Réécriture de programmes C-- en équations logiques

Olivier Ponsini

Laboratoire I3S - UNSA - CNRS
2000, route des Lucioles, B.P. 121
O6 903 Sophia Antipolis, FRANCE
ponsini@i3s.unice.fr

RÉSUMÉ. *Cet article présente un système pour transformer, de façon automatique, des programmes écrits en C--, un langage impératif simple, en un ensemble d'équations du premier ordre. Cet ensemble d'équations utilisé pour représenter un programme C-- a une signification mathématique précise et les techniques standards de mécanisation du raisonnement équationnel peuvent être déployées pour vérifier des propriétés des programmes. Le travail présenté montre que des programmes impératifs simples peuvent être vus comme des systèmes logiques totalement formalisés, dans lesquels il est possible de prouver des théorèmes. Le système lui-même est exprimé abstraitement par un ensemble de règles de réécriture du premier ordre.*

ABSTRACT. *This paper describes a system for automatically transforming programs written in a simple imperative language (called C--), into a set of first-order equations. This means that a set of first-order equations used to represent a C-- program already has a precise mathematical meaning; moreover, the standard techniques for mechanizing equational reasoning can be used for verifying properties of programs. This work shows that simple imperative programs can be seen as fully formalized logical systems, within which theorems can be proved. The system itself is formulated abstractly as a set of first-order rewrite rules.*

MOTS-CLÉS : *vérification de programme, réécriture, sémantique opérationnelle et équationnelle*

KEYWORDS: *Program Verification, Rewriting, Operational and Equational Semantics*

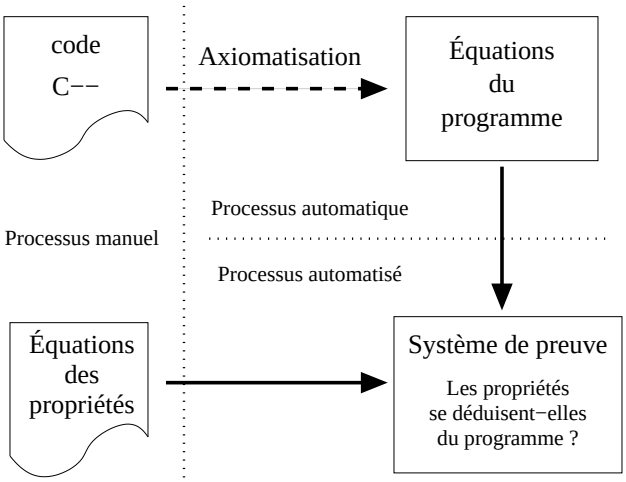

Figure 1. *Le processus de preuve.*

1. Introduction

La nécessité de pouvoir raisonner de façon rigoureuse et formelle sur les programmes informatiques est évidente. Afin d'accroître le niveau de confiance dans les programmes réalisés, il est important de s'efforcer de vérifier que les programmes sont valides vis-à-vis de leur spécification. Dans un article précédent, [FÉD 99], FÉDÈLE et KOUNALIS ont introduit un cadre théorique pour prouver de façon automatique des propriétés de programmes écrits en C--, un langage impératif simple. L'idée est de traduire le code source du programme en un ensemble d'équations du premier ordre exprimant la sémantique algébrique du programme. L'utilisation de la *logique équationnelle* a plusieurs avantages par rapport à d'autres logiques plus complexes :

– elle est très simple — logique de substitution de termes égaux ;

– de nombreux problèmes associés aux équations sont décidables dans cette logique ;

– il existe des algorithmes efficaces pour résoudre la plupart de ces problèmes.

Le schéma général de notre cadre de preuve est présenté dans la figure 1. Les utilisateurs écrivent le code C-- d'un programme ; ils donnent aussi la spécification du programme sous la forme d'un ensemble de propriétés exprimées en logique équationnelle. Le code source est alors transformé automatiquement par notre système (SOS C--) en un ensemble d'équations. Les équations du programme peuvent être vues comme des axiomes, et les propriétés à prouver comme des conjectures. Par conséquent, prouver que ces conjectures sont des théorèmes à partir des axiomes équivaut à prouver que le programme respecte sa spécification. Cette dernière partie est effectuée automatiquement en utilisant des *prouveurs de théorèmes* capables d'induction

mathématique comme NICE [KOU 99], ou interactivement en utilisant des *vérificateurs de preuve* comme COQ [BAR 97].

Cette approche est conceptuellement différente d'autres développements récents comme COGITO [TRA 97] ou SPECWARE [JUE 96], puisque ces systèmes génèrent du code à partir de spécifications. Elle diffère aussi de [FIL 98] et [ANT 94] puisque ces systèmes ont recours à des *annotations* pour prouver le comportement du programme. Notre approche est similaire à celle des systèmes [WAR 93] et [SCH 97]. Cependant, [SCH 97] ne gère pas les programmes contenant des boucles et [WAR 93] emploie une logique qui n'est pas aisément automatisable.

Dans cet article, nous étendons [FÉD 99] dans plusieurs directions :

1) Nous donnons un cadre abstrait au processus de traduction du programme en équations. Dans [FÉD 99], il n'apparaissait que les bases d'un système de réécriture et aucune propriété du système n'avait été prouvée.

- En particulier, nous formulons une grande partie du système SOS C-- comme un système de réécriture. Nous avons aussi revu les règles pour enrichir la sémantique du C--.

- Nous prouvons la *convergence* du système de réécriture (*i.e.* terminaison et confluence) en utilisant le système RRL[1]. De façon informelle, la convergence assure que tout programme C-- peut-être transformé en un unique programme équationnel.

- Nous donnons une description formelle de la génération des équations à partir des environnements.

2) Dans [FÉD 99], aucune implémentation n'avait été faite. Nous affirmons ici que le processus d'axiomatisation peut être automatisé puisque nous avons réalisé une implémentation en Java.

- JavaCC[2], un générateur d'analyseur, a été utilisé pour l'étape de génération des termes.

- Nous avons développé une version Java d'un algorithme de réécriture générique. Les règles de réécritures sont lues dans un fichier afin de faciliter l'élaboration des règles.

- Nous avons défini un algorithme pour générer les équations à partir des environnements.

Dans ce qui suit, nous donnons d'abord quelques définitions et notations utiles pour la compréhension de cet article. Ensuite, la section 3 décrit le système SOS C-- et illustre son fonctionnement sur un exemple.

---

1. Rewrite Rule Laboratory [KAP 89]. La preuve est dans la version complète de cet article.
2. Java Compiler Compiler, Metamata.

2. Définitions

2.1. *Systèmes de réécriture*

Nous supposons des connaissances en logique équationnelle et en système de réécriture (*cf.* [BAA 98] par exemple). Soit F un ensemble fini de symboles de fonction appelé *vocabulaire* et X un ensemble dénombrable de *variables*, $T(F, X)$ désigne l'ensemble des termes construit à partir des symboles de F et des variables de X. Si t est un terme et θ est une *substitution* de termes pour des variables dans t, alors $t\theta$ est une *instance* de t. Une *équation* e est un élément de $T(F, X) \times T(F, X)$ et s'écrit $t = s$. Un *système de réécriture* $\mathcal{R}$ est un ensemble d'équations orientées $l \rightarrow r$, appelées *règles de réécriture*. Une règle est appliquée à un terme t en cherchant un sous terme s de t qui soit une instance de la partie gauche l de la règle (*i.e.* $s = l\theta$) et en remplaçant s par l'instance correspondante de la partie droite de la règle ($r\theta$). Les calculs avec $\mathcal{R}$ se font par l'application répétée des règles pour réécrire (ou réduire) un terme donné en entrée jusqu'à obtenir une *forme normale* (un terme irréductible). Soit A un ensemble d'équations, dans l'hypothèse où A peut être ramené à un système de réécriture $\mathcal{R}$ *convergent* (*i.e. terminant* et *confluent*), on peut décider la validité de $t =_A s$ en regardant si les $\mathcal{R}$-formes normales de t et s sont identiques (*i.e.* $\mathrm{nf}(t) \stackrel{?}{=} \mathrm{nf}(s)$, où $\mathrm{nf}(t)$ (resp. $\mathrm{nf}(s)$) désigne la forme normale de t (resp. s)).

2.2. *Le langage C--*

Pour l'élaboration de notre système, nous utilisons un langage impératif très simple. La syntaxe du C-- est proche de celle du C. Les principales caractéristiques du langage sont :

– l'affectation ;

– les instructions de contrôle : *if . . . else*, *while* et *return* ;

– deux types prédéfinis : les entiers (*int*), et les listes (*list*) ;

– les opérateurs arithmétiques usuels ;

– des opérateurs sur les listes : *getHead* qui retourne le premier élément d'une liste, *getQueue* qui retourne la copie d'une liste privée de son premier élément, *NULL* qui représente une liste vide, et *cons* qui construit une liste en ajoutant un élément en tête d'une autre liste.

Cependant, plusieurs caractéristiques communes dans les langages impératifs ne sont pas disponibles en C-- : pas de types utilisateurs ; pas de variables globales ; pas de pointeurs directement accessibles — des pointeurs sont utilisés pour implémenter le type *list* mais ils ne sont pas visibles pour le programmeur.

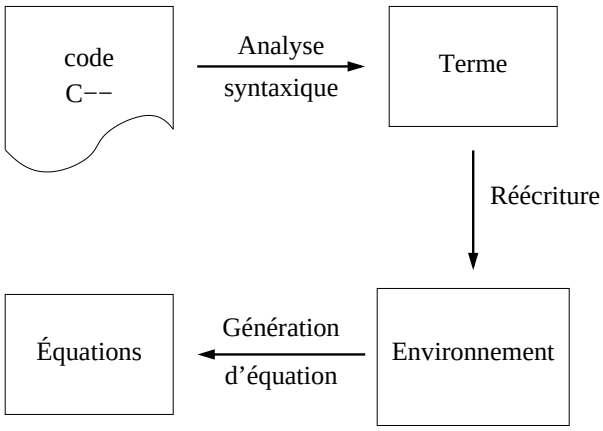

Figure 2. *Axiomatisation.*

3. Description du système SOS C--

Le système SOS C--, pour la majeure partie, peut être représenté comme un système de réécriture $\mathcal{R}$ sur un langage du premier ordre $\mathcal{L}$ construit à partir d'un ensemble de symboles de fonction. Ces symboles sont la traduction des constructions du langage source (le C--). Par exemple, l'instruction d'*affectation*, écrite $x = y$ en C--, se traduit par Assign(x, y), où *Assign* appartient au vocabulaire de $\mathcal{L}$.

Le système prend en entrée un *programme C--* et retourne en sortie un *ensemble d'équations* sémantiquement équivalent au programme : le résultat de l'exécution du programme C-- avec l'entrée E est identique au résultat (*i.e.* théorème) de la déduction équationnelle débutée avec la même entrée E. Le processus de transformation, appelé *axiomatisation*, est réalisé en trois étapes sans intervention de l'utilisateur. La figure 2 montre les étapes nécessaires à l'axiomatisation.

Pour illustrer ces étapes, nous utiliserons le *programme de tri* du listing 1. Ce programme est une version C-- du tri par insertion. La fonction *ins* prend un entier e et une liste triée L et retourne une nouvelle liste triée qui est une copie de L contenant e. *ISort* prend une liste L en argument et retourne une copie de L triée en insérant à la bonne position (appel à la fonction *ins*) le premier élément de L dans la queue de L déjà triée.

3.1. *Les Programmes sont des Termes*

La première étape consiste en l'analyse des fonctions du programme source P. Le résultat de cette analyse syntaxique est une liste de terme T_P^f sur $\mathcal{L}$; un terme par fonction f de P. Intuitivement, un terme est équivalent à une fonction du code source et forme une entrée convenable pour la réécriture.

Listing 1: Programme de tri.

```
list  ins(int e, list L) {
list  ret=NULL;
if (L==NULL) ret=cons(e, NULL);
  else  if (e<=getHead(L)) ret=cons(e, L);
    else  { ret=ins(e, getQueue(L));
            ret=cons(getHead(L), ret ); }
return ret ; }

list  ISort( list L) {
list  ret=NULL;
if (L==NULL) ret=NULL;
  else  ret=ins(getHead(L), ISort(getQueue(L)));
return ret ; }
```

Chaque *fonction* C-- est vue comme une liste d'instructions et un environnement initial assemblés dans un terme *GE*[3]. L'environnement initial est composé des paramètres formels de la fonction combinés dans un terme *Pair* avec des termes *EP*[4] figurant la valeur donnée à un paramètre quand la fonction est appelée. Un terme *EP* désigne donc un paramètre effectif. Ainsi, les paramètres formels se comportent comme des variables locales d'une fonction auxquelles sont affectés les paramètres effectifs.

Les *déclarations de variable* sont considérées comme des affectations et font donc partie de la liste des instructions de la fonction.

Une *séquence* d'instruction est représentée par une liste de termes.

L'instruction *return* engendre une paire liant le nom de la fonction et son résultat.

Les *expressions* apparaissant en partie droite des affectations ou dans un appel de fonction sont laissées telles quelles.

Les autres constructions du langage C-- sont simplement associées aux termes équivalents du système de réécriture : l'instruction *if*, if(c) $s1$ else $s2$, est traduite en If$(c, s1, s2)$; l'instruction *while*, $while(c)$ s, est traduite en While$(while_num, c, s)$.

Par exemple, le terme qui correspond à la fonction *ISort* du *programme de tri* est :

GE({ Assign(*ret*, *NULL*),
  If($L = NULL$,
    {Assign(*ret*, *NULL*)},
    {Assign(*ret*, *ins*(*getHead*(L),

3. *GE* signifie Generate Environment.
4. *EP*(x) signifie paramètre effectif de x.

ISort(*getQueue*(L))))}),
Return(*ISort*(L), *ret*) }, { Pair(L, EP(L)) }) .

Un terme *GE* contient deux éléments. Le premier représente la séquence d'instructions de la fonction source :

– la déclaration de *ret* donne le premier terme *Assign* ;

– ensuite vient le terme *If* avec la condition et ses deux listes d'instruction, une pour chaque alternative ;

– et finalement le terme *return*.

Le second représente l'environnement initial de la fonction. C'est une liste de terme *Pair*. Un terme *Pair* associe une variable et une valeur. Ainsi, l'environnement initial contient les valeurs des variables apparaissant dans la fonction après l'appel mais avant qu'aucune instruction ne soit évaluée.

De même, le terme qui correspond à la fonction *ins* est :

GE({ Assign(*ret*, *NULL*) ,
If($L = NULL$,
{Assign(*ret*, *cons*(e, *NULL*))},
{If($e \leq getHead(L)$,
{Assign(*ret*, *cons*(e, L))},
{Assign(*ret*, *ins*(e, *getQueue*(L))),
Assign(*ret*, *cons*(*getHead*(L), *ret*))}) }
Return(*ins*(e, L), *ret*) },
{ Pair(L, EP(L)) , Pair(e, EP(e)) }) .

3.2. *Les Termes se réécrivent en Environnements*

À la seconde étape, le système transforme un terme T_P^f en un environnement $E_{T_P^f}$. Intuitivement, cet environnement contient toutes les informations sur les variables de la fonction f et l'expression qui leur correspond (la valeur de f lors d'une exécution de P résulte de l'évaluation de ces expressions).

3.2.1. *Description*

Pour obtenir cet environnement, chaque terme T_P^f est normalisé selon les règles de réécriture[5] de $\mathcal{R}$. Les règles sont divisées en deux catégories. La première contient les règles qui définissent la sémantique opérationnelle du langage C--. La deuxième catégorie contient les règles qui mettent à jour les environnements — principalement des règles pour manipuler des listes.

Les instructions des fonction sont exécutées suivant un ordre qui dépend des instructions de contrôle et des conditions qui leur sont associées. Ces différents ordres

5. L'ensemble des règles est disponible dans la version complète de cet article.

possibles constituent les chemins d'exécution d'une fonction. L'environnement produit par réécriture représente ces différents chemins ainsi que les conditions associées à chacun et l'expression finale des variables et de la fonction. L'état des variables est représenté par une liste de terme *Pair*. Un terme *Branch* associe une condition à un état des variables. Les chemins sont contenus dans un terme *Choice*.

Par exemple, l'environnement de la fonction *ISort* du listing 1 est :

Choice(
  [Branch($L = NULL$,
    { Pair(L, L), Pair(*ret*, *NULL*),
      Pair(*ISort*(L), *NULL*) })],
  [Branch($L \neq NULL$,
    { Pair(L, L), Pair(*ret*, *ins*(*getHead*(L),
                                ISort(*getQueue*(L)))),
      Pair(*ISort*(L), *ins*(*getHead*(L),
                                ISort(*getQueue*(L)))) })]) .

La fonction *ISort* contient une instruction *if*, donc nous trouvons dans l'environnement un terme *Choice* composé des deux alternatives : les deux termes *Branch*. Ces deux termes partitionnent les instructions de la fonction entre celles qui sont exécutées quand la condition $L = NULL$ est vraie et celles qui sont exécutées quand la même condition est fausse. Dans chaque terme *Branch* nous trouvons une liste de termes *Pair* qui représentent l'état des variables à la fin d'une exécution "abstraite" de la fonction *ISort*. Par exemple, dans le cas où $L = NULL$, Pair(L, L) signifie que L n'est pas modifiée par la fonction ; Pair(*ret*, *NULL*) signifie que la valeur de la variable *ret* est *NULL* ; le terme *Pair* contenant le nom de la fonction, Pair(*ISort*(L), *NULL*) signifie que le résultat de la fonction est *NULL*.

De même, l'environnement de la fonction *ins* est :

Choice(
  [Branch($L = NULL$,
    { Pair(L, L), Pair(e, e), Pair(*ret*, *cons*(e, *NULL*)),
      Pair(*ins*(e, L), *cons*(e, *NULL*)) })],
  [Choice(
    [Branch($L \neq NULL$ and $e \leq getHead(L)$,
      { Pair(L, L), Pair(e, e), Pair(*ret*, *cons*(e, L)),
        Pair(*ins*(e, L), *cons*(e, L)) })],
    [Branch($L \neq NULL$ and $e > getHead(L)$,
      { Pair(L, L), Pair(e, e),
        Pair(*ret*, *cons*(*getHead*(L), *ins*(e, *getQueue*(L)))),
        Pair(*ins*(e, L), *cons*(*getHead*(L),
                                ins(e, *getQueue*(L)))) })])]) .

3.2.2. *L'instruction while*

Nous abordons maintenant la construction itérative *while*. Cette section donne un aperçu de la façon dont sont traitées les instructions *while*. Les sections 3.2.3 et 3.3 formalisent les idées présentées ici.

La sémantique des instructions *while* est un peu particulière. En effet, une boucle est considérée comme une famille de fonctions récursives. À chaque boucle sont associés des paramètres et un corps. L'idée poursuivie est que chaque boucle est une fonction qui s'appelle récursivement avec en paramètre la valeur des variables modifiées dans la boucle selon les instructions du corps de la boucle. Une telle fonction est définie pour chaque variable modifiée dans le corps de la boucle, formant ainsi une famille de fonctions. Par conséquent, dans une fonction où apparaît une boucle, la valeur d'une variable modifiée par la boucle est le résultat d'un appel à la fonction correspondant à la variable parmi la famille de fonctions définissant la boucle. Voici comment est modifié l'environnement lorsqu'une boucle est rencontrée :

– Un terme LT[6] est créé, il contient toutes les informations nécessaires à la génération de la famille de fonctions de la boucle. On y trouve un numéro identifiant la boucle, la condition de sortie, les instructions du corps de la boucle et une liste des variables visibles par la boucle. La génération de la famille de fonctions aura effectivement lieu lors de la troisième étape (*cf.* section 3.3).

– La valeur de chaque variable modifiée dans le corps d'une boucle devient le résultat d'un appel à la fonction de boucle correspondante. Cette fonction reçoit en argument l'état courant des variables dans la fonction appelante.

L'exemple suivant montre comment nous traitons les instructions *while*. Nous supposons qu'une fonction C-- déclare trois variables — x, y, z — parmi lesquelles deux sont modifiées dans une boucle conformément au listing 2. Au cours de la réécriture

Listing 2: Une boucle.

```
int f () {
int x,y,z;

x=1; y=2; z=3;
while(y>0) {
  x=x+z;
  y=y−1; }
... }
```

du terme correspondant à la fonction f, le terme LT suivant est créé :

$$\mathrm{LT}(1,\ y > 0,\ \mathrm{GE}(\mathit{instructions},\ \mathit{environnement\ initial}),\ (x, y, z))\ .$$

6. Loop Term.

Les *instructions* sont les deux affectations modifiant x et y. L'*environnement initial* est la liste de paires $(x, \text{EP}(x))$, $(y, \text{EP}(y))$ et $(z, \text{EP}(z))$. Le terme GE signifie qu'un nouvel environnement sera évalué pour le corps de la boucle. À la troisième étape du processus, ceci conduira à la définition de deux fonctions, LOOP_x^1 et LOOP_y^1, une pour chaque variable modifiée dans la boucle.

$$\begin{cases} \text{If } y > 0 \text{ then} \\ \quad \text{LOOP}_x^1(x, y, z) = \text{LOOP}_x^1(x + z, y - 1, z) \\ \text{If not}(y > 0) \text{ then} \\ \quad \text{LOOP}_x^1(x, y, z) = x \end{cases}$$

$$\begin{cases} \text{If } y > 0 \text{ then} \\ \quad \text{LOOP}_y^1(x, y, z) = \text{LOOP}_y^1(x + z, y - 1, z) \\ \text{If not}(y > 0) \text{ then} \\ \quad \text{LOOP}_y^1(x, y, z) = y \end{cases}$$

De plus, les paires

$$\text{Pair}(x, \text{Loop}_x^1(\mathit{état\ courant\ des\ variables}))$$

et

$$\text{Pair}(y, \text{Loop}_y^1(\mathit{état\ courant\ des\ variables})),$$

sont insérées dans l'environnement de la fonction f pour refléter le nouvel état de ces variables. L'*état courant des variables* fait référence à l'état des variables juste avant l'instruction *while*. Cela revient à remplacer la boucle dans la fonction f par les appels de fonction : $x = \text{LOOP}_x^1(1, 2, 3)$ et $y = \text{LOOP}_y^1(1, 2, 3)$.

3.2.3. *Règles du système de réécriture*

Les règles sont divisées en deux catégories. Parmi les règles décrivant la sémantique opérationnelle du langage, nous trouvons :

– les règles GE : elles traduisent le comportement de l'instruction de *séquence* ;

– les règles $Comp$: elles évaluent une nouvelle instruction dans l'environnement courant ;

- Règles pour l'*affectation* et l'instruction *return*. Ces règles ajoutent une nouvelle paire *Pair*(variable, valeur) à l'environnement ou modifient une paire existante.

- Règles pour l'instruction *if*. Ces règles ont pour but de diviser l'environnement en deux parties à travers un terme $Choice$. Chaque partie est incluse dans un terme $Branch$ et contient une alternative du *if* selon que sa condition est valide ou non.

- Règles pour l'instruction *while*. Ces règles créent un terme LT et ajoutent une nouvelle paire à l'environnement pour chaque variable modifiée dans le corps de la boucle.

- Deux autres règles $Comp$ expriment que les instructions suivant une instruction *if* doivent être exécutées quelle que soit l'alternative choisie ;

– les règles $Branch$ regroupent deux instructions *if* successives en fusionnant leur condition.

Parmi les règles de mise à jour de l'environnement, nous trouvons par exemple :

– $Merge\_env$ et $Merge\_L\_var$ pour fusionner deux listes ;

– $Insert\_pair$ et $Insert\_var$ pour ajouter une paire ou une variable à une liste.

3.3. *Les Environnements génèrent des Équations*

Au cours de la troisième étape, un ensemble d'équations est généré à partir de chaque environnement $E_{T_P^f}$. Cet ensemble d'équations définit la sémantique algébrique de P. Dans l'exemple du *programme de tri*, nous obtenons :

$$
\begin{array}{l}
L = NULL \Rightarrow \text{ISort}(L) = NULL \\
L \neq NULL \Rightarrow \text{ISort}(L) = \text{ins}(\text{getHead}(L), \text{ISort}(\text{getQueue}(L))) \\
L = NULL \Rightarrow \text{ins}(e, L) = \text{cons}(e, NULL) \\
L \neq NULL \text{ and } e \leq \text{getHead}(L) \Rightarrow \text{ins}(e, L) = \text{cons}(e, L) \\
L \neq NULL \text{ and } e > \text{getHead}(L) \Rightarrow \\
\qquad\qquad \text{ins}(e, L) = \text{cons}(\text{getHead}(L), \text{ins}(e, \text{getQueue}(L)))
\end{array}
$$

Seuls certains éléments dans un environnement vont générer des équations : ce sont les *générateurs d'équation*. La troisième et dernière étape de l'axiomatisation affine l'environnement, extrait les générateurs d'équation et génère les équations.

Les générateurs d'équation sont :

– les *listes de paires*. Elles représentent l'état des variables à la fin d'une exécution. Mais seule la valeur de retour de la fonction est utile, donc, seule la paire contenant le nom et la valeur de la fonction générera une équation.

$$
\begin{array}{ll}
\texttt{Générateur} & : \\
\quad \text{Pair}(\dots) \cdot \dots \cdot \text{Pair}(nom\_fonction, expression) & \\
\texttt{Équation} & : \\
\qquad nom\_fonction = expression &
\end{array}
$$

– les termes *Branch*. Ils apparaissent à la suite d'une instruction *if* et représentent une alternative. Ils lient une condition et une liste de paires. De nouveau, seule la paire avec le nom de la fonction est utile pour la génération d'équation. Chaque terme $Branch$ génère une équation conditionnelle.

$$
\begin{array}{ll}
\texttt{Générateur} & : \\
\quad \text{Branch}(condition, \text{Pair}(\dots) \cdot \dots \cdot \text{Pair}(nom\_fonction, exp)) & \\
\texttt{Équation} & : \\
\qquad condition = True \Rightarrow nom\_fonction = exp &
\end{array}
$$

– les termes *LT*. Ils génèrent une famille d'équations conditionnelles qui définissent des boucles sous la forme de fonctions récursives — une fonction de boucle par variable modifiée dans le corps de la boucle. Deux équations sont nécessaires :

- une pour l'appel récursif. L'état des variables est modifié en fonction des instructions du corps de la boucle.

- une pour le cas d'arrêt. Cette équation donne le résultat de la fonction de boucle, c'est-à-dire, la valeur courante de la variable modifiée considérée.

`Générateur` :
$\mathrm{LT}(num, cond, \mathrm{Pair}(v_1, e_1) \cdot \ldots \cdot \mathrm{Pair}(v_n, e_n), \{v_1, \ldots, v_n\})$
`Équation` :

$$\bigcup_{1 \leq i \leq m} \left\{ \begin{array}{l} cond = True \Rightarrow \\ \quad \mathrm{LOOP}^{num}_{v_{\mathrm{mod}_i}}(v_1, \ldots, v_n) = \mathrm{LOOP}^{num}_{v_{\mathrm{mod}_i}}(e_1, \ldots, e_n)\ , \\ cond = False \Rightarrow \\ \quad \mathrm{LOOP}^{num}_{v_{\mathrm{mod}_i}}(v_1, \ldots, v_n) = v_{\mathrm{mod}_i} \end{array} \right\}$$

Ici, $v_{\mathrm{mod}_1}, \ldots, v_{\mathrm{mod}_m}$ sont les variables modifiées dans le corps de la boucle et $v_1, \ldots, v_n$ sont toutes les variables apparaissant dans la fonction C--. Pour savoir si une variable v a été modifiée, on compare sa valeur à $\mathrm{EP}(v)$ qui est la valeur qui lui est assignée avant d'entrer dans la boucle.

4. Conclusion

Dans cet article nous avons présenté un système qui permet d'obtenir automatiquement une formulation équationnelle d'un programme C-- à partir de son code source. Le processus conduisant aux équations requiert trois étapes.

L'axe central de la méthode exposée est la génération d'un environnement au moyen d'un système de réécriture qui décrit la sématique opérationnelle du langage C--.

La première étape consiste à construire, grâce à l'analyse syntaxique du programme source, un terme apte à être réécrit.

La dernière étape consiste à traduire les environnements en équations.

Une implémentation de ce système a été réalisée en Java. Il reste encore des tâches à accomplir, parmi lesquelles :

– Ajouter des fonctionnalités au langage C-- afin de le rendre plus proche des langages impératifs réels. En effet, malgré l'existence de constructions telles que l'affectation et les boucles *while*, la programmation en C-- est très proche du style fonctionnel.

– Implémenter des interfaces vers les systèmes de preuve, c'est-à-dire, fournir les équations dans le format requis par le système de preuve.

– Expérimenter de façon plus approfondie la preuve de propriétés à partir des équations délivrées par notre système. Ceci afin d'identifier des classes de propriétés et de programmes qui peuvent être prouvées correctes grâce à l'utilisation de notre système.

NOTE. — Une version augmentée de cet article ainsi qu'une implémentation du système SOS C-- sont disponibles à l'adresse suivante : http ://www.i3s.unice.fr/˜ponsini.

5. Bibliographie

[ANT 94] ANTOY S., GANNON J., « Using Term Rewriting to Verify Software », *IEEE Transactions on Software Engineering*, vol. 20, n° 4, 1994, p. 259-274.

[BAA 98] BAADER F., NIPKOW T., *Term Rewriting and All That*, (Cambridge : Cambridge University Press), 1998.

[BAR 97] BARRAS B., BOUTIN S., CORNES C., COURANT J., FILLIÂTRE J., GIMÉNEZ E., HERBELIN H., HUET G., NOZ C. M., MURTHY C., PARENT C., PAULIN C., SAÏBI A., WERNER B., « The Coq Proof Assistant Reference Manual – Version V6.1 », rapport n° 0203, août 1997, INRIA.

[FÉD 99] FÉDÈLE C., KOUNALIS E., « Automatic Proofs of Properties of Simple C-- Modules », *14th IEEE Conf. on Automated Software Engineering*, Cocoa Beach, USA, octobre 1999, IEEE Computer Society Press, p. 283-286.

[FIL 98] FILLIÂTRE J.-C., « Proof of Imperative Programs in Type Theory », *2nd Workshop on Types for Proofs and Programs*, vol. 1657 de *Lecture Notes in Computer Science*, Kloster Irsee, Germany, mars 1998, Springer-Verlag, p. 78-92.

[JUE 96] JUELLIG R., SRINIVAS Y., LIU J., « SPECWARE : An Advanced Environment for the Formal Development of Complex Software Systems », *5th Conf. on Algebraic Methodology and Software Technology*, Munich, Germany, 1996, page 551.

[KAP 89] KAPUR D., ZHANG H., « RRL : Rewrite Rule Laboratory », mai 1989.

[KOU 99] KOUNALIS E., URSO P., « Generalization Discovery for Proofs by Induction in Conditional Theories », *12th FLAIRS Conf.*, Orlando, USA, 1999, AAAI Press, p. 250-256.

[SCH 97] SCHWEIZER D., DENZLER C., « Verifying the Specification–to–Code Correspondence for Abstract Data Types », *6th Conf. on Dependable Computing for Critical Applications*, vol. 11 de *Dependable Computing and Fault-Tolerant Systems*, Garmisch-Partenkirchen, Germany, 1997, IEEE Computer Society Press.

[TRA 97] TRAYNOR O., HAZEL D., KEARNEY P., MARTIN A., NICKSON R., WILDMAN L., « The Cogito development system », *6th Conf. on Algebraic Methodology and Software Technology*, vol. 1349 de *LNCS*, Sidney, Australia, décembre 1997, Springer-Verlag, p. 586-591.

[WAR 93] WARD M., « Abstracting a Specification from Code », *Journal of Software Maintenance : Research and Practice*, vol. 5, n° 2, 1993, p. 101-122.

Annexe I : Recherche dans un arbre binaire

Voici le code et les équations d'un programme C-- qui cherche un élément e dans un arbre binaire de recherche. Les équations sont obtenues par le système SOS C--.

Un nœud dans l'arbre est une liste composée d'un entier et de deux autres nœuds pour les fils gauche et droit (*e.g.* (*i fils-gauche fils-droit*)).

Listing 3: Recherche dans un arbre de consultation.

```
int  root( list  tree ) { return getHead(tree ); }
list  lc( list  tree ) { return getHead(getQueue(tree )); }
list  rc( list  tree ) { return getHead(getQueue(getQueue(tree))); }

int  bs( int  e , list  tree ) {
  int  ret ;
  if  ( tree ==null ) ret =0;
    else  if  ( e==root( tree )) ret =1;
      else  if (e<root( tree )) ret =bs(e, lc( tree ));
        else  ret =bs(e, rc( tree ));
  return  ret ; }
```

$$
\begin{aligned}
&BS(e,\ \emptyset) = 0\\
&BS(e,\ e \cdot lc \cdot rc) = 1\\
&e < r \Rightarrow BS(e,\ r \cdot lc \cdot rc) = BS(e,\ lc)\\
&e > r \Rightarrow BS(e,\ r \cdot lc \cdot rc) = BS(e,\ rc)
\end{aligned}
$$

Annexe II : Parcours en profondeur

Voici le code et les équations d'un programme C-- qui visite tous les nœuds d'un graphe en effectuant un parcours en profondeur. Les équations sont obtenues par le système SOS C--. Les nœuds du graphe sont des entiers. Le graphe est représenté par une liste de listes d'adjacence $Ladj$. Le $i^{ème}$ élément de $Ladj$ est la liste de tous les nœuds adjacents au nœud i.

Listing 4: Parcours en profondeur.

```
int  member(int s , list  L) {
  int  ret ;
  if  ( L==NULL) ret=0;
    else
      if  ( s==getHead(L))
        ret =1;
      else
        ret =member(s, getQueue(L));
  return  ret ; }

list  adj ( int  s , list  Ladj ) {
  list  ret ;
  if  ( s==1)
    ret =getHead(Ladj);
```

```
  else
    ret=adj(s−1, getQueue(Ladj));
  return ret ; }

int dfs( list sommets, list visites , list Ladj) {
  int ret , s;
  if (sommets==NULL) ret=1;
    else { s=getHead(sommets);
           ret=depth(sommets, adj(s , Ladj ), cons(s , visites ), Ladj ); }
  return ret ; }

int depth( list sommets, list adjacents , list visites , list Ladj) {
  int ret , s;
  if ( adjacents==NULL) ret=dfs(getQueue(sommets), visites , Ladj );
    else { s=getHead(adjacents);
           if (member(s, visites )==1)
             ret=depth(sommets, getQueue(adjacents ), visites , Ladj );
           else
             ret=dfs(cons(s , sommets), visites , Ladj ); }
  return ret ; }
```

$$
\begin{aligned}
&member(s,\ \emptyset) = 0\\
&s \neq c \Rightarrow member(s,\ c \cdot L) = member(s,\ L)\\
&member(s,\ s \cdot L) = 1\\
&adj(1,\ L_1 \cdot L) = L_1\\
&adj(n+1,\ L_1 \cdot L_2 \cdot L) = adj(n,\ L_2 \cdot L)\\
&DFS(\emptyset,\ M,\ A) = 1\\
&DFS(s \cdot L,\ M,\ A) = depth(s \cdot L,\ adj(s,\ A),\ s \cdot M,\ A)\\
&depth(s \cdot L,\ \emptyset,\ M,\ A) = DFS(L,\ M,\ A)\\
&member(c,\ M) = 1 \Rightarrow\\
&\quad depth(L,\ c \cdot L_1,\ M,\ A) = depth(L,\ L_1,\ M,\ A)\\
&member(c,\ M) = 0 \Rightarrow\\
&\quad depth(L,\ c \cdot L_1,\ M,\ A) = DFS(c \cdot L_1,\ M,\ A)
\end{aligned}
$$