro-canadien

Objectif : développer une boîte à outils intégrée pour traiter les spécifications LOTOS

Canada

- **Université de Montréal :** génération de tests (TETRA)

- **Université d'Ottawa :** simulation (ISLA)

Europe

- **VERIMAG (Grenoble) :** compilation et vérification (CÆSAR/ALDEBARAN)

- **Université de Liège :** extensions à ACT-ONE et évaluation des outils sur les spécifications OSI95