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

Hubert Garavel, Philippe Turlier

VERIMAG
B.P. 53X
38041 GRENOBLE cedex
FRANCE

Résumé. *Cet article présente le compilateur* CÆSAR.ADT *dont le rôle est de traduire en langage C les définitions de types abstraits algébriques présents dans les programmes* LOTOS. *Grâce à ce compilateur, il est possible d'exécuter une spécification algébrique, ce qui autorise des applications immédiates pour le prototypage et la vérification formelle. La rapidité de la traduction et l'efficacité du code engendré permettent de traiter des programmes de taille significative avec des performances réalistes.*
Mots-clés : compilation, filtrage, génération de code, LOTOS, *méthodes formelles, réécriture de termes, spécification algébrique, traduction, types abstraits algébriques.*
Abstract. *This paper presents the* CÆSAR.ADT *compiler that translates the definitions of abstract data types contained in* LOTOS *programs into C code.*
Key words : abstract data types, algebraic specification, code generation, compilation, formal methods, LOTOS, *pattern-matching, term rewriting, translation.*

Introduction

Le langage LOTOS est issu des travaux de l'ISO (Organisation Internationale de Normalisation) concernant la définition de protocoles normalisés. C'est un langage de spécification moderne, qui comprend deux parties strictement orthogonales :

Une partie "contrôle", destinée à décrire le comportement observable des systèmes parallèles. Il s'agit d'une algèbre de processus qui s'inspire des concepts de CCS et CSP.

Une partie "données", destinée à décrire les valeurs manipulées et échangées par les composants du système. Cette partie est basée sur le formalisme des types abstraits algébriques, et plus précisément sur le langage ACTONE [EM85] [dMRV92].

Le succès rencontré par LOTOS a donné naissance à de nombreux outils pour ce langage. Le traitement de la partie "types abstraits algébriques" a soulevé des difficultés, qui ont conduit les développeurs d'outils à adopter diverses solutions :

- Certains ont choisi d'ignorer la partie données. Il existe donc des outils qui ne traitent que "*basic* LOTOS", c'est-à-dire le sous-ensemble de LOTOS limité à la partie contrôle. Cette approche n'est pas réaliste, car la quasi-totalité des spécifications LOTOS contiennent des données.

- D'autres ont proposé de s'écarter de la norme ISO en remplaçant les types abstraits algébriques par un formalisme impératif, plus facile à traiter, par exemple le langage de description de données ASN.1.

- D'autres encore ont préféré ne considérer les définitions de types abstraits que comme des spécifications d'interface, l'implémentation des types et des opérations étant assurée manuellement (sous forme de fichiers externes ou bien de fragments de code C insérés directement dans les spécifications LOTOS [MdM88]).

- Une approche plus sophistiquée consiste à interpréter les spécifications algébriques. Pour cela, les équations sont orientées et considérées comme des règles de réécriture. Diverses stratégies peuvent être appliquées : appel par valeur, évaluation paresseuse, surréduction, surréduction paresseuse [EW92], etc. Cette approche présente cependant l'inconvénient d'être souvent coûteuse en temps et en mémoire.

- C'est pourquoi, une autre approche peut être préférée, qui consiste à compiler les spécifications algébriques, c'est-à-dire à les traduire vers un langage-cible susceptible d'être exécuté efficacement. Parmi les langages-cibles proposés, certains sont des langages abstraits spécialement conçus pour supporter les langages fonctionnels [WB89], d'autres sont des langages algorithmiques généraux tels que LISP [Kar92] ou C [MdM88] [Gar89].

Le compilateur CÆSAR.ADT décrit ici participe de cette dernière approche. Il traduit automatiquement les types abstraits LOTOS en C, en essayant de produire un code C efficace, grâce une "vraie" compilation des équations et grâce une implémentation optimisée des structures de données.

Ce compilateur fait partie d'une boîte à outils pour la vérification de programmes LOTOS [FGM+92]. Il a été conçu comme un complément à l'outil CÆSAR [GS90] qui traite la partie contrôle des programmes LOTOS et qui ne peut fonctionner que s'il dispose d'une implémentation en langage C pour les sortes et les opérations définies dans le programme LOTOS.

Une version antérieure du compilateur CÆSAR.ADT a été présentée dans [Gar89]. Plus récemment, une refonte radicale a été entreprise, conduisant à une réécriture complète du compilateur. [Tur92]. Cet article présente la nouvelle version 4.0 en soulignant plus particulièrement les améliorations apportées par rapport aux versions précédentes.

1 L'architecture générale du compilateur

A l'origine, les premières versions de CÆSAR.ADT étaient uniquement destinées à la traduction en langage C des types abstraits LOTOS. Le nouveau compilateur CÆSAR.ADT a été conçu dans un but plus général :

- il pourrait accepter en entrée n'importe quel langage source — autre que LOTOS — basé sur les types abstraits algébriques. Cette possibilité est particulièrement intéressante dans la mesure où il existe de nombreux autres langages basés sur les types abstraits ([Wir90] en donne une liste assez complète) et notamment, dans le domaine des protocoles, le langage SDL, dont la partie données est quasiment identique à celle de LOTOS.
- il pourrait produire en sortie n'importe quel langage cible — autre que C — pourvu que ce langage dispose du mécanisme d'appel de fonction et du test "**if**... **then**... **else**...". C'est dire que le nouveau compilateur CÆSAR.ADT pourrait produire du code pour la quasi-totalité des langages impératifs existants, dont ADA, C++, etc.

Cette indépendance du compilateur par rapport à LOTOS et à C est obtenue au moyen de deux formes intermédiaires, appelées *langage d'entrée* et *langage de sortie*, qui seront définies aux sections 2 et 3.

La figure 1 définit l'architecture générale du nouveau compilateur CÆSAR.ADT. Les différentes étapes de la traduction sont les suivantes :

- Le passage du *langage source* (LOTOS, SDL...) au *langage d'entrée* est assuré au moyen d'un *pré-processeur* chargé d'effectuer l'analyse lexicale et syntaxique, ainsi que les premières vérifications de sémantique statique (notamment la liaison des identificateurs, la vérification des types, la résolution des surcharges d'opérateurs, le calcul des signatures, etc.)

  A l'heure actuelle, seul le pré-processeur LOTOS a été réalisé ; il repose sur la partie avant (*front-end*) du compilateur CÆSAR. Ce pré-processeur traite des programmes LOTOS complets, mais seule la partie données est prise en compte (la partie contrôle étant ignorée).
- La phase de *vérification et transformation* analyse le programme source après que celui-ci ait été traduit en langage d'entrée. Elle s'assure en particulier que les restrictions décrites section 2 sont bien satisfaites. Elle modifie certaines équations pour les mettre sous la forme requise par les algorithmes de compilation.
- L'essentiel de la traduction porte sur le passage du langage d'entrée au langage de sortie. C'est là que sont mis en œuvre les algorithmes complexes. Ce passage est réalisé en deux phases rigoureusement indépendantes : la compilation des sortes et des constructeurs, et la compilation des non-constructeurs, qui seront respectivement présentées aux sections 4 et 5.
- Ensuite, une phase d'optimisation applique un certain nombre de transformations sur le code produit en langage de sortie, en vue d'améliorer son efficacité. Ces transformations sont du type de celles rencontrées habituellement dans l'optimisation à lucarne (*peephole optimization*).

- Finalement, le passage du *langage de sortie* au *langage cible* (C, Ada…) est effectué par un *post-processeur*. Actuellement, seul le post-processeur C a été réalisé.

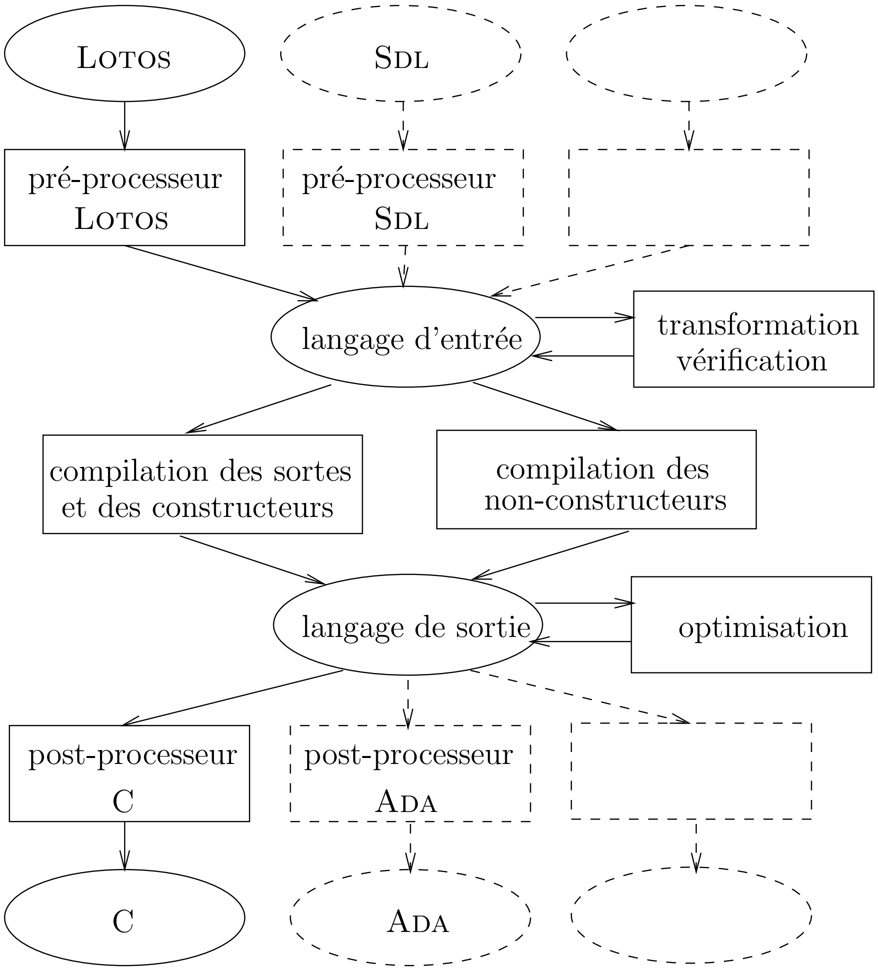

Figure 1 : Architecture du nouveau compilateur Cæsar.adt

Les versions antérieures de Cæsar.adt étaient entièrement écrites en langage C. La version 4.0 innove sur ce point, car la quasi-totalité des traitements sémantiques a été programmée, non pas en langage C, mais en types abstraits algébriques Lotos. Autrement dit, le compilateur s'auto-compile, l'amorçage (*bootstrap*) étant assuré par la précédente version 3.2.

Cette technique d'auto-compilation présente des avantages importants. Par rapport aux versions antérieures écrites en langage C, la nouvelle version du compilateur est plus simple, plus fiable et plus facile à maintenir.

2 Le langage d'entrée

Le langage d'entrée de CÆSAR.ADT est une forme intermédiaire qui se situe au niveau d'une syntaxe abstraite. Du fait de sa simplicité, il devrait pouvoir servir de "dénominateur commun" à bon nombre de langages basés sur les types abstraits algébriques. Ce langage se compose de deux parties "orthogonales", présentées à tour de rôle.

2.1 Sortes et constructeurs

La première partie du langage d'entrée comprend la liste des *sortes* définies dans le programme source LOTOS. A chaque sorte S est associée la liste des *constructeurs* qui renvoient un résultat de sorte S. A chaque constructeur C est associée la liste des sortes des arguments de C.

Comme LOTOS ne prévoit pas de syntaxe particulière pour distinguer les opérations constructeurs des opérations non-constructeurs, c'est l'utilisateur qui signale les constructeurs au compilateur en faisant suivre leur déclaration d'un commentaire spécial de la forme `(*! constructeur *)`.

Le compilateur CÆSAR.ADT permet à l'utilisateur de déclarer "externes" certaines sortes et constructeurs au moyen d'un commentaire `(*! external *)`. Ces informations se retrouvent également dans le langage d'entrée. Le compilateur n'engendre aucun code pour les objets externes : c'est à l'utilisateur de fournir les implémentations correspondantes. En pratique, cette possibilité est très utile pour le développement d'applications réalistes puisqu'elle permet :

- d'implémenter manuellement certains types pour lesquels on dispose d'implémentations efficaces (par exemple, les nombres réels en virgule flottante) ;

- de pouvoir interfacer du code C déjà existant, en définissant "vues" en types abstraits qui permettent de manipuler les objets existants à l'intérieur d'un programme LOTOS.

Concernant les sortes et les constructeurs, le compilateur effectue un certain nombre de vérifications :

- Il détecte les sortes qui ne possèdent aucun constructeur et qui ne sont pas externes ; de telles sortes sont alors considérées comme externes, car il est impossible d'engendrer automatiquement une implémentation pour elles.

- Il détecte les sortes *improductives*, c'est-à-dire les sortes pour lesquelles il n'existe aucun terme sous forme normale, algébriquement clos et bien typé (leur algèbre initiale est vide, bien que ces sortes possèdent un ou plusieurs constructeurs).

- Il vérifie que tous les constructeurs d'une sorte externe sont eux aussi externes, et qu'inversement aucun constructeur d'une sorte non externe n'est externe. En effet, une sorte peut être implémentée soit automatiquement, soit manuellement, mais il n'est pas possible de combiner les deux.

2.2 Non-constructeurs et équations

La seconde partie du langage d'entrée comprend la liste des non-constructeurs définis dans le programme source. A chaque non-constructeur F est associée la liste des sortes des arguments de F, la sorte du résultat de F et la liste des équations qui définissent F en partie gauche (cette notion sera précisée plus loin).

Comme précédemment, il est possible de définir des non-constructeurs externes, pour lequel le compilateur n'engendre pas de code.

La forme des équations fait l'objet de restrictions syntaxiques sur l'emploi des constructeurs et des non-constructeurs. Les équations conditionnelles sont admises. La syntaxe autorisée est décrite par la grammaire ci-dessous, où les symboles terminaux X, C, et F dénotent respectivement une variable, un constructeur et un non-constructeur, et où les symboles non-terminaux V, T, P et E dénotent respectivement un terme sous forme normale, un terme, une prémisse et une équation :

$$
\begin{array}{lcl}
V & ::= & X \\
 & | & C(V_1, \ldots, V_n) \\
T & ::= & X \\
 & | & C(T_1, \ldots, T_n) \\
 & | & F(T_1, \ldots, T_n) \\
P & ::= & T_1 \texttt{ = } T_2 \\
E & ::= & F(V_1, \ldots\ V_n) = T \\
 & | & P_1 \textbf{ and } \ldots \textbf{ and } P_m \texttt{ => } F(V_1, \ldots\ V_n) \texttt{ = } T
\end{array}
$$

C'est ainsi que les équations doivent respecter certaines contraintes :

- En partie gauche d'une équation E doit donc obligatoirement figurer un non-constructeur F (on dit alors que E *définit* F).

  Il est donc interdit de définir un constructeur à l'aide d'équations (pas d'"équations entre constructeurs"). Il s'agit d'une contrainte forte qui impose à l'utilisateur une discipline de "programmation par constructeurs", analogue aux définitions par cas (*pattern-matching*) que l'on trouve dans des langages comme ML ou MIRANDA. Dans une telle approche, les types abstraits sont utilisés davantage comme un langage de programmation fonctionnelle que comme un langage de spécification algébrique.

- Tous les arguments de F en partie gauche ne peuvent contenir que des constructeurs et des variables libres.

- Toute variable figurant en prémisse ou en partie droite d'une équation doit également apparaître en partie gauche de cette même équation.

- Toute équation doit être *linéaire à gauche*, c'est-à-dire qu'une même variable X ne peut apparaître deux fois en partie gauche. Cette condition est requise par l'algorithme de compilation des non-constructeurs. Toutefois, lorsqu'une équation ne satisfait pas cette condition, le compilateur la transforme automatiquement pour la rendre linéaire à gauche, grâce à l'introduction d'une nouvelle variable et à l'ajout d'une prémisse.

Le compilateur effectue aussi d'autres vérifications :

- Il détecte les non-constructeurs qui ne sont définis par aucune équation et qui ne sont pas externes.
- Il détecte les opérations externes (constructeurs et non-constructeurs) qui sont définies par des équations.

D'un point de vue sémantique, les équations figurant dans le langage d'entrée sont interprétées de la manière suivante :

- Tout d'abord, le compilateur considère que les équations sont orientées de la gauche vers la droite, ce qui revient en fait à considérer une définition équationnelle comme un système de réécriture.
- Le compilateur adopte une stratégie de réécriture particulière que l'on peut caractériser comme étant l'*appel par valeur* (*call-by-value*) avec *priorité décroissante entre équations*. Selon cette stratégie, lorsque plusieurs termes peuvent être réécrits simultanément, les plus imbriqués sont réécrits les premiers. Lorsque, pour un même terme, plusieurs équations (considérées comme des règles de réécriture) peuvent simultanément s'appliquer, on donne priorité à celle qui apparaît la première dans le texte du programme source.

  Il faut noter que cette stratégie n'est pas entièrement déterministe puisque l'ordre dévaluation des arguments d'une même fonction n'est pas précisé (ce qui est également le cas dans la plupart des langages-cibles, notamment C et ADA).

Formellement, la sémantique du langage d'entrée peut être définie comme suit :

- on note $\mathcal{T}$ l'ensemble des termes formés de constructeurs et de non-constructeurs, mais ne comportant aucune variable ;
- on note $\mathcal{V}(X)$ l'ensemble des termes formés de constructeurs et de variables, mais ne comportant aucun non-constructeur ;
- on note $\mathcal{V}$ l'ensemble des termes formés uniquement de constructeurs, sans aucune variable ni non-constructeur ;
- classiquement, on note Σ l'ensemble des substitutions de $\mathcal{V}(X)$ vers $\mathcal{V}$, c'est-à-dire l'ensemble des applications de $\mathcal{V}(X)$ vers $\mathcal{V}$ qui sont des morphismes pour les opérations constructeurs ;
- on note "*eqns* $[F]$" la liste des équations qui définissent le non-constructeur F, dans l'ordre où elles apparaissent dans le programme source.

Evaluer un terme de $\mathcal{T}$, c'est lui faire correspondre une valeur dans $\mathcal{V}$. Selon [Sch88], le mécanisme d'évaluation est défini par deux fonctions mutuellement récursives :

- la fonction "*rewr* $[T]$" évalue le terme T de $\mathcal{T}$ et renvoie sa valeur dans $\mathcal{V}$ ou bien "$\perp$" si les équations ne définissent pas comment T doit être évalué (cas d'incomplétude) ;

- la fonction "*apply* $[F][v_1, \ldots, v_n][E_1, \ldots, E_p]$" calcule la valeur renvoyée par le non-constructeur F appliqué à la liste d'arguments $v_1, \ldots, v_n$ (qui sont des termes de $\mathcal{V}$) lorsque la liste des équations définissant F est $E_1, \ldots, E_p$, ou bien "$\perp$" si cette valeur n'est pas définie ;

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

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

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

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

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

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

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

$$\frac{\textit{dans tous les autres cas}}{\mathit{apply}\ [F][v_1, \ldots, v_n][E_1, \ldots, E_p] = \mathit{apply}\ [F][v_1, \ldots, v_n][E_2, \ldots, E_p]}$$

3 Le langage de sortie

Le langage de sortie de CÆSAR.ADT constitue une forme intermédiaire susceptible d'être traduite de manière simple et efficace dans un langage algorithmique tel que le langage C. Comme le langage d'entrée, le langage de sortie se compose de deux parties orthogonales.

3.1 Types

La première partie du langage de sortie consiste en une liste de couples $\langle s, S \rangle$ qui, à chaque sorte LOTOS s définie dans le langage d'entrée, associe son implémentation S.

Cette implémentation est une *expression de type* qui distingue cinq cas : quatre cas particuliers (types entiers, types énumérés, types tuples, types externes) et un cas général (types ne rentrant dans aucun des quatre cas particuliers).

3.2 Fonctions

La seconde partie du langage de sortie contient une liste de couples $\langle f, F \rangle$ qui, à chaque non-constructeur LOTOS f défini dans le langage de sortie, associe son implémentation F.

Deux cas sont à distinguer selon que f est externe ou non. Dans le premier cas, aucun code n'est à engendrer pour f. Dans le second cas, l'implémentation de f est une fonction dont le types des arguments et du résultat sont ceux de f et dont le corps est une instruction I, définie par la syntaxe abstraite suivante, où les symboles non-terminaux $I_0, \ldots, I_n$ dénotent des *instructions*, les symboles non-terminaux $E_0, \ldots\ E_n$ dénotent des *expressions*, et les symboles terminaux C, F et m dénotent respectivement un constructeur, un non-constructeur et un nombre entier :

$$
\begin{array}{lcl}
I & ::= & \textbf{return } E \\
 & \mid & \textbf{if } E \textbf{ then } I_1 \textbf{ else } I_2 \\
 & \mid & \textbf{error} \\
E & ::= & \$m \\
 & \mid & \textbf{apply } C, E_1, \ldots, E_n \\
 & \mid & \textbf{apply } F, E_1, \ldots, E_n \\
 & \mid & E_1 \textbf{ and } E_2 \\
 & \mid & E_1 \texttt{ = } E_2 \\
 & \mid & \textbf{test } C, E_0 \\
 & \mid & \textbf{select } C, m, E_0
\end{array}
$$

Si I (*resp.* E) est une instruction (*resp.* une expression) située dans le corps d'une fonction f appelée avec une liste L de termes $T_1, \ldots, T_n$ comme paramètres effectifs, on note $\mathit{exec}\ [I][T_1, \ldots, T_n]$ (*resp.* $\mathit{eval}\ [E][T_1, \ldots, T_n]$) le résultat de l'exécution de I (*resp.* l'évaluation de E) :

- "**return** E" a pour effet d'évaluer l'expression E et de renvoyer sa valeur comme résultat de la fonction ;

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

- "**if** E **then** I_1 **else** I_2" a pour effet d'évaluer l'expression booléenne E et, selon que le résultat est vrai ou faux, d'exécuter soit I_1, soit I_2 ;

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

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

- "**error**" interrompt l'exécution en affichant un message d'erreur ; cette instruction est exécutée dans les cas d'incomplétude, lorsque les équations ne spécifient pas quel résultat doit être renvoyé par la fonction.

$$\frac{-}{exec\ [\textbf{error}][L] = \perp}$$

De même, la sémantique d'évaluation des expressions est la suivante (en notant que l'évaluation d'une expression renvoie toujours un terme sous forme normale et algébriquement clos) :

- "$\$m$" renvoie la valeur du $m^{\text{ième}}$ paramètre effectif de la fonction ;

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

- "**apply** $C, E_1, \ldots, E_n$" évalue les expressions $E_1, \ldots, E_n$ (dans un ordre non défini) et appelle ensuite le constructeur C en lui passant comme paramètres effectifs les valeurs ainsi obtenues ;

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

$$\frac{(\forall i \in \{1, \ldots, n\})\ eval\ [E_i][L] \neq \perp}{eval\ [\textbf{apply}\ C, E_1, \ldots, E_n][L] = C(eval\ [E_1][L], \ldots, eval\ [E_n][L])}$$

- "**apply** $F, E_1, \ldots, E_n$" fait de même avec le non-constructeur F ;

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

$$\frac{\begin{array}{c}(\forall i \in \{1, \ldots, n\})\ eval\ [E_i][L] \neq \perp \\ \textit{la fonction qui implémente } F \textit{ a comme corps l'instruction } I\end{array}}{eval\ [\textbf{apply}\ F, E_1, \ldots, E_n][L] = exec\ [I][eval\ [E_1][L], \ldots, eval\ [E_n][L]]}$$

- "E_1 **and** E_2" évalue les deux expressions booléennes E_1 et E_2 et effectue un "et logique" de leurs valeurs ;
- "E_1 = E_2" évalue les deux expressions E_1 et E_2 et effectue le test d'égalité sur leurs valeurs ;
- "**test** C, E_0" évalue l'expression E_0 et renvoie vrai si et seulement si la valeur obtenue est de la forme $C(V_1, \ldots, V_n)$;
- "**select** C, m, E_0" évalue l'expression E_0 — dont la valeur doit nécessairement être de la forme $C(T_1, \ldots, T_n)$ — et renvoie la valeur du $m^{\text{ième}}$ argument T_m.

4 L'implémentation des sortes et des constructeurs

Cette phase a pour objet de passer de la première partie du langage d'entrée (c'est-à-dire les sortes et constructeurs LOTOS) à la première partie du langage de sortie (c'est-à-dire les types C).

Pour implémenter une sorte S, CÆSAR.ADT examine la liste des constructeurs associés à S, c'est-à-dire ceux qui renvoient un résultat de sorte S.

- Si S ne possède que deux constructeurs C_1 et C_2 de profils respectifs $C_1 :\longrightarrow S$ et $C_2 : S \longrightarrow S$, elle est implémentée comme un type entier ; les constructeurs C_1 et C_2 sont respectivement implémentés par la fonction constante $\lambda x.0$ et la fonction $\lambda x.x + 1$.
- Si S possède $n \geq 1$ constructeurs $C_1, \ldots, C_n$ ayant tous le profil $C_i :\longrightarrow S$, elle est implémentée comme un type énuméré à n valeurs. Chacun des constructeurs C_i est implémenté comme l'une des valeurs de ce type.
- Si S ne possède qu'un seul constructeur et que ce constructeur comporte $m \geq 1$ arguments, elle est implémenté comme un type tuple (ou enregistrement) comportant m champs.
- Si S a été déclarée externe, ou si elle ne possède aucun constructeur, le compilateur n'engendre pas de code pour elle.
- si aucun des cas particuliers ci-dessus ne convient, alors l'algorithme applique le cas général. Si S possède n constructeurs $C_1, \ldots, C_n$ ayant $m_1, \ldots, m_n$ arguments respectivement, S est alors implémentée comme un type pointeur vers un enregistrement à n champs variants, chaque champ correspondant à l'un des constructeurs ; le $i^{\text{ème}}$ champ de ce type est lui-même est un tuple comportant m_i champs, chacun de ces champs correspondant à l'un des m_i arguments du constructeur C_i.

  Par exemple, une sorte LOTOS décrivant un ensemble, une pile, une file, etc., relève du général et sera implémentée comme une liste chaînée.

Vient ensuite le post-processeur, dont le rôle est de traduire en C le langage de sortie. Pour chaque sorte non-externe S, le post-processeur doit engendrer :

- une définition de type qui implémente S ;
- une fonction de comparaison, de profil $S \times S \longrightarrow$ **bool** qui réalise la comparaison (identité des forme normales) entre deux valeurs de sorte S ;

- un itérateur qui permet de faire décrire à une variable l'ensemble des valeurs de la sorte S (lorsque cet ensemble est fini) ;
- une procédure d'impression qui permet d'afficher une valeur de sorte S sous forme de chaîne de caractères lisible par l'utilisateur.

Pour chaque constructeur C_i de profil $C_i : S_1 \times \ldots \times S_{m_i} \longrightarrow S$, le post-processeur engendre :

- une fonction qui implémente C_i ;
- un *testeur*, c'est-à-dire une fonction de profil $S \longrightarrow$ **bool** qui renvoie vrai si et seulement si la valeur de son argument est de la forme $C_i(V_1, \ldots, V_{m_i})$ où $V_1, \ldots, V_{m_i}$ sont des valeurs quelconques de sortes respectives $S_1, \ldots, S_{m_i}$;
- des *sélecteurs*, c'est-à-dire m_i fonctions de profils respectifs $S \longrightarrow S_j$ avec $1 \leq j \leq m_i$, où la $j^{\text{ème}}$ fonction — qui n'est définie que lorsque son argument est de la forme $C_i(V_1, \ldots, V_{m_i})$ — renvoie V_j.

5 L'implémentation des non-constructeurs

Cette phase a pour objet de transformer les équations associées à un non-constructeur F en une instruction I du langage de sortie. Il s'agit donc de passer d'un formalisme déclaratif à un formalisme impératif.

Pour compiler le filtrage, plusieurs algorithmes ont été proposés, notamment [Aug85] [Wad87] [Kap87] [Sch88] et plus récemment [Pet92] et [PS93]. L'algorithme utilisé dans CÆSAR.ADT est celui proposé par [Sch88]. Cet algorithme a l'avantage de prendre en compte le domaine de la fonction à compiler. De plus, dans sa version étendue, il traite les équations conditionnelles.

Cet algorithme étant trop complexe pour être présenté en détail dans le cadre de cet article, on se contente ici de présenter ses effets sur divers exemples.

Dans les exemples ci-dessous, on considère la sorte `bool` — dont les constructeurs sont `false` et `true` — et la sorte `nat` — dont les constructeurs sont `0` et `succ`. X et Y sont deux variables de sorte `bool` ; M et N sont deux variables de sorte `nat`.

Pour chaque tableau, la colonne de gauche donne le profil du non-constructeur et les équations qui le définissent ; la colonne de droite contient l'instruction constituant le corps de la fonction engendrée pour cet opérateur.

Il faut noter que le code produit pour les non-constructeurs utilise le code engendré au moment de la compilation des sortes (constructeurs, testeurs, sélecteurs et relations de comparaison). Néanmoins, l'algorithme de compilation des non-constructeurs est complètement indépendant de l'implémentation concrète des sortes. Le code produit pour les exemples ci-dessous est donc représentatif du comportement de l'algorithme de compilation lorsque celui-ci est appliqué à des opérateurs et des sortes autres que `bool` et `nat`.

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

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

`-` : `nat` × `nat` ⟶ `nat`
M `- 0` = M
`succ` (M) `- succ` (N) = M `-` N
`(* otherwise: error *)`
	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 `(* otherwise: *) max` (M, N) = N	**if** ((**apply** $\geq$, $\$1$, $\$2$) `= true`) **then return** $\$1$ **else return** $\$2$

La phase d'optimisation améliore l'efficacité du code en langage de sortie engendré par l'algorithme de compilation des opérations. Par exemple, elle est capable :

- d'implémenter par des macro-définitions (`#define` en C) les fonctions dont le corps se réduit à une instruction de la forme "**return** E" ;
- d'effectuer des "réductions de force" pour certains opérateurs, en utilisant des règles de transformation dont voici quelques exemples :

$$\frac{\textbf{if } \neg E \textbf{ then } I_1 \textbf{ else } I_2}{\textbf{if } E \textbf{ then } I_2 \textbf{ else } I_1}$$

$$\frac{\textbf{if } E_1 \textbf{ then } I_1 \textbf{ else if } E_2 \textbf{ then } I_1 \textbf{ else } I_2}{\textbf{if } (E_1 \textbf{ and\_then } E_2) \textbf{ then } I_1 \textbf{ else } I_2}$$

6 Applications

Le compilateur CÆSAR.ADT a été utilisé sur des programmes LOTOS de grande taille :

MAA : Il s'agit de l'algorithme cryptographique *Message Authentication Algorithm* qui permet de calculer une signature numérique sur un fichier dans un but d'authentification. Cet algorithme a été décrit formellement en LOTOS [Mun91].

Le code C produit par CÆSAR.ADT a été utilisé pour encrypter divers fichiers dont les tailles s'échelonnent entre 800 et 30000 octets. Le temps d'encryption pour ces fichiers varie entre 1 seconde et 51 secondes sur SUN4.

VTT : Il s'agit de la spécification en types abstraits d'un routeur de messages. Cet exemple a été étudié au sein des projets ESPRIT SPECS et METEOR.

Le code C produit par CÆSAR.ADT a permis de vérifier certaines propriétés de bon fonctionnement du routeur.

FWC : Il s'agit des spécifications en LOTOS des calculateurs FWC (*Flight Warning Computer*) embarqués à bord des Airbus A340. Deux spécifications LOTOS ont été développées par la société Aérospatiale, correspondant aux cartes CPU1 et CPU2 du calculateur embarqué FWC.

Le code C produit a été utilisé — dans le cadre de l'environnement ouvert OPEN/CÆSAR — pour simuler le fonctionnement du calculateur FWC.

Le tableau ci-dessous donne, pour chaque exemple, le nombre de lignes du programme LOTOS (L_L), la taille du programme LOTOS (T_L, en kilo-octets), le nombre de types (N_T), le nombre de sortes (N_S), le nombre de constructeurs (N_C), le nombre de non-constructeurs (N_F) et le nombre d'équations (N_E). Les chiffres entre crochets indiquent, s'il y a lieu, le nombre de sortes et de non-constructeurs implémentés en externe.

application	L_L	T_L	N_T	N_S	N_C	N_F	N_E
MAA	1126	34.178	10	11 [1]	16	255 [71]	167
VTT	1130	39.444	17	16	39	111	345
FWC	13524	580.119	145	142 [3]	515	847 [61]	1432

Le tableau ci-dessous donne, pour chaque exemple, le temps mis par CÆSAR.ADT pour compiler le programme LOTOS sur une station SUN4, (t, en secondes), le nombre de lignes de code C engendrées (L_C), la taille du code C produit (T_C, en kilo-octets) ainsi que la taille du code objet obtenu en compilant ce code C sur SUN4 (T_O, en kilo-octets) :

application	t	L_C	T_C	T_O
MAA	9.1	2041	110.878	40.960
VTT	7.7	3202	161.159	57.940
FWC	79	21502	1074.158	320.468

Conclusion

Cet article a présenté les principes du nouveau compilateur CÆSAR.ADT qui permet d'engendrer du code exécutable à partir des spécifications en types abstraits algébriques LOTOS. Il est donc possible d'exécuter efficacement des spécifications algébriques, ce qui ouvre un large champ d'applications en matière de prototypage.

A l'heure actuelle, le compilateur CÆSAR.ADT a été diffusé dans plus 60 sites. Il est gracieusement mis à la disposition des universités et centres de recherche publics. Il peut être obtenu sur demande à l'adresse électronique `caesar@imag.fr`.

Remerciements

Les auteurs tiennent à remercier Florence Baudin, Christophe Diot, Suzanne Graf, Joseph Sifakis et les rapporteurs anonymes pour leurs remarques et suggestions concernant cet article.

Références

[Aug85] L. Augustsson. Compiling pattern matching, 1985.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to Algebraic Specifications Based on the Language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[EW92] Henk Eertink and Dietmar Wolz. Symbolic Execution of LOTOS Specifications. In Michel Diaz and Roland Groz, editors, *Proceedings of the 5th International Conference on Formal Description Techniques FORTE'92 (Lannion, France)*, pages 289–304. IFIP, octobre 1992.

[FGM+92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A Toolbox for the Verification of LOTOS Programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, mai 1992.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, décembre 1989.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, juin 1990.

[Kap87] S. Kaplan. A Compiler for Conditional Term Rewriting Systems. In *Proceedings of the 1st International Conference on Rewriting Techniques and Applications (Bordeaux, France)*, volume 256 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

[Kar92] Günter Karjoth. Generating Transition Graphs from LOTOS Specifications. In Michel Diaz and Roland Groz, editors, *Proceedings of the 5th International Conference on Formal Description Techniques FORTE'92 (Lannion, France)*, pages 275–288. IFIP, octobre 1992.

[MdM88] J. A. Manas and T. de Miguel. From LOTOS to C. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 79–84. North-Holland, septembre 1988.

[Mun91] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, septembre 1991.

[Pet92] Mikael Pettersson. A Term Pattern-Matching Compiler Inspired by Finite Automata Theory. In U. Kastens and P. Pfahler, editors, *Proceedings of the 4th International Conference on Compiler Construction, CC'92 (Paderborn, FRG)*, volume 641 of *Lecture Notes in Computer Science*, pages 258–270. Springer Verlag, octobre 1992.

[PS93] Laurence Puel and Ascander Suarez. Optimal solutions to pattern matching problems. In *TAPSOFT'93 — Proceedings of the 4th International Joint Conference CAAP/FASE (Orsay, France)*, volume 668 of *Lecture Notes in Computer Science*, pages 501–518. Springer Verlag, avril 1993.

[Sch88] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming*, 11:133–159, 1988.

[Tur92] Philippe Turlier. La compilation des types abstraits algébriques du langage LOTOS. Mémoire d'ingénieur CNAM, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, décembre 1992.

[Wad87] Philip Wadler. *Efficient Compilation of Pattern-Matching*. In Simon L. Peyton-Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5, pages 78–103. Prentice-Hall, 1987.

[WB89] Dietmar Wolz and Paul Boehm. Compilation of LOTOS Data Type Specifications. In Ed Brinksma, Guiseppe Scollo, and Chris Vissers, editors, *Proceedings of the 9th IFIP International Workshop on Protocol Specification, Testing and Verification (Enschede, The Netherlands)*. IFIP, 1989.

[Wir90] Martin Wirsing. *Algebraic Specification*. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, Formal Models and Semantics, chapter 13, pages 675–788. Elsevier Science Publishers, 1990.

Hubert Garavel, ingénieur ENSIMAG, docteur en informatique, a consacré sa thèse à la compilation et à la vérification des programmes LOTOS, travail qui lui a valu le prix IBM du Jeune Chercheur en Informatique en 1990. Depuis 1989 il fait partie du groupe d'Etudes Avancées de VERILOG et, depuis 1993, de l'Unité Mixte de Recherche VERIMAG.

Philippe Turlier a obtenu en 1992 le diplôme d'ingénieur du Conservatoire National des Arts et Métiers (CNAM) pour son travail sur la compilation des types abstraits algébriques du langage LOTOS. Il travaille actuellement dans la société MAATEL.