Overview of the CÆSAR.ADT abstract data type compiler

Hubert Garavel
Christian Bard (1988)
Philippe Turlier (1991–1992)
Radu Mateescu (1993)
Mihaela Sighireanu (1994)

INRIA projet SPECTRE – VERIMAG
Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE

Plan

1. An "operational" subset of ActOne

2. Verifications performed by the compiler

3. Translation of sorts and constructors

4. Translation of non-constructors and equations

5. Applications

6. Conclusion

Introduction

Algebraic data types exist in LOTOS (and SDL).

How can they be handled?

Various attitudes:

indifference: restriction to "basic LOTOS"

subversion: importation of external types or replacement with concrete data type definitions

interpretation: term rewriting techniques, possibly with resolution or narrowing

compilation: translation into an imperative language (LISP, C, Ada...)

Subset of LOTOS accepted

Constructors must be identified explicitly:

```
type Boolean is
   sort
      Bool
   opns
      true (*! constructor *),
      false (*! constructor *) : -> Bool}
   ...
```

The form of equations is restricted:

1. The left-hand side of each equation has the form $F(V_1, \ldots, V_n)$, where F is a non-constructor

2. Terms $V_1, \ldots, V_n$ may only contain constructors and variables

3. Any variable occurring on the right-hand side must also occur on the left-hand side

   $F(X)$ = $Y + 1$ is rejected

4. Any variable occurring in a premiss must also occur on the left-hand side

   $Y \neq 0$ => $F(X)$ = 1 is rejected

Chosen rewrite strategy

1. orientation of equations (from left to right)

2. special rewrite strategy combining:

   - **call by value**, or functional evaluation
     "*when several terms can be rewritten, innermost ones are rewritten first*"
     $\Longleftrightarrow$
     "*all the sub-terms of a term are rewritten before the term itself*"

   - **decreasing priority between equations**
     "*when several equation simultaneously apply, the first one is selected*"

- this strategy is not completely deterministic
- confluence is not always a desirable property
  Example :

$$X \texttt{ equal } X \texttt{ = true}$$
$$X \texttt{ equal } Y \texttt{ = false}$$

Source language: semantics

- $\mathcal{T}$: terms without variables
- $\mathcal{V}(X)$: terms without non-constructors
- $\mathcal{V}$: terms without variables nor non-constructors
- *eqns* $[F]$: list of equations associated to F
- Σ : set of substitutions from $\mathcal{V}(X)$ to $\mathcal{V}$

"*rewr* $[T]$" evaluates the term T belonging to $\mathcal{T}$ and returns a value belonging to $\mathcal{V}$ (ou "$\perp$" if the equations do not specify how T has to be evaluated).

$$\frac{(\exists i \in \{1,\ldots,n\})\ \mathit{rewr}\ [T_i] = \perp}{\mathit{rewr}\ [C(T_1,\ldots,T_n)] = \perp}$$

$$\frac{(\forall i \in \{1,\ldots,n\})\ \mathit{rewr}\ [T_i] \neq \perp}{\mathit{rewr}\ [C(T_1,\ldots,T_n)] = C(\mathit{rewr}\ [T_1],\ldots,\mathit{rewr}\ [T_n])}$$

$$\frac{(\exists i \in \{1,\ldots,n\})\ \mathit{rewr}\ [T_i] = \perp}{\mathit{rewr}\ [F(T_1,\ldots,T_n)] = \perp}$$

$$\frac{(\forall i \in \{1,\ldots,n\})\ \mathit{rewr}\ [T_i] \neq \perp}{\begin{array}{c}\mathit{rewr}\ [F(T_1,\ldots,T_n)] = \\ \mathit{apply}\ [F][\mathit{rewr}\ [T_1],\ldots,\mathit{rewr}\ [T_n]][\mathit{eqns}\ [F]]\end{array}}$$

Source language: semantics

"*apply* $[F][v_1, \ldots, v_n][E_1, \ldots, E_p]$" computes the value returned by the non-constructor F applied to the list of actual parameter $v_1, \ldots, v_n$ belonging to $\mathcal{V}$, where $E_1, \ldots, E_p$ is the list of equations associated to F

$$\frac{-}{\mathit{apply}\ [F][v_1, \ldots, v_n][\ \emptyset] = \bot}$$

$$\frac{\begin{array}{c} E_1 ::= F(V_1, \ldots, V_n)\ \texttt{=}\ T \\ (\exists \sigma \in \Sigma)\ (\forall i \in \{1, \ldots, n\})\ \sigma(V_i) = v_i \end{array}}{\mathit{apply}\ [F][v_1, \ldots, v_n][E_1, \ldots, E_p] = \mathit{rewr}\ [\sigma(T)]}$$

$$\frac{\begin{array}{c} E_1 ::= P_1\ \textbf{and}\ \ldots\ \textbf{and}\ P_m\ \texttt{=>}\ F(V_1, \ldots, V_n)\ \texttt{=}\ T \\ (\forall j \in \{1, \ldots, m\})\ P_j ::= T_1^j = T_2^j \\ (\exists \sigma \in \Sigma)\ (\forall i \in \{1, \ldots, n\})\ \sigma(V_i) = v_i \\ (\forall j \in \{1, \ldots, m\})\ \mathit{rewr}\ [\sigma(T_1^j)] = \mathit{rewr}\ [\sigma(T_2^j)] \end{array}}{\mathit{apply}\ [F][v_1, \ldots, v_n][E_1, \ldots, E_p] = \mathit{rewr}\ [\sigma(T)]}$$

$$\frac{\textit{in any other case}}{\begin{array}{c} \mathit{apply}\ [F][v_1, \ldots, v_n][E_1, \ldots, E_p] = \\ \mathit{apply}\ [F][v_1, \ldots, v_n][E_2, \ldots, E_p] \end{array}}$$

Verifications for sorts and constructors

1. detection of sorts without constructors
   $\Longrightarrow$ considered as external sorts
2. detection of improductive sorts

   sort S_1
   - *constructor 1:* $F_1 : \texttt{bool}, S_2 \longrightarrow S_1$
   - *constructor 2:* $F_2 : S_1 \longrightarrow S_1$

   sort S_2
   - *constructor:* $F_3 : S_1, \texttt{nat} \longrightarrow S_2$

3. the constructors of an external sort must be external

4. the constructors of a non-external sort must not be external

5. new constructors cannot be added to a renamed sort

6. the constructors of a given sort S should be declared in the same type as S (modularity)

Verifications for non-constructors

1. detection of non-constructeurs without equations
   $\Longrightarrow$ considered as external functions

2. an external operation (constructor or non-constructor) must not have associated equations

3. the equations associated to a given non-constructor F should occur in the type where F is declared (modularity)

4. new equations cannot be added to a renamed non-constructor

5. left-hand sides of equations are made linear:

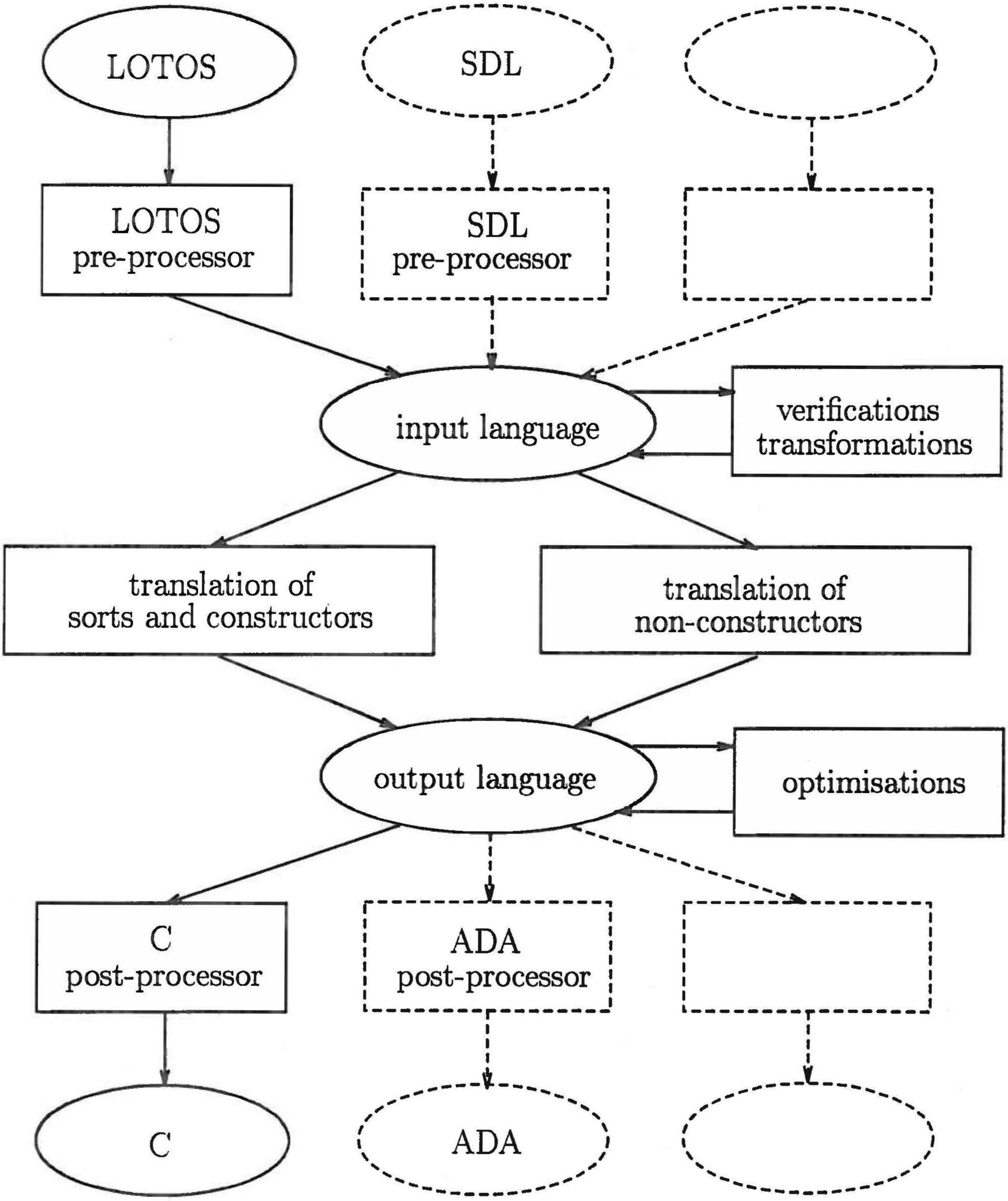
   $$P \texttt{ => } F(X, C(X), \ldots) \texttt{ = } T$$

   is replaced with:

   $$P \textbf{ and } (X \texttt{ = } X') \texttt{ => } F(X, C(X'), \ldots) \texttt{ = } T$$

   where X' is a new variable of the same sort as X

The CÆSAR.ADT compiler



Compiling sorts and non-constructors

For each (non-external) sort S, one must produce:

- a type $\boxed{\texttt{TYPE}_S}$
- a comparison function $\boxed{\texttt{CMP}_S : S \times S \rightarrow \texttt{bool}}$
- an iteration macro $\boxed{\texttt{ITR}_S}$
- a printing procedure $\boxed{\texttt{PRT}_S : \texttt{file} \times S}$

- For each constructor $C : S_1, \ldots, S_n \rightarrow S$ one must produce:
  - a function $\boxed{\texttt{FUNC}_C : S_1, \ldots, S_n \rightarrow S}$

    $$\texttt{FUNC}_C(v_1, \ldots, v_n) = C(v_1, \ldots, v_n)$$

  - a test predicate $\boxed{\texttt{TEST}_C : S \rightarrow \texttt{bool}}$

    $$v \textit{ has the form } C(v_1, \ldots, v_n) \Longleftrightarrow \texttt{TEST}_C(v) = \textit{true}$$

  - n selection functions $\boxed{\texttt{SEL}_C^i : S \rightarrow S_i}$ $(1 \leq i \leq n)$

    $$v \textit{ has the form } C(v_1, \ldots, v_n) \Longrightarrow \texttt{SEL}_C^i(v) = v_i$$

General sort implementations

Principle 1: The implementation of a given sort only depends on the profile of its constructors

Principle 2: Any sort can be implemented using only pointers and discriminated unions

Example:

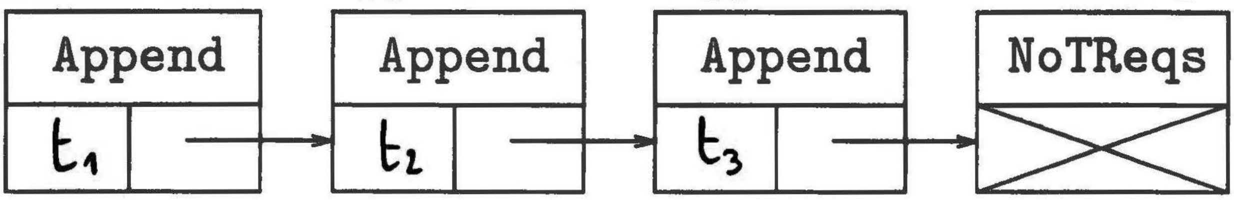
```
sorts
   History
cons
   NoTReqs  : -> History
   Append   : TSP, History -> History
```

Context-free definition of **`History`** terms:

```
<History> ::= NoTReqs | Append (<TSP>, <History>)
```

Representation by a linked list:

```
Append (t1, Append (t2, Append (t3, NoTReqs)))
```



Optimized sort implementations

Special case 1:

sort ADDR
constructor 1: FIRST :$\longrightarrow$ ADDR
constructor 2: NEXT : ADDR $\longrightarrow$ ADDR

$\Longrightarrow$ implemented using an integer type

Special case 2:

sort SIGNAL
constructor 1: SIGHUP :$\longrightarrow$ SIGNAL
constructor 2: SIGINT :$\longrightarrow$ SIGNAL
constructor 3: SIGQUIT :$\longrightarrow$ SIGNAL
constructor 4: SIGILL :$\longrightarrow$ SIGNAL

$\Longrightarrow$ implemented using an enumerated type (2 bits)

Special case 3:

sort TIMEVAL
constructor: TIME : SEC, USEC $\longrightarrow$ TIMEVAL

$\Longrightarrow$ implemented using a record type

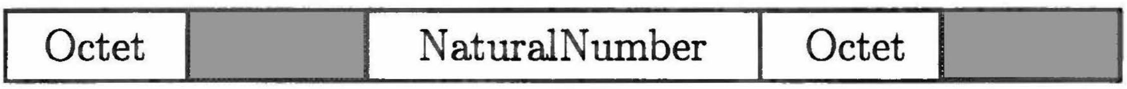
Optimizations

1. Using the minimal number of bits

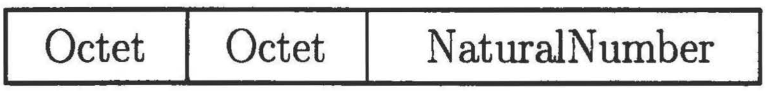
`Boolean` → 1 bit
`Bit` → 1 bit
`Octet` → 8 bits
...

2. Permutation of record fields

Random ordering



Optimal ordering



Compiling non-constructors and equations

source language (LOTOS)
(declarative)
↓
target language (C)
(imperative)

Objectif : **compiling instead of interpreting**

Several algorithms for pattern-matching compiling:

- [Augustsson, 1985]
- [Wadler, 1987]
- [Kaplan, 1987]
- [Schnœbelen, 1988]
- [Pettersson, 1992]
- [Puel-Suarez, 1993]

Chosen algorithm: [Schnœbelen, 1988]

- orthogonal to the implementation of sorts
- compiles a function on a given domain
- handles conditional equations
- handles non-free constructors

Target language: syntax

Terminal symbols:

- C : constructor
- F : non-constructor
- m : integer

Non-terminal symbols:

- I : instruction
- E : expression

$$
\begin{array}{rcl}
I & ::= & \textbf{return } E \\
 & | & \textbf{if } E \textbf{ then } I_1 \textbf{ else } I_2 \\
 & | & \textbf{error}
\end{array}
$$

$$
\begin{array}{rcl}
E & ::= & \$m \\
 & | & \textbf{apply } C, E_1, \ldots, E_n \\
 & | & \textbf{apply } F, E_1, \ldots, E_n \\
 & | & E_1 \textbf{ and } E_2 \\
 & | & E_1 = E_2 \\
 & | & \textbf{test } C, E_0 \\
 & | & \textbf{select } C, m, E_0
\end{array}
$$

The body of each generated function is an instruction

Target language: semantics

Notations:

- F: non-constructor considered
- L: list of actual parameters supplied to F
- I: instruction occurring in the body of F
- E: expression occurring in the body of F

Two mutually recursive functions:

- "$\mathit{exec}\ [I][L]$" executes instruction I and returns its result
- "$\mathit{eval}\ [E][L]$" evaluates expression E and returns its value

Rule for "return":

$$\frac{-}{\mathit{exec}\ [\mathbf{return}\ E][L] = \mathit{eval}\ [E][L]}$$

Rules for "if":

$$\frac{\mathit{eval}\ [E][L] = \mathit{true}}{\mathit{exec}\ [\mathbf{if}\ E\ \mathbf{then}\ I_1\ \mathbf{else}\ I_2][L] = \mathit{exec}\ [I_1][L]}$$

$$\frac{\mathit{eval}\ [E][L] = \mathit{false}}{\mathit{exec}\ [\mathbf{if}\ E\ \mathbf{then}\ I_1\ \mathbf{else}\ I_2][L] = \mathit{exec}\ [I_2][L]}$$

Target language: semantics

Rule for "error":

$$\frac{-}{exec\ [\mathbf{error}][L] = \bot}$$

Rule for "$":

$$\frac{-}{eval\ [\$m][T_1, \ldots, T_m] = T_m}$$

Rules for "apply" (case of a constructor):

$$\frac{(\exists i \in \{1, \ldots, n\})\ eval\ [E_i][L] = \bot}{eval\ [\mathbf{apply}\ C, E_1, \ldots, E_n][L] = \bot}$$

$$\frac{(\forall i \in \{1, \ldots, n\})\ eval\ [E_i][L] \neq \bot}{\begin{array}{c} eval\ [\mathbf{apply}\ C, E_1, \ldots, E_n][L] = \\ C(eval\ [E_1][L], \ldots, eval\ [E_n][L]) \end{array}}$$

Rules for "apply" (case of a non-constructor):

$$\frac{(\exists i \in \{1, \ldots, n\})\ eval\ [E_i][L] = \bot}{eval\ [\mathbf{apply}\ F, E_1, \ldots, E_n][L] = \bot}$$

$$\frac{\begin{array}{c} (\forall i \in \{1, \ldots, n\})\ eval\ [E_i][L] \neq \bot \\ body\ of\ function\ F\ is\ instruction\ I \end{array}}{\begin{array}{c} eval\ [\mathbf{apply}\ F, E_1, \ldots, E_n][L] = \\ exec\ [I][eval\ [E_1][L], \ldots, eval\ [E_n][L]] \end{array}}$$

Target language: semantics

Rules for "and":

$$\frac{(\mathit{eval}\ [E_1][L] = \bot)\ \vee\ (\mathit{eval}\ [E_2][L] = \bot)}{\mathit{eval}\ [E_1\ \textbf{and}\ E_2][L] = \bot}$$

$$\frac{(\mathit{eval}\ [E_1][L] \neq \bot)\ \wedge\ (\mathit{eval}\ [E_2][L] \neq \bot)}{\mathit{eval}\ [E_1\ \textbf{and}\ E_2][L] = (\mathit{eval}\ [E_1][L]\ \wedge\ \mathit{eval}\ [E_2][L])}$$

Rules for "=":

$$\frac{(\mathit{eval}\ [E_1][L] = \bot)\ \vee\ (\mathit{eval}\ [E_2][L] = \bot)}{\mathit{eval}\ [E_1\ \texttt{=}\ E_2][L] = \bot}$$

$$\frac{(\mathit{eval}\ [E_1][L] \neq \bot)\ \wedge\ (\mathit{eval}\ [E_2][L] \neq \bot)}{\mathit{eval}\ [E_1\ \texttt{=}\ E_2][L] = (\mathit{eval}\ [E_1][L] = \mathit{eval}\ [E_2][L])}$$

Rules for "test":

$$\frac{\mathit{eval}\ [E_0][L] = \bot}{\mathit{eval}\ [\textbf{test}\ C, E_0][L] = \bot}$$

$$\frac{\mathit{eval}\ [E_0][L]\ \textit{has the form}\ C'(T_1, \ldots, T_n)}{\mathit{eval}\ [\textbf{test}\ C, E_0][L] = (C = C')}$$

Rules for "select":

$$\frac{\mathit{eval}\ [E_0][L] = \bot}{\mathit{eval}\ [\textbf{select}\ C, m, E_0][L] = \bot}$$

$$\frac{\mathit{eval}\ [E_0][L]\ \textit{has the form}\ C(T_1, \ldots, T_n)}{\mathit{eval}\ [\textbf{select}\ C, m, E_0][L] = T_m}$$

Compiling non-constructors: examples

implies : bool × bool ⟶ bool
X implies Y = not (X) or Y
apply or, (**apply** not, $1), $2

not : bool ⟶ bool
not (true) = false not (false) = true
if (**test** true, $1) **then return** (**apply** true) **else return** (**apply** false)

and : bool × bool ⟶ bool
X and true = X X and false = false
if (**test** true, $2) **then return** $1 **else return** (**apply** false)

Compiling non-constructors: examples

+ : nat × nat ⟶ nat
M + 0 = M M + succ (N) = succ (M) + N
if (**test** 0, \$2)   **then return** \$1   **else return** (**apply** +, (**apply** succ, \$1), (**select** succ, 1, \$2))

- : nat × nat ⟶ nat
M - 0 = M succ (M) - succ (N) = M - N
if (**test** 0, \$2)   **then return** \$1   **else if** (**test** succ, \$1)   **then return** (**apply** -, (**select** succ, 1, \$1), (**select** succ, 1, \$2))   **else** error

max : nat × nat ⟶ nat
$M \geq N$ => max (M, N) = M max (M, N) = N
if ((**apply** ≥, \$1, \$2) = true)   **then return** \$1   **else return** \$2

Compiling non-constructors: examples

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

if (**test** 0, $1)
then if (**test** 0, $2)
    then return (**apply** true)
    else return (**apply** false)
else if (**test** 0, $2)
    then return (**apply** false)
    else return (**apply** eq, (**select** succ, 1, $1),
(**select** succ, 1, $2))

equal : nat × nat ⟶ bool
M equal M = true M equal N = false
if ($1 = $2) **then return** (**apply** true) **else return** (**apply** false)

Optimisations

Constant detection

Functions are replaced with once-computed variables

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

"Simple functions" detection

Functions are replaced with macro-definitions

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

Test reduction

Short-circuits are used: `&&`, `||`, `?:`

$$\frac{\texttt{if P then if Q then X else Y else Y}}{\texttt{if (P \&\& Q) then X else Y}}$$

Redundant code elimination

"`goto`" instructions are introduced for factorizing code

Applications

ATP : compiler for a timed process algebra

MAA : cryptographic signature algorithm [ISO 8731]

VTT : transit node

OTS : OSI transport service [ISO 8072]

XTL : compiler for an extended temporal logic

FWC : Airbus A320 flight warning computer

Source LOTOS programs

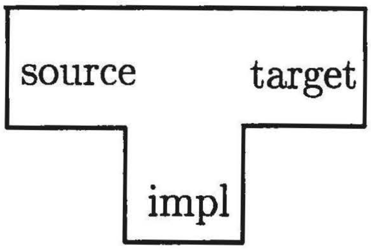
	lines	Kbytes	types	sorts	cons	n-cons	eqns
ATP	376	12.638	12	12	30	73	81
MAA	1126	34.178	10	10	16	184	167
VTT	1130	39.444	17	16	39	111	345
OTS	1802	72.170	40	32	87	245	338
XTL	3100	147.345	24	15	135	271	761
FWC	13524	580.119	145	139	515	786	1432

Code generated by CÆSAR.ADT 4.1 on a SUN 4/40

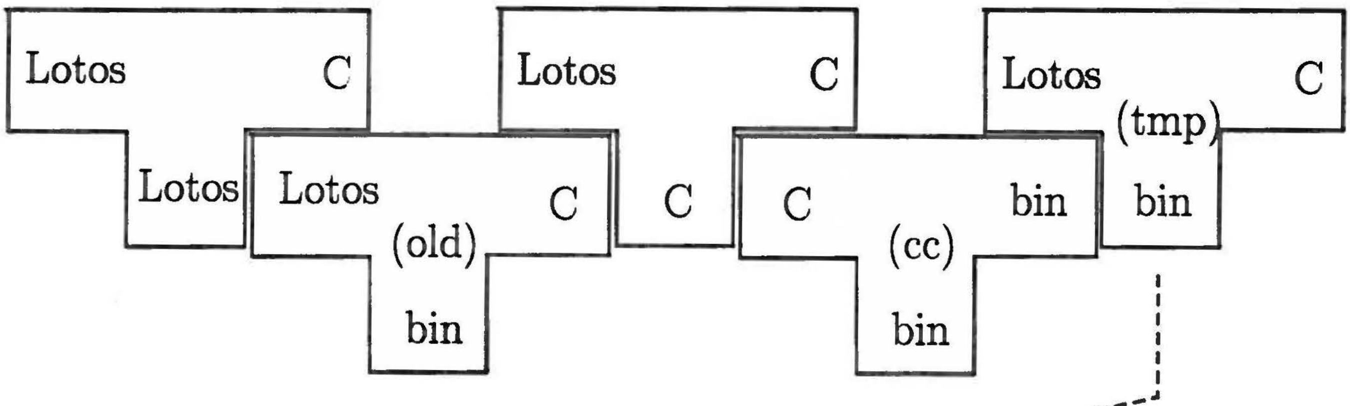
	time	lines (C)	Kbytes (C)	Kbytes (obj)
ATP	4.5	2294	90.165	49.992
MAA	9.1	2041	110.878	40.960
VTT	7.7	3202	161.159	57.940
OTS	19	5252	251.407	72.144
XTL	32	8386	463.823	221.944
FWC	79	21502	1074.158	320.468

A self-compiling compiler

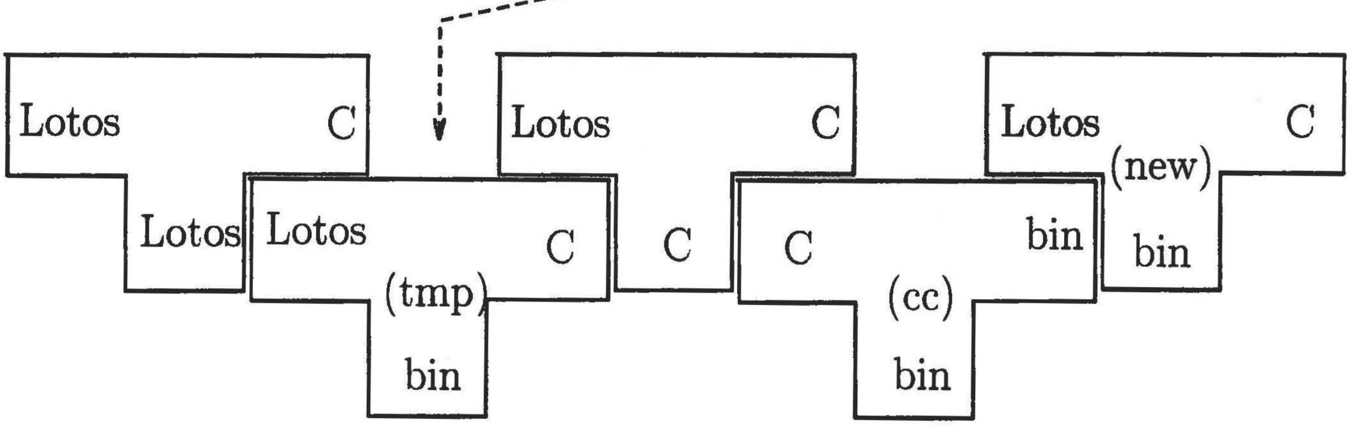
"T" diagrams [Bratman, 1961]



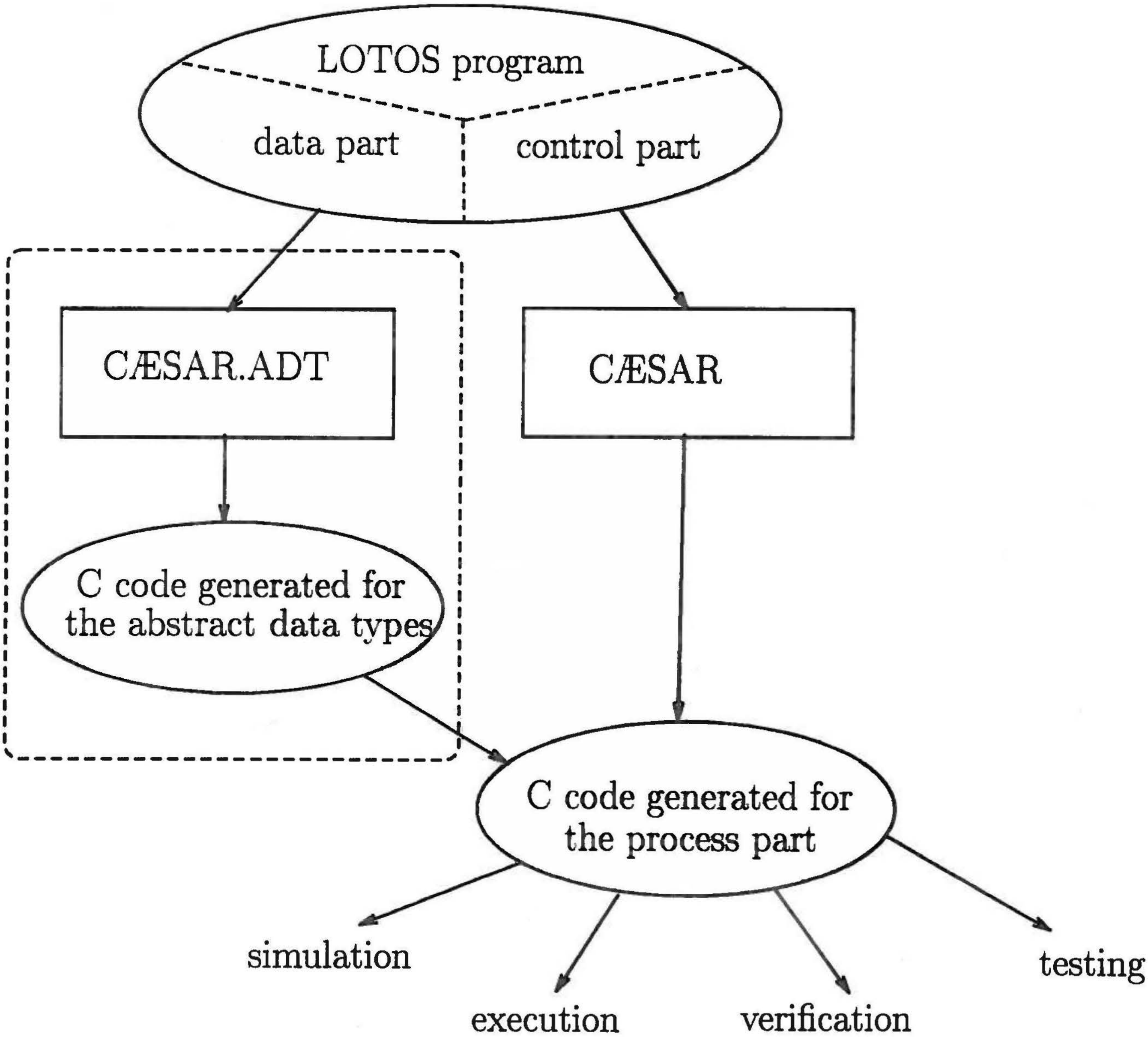
Second pass of boostrap



First pass of bootstrap



The CÆSAR and CÆSAR.ADT compilers



The EUCALYPTUS consortium

In Europe

- **INRIA project SPECTRE / VERIMAG**

  supported by the Commission of the European Communities.

- **University of Liège**

  supported by the Commission of the European Communities.

In Canada

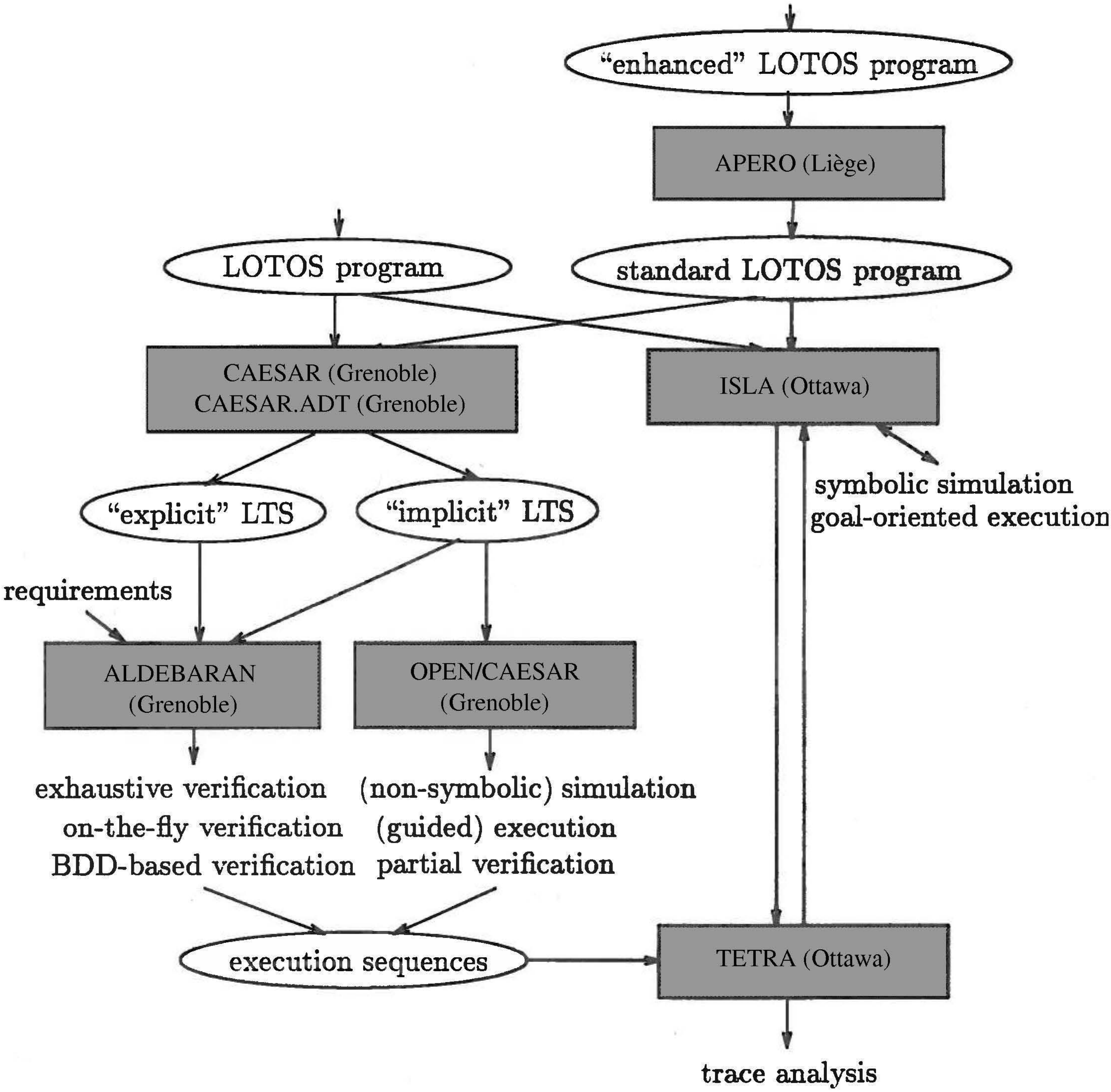
- **University of Montréal**

  supported by the IDACOM-NSERC-CWARC Industrial Research Chair on Communications Protocols at the University of Montreal.

- **University of Ottawa**

  supported by the Telecommunications Research Institute of Ontario (TRIO).

The EUCALYPTUS toolset architecture



Conclusion

- pragmatic restrictions on the source language:
  - orientation of equations
  - explicit indication of constructors
  - no equations between constructors

- efficient implementation for sorts and constructors:
  - general representation scheme
  - ad hoc optimizations for common cases

- efficient implementation for non-constructors:
  - pattern-matching compiling algorithm
  - many optimizations

- a workable compiler, CÆSAR.ADT:
  - fast and robust
  - static semantics verifications
  - debugging features (debug, trace)
  - importation of external sorts and operations

Current research directions

Support for parameterized types:

- new verifications needed
- adapt "flattening" to constructors

New optimizations for sorts:

- recognition of lists, binary trees, etc.
- garbage-collection

New optimisations for non-constructors:

- reduction of nested tests
- recursion elimination