Université de Nice-Sophia Antipolis **UFR Sciences**

École doctorale STIC
Sciences et Technologies de l'Information et de la Communication

Des programmes impératifs vers la logique équationnelle pour la vérification

THÈSE

présentée et soutenue publiquement le 24 novembre 2005

pour obtenir le titre de

Docteur ès Sciences
de l'Université de Nice-Sophia Antipolis
spécialité informatique

par

Olivier Ponsini

Composition du jury

Président :	M. Yves Bertot	Directeur de Recherche à l'INRIA Sophia Antipolis
Rapporteurs :	M. Pierre Marquis	Professeur à l'Université d'Artois
	M. Michaël Rusinowitch	Directeur de Recherche à l'INRIA Lorraine
Examinateur :	M. Jean-Paul Bodeveix	Professeur à l'Université Toulouse III Paul Sabatier
Directeurs :	M. Emmanuel Kounalis	Professeur à l'Université de Nice-Sophia Antipolis
	Mlle Carine Fédèle	Maître de conférences à l'Université de Nice-Sophia Antipolis

Laboratoire Informatique, Signaux et Systèmes de Sophia Antipolis

À mes parents

Résumé

L'omniprésence des systèmes informatiques dans les applications technologiques modernes rend capitale la question de leur fiabilité. Dans cette perspective, la correction des logiciels mis en œuvre est indispensable. Nous nous sommes intéressé dans cette thèse à la logique équationnelle en tant que support de la vérification des programmes écrits dans un langage impératif. Notre approche vise le double objectif d'automatiser la vérification des propriétés de programmes tout en proposant un formalisme pour raisonner sur les programmes adapté aux acteurs du développement des logiciels.

Précisément, les travaux de cette thèse portent sur la traduction automatique des programmes impératifs vers la logique équationnelle afin d'en vérifier la correction. Nous avons été amené à considérer deux classes de programmes impératifs.

Dans la première classe de programmes, la seule instruction avec effet de bord du langage de programmation est l'affectation. Nous présentons un algorithme de traduction d'un programme de cette classe en un ensemble d'équations. Cet algorithme est donné sous la forme d'un système de réécriture définissant la sémantique du langage. Nous montrons la convergence de notre système de réécriture à l'aide d'un démonstrateur de théorèmes.

Pour définir la seconde classe de programmes, nous ajoutons au langage impératif une forme d'appel par référence ainsi qu'un type des listes mutables. Ces deux mécanismes introduisent la possibilité de manipuler des alias dans les programmes. Nous énonçons des restrictions sur la création et l'utilisation des alias moyennant lesquelles nous proposons un algorithme pour la traduction en équations des programmes de la nouvelle classe. La définition équationnelle ainsi obtenue ne s'appuie pas sur une modélisation explicite de la mémoire.

Les équations produites par la traduction d'un programme avec notre méthode peuvent alors être utilisées dans des systèmes de preuve afin de vérifier des propriétés du programme, elles-mêmes exprimées sous forme d'équations. Nous validons notre approche en implantant une version de la traduction pour chaque classe de programmes et en prouvant des propriétés de programmes non triviales à l'aide des équations produites par notre méthode.

Mots-clés: Méthodes formelles ; Vérification de programmes ; Logique équationnelle ; Sémantique algébrique ; Systèmes de réécriture.

Abstract

Omnipresence of computer systems in modern technological applications makes the question of their reliability essential. In this perspective, correctness of the involved software cannot be neglected. In this thesis, we investigate equational logic as a foundation for the verification of programs written in an imperative language. Our approach aims at automating the verification of program properties while offering a formalism suited to software developers for reasoning about programs.

Precisely, our work addresses the automatic translation of imperative programs into equational logic in order to verify their correctness. We studied two classes of imperative programs.

In the first class of programs, assignment is the sole programming language statement with side effect. We developed an algorithm to translate the programs in this class into a set of equations. This algorithm is expressed as a rewriting system defining the language's semantics. We showed, with the assistance of a theorem prover, the convergence of our rewriting system.

The second class of programs is defined by adding to the programming language a call-by-reference parameter passing mode and a type of mutable lists. These two mechanisms introduce the ability of handling aliases in programs. We define restrictions on the creation and use of aliases which allow us to propose an algorithm for the translation into equations of the programs in the new class. The equational definition we obtain does not involve an explicit model of the program's memory.

The equations produced by the translation of a program with our method can then be used in proof systems in order to verify properties of the program. The properties are also expressed as equations. We validate our approach by implementing the translation algorithms for each class of programs and by proving interesting program properties from the equations produced by our method.

Keywords: Formal methods ; Program verification ; Equational logic ; Algebraic semantics ; Rewriting system.

Table des matières

Table des figures

Préambule

Il a longtemps été fait référence à l'écriture de programmes pour ordinateur comme à un art, connotant ainsi que les programmes échapperaient en quelque sorte au raisonnement. Une des séries d'ouvrages informatiques parmi les plus célèbres ne s'intitule-t-elle pas *The Art of Computer Programming* [Knuth, 1968] ?

Lors de l'allocution qu'il prononça en recevant le prix Turing [Knuth, 1974], D. E. Knuth s'est longuement expliqué sur le choix de ce titre et sur l'apparent paradoxe qui consiste à conférer à la programmation des ordinateurs le statut d'art tout en prônant une approche rigoureuse. Sans entrer dans les détails d'une esthétique des programmes telle que la conçoit D. E. Knuth, nous pouvons toutefois constater qu'entre la deuxième partie des années 1930, quand les mathématiciens pionniers de l'informatique donnèrent une définition rigoureuse de ses fondements, et les premiers travaux établissant un cadre mathématique pour la preuve des programmes par R. W. Floyd (1967) et C. A. R. Hoare (1969) puis par E. W. Dijkstra (1976), il y eut une période durant laquelle le concept de *correction*[1] de programme n'avait tout simplement pas d'expression formelle. Dans les faits, la programmation apparaissait alors comme une habile manipulation de symboles dont le résultat était correct « par intuition ».

Depuis, malgré les progrès théoriques évidents dans la formalisation des programmes et de leur correction, la situation n'a que peu évolué en pratique et cette citation de D. E. Knuth, pourtant de 1974, conserve obstinément son actualité [Knuth, 1974] :

> « *The point is that when we write programs today, we know that we could in principle construct formal proofs of their correctness if we really wanted to, now that we understand how such proofs are formulated.* »

De nos jours, en dépit de l'existence de méthodes formelles, la validation des programmes par des jeux de *tests* constitue toujours la méthode prédominante. Afin de s'assurer de la correction de son programme, le développeur fournit en entrée du programme un ensemble de données sélectionnées pour être à la fois *représentatives* des cas d'usage normal et limite du programme. Le programme est alors tenu pour correct si les réponses qu'il produit pour ces entrées sont celles attendues. La validation d'un programme par des tests soulève les difficiles problèmes du choix des données pour les tests et de leur génération automatique. Mais surtout, elle fait face à une impossibilité théorique que E. W. Dijkstra (1972) énonçait ainsi :

[1] Correction : *qualité de ce qui est correct, juste* (définition du dictionnaire Le Robert) ; *conformité à un modèle, à un ensemble de principes ou de règles* (définition du dictionnaire Trésor de la Langue Française : `http://atilf.atilf.fr`).

> « *Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.* »

Effectivement, le domaine d'un programme, c'est-à-dire l'ensemble des valeurs valides en entrée du programme, est le plus souvent infini ou tellement grand qu'il est intraitable en pratique. Par conséquent, un jeu de tests, aussi bien conçu qu'il soit, ne constitue jamais une *preuve* de la correction d'un programme.

Pourtant, les exemples en faveur d'une vérification plus formelle de la correction des programmes abondent. Ce sont d'une part les échecs retentissants dus à des programmes informatiques défectueux : l'explosion en plein vol de la fusée Ariane 5 pour une conversion de types erronée [Jézéquel et Meyer, 1997], l'algorithme de division erroné du processeur Pentium [Pratt, 1995], le système de réservation de la SNCF désorganisé par un programme mal dimensionné [Campagnolle, 2004], etc. Ces exemples, et malheureusement quantité d'autres parfois tragiques, sont régulièrement relatés dans la presse et recensés par des listes de diffusion spécialisées [RISKS, 2005].

D'autre part, plusieurs projets de grande envergure ont été de considérables succès, aussi bien au niveau de la qualité des logiciels que des coûts de développement, grâce notamment à la prise en considération de la correction formelle des logiciels mis en œuvre. Citons par exemple la réussite du système de contrôle de la vitesse des trains de la ligne A du RER parisien, grâce à la spécification formelle de l'ensemble du système et à la preuve mathématique de la correction des portions critiques du système [Guiho et Hennebert, 1990] ; la découverte, par une technique formelle de vérification de modèle, d'une erreur dans un système de contrôle actif des structures qui, au lieu de protéger les bâtiments lors de tremblements de terre, les aurait rendus plus sensibles aux vibrations [Elseaidy *et al.*, 1997] ; la vérification d'un algorithme de calcul de la racine carrée pour un processeur AMD, mécanisée à l'aide d'un système de preuve formel [Russinoff, 1999] ; d'autres exemples sont rapportés dans [Clarke et Wing, 1996].

Dans notre société, les systèmes informatiques envahissent tous les champs de la vie quotidienne : transport, habitat, communication, santé, etc. Comme le montrent les exemples précédents, des fonctions *critiques* sont aujourd'hui confiées à des logiciels dont la défaillance entraînerait des pertes humaines ou des coûts matériels et financiers énormes. Ce constat plaide donc pour un accroissement significatif de la fiabilité des logiciels. Un critère indispensable à la réussite de cet objectif est la vérification de la correction des programmes. La correction d'un programme n'est pas une notion universelle et aisément définie. Elle est usuellement employée de façon restrictive pour signifier l'absence d'erreur syntaxique. Cette dernière n'est pourtant que préalable à la correction, dans son sens plein, qui dépend de la signification du programme, ce qu'il est supposé faire : sa *spécification*. *Un programme est correct s'il satisfait sa spécification.*

Les méthodes formelles permettent d'établir une preuve mathématique qu'un programme est correct vis-à-vis de sa spécification pour *toutes* les entrées valides du programme. Elles comportent généralement un langage de spécification, un langage de description du programme et une méthode pour vérifier que la description satisfait la spécification. Lorsque l'ensemble de ces éléments est formalisé, le processus de vérification peut être automatisé, partiellement ou totalement selon la méthode et la classe de programmes

à laquelle elle s'applique. La spécification formelle *complète* d'un système informatique est rarement réalisable : la description initiale d'un problème réel est par nature informelle, comment s'assurer dans ces conditions de la justesse de la spécification formelle ? Pour autant, cet argument ne doit pas nous faire renoncer à notre légitime ambition de fiabilité. Le processus de spécification formelle d'un problème réel est vertueux dans le sens où il améliore la compréhension du problème et est le seul à admettre par la suite un traitement rigoureux.

Les méthodes formelles ont aujourd'hui atteint un degré de maturité qui les rend économiquement viables et, partant, attractives pour les industriels. Elles restent cependant encore d'un abord ardu et requièrent un investissement important pour le commun des programmeurs qui les considère comme une ingrate perte de temps. De ce fait, leur mise en œuvre est souvent confiée à des spécialistes.

Contributions de cette thèse

Dans cette thèse, nous proposons une méthode pour aider à raisonner sur les programmes impératifs et faciliter la preuve de propriétés de programmes. Notre contribution principale est la conception et le développement d'un système, appelé $SOS\mathrm{Sub}_{\mathcal{C}}$, permettant d'exprimer la sémantique des programmes dans la logique équationnelle.

L'idée centrale de cette thèse consiste à traduire automatiquement le code source d'un programme impératif en un ensemble d'équations. Dans cette perspective, la vérification qu'un programme P satisfait une propriété générique π comporte trois étapes :

1. la formulation de la propriété π dans la logique du premier ordre ;
2. la traduction *automatique* par notre système du programme P en un ensemble d'équations E_P, qui expriment la sémantique algébrique de P ;
3. la preuve, généralement par l'application d'un principe de récurrence, que la propriété π est une conséquence des équations de E_P.

Un des traits saillants de notre approche est la *richesse du langage de programmation source* (*e.g.*, structures de contrôle, données structurées, allocation dynamique, effets de bord, etc.) qui contraste avec la *simplicité du langage équationnel cible*. La logique équationnelle constitue une base mathématique traditionnellement enseignée qui pourrait ainsi prétendre à être le langage commun entre tous les acteurs d'un développement logiciel. En outre, la syntaxe concise et la sémantique claire de la logique équationnelle recèlent des principes de raisonnement puissants. À notre connaissance, cette thèse semble être la première à traiter des effets de bord dans la logique équationnelle *sans expliciter un modèle de la mémoire*.

Une deuxième caractéristique de notre méthode est son *degré d'automatisation* : une large classe de programmes est automatiquement traduisible en équations par notre système. De plus, le choix de la logique équationnelle a pour avantage de présenter la sémantique des programmes sous une forme commode pour l'automatisation des traitements postérieurs. En effet, de nombreux outils et algorithmes facilitent le raisonnement dans la logique équationnelle.

Notre système, $SOS\mathrm{Sub}_{\mathcal{C}}$, se décline en deux versions :

***SOS*Sub$^{\mathcal{R}}_{\mathcal{C}}$** Dans cette première version du système, la sémantique du langage impératif source, le langage $\mathrm{Sub}_{\mathcal{C}}$, est formalisée par les règles d'un *système de réécriture* dont nous avons prouvé la propriété de convergence. Cette propriété assure que le processus de réécriture des programmes termine et que son résultat est unique. Ainsi, la propriété de convergence garantit que *notre système définit pour tout programme $Sub_{\mathcal{C}}$ une, et une seule, sémantique algébrique du programme.* En outre, la démonstration de cette propriété a été menée automatiquement avec un démonstrateur de théorèmes.

***SOS*Sub$^{ext}_{\mathcal{C}}$** Dans cette seconde version du système, nous munissons le langage $\mathrm{Sub}_{\mathcal{C}}$ de constructions qui permettent de créer des alias et d'effectuer des effets de bord. Le caractère indécidable du traitement automatique de la plupart des problèmes ayant trait aux alias nous a conduits à définir une classe de programmes pour laquelle notre approche reste automatique. L'appartenance d'un programme à cette classe se décide, hors de notre système, en vérifiant que le programme satisfait un ensemble de *prérequis*.

Plan de la thèse

Le manuscrit débute par un chapitre qui offre une vue d'ensemble de la thèse. Nous y exposons la problématique et nos motivations à travers des exemples, ainsi que les travaux qui constituent notre contribution.

La première partie est consacrée, d'une part, à l'examen des directions de recherche explorées dans le domaine de la vérification de programme, et d'autre part, à la présentation des concepts et des notions qui seront utiles à la lecture du manuscrit. Nous présentons notamment la logique équationnelle et les systèmes de réécriture.

La deuxième partie traite de notre méthode proprement dite. Le premier chapitre aborde la version $SOS\mathrm{Sub}^{\mathcal{R}}_{\mathcal{C}}$ de notre système. Nous introduisons le langage $\mathrm{Sub}_{\mathcal{C}}$ et les principes de notre approche, puis nous revenons en détail sur la sémantique du langage et son traitement dans le système. Nous concluons le chapitre en prouvant la convergence du système de réécriture au cœur de $SOS\mathrm{Sub}^{\mathcal{R}}_{\mathcal{C}}$. Le deuxième chapitre de cette partie traite de la version $SOS\mathrm{Sub}^{ext}_{\mathcal{C}}$ de notre système. Cette version du système est construite sur $\mathrm{Sub}^{ext}_{\mathcal{C}}$ une extension de $\mathrm{Sub}_{\mathcal{C}}$ qui procure de nouvelles possibilités d'effet de bord. Nous commençons le chapitre par une discussion des conséquences sur le système de ces nouveaux effets de bord. Nous décrivons ensuite la syntaxe et la sémantique des constructions nouvelles, ainsi que les répercussions sur les autres éléments du système et sur l'approche elle-même.

La troisième partie rapporte une série d'expérimentations qui ont été menées avec les deux versions de notre système. Nous y développons notamment les exemples qui ont servi à exposer les motivations de notre approche lors du tout premier chapitre.

Dans la conclusion, nous faisons une synthèse des travaux accomplis ainsi que leur critique. Enfin, nous discutons brièvement des perspectives de recherche que cette thèse suscite.

Aperçu global de la thèse

L'objet de ce chapitre est d'introduire concrètement, par des exemples simples mais non triviaux, les motivations et les contributions des travaux dont ce mémoire rend compte. Le détail des solutions mises en œuvre se trouve dans la Partie II.

Nous abordons dans la Section 1.1 la problématique de la preuve de propriétés de programmes par deux cas pratiques. La Section 1.2 présente le système $SOS\mathrm{Sub}_{\mathcal{C}}$ qui entreprend de répondre, pour une classe réduite de programmes, aux questions ainsi soulevées. Nous mettons alors en évidence dans la Section 1.3 l'apport du système $SOS\mathrm{Sub}_{\mathcal{C}}$ sur les deux exemples introductifs de ce chapitre.

1.1 La problématique : exemples motivants

Si, à des degrés très divers, les acteurs du monde du logiciel s'accordent majoritairement à reconnaître l'intérêt de spécifier et de vérifier les programmes qu'ils développent, la question de la méthode reste ouverte. Nous illustrons le problème de la vérification de propriétés des programmes sur deux exemples :

- le premier est tiré du domaine de l'arithmétique et s'implante simplement dans tout langage impératif ;
- le second provient du domaine des tris en informatique, fécond en algorithmes, et utilise des constructions plus avancées des langages impératifs.

À travers ces exemples, notre intention est de donner l'intuition de notre approche en faisant percevoir quels sont les avantages de raisonner formellement sur les programmes aussi naturellement que nous avons l'habitude de le faire en mathématiques.

1.1.1 Plus Grand Commun Diviseur

Considérons l'exemple du calcul du Plus Grand Commun Diviseur (pgcd). Le pgcd de deux entiers naturels non tous deux nuls est défini comme le plus grand entier qui divise les deux nombres. Un entier d *divise* un entier n si le reste de la division entière de n par d est nul.

La Figure 1.1 contient une proposition de programme pour calculer le pgcd de deux nombres. Ce fragment de programme est écrit dans un langage de programmation impératif, `C` en l'occurrence. Il comporte une unique fonction `pgcd` qui prend en paramètre deux entiers, `a` et `b`, et renvoie la valeur de $\mathrm{pgcd}(a, b)$.

```c
int pgcd(int a, int b)
{
  int r;

  while(b != 0)
  {
    r = b;
    b = a % b;
    a = r;
  }
  return a;
}
```

FIG. 1.1 – Un calcul du Plus Grand Commun Diviseur en `C` : `pgcd`. *Le symbole* `%` *est l'opérateur modulo du* `C` *:* `a%b` *est le reste de la division entière de* `a` *par* `b` (*i.e.*, $a \bmod b$).

Quelle garantie avons-nous que ce texte, une fois exécuté sur un ordinateur, calcule effectivement $\mathrm{pgcd}(a, b)$? Quand bien même nous connaîtrions l'*algorithme d'Euclide*[2] dont ce programme est une variante, nous pourrions néanmoins nous poser la question de la validité du programme : ne serait-ce que pour des valeurs négatives ou nulles de `a` ou `b`.

D'autre part, un mathématicien saura dire qu'une propriété fondamentale du pgcd est la suivante[3] :

Propriété 1.1. *Pour a et b entiers naturels et $a > b$,* $\mathrm{pgcd}(a, b) = \mathrm{pgcd}(a - b, b)$.

Cependant, comment prouver que le programme `pgcd` vérifie cette propriété ?

Deux approches s'offrent à nous :

- Pour nous convaincre, nous pouvons effectuer des *tests* et vérifier que pour certaines valeurs de `a` et `b` le résultat obtenu est bien celui attendu. Cette solution, bien qu'étant la plus usitée, n'est pas satisfaisante. En effet, cette façon de procéder montre l'adéquation du programme proposé avec le calcul souhaité pour les seules valeurs de `a` et `b` qui auront été essayées, mais aucunement pour toutes les autres valeurs. Or, l'espace des valeurs possibles, les entiers, est infini : cette méthode s'avère impraticable pour démontrer la correction du programme.

- Une autre démarche consiste à *raisonner* sur le texte du programme lui-même avec une rigueur équivalente à celle des mathématiques. Pour atteindre cet objectif, il faut donner un sens formel aux programmes, c'est-à-dire, une signification symbolique et non ambiguë avec laquelle des démonstrations pourront être construites.

L'idée que nous préconisons dans cette thèse est de traduire les programmes dans la logique équationnelle. Plus précisément, nous proposons de traduire le programme

[2] Cet algorithme consiste à soustraire itérativement le plus petit nombre au plus grand jusqu'à ce qu'ils soient égaux, cette dernière valeur est le pgcd.

[3] L'algorithme d'Euclide repose d'ailleurs sur cette propriété.

```
typedef struct CELL *LIST;
struct CELL {
 int element;
 LIST next;
};
```

FIG. 1.2 – Une implantation des listes simplement chaînées en C

pgcd en un ensemble d'équations. Nous disposons alors de tout « l'arsenal des mathématiques » pour prouver des propriétés exprimées par des assertions logiques telles que la Propriété 1.1. De plus, cette tâche peut être accomplie manuellement ou à l'aide d'outils logiciels de preuve.

L'étape essentielle dans la réalisation de ce projet réside dans la traduction du texte du programme vers sa formulation mathématique. L'ambition du système $SOS\mathrm{Sub}_C$, présenté dans la Section 1.2, est d'effectuer automatiquement cette traduction. La traduction du programme pgcd et la preuve de la Propriété 1.1 sont discutées dans la Section 1.3.1. Mais avant cela, considérons un deuxième exemple de programme plus complexe.

1.1.2 Tri par fusion

Les *pointeurs* occupent une place importante en langage C. Ils permettent, entre autres, de passer par *adresse* des paramètres à un sous-programme ou de définir des *types de données récursifs*. L'exemple le plus simple et le plus représentatif des types de données récursifs est peut-être la liste simplement chaînée. Celle-ci consiste en une séquence de *maillons* qui portent chacun une information[4] et un *lien* vers le maillon suivant. Une implantation de cette structure de données en C est fournie par le type LIST de la Figure 1.2.

Employés à bon escient, les pointeurs conduisent à des programmes compacts et efficaces aussi bien en temps qu'en espace mémoire. Le programme de tri fusion MergeSort de la Figure 1.3, extrait du livre *Foundations of Computer Science* [Aho et Ullman, 1994, Section 2.8 (page 84)], en est une remarquable illustration. Une analyse informelle de ce programme révèle qu'il est composé de trois sous-programmes manipulant des listes d'entiers :

- Le sous-programme MergeSort trie la liste list passée en paramètre et renvoie le résultat du tri. L'algorithme est le suivant :
  - Si list est vide ou ne contient qu'un seul maillon, alors elle est déjà triée et elle est renvoyée.
  - Dans le cas contraire, la moitié des éléments sont retirés de list et placés dans la liste SecondList (appel du sous-programme split, ce qui implique potentiellement un effet de bord sur le paramètre de MergeSort). Les deux listes sont alors récursivement triées avant d'être fusionnées en une seule par merge. Cette dernière liste est renvoyée.

[4] Nous dirons aussi *élément* ; et par abus de langage, nous pourrons employer élément pour maillon.

```
LIST merge(LIST list1, LIST list2);
LIST split(LIST list);
LIST MergeSort(LIST list);

LIST MergeSort(LIST list)
{
  LIST SecondList;

  if (list == NULL) return NULL;
  else if (list->next == NULL) return list;
  else {
    SecondList = split(list);
    return merge(MergeSort(list), MergeSort(SecondList));
  }
}

LIST merge(LIST list1, LIST list2)
{
  if (list1 == NULL) return list2;
  else if (list2 == NULL) return list1;
  else if (list1->element <= list2->element) {
    list1->next = merge(list1->next, list2);
    return list1;
  }
  else {
    list2->next = merge(list1, list2->next);
    return list2;
  }
}

LIST split(LIST list)
{
  LIST pSecondCell;

  if (list == NULL) return NULL;
  else if (list->next == NULL) return NULL;
  else {
    pSecondCell = list->next;
    list->next = pSecondCell->next;
    pSecondCell->next = split(pSecondCell->next);
    return pSecondCell;
  }
}
```

FIG. 1.3 – Le programme de tri `MergeSort` en `C` [Aho et Ullman, 1994]

Dans les deux cas, la liste renvoyée est constituée exactement des maillons de `list` réordonnés.

– Le rôle du sous-programme `split` est de retirer un élément sur deux de la liste qui lui est passée en paramètre (elle a donc un effet de bord) et de les regrouper dans une seconde liste qui est renvoyée.

Exemple 1.2. Supposons que la liste `list` contienne les éléments 1, 2, 3, 4 et 5 dans cet ordre. L'appel du sous-programme `split` retourne une liste contenant les éléments 2 et 4 dans cet ordre. Dans le même temps, `list` a été modifiée et ne contient plus que les éléments 1, 3 et 5 dans cet ordre. ♦

– Le sous-programme `merge` prend deux listes triées et arrange leurs éléments en une liste triée, qui est renvoyée. Les deux listes passées en paramètre sont modifiées au cours du traitement, ce qui est un effet de bord du sous-programme.

Selon le même schéma que pour le programme du calcul du pgcd, la preuve de la correction du programme `MergeSort` nécessite en premier lieu d'établir sa spécification. La vérification d'un programme de tri de listes requiert la démonstration de deux propriétés :

– il faut bien évidemment que la liste résultat soit triée, c'est-à-dire que les éléments qui la composent soient ordonnés ;

– mais il faut aussi s'assurer que la liste résultat comporte les mêmes éléments que la liste initiale et en même nombre.

Ceci s'énonce dans le cas du `MergeSort` par les deux propriétés suivantes, les fonctions sorted et permutation étant définies par ailleurs (*cf.* Sections 6.3 et 7.1) :

Propriété 1.3. *Pour toute liste l d'entiers,* $\mathrm{sorted}(\mathrm{MergeSort}(l)) = \mathrm{vrai}$.

Propriété 1.4. *Pour toute liste l d'entiers,* $\mathrm{permutation}(l, \mathrm{MergeSort}(l)) = \mathrm{vrai}$.

Le caractère informel de l'analyse esquissée précédemment ne permet de décider sur la validité d'aucune propriété. Nous rencontrons la même difficulté à raisonner sur les programmes qu'avec l'exemple du calcul du pgcd de la section précédente. De surcroît, la sémantique des constructions du langage de programmation auxquelles nous nous intéressons est plus complexe. Nous tirerions donc encore davantage profit de la possibilité de raisonner formellement sur les programmes et de déléguer à la machine autant que possible de cette tâche hautement sujette à erreur.

La section qui suit présente le système $SOS\mathrm{Sub}_{\mathcal{C}}$ qui nous permettra de traduire le programme du tri par fusion en équations et de prouver sa correction dans la Section 1.3.2.

1.2 Notre contribution : le système *SOS*Sub$_{\mathcal{C}}$

Nous avons introduit dans la section précédente la problématique de la vérification de propriétés des programmes. Le système *SOS*Sub$_{\mathcal{C}}$, qui est l'objet des travaux présentés dans ce mémoire, répond au besoin de raisonner sur les programmes impératifs de façon formelle en se fondant sur la *logique équationnelle* (*cf.* Section 3.1 pour une définition). Dans cette logique, la description formelle des programmes et des spécifications des programmes est formulée par des *équations conditionnelles*. La vérification d'un programme consiste alors à démontrer, au moyen des règles d'inférence de la logique, que les équations des spécifications du programme se déduisent des équations du programme.

Le système *SOS*Sub$_{\mathcal{C}}$ s'intègre dans un atelier où les programmes sont développés dans un langage impératif, le langage Sub$_{\mathcal{C}}$, avant d'être *automatiquement* traduits en équations pour que leur correction soit formellement prouvée à l'aide de systèmes de preuve.

Les trois sections qui suivent donnent un aperçu du système *SOS*Sub$_{\mathcal{C}}$, qui sera pleinement détaillé dans les prochains chapitres. Nous exposons tout d'abord les principes du système, puis nous présentons la version du système sur les constructions de base du langage Sub$_{\mathcal{C}}$, enfin, nous abordons la version du système étendu à des constructions plus complexes.

1.2.1 Principes

La nécessité de pouvoir raisonner sur les programmes conduit à attribuer un sens formel aux différentes constructions du langage de programmation (*e.g.*, `while`, `if`, etc.). Ceci est le rôle d'une *sémantique*, qui est un « codage symbolique » dans un système mathématique formel des *calculs* possibles dans le langage. Le système formel est habituellement une logique dans laquelle il sera possible de construire des démonstrations et de prouver des formules.

> *Le principe de notre approche est de produire automatiquement la sémantique équationnelle d'un programme à partir de son code source : c'est-à-dire, produire un ensemble d'équations qui constituent les axiomes de la théorie équationnelle du programme.*
> *À cette fin, nous décrivons la sémantique du langage impératif de programmation par les modifications que les différentes constructions du langage entraînent sur un ensemble d'équations caractéristiques de l'état d'un programme. L'évaluation symbolique d'un programme selon cette sémantique génère l'axiomatisation du programme. L'évaluation des programmes est « symbolique » au sens que les données en entrée des programmes sont manipulées comme des symboles dont la valeur n'est pas connue. Le résultat de l'évaluation d'un programme est par conséquent un ensemble d'équations dont les variables sont les entrées du programme.*
> *Ces équations pourront alors être utilisées pour prouver des propriétés de programmes.*

Traditionnellement, les sémantiques dénotationnelles rendent apparent un modèle de la mémoire (même lorsque ce modèle est abstrait comme dans le cas des sémantiques algébriques) et sont plus appropriées pour raisonner sur les langages que sur les programmes. De même, les sémantiques opérationnelles échouent à fournir un cadre bien adapté à la preuve de propriétés de programmes et servent plutôt dans le domaine de l'équivalence des programmes. En revanche, les sémantiques axiomatiques ont été conçues pour la preuve de propriétés de programmes mais donnent en général une sémantique des itérations peu naturelle.

Notre approche se distingue des travaux similaires par :

- son aptitude à être automatisée grâce au choix de la logique équationnelle pour exprimer la sémantique des programmes ;
- l'absence de modèle de la mémoire ;
- la distinction claire entre la sémantique du langage et celle des programmes ;
- les caractéristiques du langage impératif traité (boucles, mémoire dynamique, effets de bord, etc.).

Le système *SOS*Sub$_{\mathcal{C}}$ met en œuvre notre approche en réalisant l'*axiomatisation* des programmes écrits en Sub$_{\mathcal{C}}$: c'est-à-dire, la traduction automatique des programmes vers un ensemble d'équations qui définit leur sémantique. Plus précisément, l'axiomatisation comporte trois étapes articulées autour de l'évaluation symbolique des programmes :

1. *traduction des programmes en termes ;*
2. *évaluation symbolique des termes ;*
3. *formulation des équations.*

Traduction des programmes en termes

À chaque construction du langage Sub$_{\mathcal{C}}$ (*e.g.*, instruction, opérateur) correspond un symbole de fonction de la signature Σ_{sos} (*cf.* Section 4.2). L'ensemble des symboles de fonction de la signature contient aussi les symboles des variables, les symboles des littéraux et les symboles des fonctions qui peuvent être utilisés en Sub$_{\mathcal{C}}$.

Exemple 1.5. L'affectation du langage Sub$_{\mathcal{C}}$ est associée au symbole de fonction de la signature Σ_{sos} *C_Assign*. ♦

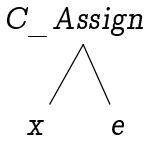
L'analyse lexico-syntaxique du code source d'un programme Sub$_{\mathcal{C}}$ produit l'*arbre de syntaxe abstraite* du programme :

- les nœuds de cet arbre sont essentiellement les symboles des instructions et des opérateurs du langage Sub$_{\mathcal{C}}$, ainsi que des fonctions définies dans le programme analysé ;

– les feuilles de cet arbre sont les noms des variables du programme analysé et les littéraux (les nombres entiers).

Ainsi, l'arbre de syntaxe abstraite du programme constitue la représentation arborescente d'un terme de la signature Σ_{sos}.

Exemple 1.6. Dans un programme $\text{Sub}_{\mathcal{C}}$, au cours de la première étape de l'axiomatisation, une affectation `x = e` produit l'arbre de syntaxe abstraite suivant :



Cet arbre correspond au terme *C_Assign(x, e)* où *x* et *e* sont des symboles de constante et *C_Assign* un symbole de fonction de la signature Σ_{sos}. ♦

Note 1.7. Par commodité d'écriture, dans les termes des programmes, une liste ne sera pas représentée à l'aide de ses deux constructeurs (liste vide et ajout en tête d'un élément) sous la forme d'un terme. Nous adopterons une notation où les listes sont délimitées par des accolades et leurs éléments séparés par un point. Ainsi, $\{\}$ dénote la liste vide et, par exemple, $\{e_1 \cdot e_2\}$ dénote une liste à deux éléments.

Cette étape est détaillée dans le Chapitre 4, et plus particulièrement dans la Section 4.2.2.

Évaluation symbolique des termes

La sémantique des constructions du langage $\text{Sub}_{\mathcal{C}}$ est exprimée par des modifications dans un ensemble d'équations qui constitue l'*environnement* d'un programme.

Les termes obtenus de la première étape sont évalués conformément à la sémantique du langage. L'évaluation est symbolique : nous représentons des *entrées arbitraires* des programmes par des symboles. Cette opération fait donc évoluer un environnement *initial* vers un environnement *final* qui décrit la sémantique du programme, c'est-à-dire, toutes les exécutions possibles du programme.

Cette étape est réalisée de deux manières différentes selon la version du système $SOS\text{Sub}_{\mathcal{C}}$:

***SOS*Sub$^{\mathcal{R}}_{\mathcal{C}}$** s'applique à une classe de programmes où les possibilités d'effet de bord sont limitées. La sémantique du langage $\text{Sub}_{\mathcal{C}}$ est exprimée formellement par un système de réécriture. (*cf.* Chapitre 4, et plus particulièrement Section 4.2). Cette formalisation nous permet de prouver certaines propriétés de l'axiomatisation (*cf.* Section 4.4).

***SOS*Sub$^{ext}_{\mathcal{C}}$** traite une classe de programmes plus vaste définie par $\text{Sub}^{ext}_{\mathcal{C}}$, une extension du langage $\text{Sub}_{\mathcal{C}}$. Afin de réaliser la traduction en équations des nouvelles constructions, nous proposons des algorithmes qui ne mettent plus en œuvre de système de réécriture (*cf.* Chapitre 5). Les propriétés prouvées pour $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$ ne sont donc pas directement transposables à $SOS\text{Sub}^{ext}_{\mathcal{C}}$.

Exemple 1.8. Dans un environnement, nous représentons l'équation $x = v$ par une paire *C_Pair(x, v)*. Un ensemble d'équations est alors une liste de paires. Ainsi, les valeurs des variables d'un programme forment une liste de paires associant le nom d'une variable à sa valeur.

Supposons par exemple qu'une variable `x` d'un programme ait pour valeur `v` dans l'environnement courant du programme (*i.e.*, l'environnement résultant de l'évaluation des instructions précédentes du programme), alors la liste de paires dans l'environnement aura la forme suivante :

$$\{\ldots \cdot \textit{C\_Pair(x, v)} \cdot \ldots\}$$

La sémantique d'une affectation *C_Assign(x, e)* est de remplacer dans l'environnement courant la paire concernant la variable `x` par la paire *C_Pair(x, e')*, où *e'* est l'expression *e* partiellement évaluée dans l'environnement courant. L'évaluation de cette affectation produira donc le changement d'environnement suivant :

$$\{\ldots \cdot \textit{C\_Pair(x, v)} \cdot \ldots\} \xrightarrow{\textit{C\_Assign(x, e)}} \{\ldots \cdot \textit{C\_Pair(x, e')} \cdot \ldots\} \quad \blacklozenge$$

Formulation des équations

Dans un environnement final, la valeur des variables est une expression algébrique définie sur les symboles des entrées du programme. Éventuellement, pour une variable donnée plusieurs valeurs sont définies et associées à des conditions. Ces conditions sont formulées par des relations sur les entrées du programme, elles proviennent des conditions des structures de contrôle rencontrées lors de l'évaluation du programme et qu'il n'a pas été possible de résoudre puisque l'évaluation est symbolique.

Parmi toutes les variables, seules les valeurs de certaines nous intéressent : la troisième étape de l'axiomatisation consiste alors à rechercher les valeurs pertinentes contenues dans un environnement et à les présenter sous forme équationnelle pour une utilisation dans un système de preuve (*cf.* Sections 4.2.4 et 5.4).

Exemple 1.9. Pour illustrer cette troisième étape, très simplement, si nous sommes intéressés par la valeur d'une variable `x`, nous recherchons dans l'environnement final la paire *C_Pair(x, v)* et nous écrivons l'équation $x = v$. $\blacklozenge$

Exemple 1.10 (somme des n premiers entiers). Nous exemplifions ici les trois étapes de l'axiomatisation avec $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$ de la fonction `somme(n)` qui calcule la somme des n premiers entiers naturels. Cette fonction est définie par le programme $\text{Sub}_{\mathcal{C}}$ de la Figure 1.4. Le programme consiste principalement en une boucle où la variable `n` décroît jusqu'à zéro, pendant que la variable `sum` sert d'accumulateur des valeurs successives de `n`.

Pour ce premier exemple, nous allons simplifier la notation et masquer certains détails de la méthode. L'exemple complet est traité dans la Section 4.3.3 (Exemple 4.10 en Page 76).

1. Lors de la première étape, les constructions du langage $\text{Sub}_{\mathcal{C}}$ sont traduites en termes. Par exemple, pour l'instruction `while`, nous construisons un terme associant la condition de la boucle à la liste d'instructions formant le corps de la boucle :

```
int somme(int n) {
 int sum = 0;

 while (n != 0) {
  sum = sum + n;
  n = n − 1;
 }

 return sum;
}
```

FIG. 1.4 – Un calcul de la somme des n premiers entiers en $\text{Sub}_{\mathcal{C}}$

$$C\_While(n \neq 0, \{C\_Assign(sum, (sum + n)) \cdot C\_Assign(n, (n - 1))\})$$

Pour l'instruction `return sum`, nous construisons un terme qui associe le nom de la fonction à l'expression de retour de la fonction, comme s'il s'agissait d'une affectation :

$$C\_Assign(somme, sum)$$

Au final, l'analyse lexico-syntaxique du code $\text{Sub}_{\mathcal{C}}$ de la fonction `somme` produit le terme $Tsomme$ suivant :

$$\begin{array}{l}\{C\_Assign(sum, 0) \cdot \\ \ C\_While(n \neq 0, \{C\_Assign(sum, (sum + n)) \cdot C\_Assign(n, (n - 1))\}) \cdot \\ \ C\_Assign(somme, sum)\}\end{array}$$

Dans notre contexte, l'*environnement initial* d'une fonction est une liste de paires qui associent :

- les paramètres *formels* aux paramètres *effectifs* de la fonction ;
- les variables locales à une valeur par défaut (zéro pour le type entier) s'il n'y a pas eu d'initialisation explicite.

Soit n_{eff} le paramètre effectif associé à `n`, l'environnement initial de `somme` est la liste de deux paires :

$$\{C\_Pair(sum, 0) \cdot C\_Pair(n, n_{eff})\}$$

2. Le terme $Tsomme$ est ensuite évalué symboliquement dans l'environnement initial selon les règles de la sémantique du langage $\text{Sub}_{\mathcal{C}}$. L'évaluation d'une liste de termes est l'évaluation de la queue de la liste dans l'environnement produit par l'évaluation de la tête de la liste.

Dans cet exemple, la tête de la liste est un terme qui correspond à une affectation (nous en avons vu la sémantique dans l'Exemple 1.8). L'évaluation de ce terme laisse l'environnement initial inchangé.

Ensuite, l'évaluation du terme de racine *C_While* ajoute dans l'environnement l'élément suivant (*cf.* Section 4.3.3) :

$$C\_While\_Closure(n \neq 0, \{C\_Pair(n, n-1) \cdot C\_Pair(sum, sum+n)\})$$

Lors de l'étape suivante, la boucle donnera lieu à la définition d'une nouvelle fonction. La valeur des variables est mise à jour pour tenir compte de cette nouvelle fonction, dénotée par un terme *C_Loop*. Par exemple pour la variable `n` :

$$C\_Pair(n, C\_Loop(n, \{C\_Pair(n, n_{\mathit{eff}}) \cdot C\_Pair(sum, 0)\}))$$

Pour finir, la sémantique de l'instruction `return` est d'ajouter dans l'environnement une paire associant le nom de la fonction à sa valeur de retour une fois mise à jour dans l'environnement courant. Ainsi, l'environnement final est le suivant :

$$\begin{array}{l}\{C\_While\_Closure(n \neq 0, \{C\_Pair(n, n-1) \cdot C\_Pair(sum, sum+n)\}) \cdot \\ \ C\_Pair(n, C\_Loop(n, \{C\_Pair(n, n_{\mathit{eff}}) \cdot C\_Pair(sum, 0)\})) \cdot \\ \ C\_Pair(sum, C\_Loop(sum, \{C\_Pair(n, n_{\mathit{eff}}) \cdot C\_Pair(sum, 0)\})) \cdot \\ \ C\_Pair(somme, C\_Loop(sum, \{C\_Pair(n, n_{\mathit{eff}}) \cdot C\_Pair(sum, 0)\}))\}\end{array}$$

3. L'environnement contient trois paires, cependant, lors de la troisième étape, nous ne nous intéressons qu'à celle qui porte la valeur de retour de la fonction `somme`. Cette valeur est un terme de racine *C_Loop* qui correspond à un appel à une fonction somme^l définie récursivement par deux équations à partir du terme de racine *C_While_Closure* :

$$n \neq 0 \Rightarrow \text{somme}^l(n, sum) = \text{somme}^l(n-1, sum+n) \tag{1.11}$$
$$n = 0 \Rightarrow \text{somme}^l(n, sum) = sum \tag{1.12}$$
$$\text{somme}(n) = \text{somme}^l(n, 0) \tag{1.13}$$

Ces équations expriment la sémantique du programme `somme` dans la logique équationnelle. Elles pourront servir à montrer des propriétés du programme. ♦

Nous avons développé en Java deux versions de $SOS\text{Sub}_{\mathcal{C}}$ qui diffèrent par la classe de programmes concernée et la méthode employée pour l'évaluation symbolique :

- $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$, pour le langage $\text{Sub}_{\mathcal{C}}$ avec une modélisation fonctionnelle des listes ;
- $SOS\text{Sub}_{\mathcal{C}}^{ext}$, pour le langage $\text{Sub}_{\mathcal{C}}^{ext}$ intégrant des listes mutables et des effets de bord.

Nous présentons dans les deux sections suivantes les particularités de chacune des versions du système.

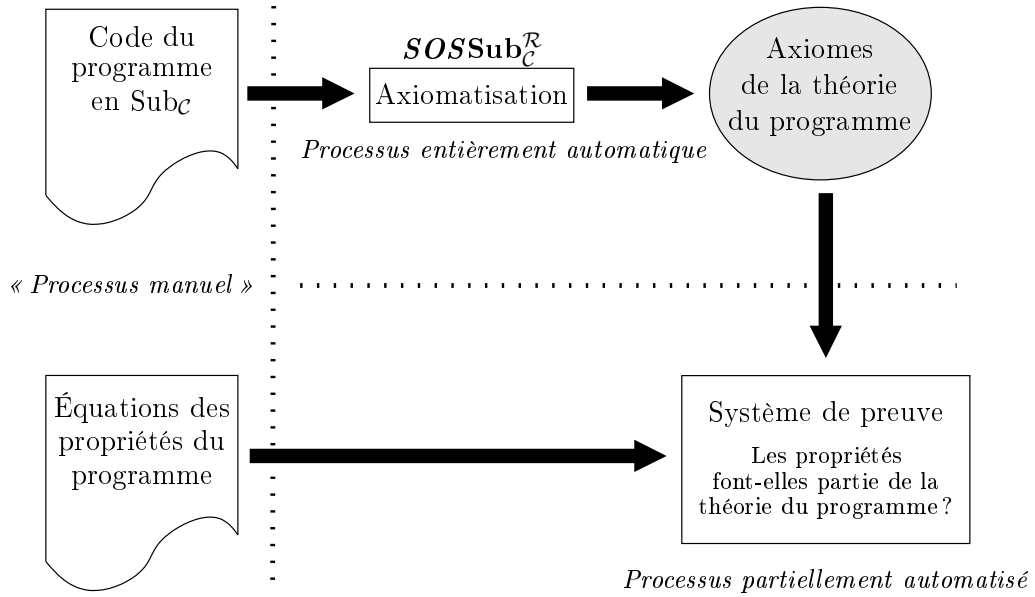


FIG. 1.5 – L'atelier de preuve avec $SOS\text{Sub}_C^R$

1.2.2 $SOS\text{Sub}_C^R$

Le langage de programmation impératif Sub_C se présente comme un sous-ensemble du langage C avec deux types primitifs de données : les *entiers* et les *listes*. Les données se manipulent à l'aide d'opérateurs prédéfinis pour chaque type et de fonctions définies par l'utilisateur. L'affectation et les structures de contrôle typiques des langages impératifs sont présentes.

Dans la version $SOS\text{Sub}_C^R$ du système $SOS\text{Sub}_C$, le mode de passage des paramètres est par valeur uniquement. De plus, le type liste ne suit pas la définition habituelle dans les langages impératifs de cette structure de données : une liste, bien qu'elle soit une séquence d'éléments, est manipulée dans le langage comme une unique valeur.

Comme énoncé dans la section précédente, le principe du système $SOS\text{Sub}_C$ est de « traduire » *automatiquement* le code source Sub_C vers un ensemble d'équations exprimant la sémantique du programme. Cette traduction, appelée *axiomatisation*, s'intègre dans un atelier pour prouver interactivement des propriétés de programmes. La description synoptique de notre atelier pour la version $SOS\text{Sub}_C^R$ est montrée Figure 1.5.

Les concepteurs d'une application écrivent les spécifications du programme comme un ensemble de propriétés désirées : les *équations des propriétés du programme*. Elles sont exprimées dans une logique cohérente avec la logique équationnelle, *e.g.*, la logique du premier ordre. Les développeurs produisent le code Sub_C du programme selon les indications des concepteurs. Ces deux étapes sont essentiellement l'œuvre d'acteurs humains. Ensuite, le système $SOS\text{Sub}_C^R$ traduit le code source en un ensemble d'équations. Ces équations constituent les axiomes d'une *théorie* pour la logique équationnelle : les *axiomes de la théorie du programme*. Enfin, lors de la dernière étape, les équations des propriétés deviennent des conjectures. La vérification de la correction du programme consiste alors à démontrer, dans la logique de la spécification, que ces conjectures appartiennent à la

théorie équationnelle du programme. *Si la démonstration est possible, alors il aura été prouvé que le programme respecte ses spécifications.*

Puisque les axiomes du programme sont exprimés dans la logique équationnelle, la preuve peut être faite au moyen d'un des nombreux systèmes de preuve à même de manipuler la logique équationnelle : automatiquement avec les *démonstrateurs de théorèmes* capables d'induction mathématique comme *RRL* [Kapur et Zhang, 1988] ou Spike [Bouhoula *et al.*, 1992], ou interactivement en utilisant des *assistants de preuve*[5] comme PVS [Owre *et al.*, 1992] ou Coq [Barras *et al.*, 1997] (*cf.* Section 2.4 pour des détails sur les systèmes de preuve).

L'axiomatisation est l'opération qui prend en entrée un programme $\text{Sub}_{\mathcal{C}}$ et produit en sortie un ensemble d'équations exprimant la sémantique du programme. Dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$, nous utilisons un système de réécriture comme le moyen formel de définir la sémantique du langage $\text{Sub}_{\mathcal{C}}$. Cette définition possède l'avantage supplémentaire d'être exécutable : la réécriture permet d'effectuer l'évaluation symbolique des programmes (*cf.* Section 3.2). Le système de réécriture que nous avons défini, nommé $SOS_{\mathcal{R}}$, figure en Annexe A et est décrit avec $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$ dans la Section 4.

1.2.3 $SOS\text{Sub}_{\mathcal{C}}^{ext}$

Dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$, le langage $\text{Sub}_{\mathcal{C}}$ ne capture pas certains des mécanismes propres au paradigme impératif, parmi les plus efficaces, tels que ceux qui sont à l'œuvre dans le programme `MergeSort` de la Section 1.1.2. En effet, les trois sous-programmes de `MergeSort` ne copient, ni ne créent, aucun maillon de liste (bien que l'algorithme ne soit pas constant en espace du fait de la pile des appels récursifs), elles se contentent de réarranger les liens entre les éléments de la liste à trier. Cette façon de procéder n'est pas réalisable avec $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$, elle est pourtant typique des langages impératifs et repose sur la notion d'effet de bord. Les objets manipulés par les programmes impératifs sont mutables : leur état peut changer au cours de l'exécution du programme.

L'approche proposée avec le système $SOS\text{Sub}_{\mathcal{C}}$ reste valide. Il convient cependant d'enrichir le langage $\text{Sub}_{\mathcal{C}}$ de nouvelles constructions semblables à celles qui sont employées dans le programme `MergeSort` et de leur donner une sémantique dans le cadre de la logique équationnelle.

Essentiellement, nous étendons le langage $\text{Sub}_{\mathcal{C}}$ en ajoutant les listes mutables et un mode de passage des paramètres par référence :

- Le type des listes est désormais semblable à la structure de liste simplement chaînée des langages impératifs et s'inspire de l'implantation en `C` de la Figure 1.2, à l'exception que le type de l'élément peut être n'importe quel type de données $\text{Sub}_{\mathcal{C}}$, et en particulier une liste. Nous ajoutons deux nouvelles constructions, `L->element` et `L->next` qui permettent de modifier, respectivement, le contenu et le suivant du maillon pointé par `L`.

[5] Cette distinction sur le degré d'automatisation entre démonstrateur de théorèmes et assistant de preuve peut paraître assez artificielle car les démonstrateurs de théorèmes sont souvent partiellement interactifs et les assistants de preuve partiellement automatisés.

– Nous enrichissons aussi le langage $\text{Sub}_{\mathcal{C}}$ de sous-programmes avec une forme d'appel par référence. Ce mode de passage des paramètres est dénoté par une perluète (caractère **&**) devant le nom du paramètre dans le prototype du sous-programme. Il autorise un sous-programme à modifier la valeur d'un paramètre.

Grâce à ces ajouts, le langage $\text{Sub}_{\mathcal{C}}^{ext}$ recouvre maintenant une plus large classe de programmes. Ainsi, nous pouvons écrire en $\text{Sub}_{\mathcal{C}}^{ext}$ le programme du tri fusion de la Figure 1.3 : le résultat, en Figure 1.6, s'obtient par un simple changement de syntaxe. Mais surtout, les algorithmes peuvent être exprimés aussi naturellement en $\text{Sub}_{\mathcal{C}}^{ext}$ que dans un langage impératif usuel comme le `C`, cependant que nous maintenons un contrôle sur l'utilisation des pointeurs. Par exemple, nous ne fournissons toujours pas de type pointeur générique, ni n'autorisons l'arithmétique de pointeur.

Néanmoins, même cet usage restreint des pointeurs soulève le problème difficile des *alias*. L'intérêt que nous portons aux alias provient de la nécessité de propager les effets de bord à tous les noms alias d'un même objet, *e.g.*, si `liste1` et `liste2` sont des alias, lorsque la valeur de `liste1` est modifiée, il faut aussi que `liste2` reflète la nouvelle valeur. Un prérequis à cet objectif est de déceler tous les alias. Malheureusement, ce problème est *indécidable* pour un langage tel que $\text{Sub}_{\mathcal{C}}^{ext}$ (*cf.* Section 5.1.1).

Pour nous, cela implique très concrètement l'impossibilité d'automatiser la traduction des programmes $\text{Sub}_{\mathcal{C}}^{ext}$ en équations dans le cas général. Nonobstant, nous énonçons un ensemble de restrictions sur l'emploi des constructions problématiques qui nous permet de proposer un algorithme pour l'axiomatisation (*cf.* Chapitre 5, et plus particulièrement Section 5.2.2 pour les restrictions). Cet algorithme ne repose plus sur la réécriture et se base sur un *schéma de nommage*, ou *dénomination systématique*, des zones de mémoire accessibles au programme qui donne lieu à une représentation « naturelle » des listes et des opérations sur les listes.

Le processus de preuve décrit par la Figure 1.5 doit être remanié pour inclure une étape supplémentaire qui consiste en la démonstration, non automatisée, que le programme respecte les règles de création et d'emploi des alias. Ce n'est qu'à cette condition que l'axiomatisation par $SOS\text{Sub}_{\mathcal{C}}^{ext}$ est valide. Le nouveau processus est illustré par la Figure 1.7.

1.3 $SOS\text{Sub}_{\mathcal{C}}$ *versus* les exemples motivants

Nous revenons dans cette section sur les deux exemples introductifs de la Section 1.1. Nous montrons comment le système $SOS\text{Sub}_{\mathcal{C}}$ donne une réponse aux questions que nous avions alors laissées en suspens.

1.3.1 Sur l'exemple du Plus Grand Commun Diviseur

Nous avons présenté dans la Section 1.1.1, en Figure 1.1, le programme `pgcd` supposé calculer le pgcd de deux entiers naturels. Nous avons discuté de la difficulté de raisonner sur le code source et de vérifier que le programme satisfait les propriétés du pgcd.

```
list split(list L);
list merge(list La, list Lb);

list mergeSort(list L)
{
 list secondList, La, Lb;

 if(L == NULL) return NULL;
 else if(next(L) == NULL) return L;
 else {
  secondList = split(L);
  La = mergeSort(L);
  Lb = mergeSort(secondList);
  return merge(La, Lb);
 }
}

list merge(list La, list Lb)
{
 if(La == NULL) return Lb;
 else if(Lb == NULL) return La;
 else if(element(La) <= element(Lb)) {
  La->next = merge(next(La), Lb);
  return La;
 }
 else {
  Lb->next = merge(La, next(Lb));
  return Lb;
 }
}

list split(list L)
{
 list pSecondCell;

 if(L == NULL) return NULL;
 else if(next(L) == NULL) return NULL;
 else {
  pSecondCell = next(L);
  L->next = next(pSecondCell);
  pSecondCell->next = split(next(pSecondCell));
  return pSecondCell;
 }
}
```

FIG. 1.6 – Le programme de tri `MergeSort` en $\text{Sub}_{\mathcal{C}}^{ext}$

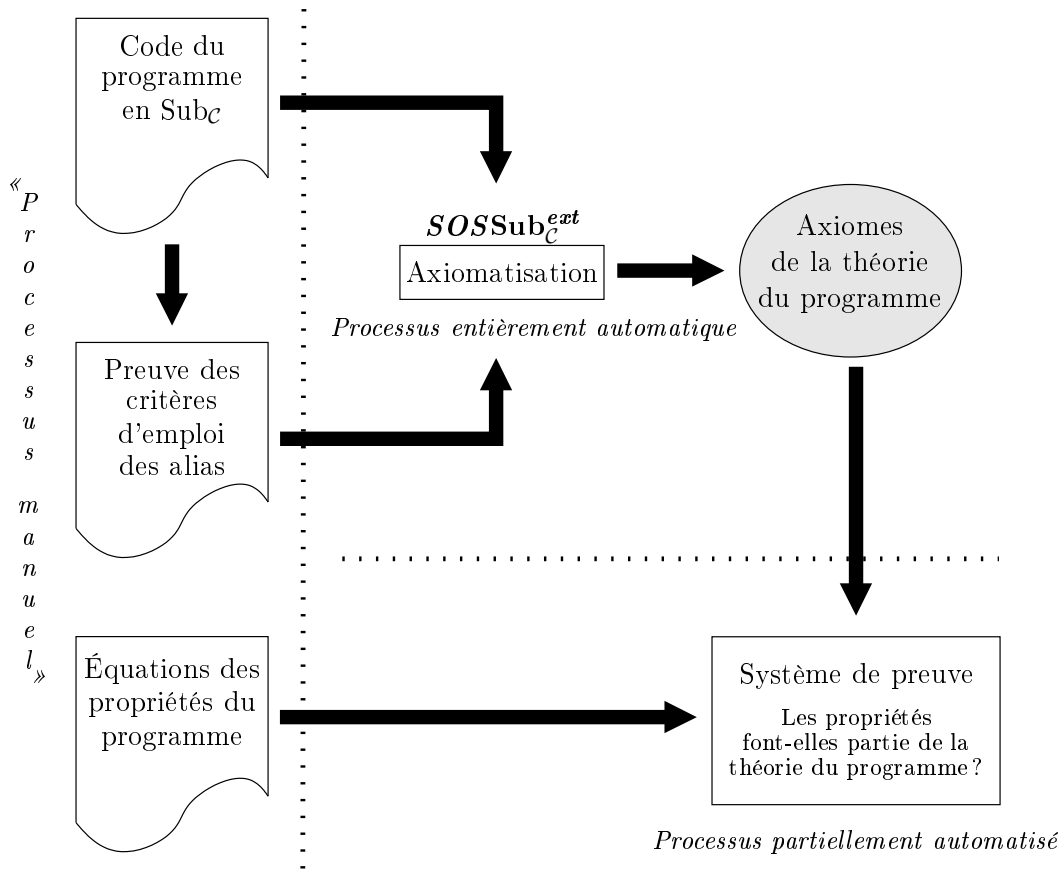


FIG. 1.7 – L'atelier de preuve avec *SOS*Sub$_\mathcal{C}^{ext}$. *L'atelier remanié inclut une étape manuelle de preuve que les critères de création et d'emploi des alias dans le programme source sont respectés.*

Grâce au système *SOS*Sub$_\mathcal{C}$, nous pouvons traduire automatiquement le code source d'un programme Sub$_\mathcal{C}$ en équations conditionnelles, avec lesquelles nous savons raisonner. Le programme `pgcd` n'utilisant aucune des constructions spécifiques de *SOS*Sub$_\mathcal{C}^{ext}$, les deux versions du système *SOS*Sub$_\mathcal{C}$ peuvent être employées. La Figure 1.8 contient les équations obtenues pour le programme `pgcd`. Celles-ci constituent ce que nous avons appelé les axiomes de la théorie équationnelle du programme. Elles vont nous permettre de prouver le théorème suivant :

Théorème 1.17. *Soient a et b deux entiers naturels tels que $a > b$, étant donné la définition équationnelle de la fonction* pgcd *en Figure 1.8,* $\text{pgcd}(a, b) = \text{pgcd}(a - b, b)$.

Démonstration. La démonstration est triviale et s'appuie sur une propriété de mod :

$$b \neq 0 \Rightarrow a \mod b = a - b \mod b$$

La Section 6.1 discute de cette preuve avec un assistant de preuve. □

Puisque les équations de la Figure 1.8 expriment la sémantique du programme `pgcd`, nous déduisons du Théorème 1.17 que ce programme vérifie la Propriété 1.1 du pgcd.

$$b \neq 0 \Rightarrow \mathrm{pgcd}_a(a, b, r) = \mathrm{pgcd}_a(b, a \bmod b, b) \quad (1.14)$$
$$\mathrm{pgcd}_a(a, 0, r) = a \quad (1.15)$$
$$\mathrm{pgcd}(a, b) = \mathrm{pgcd}_a(a, b, 0) \quad (1.16)$$

FIG. 1.8 – Équations de `pgcd`

Cependant, ceci ne suffit pas à prouver que le programme `pgcd` calcule effectivement le pgcd de deux nombres. La propriété que nous venons de montrer ne constitue pas à elle seule une *spécification complète* du pgcd. Dans le cas d'une application logicielle réelle, la description de l'application est souvent informelle et la formalisation d'une spécification complète du problème peut s'avérer difficile. Dans le cas présent, les propriétés désirées ont déjà une définition équationnelle et peuvent sans difficulté être exprimées dans un système de preuve. Pour le programme de calcul du pgcd, nous pourrions encore vouloir vérifier les deux propriétés suivantes :

Propriété 1.18. *Pour a et b entiers naturels,* $\mathrm{pgcd}(a, b) = \mathrm{pgcd}(b, a)$.

Propriété 1.19. *Pour a entier naturel,* $\mathrm{pgcd}(a, a) = a$.

Démonstration. La démonstration de ces propriétés avec un système de preuve ne pose pas de difficulté (*cf.* Section 6.1). □

Outre leur intérêt pour la preuve de programme, les équations obtenues avec $SOS\mathrm{Sub}_{\mathcal{C}}$ pour le programme `pgcd` définissent un *programme équationnel* exécutable. En effet, les équations peuvent être orientées, de gauche à droite, en règles de réécriture. La normalisation d'un terme à l'aide de ce système fournit une sémantique opérationnelle pour l'évaluation du programme `pgcd`. En particulier, le nombre de réécritures de pgcd_a nous renseigne sur la complexité de la boucle dans le programme source.

Exemple 1.20. Voici, à partir des équations de la Figure 1.8 orientées en règles, la séquence de réécriture qui mène à l'évaluation de `pgcd(24, 9)` :

$$\begin{aligned}
\mathrm{pgcd}(24, 9) &\overset{(1.16)}{\rightarrow} \\
\mathrm{pgcd}_a(24, 9, 0) &\overset{(1.14)}{\rightarrow} \\
\mathrm{pgcd}_a(9, 6, 9) &\overset{(1.14)}{\rightarrow} \\
\mathrm{pgcd}_a(6, 3, 6) &\overset{(1.14)}{\rightarrow} \\
\mathrm{pgcd}_a(3, 0, 3) &\overset{(1.14)}{\rightarrow} \\
\mathrm{pgcd}_a(3, 0, 3) &\overset{(1.15)}{\rightarrow} 3
\end{aligned}$$

♦

1.3.2 Sur l'exemple du tri par fusion

Nous avons discuté dans la Section 1.1.2 d'un programme de tri par fusion employant des constructions avancées des langages impératifs. Dans cette section, nous montrons que le système $SOS\mathrm{Sub}_{\mathcal{C}}$, dans sa version $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$, nous permet de raisonner aussi sur ce type de programmes avec listes mutables et sous-programmes à effet de bord.

Nous donnons, Figure 1.9, les équations automatiquement générées par le système $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ à partir du programme de tri fusion en $\mathrm{Sub}_{\mathcal{C}}$ de la Figure 1.6. Des fonctions intermédiaires, split_L, merge_{La} et merge_{Lb}, ont été définies par le système. Elles proviennent du traitement des paramètres passés par référence. Ce traitement est détaillé en Section 5.3.

Note 1.21. Dans les équations, nous employons la notation l_i pour une liste et e_i pour un élément d'une liste. Une liste vide est dénotée par NULL. Nous représentons le chaînage des éléments en les séparant par un point, *e.g.*, une liste avec au moins deux éléments : $e_1 \cdot e_2 \cdot l_3$.

La preuve que ce programme respecte les Propriétés 1.3 et 1.4, autrement dit ses spécifications, est faite à l'aide d'un assistant de preuve dans le Chapitre 7.1. Cependant, comme discuté précédemment et illustré par la Figure 1.7, l'axiomatisation par le système $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$, en présence de constructions du langage $\mathrm{Sub}_{\mathcal{C}}$ pouvant créer des alias, n'est correcte que si certaines restrictions dans l'usage de ces constructions sont respectées. La démonstration que le programme `MergeSort` observe les restrictions sur la création et l'emploi des alias est faite dans la Section 7.1.1.

$$\begin{aligned}
\text{mergeSort}(\text{NULL}) &= \text{NULL}\\
\text{mergeSort}(e_{1_L} \cdot \text{NULL}) &= e_{1_L} \cdot \text{NULL}\\
\text{mergeSort}(e_{1_L} \cdot e_{2_L} \cdot l_{3_L}) &= \text{merge}(\text{mergeSort}(\text{split}_L(e_{1_L} \cdot e_{2_L} \cdot l_{3_L})),\\
&\qquad \text{mergeSort}(\text{split}(e_{1_L} \cdot e_{2_L} \cdot l_{3_L})))
\end{aligned}$$

(a) sous-programme `mergeSort`

$$\begin{aligned}
\text{merge}(\text{NULL}, L) &= L\\
\text{merge}(e_{1_L} \cdot l_{2_L}, \text{NULL}) &= e_{1_L} \cdot l_{2_L}\\
e_{1_{La}} \leq e_{1_{Lb}} &\Rightarrow \text{merge}(e_{1_{La}} \cdot l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}}) = e_{1_{La}} \cdot \text{merge}(l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}})\\
e_{1_{La}} > e_{1_{Lb}} &\Rightarrow \text{merge}(e_{1_{La}} \cdot l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}}) = e_{1_{Lb}} \cdot \text{merge}(e_{1_{La}} \cdot l_{2_{La}}, l_{2_{Lb}})
\end{aligned}$$

$$\begin{aligned}
\text{merge}_{La}(\text{NULL}, L) &= \text{NULL}\\
\text{merge}_{La}(e_{1_L} \cdot l_{2_L}, \text{NULL}) &= e_{1_L} \cdot l_{2_L}\\
e_{1_{La}} \leq e_{1_{Lb}} &\Rightarrow \text{merge}_{La}(e_{1_{La}} \cdot l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}}) = e_{1_{La}} \cdot \text{merge}(l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}})\\
e_{1_{La}} > e_{1_{Lb}} &\Rightarrow \text{merge}_{La}(e_{1_{La}} \cdot l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}}) = \text{merge}_{La}(e_{1_{La}} \cdot l_{2_{La}}, l_{2_{Lb}})
\end{aligned}$$

$$\begin{aligned}
\text{merge}_{Lb}(\text{NULL}, L) &= L\\
\text{merge}_{Lb}(e_{1_L} \cdot l_{2_L}, \text{NULL}) &= \text{NULL}\\
e_{1_{La}} \leq e_{1_{Lb}} &\Rightarrow \text{merge}_{Lb}(e_{1_{La}} \cdot l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}}) = \text{merge}_{Lb}(l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}})\\
e_{1_{La}} > e_{1_{Lb}} &\Rightarrow \text{merge}_{Lb}(e_{1_{La}} \cdot l_{2_{La}}, e_{1_{Lb}} \cdot l_{2_{Lb}}) = e_{1_{Lb}} \cdot \text{merge}(e_{1_{La}} \cdot l_{2_{La}}, l_{2_{Lb}})
\end{aligned}$$

(b) sous-programme `merge`

$$\begin{aligned}
\text{split}(\text{NULL}) &= \text{NULL}\\
\text{split}(e_{1_L} \cdot \text{NULL}) &= \text{NULL}\\
\text{split}(e_{1_L} \cdot e_{2_L} \cdot l_{3_L}) &= e_{2_L} \cdot \text{split}(l_{3_L})
\end{aligned}$$

$$\begin{aligned}
\text{split}_L(\text{NULL}) &= \text{NULL}\\
\text{split}_L(e_{1_L} \cdot \text{NULL}) &= e_{1_L} \cdot \text{NULL}\\
\text{split}_L(e_{1_L} \cdot e_{2_L} \cdot l_{3_L}) &= e_{1_L} \cdot \text{split}_L(l_{3_L})
\end{aligned}$$

(c) sous-programme `split`

FIG. 1.9 – Équations du programme `MergeSort`

Première partie

État des lieux

Chapitre 2

Approches de la vérification de programme

La vérification de programme et, plus génériquement, les méthodes formelles constituent un vaste domaine scientifique toujours activement défriché qui se révèle fertile en techniques et outils. La grande variété des formalismes et des méthodes, parfois spécifiques à un besoin, rend vaine et fastidieuse toute prétention à l'exhaustivité. Dans le cadre de ce mémoire, nous nous limiterons à la présentation des principales catégories de méthodes formelles en insistant sur les approches qui possèdent des correspondances avec la nôtre quant à leur finalité ou leurs caractéristiques.

2.1 Tour d'horizon des méthodes formelles

Une définition, passablement générale, des méthodes formelles englobe l'ensemble des langages, techniques et outils fondés sur les mathématiques et dont les objectifs sont la spécification et la vérification des systèmes. Selon J. M. Wing (1990), circonscrites au développement des systèmes informatiques, les méthodes formelles fournissent un cadre mathématique pour décrire des propriétés des systèmes et *systématiquement* spécifier, développer et vérifier les systèmes.

Au moment de la conception, les méthodes formelles sont à même de révéler les ambiguïtés, incomplétudes et incohérences d'un système informatique. Ensuite, elles permettront de montrer la correction d'une implémentation ou l'équivalence de différentes implémentations. Les méthodes formelles souffrent cependant d'une limitation théorique intrinsèque : elles permettent de *vérifier* un système mais pas de le *valider*. Nous avons précédemment évoqué cette limitation à propos de l'impossibilité de « formellement formaliser » la description intuitive et informelle d'un système informatique qui constitue habituellement le point de départ de la conception du système. Une autre instance de ce problème traduit l'impossible rencontre du monde concret et du modèle abstrait. Les méthodes formelles peuvent montrer qu'une spécification est satisfaite par une implémentation sur une machine abstraite plus ou moins idéalisée. Or, le niveau d'abstraction ne peut être infiniment diminué (*e.g.*, du processeur aux transistors qui le composent, des transistors aux lois de l'électronique qui les régissent, etc.) jusqu'au point où la machine

abstraite serait l'exacte et entière description d'une réalité physique. En l'état actuel de nos connaissances, il faut déterminer un niveau d'abstraction adéquat et admettre *a priori* que le système physique concret satisfait les prérequis postulés, souvent implicitement, par la méthode formelle considérée.

Dans la suite de cette section, nous distinguons les approches selon qu'elles s'appliquent aux spécifications d'un programme (synthèse de programme) ou au programme lui-même (approches par modèles et approches déductives).

2.2 Synthèse de programme

Les méthodes présentées dans cette section traitent le problème de la vérification de programme en le supprimant par le simple constat qu'un programme correct n'a pas besoin d'être vérifié ! Il s'agit en effet pour ces méthodes d'éviter de vérifier les programmes en certifiant de préférence le « processus d'écriture » des programmes.

L'approche par synthèse de programme présente toutefois l'inconvénient que le code généré est rarement aussi efficace que celui écrit par un programmeur humain. En effet, un langage de spécification peut aider à décrire *quoi* faire mais n'est pas d'une expressivité suffisante pour indiquer *comment* le faire. En outre, ces techniques ne s'appliquent évidemment pas à la vérification ou à la maintenance des programmes existants.

2.2.1 Génération de code

Lors de la génération de code, les programmes sont dérivés de leur spécification abstraite par une succession de transformations manuelles ou automatiques, appelée *raffinement*. Le raffinement précise la spécification initiale jusqu'à un niveau de détail d'implémentation suffisant pour que le code source d'un programme puisse automatiquement être généré. Le langage de spécification est un langage formel de haut niveau (*e.g.*, Z [Spivey, 1992] basé sur la théorie des ensembles, Metaslang [Juellig *et al.*, 1996] basé sur la théorie des catégories) dont la sémantique est suffisamment bien définie pour produire automatiquement du code source dans différents langages de programmation (*e.g.*, en Lisp pour Metaslang, en Ada pour la notation Z).

Chaque étape du raffinement engendre des obligations de preuve à la charge de l'utilisateur. Le gain espéré de ces méthodes est que la preuve des obligations issues du raffinement est plus simple que la preuve du système achevé complet.

Les systèmes de génération de code diffèrent principalement par leur langage de spécification. Ce dernier est généralement optimisé pour un usage particulier. La méthode B [Abrial, 1996] et les machines à états abstraits (auparavant appelées *evolving algebra*) [Gurevich, 1985; 1991], VDM [Jones, 1990] et la notation Z, SPECWARE [Juellig *et al.*, 1996] et Metaslang sont des exemples de tels systèmes ; citons aussi le système ELAN [Borovanský *et al.*, 1996] capable de générer du code C à partir de règles de réécriture et le langage OPAL [Didrich *et al.*, 1994] qui mêle aspects fonctionnels et algébriques et qui est compilé vers le langage C.

Citons encore les systèmes formels pour la génération d'environnement de programmation, tels Centaur [Borras *et al.*, 1988] et ASF+SDF [van den Brand *et al.*, 2001], qui permettent de produire des outils logiciels pour la programmation (*e.g.*, un éditeur, un interprète, un compilateur, un débogueur) à partir d'une description du langage de programmation.

2.2.2 Programmes corrects par construction

Dans ce type d'approche, la spécification *est* le programme. Parce que le langage de spécification dispose d'une sémantique opérationnelle, une spécification est directement exécutable. Nous trouvons dans cette catégorie les langages de programmation logiques au cœur desquels se trouve la notion de *clause de Horn* tels que Prolog [Colmerauer et Roussel, 1993] et les langages de programmation purement fonctionnels basés sur le λ-calcul tels que Haskell [Hudak et Fasel, 1992].

Une autre catégorie de systèmes se fonde sur des logiques *constructives* (*e.g.*, le calcul des constructions inductives de Coq [Bertot et Casteran, 2004; Coquand et Huet, 1988]). Une propriété remarquable de ces logiques est qu'une démonstration de théorème possède une interprétation « calculatoire », c'est-à-dire qu'un programme qui réalise l'énoncé du théorème peut être *extrait* de la preuve (*cf.*, par exemple, [Paulin-Mohring, 1989]). Ainsi, sous le contrôle d'un tel système, établir la preuve qu'une spécification peut être satisfaite revient à construire le programme qui réalise la spécification.

2.3 Vérification de modèle

Dans l'approche par *vérification de modèle* (*model checking* en anglais), le système à analyser est décrit par un *modèle* sous forme de système de transitions (*e.g.*, un automate). La spécification du système est un ensemble de formules exprimées dans une logique temporelle [Pnueli, 1981; Manna et Pnueli, 1992], c'est-à-dire une logique qui intègre une notion de temps. Le principe est alors de vérifier si le modèle satisfait une propriété en parcourant la totalité des états du modèle (l'espace des états). Lorsque l'espace des états est fini, et pour certaines logiques temporelles, la vérification de modèle est décidable et les algorithmes sont rapides, même pour un grand nombre d'états. De plus, lorsqu'une propriété n'est pas vérifiée, un contre-exemple est produit, ce qui est précieux lors de la mise au point d'un système. D'autant plus que l'application la plus typique de cette approche vise les systèmes réactifs concurrents (*e.g.*, les protocoles de communication) pour lesquels les erreurs sont souvent non reproductibles.

Bien adaptée aux systèmes dont la tâche essentielle est le *contrôle*, la vérification de modèle est inappropriée lorsque les données manipulées par le système à analyser supposent un espace d'états infini (*e.g.*, les listes). Outre cette limitation inhérente aux systèmes à états finis, sauf à considérer des classes de propriétés encore plus restrictives que celles des logiques temporelles (*e.g.*, les propriétés de sûreté), l'approche doit faire face au problème de l'explosion de l'espace des états qui, malgré les efforts pour la contenir (*e.g.*, les diagrammes de décision binaire (*Binary Decision Diagram*) [Bryant, 1986]), rend l'exploration automatique des modèles impraticable lorsque leur taille grandit.

Enfin, citons par exemple les outils SMV [McMillan, 1993] ou XEVE [Bouali, 1998] qui ont adopté cette approche.

2.4 Vérification par démonstration

La dernière catégorie d'approches regroupe les méthodes qui fournissent un cadre formel à la *démonstration* de propriétés de programmes. Ces méthodes se distinguent des précédentes par :

- le fait qu'elles sont destinées à l'analyse des programmes au niveau du code source ;
- leur capacité de traiter, par des techniques d'induction, des données dont le domaine de valeur est infini.

L'ambition de ces méthodes est ainsi de pouvoir raisonner formellement et, au moins partiellement, de façon automatique sur un programme. Le programme et sa spécification sont exprimés dans un système logique formel. Les mécanismes déductifs et inductifs du système logique sont aussi formalisés et servent à élaborer la démonstration que le programme est correct relativement à sa spécification.

Les méthodes de vérification par démonstration se divisent en deux grandes familles que nous présentons après un préambule sur les *systèmes de preuve* logiciels.

2.4.1 Systèmes de preuve logiciels

Les systèmes de preuve logiciels sont des systèmes logiques (*i.e.*, composés d'un langage formel, d'un ensemble d'axiomes et de règles d'inférence), dans lesquels la construction de démonstrations est assistée par ordinateur.

Chaque système de preuve offre un compromis entre expressivité de la logique et automatisation des démonstrations.

Démonstrateurs de théorèmes

La priorité des démonstrateurs de théorèmes (*theorem prover* en anglais) est l'automatisation de la démonstration. Toutefois, pour atteindre ce but, le système doit souvent être paramétré par un expert en fonction du problème à résoudre. De plus, la trace du déroulement de la preuve est peu compréhensible pour un simple utilisateur. Le langage de ces systèmes est couramment la logique du premier ordre ou un de ses sous-ensembles décidable. Certains systèmes sont capables d'induction implicite. Des exemples de tels systèmes sont RRL [Kapur et Zhang, 1988], SPIKE [Bouhoula *et al.*, 1992] ou Otter [McCune, 1994].

Assistants de preuve

Cette fois, l'accent est mis sur l'expressivité du langage utilisé pour les preuves. Les assistants de preuve emploient la plupart du temps une logique d'ordre supérieur dans

laquelle le problème de construire une démonstration pour une propriété est indécidable dans le cas général. La vocation initiale de ces systèmes est de vérifier la validité d'une preuve construite par ailleurs. L'automatisation des tâches ingrates et répétitives tend à rendre ces systèmes plus conviviaux et facilite l'élaboration interactive d'une démonstration. Citons par exemple les systèmes Coq [Barras *et al.*, 1997] dont la logique est constructive et HOL [Gordon, 1987] ou PVS [Owre *et al.*, 1992] basés sur une logique classique d'ordre supérieur.

Ateliers logiques

Plus génériques que les outils précédents qu'ils peuvent spécifier et implanter, les ateliers logiques se veulent aussi plus flexibles en permettant à l'utilisateur de combiner les paradigmes de déduction, de définir son propre système logique et d'ajouter de nouvelles procédures de démonstration. Isabelle [Paulson, 1994] et ELAN sont des exemples d'ateliers logiques.

2.4.2 Annotation de programme

Cette première famille de méthodes par démonstration trouve ses origines dans les logiques de type Floyd-Hoare [Floyd, 1967; Hoare, 1969]. Ces logiques sont des systèmes axiomatiques dont les règles d'inférence permettent d'établir que, étant données une *pré-condition* P et une *post-condition* Q d'un programme, l'évaluation du programme conduit à Q lorsque P est vraie. La pré-condition et la post-condition sont des énoncés logiques portant sur les variables du programme analysé.

Dans ces méthodes, la spécification d'un programme est composée d'un ensemble d'énoncés logiques disséminés dans le code source du programme. Le programme ainsi annoté est traduit dans le formalisme d'un système de preuve afin de vérifier que le programme satisfait les annotations.

J.-C. Filliâtre (1999) propose une méthode autour de Coq en définissant son propre langage de programmation et d'annotation ; le langage Java est l'objet d'un effort important avec le développement de langages d'annotation comme JML [Leavens *et al.*, 1999] et des outils de vérification associés tels que Krakatoa [Marché *et al.*, 2004] s'interfaçant avec Coq, LOOP tool [Jacobs *et al.*, 1998] s'interfaçant avec PVS et Isabelle.

Un inconvénient de cette approche est l'introduction d'un langage d'annotation, spécifique à la méthode ou au langage de programmation, qui demande un investissement d'apprentissage pour être maîtrisé à la fois par les concepteurs et les développeurs d'une application.

2.4.3 Génération de spécification

La génération de spécification s'attache à extraire automatiquement la « spécification » d'un programme à partir de son code source. Cette spécification extraite est plus précisément la traduction du programme vers un langage formel avec lequel il est possible de raisonner. La vérification de la correction du programme consiste alors à montrer que la spécification extraite correspond à la spécification initiale du concepteur. Cette

vérification est conduite avec l'aide d'un système de preuve. Au cœur de ces méthodes réside l'attribution d'une sémantique aux programmes. Le choix de la sémantique et les techniques pour l'exprimer à partir du code source distinguent les méthodes.

Définition de sémantique

Une première façon d'aborder le problème est de définir la sémantique « standard » du langage de programmation dans un formalisme approprié à la démonstration de propriétés de programmes. C'est l'approche retenue dans cette thèse et dans les travaux de J. A. Bergstra *et al.* (1997) décrits plus en détail dans la Section 2.5.

Traduction préservant la sémantique

Une seconde approche de la génération de spécification est d'exprimer la signification d'un programme en se référant à une sémantique différente de la sémantique « standard » du langage de programmation concerné. Ceci s'avère particulièrement utile pour la preuve de correction des compilateurs où langage source et langage cible ont des sémantiques différentes. Il convient alors de montrer que la traduction est correcte en « décodant » la sémantique du langage source dans l'autre (*cf.* [Morris, 1973]). Mais cette technique permet aussi de remplacer la sémantique « standard » par une autre plus appropriée à la preuve de programmes, auquel cas il faut aussi assurer la correction de la traduction. C'est la démarche choisie par exemple par J.-C. Filliâtre (1999) qui prouve la correction de sa traduction d'un langage impératif vers le calcul des constructions inductives. Citons encore les travaux de M. Ward (1993) qui traduit (sans preuve de correction) des programmes impératifs vers le langage WSL [Ward, 1989] dans lequel un ensemble prédéfini de transformations préservant la sémantique permettent de prouver la correction du programme.

2.5 Preuve de propriétés de programmes impératifs

Nous abordons dans cette section les travaux concernant plus spécifiquement la preuve de propriétés de programmes impératifs. L'analyse statique formelle des programmes débute nécessairement par l'attribution d'un sens rigoureux aux programmes *via* une sémantique du langage de programmation utilisé. La sémantique d'un langage définit la sémantique de tous les programmes du langage. La sémantique d'un programme définit tous les comportements possibles de ce programme. La preuve d'un programme consiste à démontrer qu'une sémantique du programme satisfait une spécification donnée.

La sémantique est ainsi le support sur lequel seront bâtis les raisonnements sur les programmes. Ceci confère aux sémantiques une importance primordiale et explique l'abondance des formalismes qui diffèrent selon le style de leur définition et les propriétés que les sémantiques rendent observables. Nous introduisons succinctement trois grandes classes de sémantique (opérationnelle, axiomatique et dénotationnelle) ainsi que certaines des méthodes de preuve de propriétés de programmes impératifs qu'elles ont inspirées. Nous

présentons ensuite des travaux qui s'appuient sur des sémantiques dont le classement dans l'une des trois classes n'est pas possible.

2.5.1 Sémantique opérationnelle

Une sémantique opérationnelle définit la signification d'un langage de programmation en spécifiant le comportement des programmes pendant l'exécution. Autrement dit, elle décrit la façon d'exécuter un programme. Cette description fait généralement intervenir une notion d'état et des transitions entre états qui sont attachées aux opérations d'une certaine machine abstraite. Ce type de sémantique constitue par conséquent une spécification de *comment* un calcul doit être réalisé et correspond typiquement au point de vue du programmeur.

Les sémantiques opérationnelles servent notamment à la définition de compilateurs et interprètes pour les langages mais aussi à l'étude des équivalences de programmes. Cependant, le caractère très concret des sémantiques opérationnelles a pour contrepartie de les rendre verbeuses et difficilement modifiables ou extensibles.

Traduction vers un programme fonctionnel

J.-C. Filliâtre (1999) donne une définition opérationnelle de la sémantique « standard » de son langage de programmation impératif. Celle-ci sert de référence pour la correction de la traduction des programmes impératifs vers des programmes fonctionnels dans le calcul des constructions inductives de Coq.

Traduction vers un programme logique

J. C. Peralta *et al.* (1998) implantent la sémantique opérationnelle d'un langage impératif sous la forme d'un programme logique. Ce dernier constitue un interprète pour les programmes impératifs. Un évaluateur partiel, convenablement paramétré, est appliqué à l'interprète avec un programme impératif en entrée. Le résultat est un nouveau programme logique qui est une vue déclarative du programme impératif en entrée. L'analyse du programme impératif peut alors être menée avec l'aide des outils d'analyse et de transformation des programmes logiques.

2.5.2 Sémantique axiomatique

Avec une sémantique axiomatique, un système logique sert de cadre à la description des langages de programmation. Les constructions d'un langage y sont définies par des axiomes.

Une sémantique axiomatique relie des propositions logiques à propos de l'état d'un programme et constitue ainsi une logique des propriétés de programmes. Elle est par conséquent naturellement appropriée pour raisonner à propos de la correction de programmes individuels.

Logique de Floyd-Hoare

L'emploi le plus représentatif de l'approche axiomatique pour la preuve de propriétés de programmes est fourni par les logiques du type Floyd-Hoare [Floyd, 1967; Hoare, 1969]. Il s'agit d'une approche dans laquelle des assertions, et en particulier des invariants, sont associés à des points particuliers d'un programme. Le programme est correct s'il est démontré, au moyen des axiomes de la logique, que les instructions du programme satisfont les assertions et les invariants.

Première méthode historiquement à apporter une réponse théorique satisfaisante au problème de la correction des programmes, la logique de Floyd-Hoare est peu utilisée en pratique car le processus de preuve s'avère long et fastidieux. De plus, ni la terminaison des programmes, ni leur équivalence ne peuvent être exprimées dans cette logique. Notons toutefois que la transformation des annotations de programme en *conditions de vérification* [Igarashi *et al.*, 1975] a ouvert la voie à une automatisation partielle du processus de preuve (*e.g.*, un générateur de conditions de vérification certifié dans HOL [Homeier et Martin, 1994]).

Traduction vers un prédicat logique

Le calcul de *la plus faible pré-condition* [Dijkstra, 1976] consiste à trouver la condition initiale nécessaire et suffisante pour que l'exécution d'une suite d'instructions rende vraie une certaine assertion appelée *post-condition*. Autour d'une sémantique axiomatique, H. Gibbons (1998) propose un calcul de la plus faible pré-condition d'un programme impératif qui conduit à une représentation déclarative du programme par un prédicat logique. La méthode est entièrement manuelle et ne permet pas de prendre en compte les appels récursifs de fonctions.

2.5.3 Sémantique dénotationnelle

La sémantique dénotationnelle a été développée par D. Scott et C. Strachey (1971). Le principe est de représenter la signification d'un programme par une entité d'une structure mathématique appelée la *dénotation* du programme. La sémantique d'un langage est donnée par un ensemble de fonctions sémantiques qui associent les constructions du langage de programmation à leurs dénotations. Les dénotations sont souvent elles-mêmes des fonctions.

La sémantique dénotationnelle s'intéresse uniquement à la relation entre les entrées et les sorties d'un programme, elle a donc tendance à masquer les étapes de calcul intermédiaires. Par conséquent, ce type de sémantique est plus utilisé pour étudier les langages de programmation que les programmes eux-mêmes.

La sémantique dénotationnelle fait intervenir explicitement un modèle de la mémoire des ordinateurs. De plus, certains concepts de programmation familiers dans les langages impératifs (*e.g.*, les exceptions, la notion de séquence) nécessitent l'introduction d'entités particulières (*e.g.*, les *monades* pour les effets de bord [Moggi, 1991; Jacobs et Poll, 2003]) qui compliquent la compréhension intuitive de ces concepts.

Traduction vers un programme fonctionnel

La thèse de B. Moura (1997) explore une méthode pour la traduction des programmes impératifs en des programmes fonctionnels qui s'appuie sur une sémantique dénotationnelle du langage impératif. Afin d'améliorer la précision de la traduction, Moura introduit plusieurs modèles pour la mémoire de plus en plus complexes. La représentation fonctionnelle du programme impératif obtenue est mieux adaptée aux traitements ultérieurs tels que l'analyse ou l'évaluation partielle.

2.5.4 Sémantique algébrique

L'idée de base des sémantiques algébriques est de spécifier une classe de machines abstraites et de définir par des axiomes l'effet des programmes sur ces machines. Plus concrètement, il s'agit de répertorier les différents types d'objets et d'opérations sur les objets, puis de décrire leurs propriétés par des axiomes dans une logique.

Cette même approche est fréquemment employée pour définir des *types de données abstraits* : la description du type de donnée ne se réfère à aucune implantation particulière mais spécifie les propriétés logiques des opérations sur les données de ce type.

Alors que la sémantique dénotationnelle s'appuie sur un modèle particulier de la mémoire, une sémantique algébrique peut définir une classe de modèles pour la mémoire.

Nous nous sommes intéressé plus particulièrement aux sémantiques algébriques fondées sur la logique équationnelle. Ce type de sémantique emprunte des traits à chacune des trois grandes classes de sémantique précédentes. L'aspect dénotationnel des sémantiques algébriques est dû au fait que des entités mathématiques, les *algèbres*, servent de modèles aux spécifications algébriques. Ensuite, les équations des spécifications algébriques donnent un aspect axiomatique à ces sémantiques. Enfin, l'aspect opérationnel provient de ce que les équations des spécifications algébriques peuvent souvent être orientées en règles de réécriture permettant ainsi une exécution symbolique des programmes.

PIM

J. A. Bergstra *et al.* (1997) ont conçu une logique équationnelle, appelée PIM, dont le but est de servir d'atelier pour l'analyse et la transformation des programmes impératifs. PIM se veut une représentation intermédiaire des programmes avec laquelle il est possible de raisonner formellement. PIM_t, le noyau algébrique de PIM, permet de modéliser différentes classes de mémoires et est utilisé pour donner une sémantique à une classe restreinte de programmes impératifs écrits en μC et dépourvus de boucles et de fonctions. Pour des programmes plus réalistes, les constructions d'ordre supérieur de PIM sont nécessaires.

OBJ

OBJ est une famille de langages pour la programmation et la spécification algébrique. OBJ3, membre de cette famille, est le support de la méthode de vérification de programmes impératifs développée par J. A. Goguen et G. Malcolm (1996). Dans cette approche, la mémoire est définie comme un type de donnée abstrait associant les variables des

programmes à des valeurs entières. En partant de cette caractérisation de la mémoire, les auteurs formulent la sémantique de l'affectation dans leur langage de programmation. La sémantique des autres constructions de leur langage découle alors de celle de l'affectation.

La sémantique du langage de programmation forme ainsi une théorie dans la logique équationnelle qui permet de construire et de prouver des assertions logiques à propos du comportement d'un programme donné. Les spécifications des programmes sont exprimées par des pré-conditions, des post-conditions et des invariants sur le modèle de la logique de Floyd-Hoare. En plus d'être un langage de spécification, OBJ3 incorpore certains algorithmes pour la démonstration de théorèmes (*e.g.*, la réécriture) et les règles d'inférence de la logique du premier ordre qui permettent d'assister l'utilisateur lors de la preuve de correction des programmes.

PESCA

PESCA [Denzler et Schweizer, 1996] est un projet qui vise à extraire des spécifications algébriques à partir de programmes impératifs. Le langage de programmation considéré, Oberon_{T}, est particulièrement restrictif puisqu'il ne comporte ni boucle (uniquement une forme de boucle « pour »), ni sous-programme récursif, ni effet de bord. La sémantique du langage est définie de manière algébrique. Elle intervient lors de la traduction des programmes vers le langage de spécification algébrique LSL [Guttag et Horning, 1993]. Les spécifications LSL sont utilisées dans le démonstrateur de théorèmes Larch Prover [Garland *et al.*, 1993] pour prouver des propriétés du programme initial.

2.5.5 *SOS*$\text{Sub}_{\mathcal{C}}$

Nous proposons avec *SOS*$\text{Sub}_{\mathcal{C}}$ une approche équationnelle de la définition de la sémantique des programmes impératifs. Dans la classification établie précédemment, nous nous situons parmi les méthodes de génération de spécification par définition de sémantique. Le langage de programmation impératif $\text{Sub}_{\mathcal{C}}$, un sous-ensemble du langage C, sert de support à notre méthode. La *sémantique du langage* est définie de telle sorte qu'elle permette l'exécution symbolique d'un programme. Le résultat de l'exécution symbolique est une définition équationnelle de la *sémantique du programme*. La spécification du programme est exprimée dans la logique équationnelle, ou bien dans la logique du premier ordre, comme un ensemble de propriétés que doit vérifier le programme. Nous pouvons alors confronter la définition équationnelle du programme à sa spécification dans un système de preuve.

Notre méthode partage certaines idées fondamentales avec les approches algébriques citées plus haut. Nous pensons, comme J. A. Goguen et G. Malcolm (1996), J. Field *et al.* (1998), que la logique équationnelle constitue un cadre adéquat pour la sémantique et la vérification des programmes. La logique équationnelle présente en effet plusieurs avantages, parmi lesquels :

– La capacité à être automatisée. Il existe de nombreux outils logiciels intégrant la logique équationnelle qui implantent des algorithmes efficaces pour la réécriture, l'unification, la complétion, etc.

– La simplicité de son formalisme. La logique équationnelle peut de ce fait prétendre à devenir la base commune aux différents acteurs d'un développement logiciel et faciliter ainsi la pénétration des méthodes formelles dans le monde industriel.

Sur le fond, notre approche se distingue des autres approches algébriques en n'ayant pas recours à un modèle de la mémoire pour décrire la sémantique des programmes. Cette particularité conduit à une expression plus simple de la sémantique des programmes.

De plus, dans notre approche, la sémantique des programmes apparaît explicitement comme une théorie équationnelle distincte de la sémantique du langage. Ce n'est pas le cas dans [Goguen et Malcolm, 1996], où la sémantique des programmes n'est pas formulée explicitement : elle est implicitement donnée par la sémantique du langage au moment de la preuve d'une propriété d'un programme.

Dans PIM, la représentation équationnelle des programmes est une représentation intermédiaire destinée à être transformée et analysée dans l'outil lui-même. Les traits impératifs du langage sont traduits par des opérations spécifiques définies dans une théorie propre à l'outil. Dans notre approche, nous avons évité d'introduire des opérations spécifiques. Cette préoccupation permet de proposer une axiomatisation des programmes concise et intuitive qui peut être utilisée telle quelle dans la majorité des systèmes de preuve manipulant la logique équationnelle.

Un autre élément de comparaison est l'expressivité du langage de programmation et notamment les traits impératifs dont il dispose. Si dans la version $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ le langage $\mathrm{Sub}_{\mathcal{C}}$ est assez limité, il n'en reste pas moins plus puissant que l'$\mathrm{Oberon}_{\mathrm{T}}$ de PESCA ou le μC de PIM_t qui ne comporte ni boucle ni fonction. La version $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ ajoute la prise en compte d'opérations destructives sur les listes qui ne sont pas abordées dans [Goguen et Malcolm, 1996].

Chapitre 3

Notions préliminaires

Nous donnons dans cette section les définitions des principales notions utiles à la compréhension de ce mémoire. Nous supposons connus les rudiments d'un langage de programmation impératif et de la logique du premier ordre (*cf.* [Loeckx et Sieber, 1987; David *et al.*, 2001] pour une introduction). Nous commençons par présenter la logique équationnelle et les équations conditionnelles avec lesquelles nous voulons exprimer la sémantique des programmes $\mathrm{Sub}_{\mathcal{C}}$. Ces définitions servent ensuite de fondement aux systèmes de réécriture qui interviennent dans l'axiomatisation des programmes.

Nous invitons le lecteur familier avec les notions présentées dans cette section à sauter les passages qui lui sont connus ou à s'y référer au fil de sa lecture. Pour l'essentiel des définitions, cette section reprend la présentation et les notations de [Baader et Nipkow, 1998]. D'autres sources, auxquelles nous renvoyons le lecteur pour plus de détails, ont aussi été utiles [Goguen et Malcolm, 1996; Plaisted, 1993; Dershowitz *et al.*, 1988]. Lorsque nous introduisons une notion, nous pourrons être amenés pour éviter une ambiguïté à signaler entre parenthèses et en italique le terme anglais équivalent.

3.1 Logique équationnelle

La logique équationnelle est un sous-ensemble de la logique du premier ordre. Elle offre une sémantique claire et simple, qui repose sur la substitution de termes égaux, ainsi que des algorithmes efficaces avec lesquels ont été développés de nombreux outils logiciels. Ces qualités reconnues[6] font de la logique équationnelle un cadre remarquable pour l'analyse et la vérification des programmes.

La logique équationnelle dispose d'une interprétation « naturelle » dans l'*algèbre abstraite* (*universal algebra*). Il sera cependant suffisant pour nous d'en rappeler les principes de base d'un point de vue syntaxique ; nous supposons connu l'aspect sémantique de ces principes. Nous définirons ainsi précisément ce qu'est une équation puis la logique équationnelle.

[6] Se référer, par exemple, à l'article *Equations as a Uniform Framework for Partial Evaluation and Abstract Interpretation* [Field *et al.*, 1998].

Le seul prédicat de la logique équationnelle, dans son acceptation la plus étroite, est l'*égalité*. Cependant, l'interprétation des programmes que nous proposons, avec en particulier l'interprétation des instructions conditionnelles et itératives, Sections 4.3.2 et 4.3.3, mais aussi les spécifications des programmes requièrent plus d'expressivité et se décrivent naturellement par des équations conditionnelles. Par conséquent, nous considérons une extension de la logique équationnelle dans laquelle nous pouvons exprimer des équations conditionnelles.

3.1.1 Langage

Les équations sont formées d'expressions appelées *termes* qui combinent opérations, constantes et variables. L'ensemble des opérations (ou fonctions) et des constantes à partir desquelles former les termes constitue une *signature*.

Définition 3.1 (signature). Une *signature* Σ est une paire $(\mathcal{S}, \mathcal{F})$ telle que $\mathcal{S}$ est un ensemble dénombrable de *types* et $\mathcal{F}$ un ensemble dénombrable de *symboles de fonction*. De plus, tout $f \in \mathcal{F}$ est associé à une paire (w, s), le type de f, où w est une séquence de types pris dans $\mathcal{S}$, appelée l'*arité* de f, et $s \in \mathcal{S}$ est le type résultat de f. Les symboles de fonction dont l'arité est la séquence vide sont aussi appelés des *symboles de constante*.

Note 3.2. Par convention, nous noterons $f : s_1, \ldots, s_n \to s$ la paire (w, s) associée au symbole de fonction f, où la séquence $w = \langle s_1, \ldots, s_n \rangle$.

Les types correspondent aux différentes catégories d'objets considérés, qui peuvent être des booléens, des entiers, des listes, etc. Ils permettent de restreindre la définition des opérations à certains types, par exemple, l'addition pourra être définie entre deux entiers et pas entre deux listes.

Les termes sont construits à partir d'une signature et sont typés, par conséquent toute combinaison d'opérations et de variables ne produit pas un terme *bien formé*. Intuitivement, les variables et les constantes sont des termes, ainsi que l'*application* d'un symbole de fonction à des termes de types appropriés.

Définition 3.3 (terme). Soient $\Sigma = (\mathcal{S}, \mathcal{F})$ une signature et $\mathcal{V}$ un ensemble dénombrable d'éléments appelés *variables* tels que $\mathcal{F} \cap \mathcal{V} = \emptyset$. L'ensemble $\mathcal{T}_s(\Sigma, \mathcal{V})$ de tous les *termes* de type $s \in \mathcal{S}$ engendré par $\mathcal{V}$ est récursivement défini par :

- $\mathcal{V} \subseteq \mathcal{T}_s(\Sigma, \mathcal{V})$;
- pour tout $n \geq 0$ et tout $f : s_1, \ldots, s_n \to s \in \mathcal{F}$, avec s et $s_i \in \mathcal{S}$ pour $i = 1, \ldots, n$, si $t_i \in \mathcal{T}_{s_i}(\Sigma, \mathcal{V})$ pour $i = 1, \ldots, n$, alors $f(t_1, \cdots, t_n) \in \mathcal{T}_s(\Sigma, \mathcal{V})$.

Note 3.4.

1. Nous noterons $\mathcal{T}(\Sigma, \mathcal{V}) = \bigcup_{s \in \mathcal{S}} \mathcal{T}_s(\Sigma, \mathcal{V})$.
2. Un terme ne comportant pas de variables est appelé un *terme clos*.

La structure d'un terme peut être représentée par un arbre dont les nœuds sont des symboles de fonction et les feuilles sont des variables ou des symboles de constante. Une numérotation des nœuds de l'arbre identifie uniquement chaque *occurrence* (*position*) dans un terme.

Définition 3.5 (occurrence et sous-terme). Soit un terme $t \in \mathcal{T}(\Sigma, \mathcal{V})$.

1. L'ensemble des *occurrences* du terme t, noté $\mathrm{Occ}(t)$, est l'ensemble des mots sur l'alphabet $\mathbb{N}^*$ récursivement défini par :
   - si $t \in \mathcal{V}$ alors $\mathrm{Occ}(t) := \{\epsilon\}$, où ϵ est le mot vide ;
   - si $t = f(t_1, \ldots, t_n)$ alors $\mathrm{Occ}(t) := \{\epsilon\} \bigcup_{i=1}^{n} \{io \mid o \in \mathrm{Occ}(t_i)\}$.
2. Soit $o \in \mathrm{Occ}(t)$, le *sous-terme de t à l'occurrence o*, noté $t|_o$, est récursivement défini par :
   - $t|_\epsilon := t$;
   - $f(t_1, \ldots, t_n)|_{ip} := t_i|_p$.

Note 3.6. Soient $t, u \in \mathcal{T}(\Sigma, \mathcal{V})$ et $o \in \mathrm{Occ}(u)$.

1. L'occurrence ϵ correspond à la racine de l'arbre d'un terme et est appelée *occurrence racine*. Le symbole à cette occurrence est appelé *symbole racine* du terme.
2. Nous noterons $u[t]_o$ le terme obtenu en remplaçant le sous-terme de u à l'occurrence o par t.

Nous pouvons maintenant donner les définitions d'équation et d'équation conditionnelle.

Définition 3.7 (équation). Soient $\Sigma = (\mathcal{S}, \mathcal{F})$ une signature et $\mathcal{V}$ un ensemble dénombrable de variables tels que $\mathcal{F}$ et $\mathcal{V}$ sont disjoints. Une *équation* est une paire de termes $(t, u) \in \mathcal{T}_s(\Sigma, \mathcal{V}) \times \mathcal{T}_s(\Sigma, \mathcal{V})$, avec $s \in \mathcal{S}$, notée $t = u$.

Intuitivement, une équation conditionnelle est une équation qui n'est vraie que si une certaine condition est vraie.

Définition 3.8 (équation conditionnelle). Soient $\Sigma = (\mathcal{S}, \mathcal{F})$ une signature et $\mathcal{V}$ un ensemble dénombrable de variables tels que $\mathcal{F}$ et $\mathcal{V}$ sont disjoints. Une *équation conditionnelle* est le triplet de termes c, t et u, noté $c \Rightarrow t = u$, tels que c est de type booléen et $t, u \in \mathcal{T}_s(\Sigma, \mathcal{V})$, avec $s \in \mathcal{S}$. Elle signifie que pour toute substitution σ, si $\sigma(c) = \text{vrai}$ alors $\sigma(t) = \sigma(u)$.

Nous donnons encore la définition d'une substitution qui met en évidence le rôle des variables.

Définition 3.9 (substitution). Soient $\Sigma = (\mathcal{S}, \mathcal{F})$ une signature et $\mathcal{V}$ un ensemble dénombrable de variables tels que $\mathcal{F}$ et $\mathcal{V}$ sont disjoints. Une *substitution* est une fonction $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ telle que $\sigma(x) = x$ sauf pour un ensemble fini de variables noté $\mathrm{Dom}(\sigma)$.

Note 3.10. Une substitution σ est étendue en un endomorphisme $\hat{\sigma}$ de $\mathcal{T}(\Sigma, \mathcal{V})$ par les règles :

- si $x \in \mathcal{V}$, alors $\hat{\sigma}(x) = \sigma(x)$;
- si $t = f(t_1, \ldots, t_n)$, alors $\hat{\sigma}(t) = f(\hat{\sigma}(t_1), \ldots, \hat{\sigma}(t_n))$.

Pour simplifier la notation, nous écrirons σ pour désigner indifféremment σ ou son extension $\hat{\sigma}$.

Définition 3.11 (instance). Soient t et $u \in \mathcal{T}(\Sigma, \mathcal{V})$. t est une instance de u si et seulement si il existe une substitution σ telle que $\sigma(u) = t$.

Définition 3.12 (unificateur). Soient t et $u \in \mathcal{T}(\Sigma, \mathcal{V})$. t et u sont dits *unifiables* s'il existe une substitution σ, appelée *unificateur* de t et u, telle que $\sigma(t) = \sigma(u)$.

De plus, σ s'appelle un *unificateur le plus général* de t et u, si pour tout unificateur θ de t et u il existe une substitution δ telle que $\theta = \delta \circ \sigma$.

3.1.2 Règles d'inférence

Les équations peuvent être utilisées pour transformer des termes en d'autres termes équivalents en remplaçant une instance du membre gauche par l'instance correspondante du membre droit et *vice versa*. Cette substitution de termes égaux est le mécanisme clé de la logique équationnelle, et non pas le *modus ponens* comme en logique du premier ordre. Ce mécanisme déductif constitue un système de preuve au niveau syntaxique par lequel les nouvelles équations générées sont des théorèmes.

Les axiomes et règles d'inférence de la logique équationnelle conditionnelle sont présentés Figure 3.1 [Klop, 1992].

Les règles, ou *séquents*, séparent par un trait horizontal les hypothèses, au-dessus, de la conclusion : si les hypothèses peuvent être dérivées par l'application des règles d'inférence, alors la conclusion peut aussi être dérivée. L'ensemble des équations qu'il est possible de déduire depuis un ensemble d'équations E par fermeture par les règles d'inférence constitue une *théorie* dont les équations de E sont les *axiomes*.

Définition 3.13 (théorie). Soit $\mathcal{T}(\Sigma, \mathcal{V})$ l'ensemble des termes engendré par l'ensemble de variables $\mathcal{V}$ sur la signature Σ. La *théorie*, dans la logique équationnelle, induite par l'ensemble d'équations E, appelées *axiomes*, est la relation $=_E$ définie comme suit :

$$=_E := \{(t, u) \in \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V}) \mid E \vdash t = u\}.$$

3.1.3 Liens avec la logique du premier ordre

Dans les chapitres suivants, nous utiliserons le langage de la logique équationnelle pour donner une sémantique aux programmes impératifs. Cependant, il sera parfois nécessaire, ou simplement plus commode, de raisonner dans une logique plus générale telle que la logique du premier ordre. Nous le ferons notamment lorsque nous utiliserons le système de preuve PVS qui est basé sur une logique classique d'ordre supérieur (*cf.* Partie III).

$$t = u, t' = u' \Rightarrow t = u$$
$$t = t$$
$$t = v, u = v \Rightarrow t = u$$
$$t_1 = u_1, \cdots, t_n = u_n \Rightarrow f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)$$
pour tout f d'arité n

$$\frac{F \Rightarrow t' = u', E, t' = u' \Rightarrow t = u}{E, F \Rightarrow t = u}$$

$$\frac{E \Rightarrow t = u}{\sigma(E) \Rightarrow \sigma(t) = \sigma(u)}$$
pour toute substitution σ

Avec $E = \{t_1 = u_1, \ldots, t_n = u_n\}(n \geq 0)$, $\sigma(E) = \{\sigma(t_1) = \sigma(u_1), \ldots, \sigma(t_n) = \sigma(u_n)\}$,
$F = \{t'_1 = u'_1, \ldots, t'_m = u'_m\}(m \geq 0)$.

FIG. 3.1 – Axiomes et règles d'inférence de la logique équationnelle conditionnelle

Nous donnons ici un rapide aperçu de la logique du premier ordre que nous mettons en parallèle avec la logique équationnelle.

La logique équationnelle admet pour seul prédicat l'égalité. La logique du premier ordre étend la logique équationnelle en permettant la déclaration d'un ensemble dénombrable de symboles de *prédicats* qui dénotent des relations. Les équations de la logique équationnelle sont implicitement universellement quantifiées. La logique du premier ordre utilise explicitement les *quantificateurs* universel ($\forall$) et existentiel ($\exists$). De plus, la logique du premier ordre introduit aussi les *connecteurs* logiques de conjonction ($\wedge$), disjonction ($\vee$), négation ($\neg$) et implication ($\Rightarrow$).

Les règles d'inférence de la logique équationnelle peuvent être transformées en un schéma d'axiomes ayant la forme d'implications sur lesquelles la règle *modus ponens* de la logique du premier ordre pourra être appliquée. Un résultat d'importance est que les règles de la logique équationnelle sont consistantes (non contradictoires) dans la logique du premier ordre.

3.2 Systèmes de réécriture

Le mécanisme déductif de la logique équationnelle présenté Section 3.1.2 fournit une méthode pour dériver les conséquences logiques d'un ensemble d'équations, c'est-à-dire, pour montrer que deux termes sont égaux dans la théorie définie par l'ensemble d'équations. Cependant, cette méthode n'est pas efficace. L'idée à l'origine des systèmes de réécriture est de pallier cet inconvénient en restreignant les règles d'inférence de la logique équationnelle.

3.2.1 Réécriture

Le principe de la réécriture est d'orienter les équations $t = u$ en des règles $t \to u$ de façon à ce que la substitution des termes égaux ne soit plus symétrique mais se fasse dans une direction privilégiée : des instances de t peuvent être remplacées par les instances correspondantes de u mais pas l'inverse.

Définition 3.14 (règle et système de réécriture). Une *règle de réécriture* est un couple de termes $(l, r) \in \mathcal{T}_s(\Sigma, \mathcal{V}) \times \mathcal{T}_s(\Sigma, \mathcal{V})$, noté $l \to r$, tel que l ne soit pas une variable et que toute variable de r figure également dans l. Un *système de réécriture* est un ensemble de règles de réécriture.

Définition 3.15 (relation de réduction). Soit $\mathcal{R}$ un système de réécriture. La *relation de réduction* (ou de *réécriture*) $\to_{\mathcal{R}}$ est la plus petite relation sur $\mathcal{T}(\Sigma, \mathcal{V})$ qui contient $\mathcal{R}$ et qui est

- *monotone* : pour tout $n \geq 0$, si $f(t_1, \ldots, t_i, l, t_{i+1}, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$,

$$l \to_{\mathcal{R}} r \text{ implique } f(t_1, \ldots, t_i, l, t_{i+1}, \ldots, t_n) \to_{\mathcal{R}} f(t_1, \ldots, t_i, r, t_{i+1}, \ldots, t_n) \ ;$$

- *stable par substitution* : pour toute substitution σ, $l \to_{\mathcal{R}} r$ implique $\sigma(l) \to_{\mathcal{R}} \sigma(r)$.

Ainsi, t se *réécrit* en u selon $\mathcal{R}$, noté $t \to_{\mathcal{R}} u$, si $t|_o \equiv \sigma(l)$ et $u \equiv t[\sigma(r)]_o$, pour une occurrence o du terme t, une substitution σ et une règle $l \to r \in \mathcal{R}$. Autrement dit, la règle de réécriture $l \to r$ est *appliquée* à un terme t en remplaçant un sous-terme de t qui est une instance de l (*e.g.*, $\sigma(l)$) par l'instance correspondante de r (*e.g.*, $\sigma(r)$).

Note 3.16. Nous noterons $\to^*_{\mathcal{R}}$ la fermeture réflexive et transitive de la relation binaire $\to_{\mathcal{R}}$.

Le processus de réécriture est l'application répétée des règles d'un système de réécriture à un terme. La suite des applications de règle est appelée une *séquence de réécriture*. L'usage typique est de réécrire un terme jusqu'à ce qu'aucune règle ne puisse plus être appliquée : nous dirons alors que le terme est en *forme normale* ou encore qu'il a été *normalisé*.

Définition 3.17 (forme normale). Soit $\mathcal{R}$ un système de réécriture. Un terme t est *en forme normale* pour $\mathcal{R}$ si et seulement si il n'existe aucun terme u tel que $t \to_{\mathcal{R}} u$. Un terme u est *une forme normale de* t pour $\mathcal{R}$ si et seulement si $t \to^*_{\mathcal{R}} u$ et u est en forme normale.

L'implantation du processus de réécriture d'un terme nécessite un algorithme d'*appariement* (*matching*) ainsi qu'une *stratégie de normalisation* pour le choix des règles à appliquer et des termes sur lesquels appliquer les règles.

Un exemple de stratégie de normalisation est de parcourir le terme à réécrire en profondeur d'abord en essayant d'appliquer les règles du système de réécriture, l'une après l'autre, à chaque sous-terme.

Le problème de l'appariement de deux termes t et u consiste à trouver une substitution σ telle que $\sigma(t) = u$. L'appariement peut être considéré comme un cas particulier, plus simple, de l'*unification syntaxique* dans lequel l'un des deux termes à unifier (u dans notre exemple) est clos : les éventuelles variables de ce terme sont traitées comme des constantes. Ce problème est décidable et il existe des algorithmes performants pour le résoudre. Le rôle de l'algorithme d'appariement dans le processus de réécriture est de trouver une substitution, si elle existe, qui permette d'appliquer une règle de réécriture à un terme donné.

3.2.2 Terminaison

Nous nous intéressons maintenant à quelques propriétés des systèmes de réécriture en relation avec la forme normale des termes. En particulier, nous voudrions dans la suite que le processus de réécriture termine, quels que soient le terme en entrée et l'ordre d'application des règles, de sorte que tout terme ait au moins une forme normale.

Définition 3.18 (terminaison). Un système de réécriture $\mathcal{R}$ *termine* (ou est *terminant*) s'il n'existe pas de séquence de réécriture infinie $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} \cdots$

Le problème de la terminaison d'un système de réécriture est indécidable en général. En pratique, la terminaison d'un système de réécriture se montre en exhibant un *ordre de réduction* sur l'ensemble des termes. Parmi les nombreux ordres de réduction qui ont été définis, nous présentons l'*ordre lexicographique des chemins* (*lexicographic path order*). La première mention de cet ordre apparaît dans [Kamin et Levy, 1980], mais son origine remonte aux travaux de N. Dershowitz sur les *ordres de simplification* et l'*ordre récursif des chemins* (*cf.* l'article récapitulant ces travaux [Dershowitz, 1982]).

Définition 3.19 (ordre de réduction). Une relation d'ordre stricte $>$ sur $\mathcal{T}(\Sigma, \mathcal{V})$ est appelée un *ordre de réécriture* si et seulement si elle est

- *monotone* : pour tout $n \geq 0$, si $f(t_1, \ldots, t_i, l, t_{i+1}, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$,

$$l > r \text{ implique } f(t_1, \ldots, t_i, l, t_{i+1}, \ldots, t_n) > f(t_1, \ldots, t_i, r, t_{i+1}, \ldots, t_n) \ ;$$

- *stable par substitution* : pour toute substitution σ, $l > r$ implique $\sigma(l) > \sigma(r)$.

Un *ordre de réduction* est un ordre de réécriture *bien fondé* (*i.e.*, il n'existe pas de suite infinie de termes $t_0 > t_1 > \cdots$).

Le théorème suivant lie ordres de réduction et terminaison des systèmes de réécriture :

Théorème 3.20. *Un système de réécriture $\mathcal{R}$ termine si et seulement si il existe un ordre de réduction $>$ qui satisfait $l > r$ pour toute règle $l \to r \in \mathcal{R}$.*

Les *ordres de simplification* forment une classe particulièrement utile d'ordres de réduction. Notamment, un ordre lexicographique des chemins est un ordre de simplification.

Définition 3.21 (ordre de simplification). Une relation d'ordre stricte $>$ sur $\mathcal{T}(\Sigma, \mathcal{V})$ est appelée un *ordre de simplification* si et seulement si c'est un ordre de réécriture qui possède la *propriété de sous-terme* (*subterm property*) :

$$\text{pour tout terme } t \in \mathcal{T}(\Sigma, \mathcal{V}) \text{ et toute occurrence } o \in \mathrm{Occ}(t) - \{\epsilon\}, t > t|_o.$$

Théorème 3.22. *Tout ordre de simplification sur $\mathcal{T}(\Sigma, \mathcal{V})$ est un ordre de réduction.*

Définition 3.23 (ordre lexicographique des chemins). Soient $\Sigma = (\mathcal{F}, \mathcal{S})$ une signature et $>$ une relation d'ordre stricte sur $\mathcal{F}$. L'*ordre lexicographique des chemins* $>_{lpo}$ sur $\mathcal{T}(\Sigma, \mathcal{V})$ induit par $>$ est défini comme suit : $t >_{lpo} u$ si et seulement si

1. u est une variable qui figure aussi dans t et $t \neq u$, ou
2. $t = f(t_1, \ldots, t_m)$, $u = g(u_1, \ldots, u_n)$ et
   (a) il existe i, $1 \leq i \leq m$, tel que $t_i \geq_{lpo} u$, ou
   (b) $f > g$ et $t >_{lpo} u_j$ pour tout j, $1 \leq j \leq n$, ou
   (c) $f = g$ et
   - $t >_{lpo} u_j$ pour tout j, $1 \leq j \leq n$, et
   - il existe i, $1 \leq i \leq m$, tel que $t_1 = u_1, \ldots t_{i-1} = u_{i-1}$ et $t_i >_{lpo} u_i$.

Note 3.24. Nous employons la notation $t \geq_{lpo} u$ avec le sens usuel de $t >_{lpo} u \vee t = u$.

Théorème 3.25. *Soit $\Sigma = (\mathcal{F}, \mathcal{S})$ une signature. Pour toute relation d'ordre stricte $>$ sur $\mathcal{F}$, l'ordre lexicographique des chemins induit $>_{lpo}$ est un ordre de simplification sur $\mathcal{T}(\Sigma, \mathcal{V})$.*

3.2.3 Convergence

La propriété de convergence d'un système de réécriture nécessite l'introduction de celle de *confluence*. Cette dernière garantit qu'un terme n'a qu'une seule forme normale.

Définition 3.26 (confluence). Soit $\mathcal{R}$ un système de réécriture. Pour deux termes t et u, nous écrivons $t \downarrow_{\mathcal{R}} u$ pour dénoter qu'ils sont *joignables*, *i.e.*, il existe un terme v tel que $t \xrightarrow{*}_{\mathcal{R}} v$ et $u \xrightarrow{*}_{\mathcal{R}} v$. Le système de réécriture $\mathcal{R}$ est *confluent* si et seulement si pour tout terme t tel que $t \xrightarrow{*}_{\mathcal{R}} u$ et $t \xrightarrow{*}_{\mathcal{R}} v$, nous avons $u \downarrow_{\mathcal{R}} v$.

Le problème de la confluence d'un système de réécriture est indécidable dans le cas général, mais s'avère décidable lorsque le système est terminant.

Définition 3.27 (paire critique). Soient $l_1 \to r_1$ et $l_2 \to r_2$ deux règles, d'un système de réécriture $\mathcal{R}$, dont les variables ont éventuellement été renommées de façon à ce qu'aucune variable ne soit commune aux deux règles. Soit une occurrence $o \in \mathrm{Occ}(l_1)$ telle que $l_1|_o$ ne soit pas une variable. Soit θ un unificateur le plus général de $l_1|_o$ et l_2. La paire $(\theta(r_1), \theta(l_1)[\theta(r_2)]_o)$ est une *paire critique* de $\mathcal{R}$ pour les deux règles.

Théorème 3.28. *Un système de réécriture terminant $\mathcal{R}$ est confluent si et seulement si $t \downarrow_{\mathcal{R}} u$ pour toutes ses paires critiques (t, u).*

La conjonction des propriétés de terminaison et de confluence pour un système de réécriture assure que tout terme possède par ce système de réécriture une et une seule forme normale.

Définition 3.29 (convergence). Un système de réécriture est *convergent* si et seulement si il est confluent et termine.

3.2.4 Systèmes de réécriture conditionnelle

L'expressivité des systèmes de réécriture peut être augmentée en considérant des équations conditionnelles à la place de l'ensemble d'équations conventionnel. Cette extension, appelée *réécriture conditionnelle* (*conditional term rewriting*), oblige à réviser les critères de terminaison et de confluence. En effet, les résultats établis pour le cas inconditionnel ne sont pas directement applicables à cette extension.

Définition 3.30 (règle et système de réécriture conditionnelles). Une *règle conditionnelle* est le triplet de termes c, l et r, noté $c \Rightarrow l \rightarrow r$, avec $l, r \in \mathcal{T}_s(\Sigma, \mathcal{V})$ et c de la forme $t_1 = u_1 \wedge \cdots \wedge t_n = u_n$ pour $n \geq 0$. Un *système de réécriture conditionnelle* est un ensemble de règles de réécriture.

Une règle conditionnelle est utilisée pour réécrire un terme en remplaçant une instance du membre gauche l par l'instance correspondante du membre droit r pourvu que l'instance correspondante de la condition c soit satisfaite.

Note 3.31.

1. Nous simplifierons parfois l'écriture des conditions $P =$ vrai en P, et $P =$ faux en $\neg P$, pour un prédicat P et les symboles de constante de type booléen vrai et faux.

2. Plusieurs sémantiques peuvent être attribuées aux systèmes de réécriture conditionnelle selon l'interprétation du symbole $=$ dans les conditions. Par exemple, $t = u$ peut être interprété comme $t \downarrow u$ et les conditions sont alors vérifiées avec le système de réécriture lui-même. Mais il peut aussi s'agir de l'égalité au sens courant et les règles du système de réécriture sont utilisées dans les deux directions pour vérifier la condition.

La question de la terminaison des système de réécriture conditionnelle est compliquée par l'évaluation récursive des conditions à chaque étape de réécriture. Le concept de système *décroissant* (*decreasing*) capture l'idée de la terminaison de l'évaluation récursive des termes.

Définition 3.32 (système décroissant). Un système de réécriture conditionnelle est *décroissant* s'il existe un ordre bien fondé $>$ contenant la relation de réécriture $\rightarrow$ et tel que :

- l'ordre $>$ possède la propriété de sous-terme ($t > u$ si u est un sous-terme de t qui n'est pas t) ;
- pour chaque règle $t_1 = u_1 \wedge \cdots \wedge t_n = u_n \Rightarrow l \rightarrow r$ avec $n \geq 0$ et pour toute substitution σ, $\sigma(l) > \sigma(t_1), \ldots, \sigma(l) > \sigma(u_n)$.

Théorème 3.33. *Un système de réécriture décroissant est terminant.*

La notion de paire critique est étendue à celle de *paire critique conditionnelle* sur laquelle repose, similairement au cas inconditionnel, les critères de confluence des systèmes de réécriture conditionnelle.

Définition 3.34 (paire critique conditionnelle). Soient $c_1 \Rightarrow l_1 \rightarrow r_1$ et $c_2 \Rightarrow l_2 \rightarrow r_2$ deux règles, d'un système de réécriture conditionnelle $\mathcal{R}$, dont les variables ont éventuellement été renommées de façon à ce qu'aucune variable ne soit commune aux deux règles. Soit une occurrence $o \in \mathrm{Occ}(l_1)$ telle que $l_1|_o$ ne soit pas une variable. Soit θ un unificateur le plus général de $l_1|_o$ et l_2. L'équation conditionnelle $\theta(c_1 \wedge c_2) \Rightarrow \theta(r_1) = \theta(l_1)[\theta(r_2)]_o$ est une *paire critique conditionnelle* de $\mathcal{R}$ pour les deux règles.

Théorème 3.35. *Un système décroissant $\mathcal{R}$ est confluent (et donc, convergent) si pour toutes ses paires critiques $c \Rightarrow t = u$ et pour toute substitution σ telle que la condition $\sigma(c)$ est satisfaite, nous avons $\sigma(t) \downarrow_{\mathcal{R}} \sigma(u)$.*

Note 3.36. Dans le cas des systèmes décroissants, l'interprétation du symbole $=$ dans les conditions comme le prédicat d'égalité ou comme la relation $\downarrow$ ne fait pas de différence relativement à la confluence (et donc, à la convergence) : la confluence du système de réécriture selon l'une des formulations entraîne la confluence du système selon l'autre.

Deuxième partie

Le système $\boldsymbol{SOS}\mathrm{Sub}_{\mathcal{C}}$

Chapitre 4

$SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$

L'objectif du système $SOS\mathrm{Sub}_{\mathcal{C}}$ est de réaliser automatiquement l'*axiomatisation*, dans la logique équationnelle, de programmes impératifs. Nous désignons par axiomatisation la définition équationnelle de fonctions de transfert des données en entrée vers les données en sortie d'un programme.

Le système $SOS\mathrm{Sub}_{\mathcal{C}}$ se décline en deux versions : $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ et $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$. Chaque version se distingue par le langage de programmation qu'elle traite et par la méthode qu'elle met en œuvre pour définir et effectuer l'axiomatisation des programmes sources. Bien que le système $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ accepte le langage le plus riche des deux versions, le système $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ présente un intérêt propre, de par son approche originale basée sur un système de réécriture et sa complète automatisation de l'axiomatisation. En effet, en contrepartie de l'expressivité offerte dans le langage de programmation de $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$, l'axiomatisation automatique des programmes est conditionnée par la vérification en dehors du système de certains prérequis sur les programmes.

Le système $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ est décrit dans le Chapitre 5. L'objet du présent chapitre est d'exposer le fonctionnement du système $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$. Nous entamons la présentation, Section 4.1, par la définition du langage de programmation impératif $\mathrm{Sub}_{\mathcal{C}}$, qui sert de support à notre méthode. Nous poursuivons par la méthode d'axiomatisation en Section 4.2. Nous montrons que l'axiomatisation dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ s'articule autour d'un système de réécriture qui formalise la sémantique du langage $\mathrm{Sub}_{\mathcal{C}}$. Nous détaillons ensuite, Section 4.3, cette sémantique et nous commentons les règles de réécriture qui la définissent. La description de la démonstration automatique de la propriété de convergence du système de réécriture en Section 4.4 conclut ce chapitre sur $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$.

4.1 Langage $\mathrm{Sub}_{\mathcal{C}}$

Les langages informatiques sont souvent classés selon le paradigme qui sous-tend leur interprétation par le programmeur : fonctionnel, logique ou impératif pour les plus classiques. Il serait d'ailleurs plus exact de préciser que les langages de programmation sont souvent multiparadigmes : orienté-objet *et* fonctionnel, ou orienté-aspect, orienté-objet *et* impératif, etc.

Les langages impératifs manipulent un *état* du programme, constitué notamment des valeurs des variables du programme. Ils expriment un calcul par une séquence de modifications de l'état du programme. Ces langages, de loin les plus utilisés en pratique, ne disposent pas d'un modèle mathématique aussi clairement défini que le λ-calcul pour les langages fonctionnels ou la logique des prédicats pour les langages logiques. De ce fait, l'analyse des programmes écrits dans un langage impératif est plus difficile à formaliser.

Une conséquence directe de cette difficulté est que les méthodes de vérification de programmes impératifs ne considèrent qu'un nombre restreint de constructions parmi celles offertes par les langages réels. D'ailleurs, pour la programmation des applications critiques, le langage choisi est souvent très contraignant (voir par exemple les sous-ensembles de langage `C` et Ada utilisés dans l'industrie : MISRA-C [MISRA, 1998] ou SPARK [Barnes, 2003]). Le langage $\text{Sub}_{\mathcal{C}}$ que nous avons défini pour notre méthode n'échappe pas à cette règle : il se limite à un petit ensemble de constructions, dont la sémantique est bien comprise, mais qui est tout de même suffisamment riche pour être aussi puissant qu'une machine de Turing... en plus convivial !

Le langage $\text{Sub}_{\mathcal{C}}$ est un langage impératif typé, inspiré du `C` dont il conserve globalement la syntaxe. Les principales caractéristiques du langage sont les suivantes :

- l'affectation (=) ;
- la notion de sous-programme : récursivité, appel par valeur, variables locales et valeur de retour (instruction **`return`**) ;
- des structures de contrôle : la séquence (symbole infixe **`;`**), les instructions **`if`** ... **`else`** et **`while`** ;
- deux types de données prédéfinis : les entiers (**`int`**) et les listes (**`list`**) ;
- les opérateurs usuels du `C` :
  - arithmétiques ;
  - logiques (ET (**`&&`**), OU (||), NON (**`!`**)) ;
  - relationnels (égal (==), différent (**`!`**=), ...) ;
- des opérateurs sur les listes :
  - **`element(L)`** retourne le premier élément de la liste `L` (la tête de la liste) ;
  - **`next(L)`** retourne la liste `L` sans son premier élément (la queue de la liste) ;
  - **`add(elt, L)`** retourne une nouvelle liste dont la tête est l'élément `elt` et la queue est la liste `L` ;
  - **`NULL`** représente une liste vide.

Les structures de contrôle peuvent être emboîtées sans restriction. Une seule instruction `return` est autorisée par fonction : elle se trouve impérativement à la fin du corps de la fonction.

Cependant, afin de bien maîtriser la sémantique du langage et de la garder aussi simple que possible, plusieurs attributs courants dans les langages impératifs ne sont pas disponibles en Sub$_C$:

- pas d'instruction de saut du type `goto` ;
- pas de variable globale ;
- pas de type défini par le programmeur ;
- pas de pointeur directement accessible au programmeur (même si l'implantation du type prédéfini `list` peut recourir à des pointeurs, ceux-ci sont masqués au programmeur) ;
- pas de conversion de type.

Dans la famille des langages impératifs, les instructions de saut, du type `goto`, sont un héritage des premières générations. Leur utilisation inconsidérée a été assez vite décriée et le titre d'un article de E. W. Dijkstra, *Go To Statement Considered Harmful*[7] [Dijkstra, 1968], a souvent été pris au pied de la lettre par les thuriféraires de la programmation structurée. Néanmoins, les instructions de saut n'ont pas complètement disparu des langages de programmation modernes et existent sous des formes spécialisées dont la sémantique est mieux contrôlée (*e.g.*, la gestion des exceptions). Puisque théoriquement les autres structures de contrôle permettent de se passer d'instruction `goto` [Böhm et Jacopini, 1966; Ashcroft et Manna, 1972], et puisque dans le cas général cette dernière rend plus difficile la compréhension des programmes, nous ne l'avons pas incluse dans le langage Sub$_C$.

Les variables globales ont aussi été l'objet d'une accusation similaire[8] par W. Wulf et M. Shaw (1973) et ne sont pas autorisées dans notre langage.

L'absence de type défini par le programmeur présente plusieurs inconvénients, *e.g.*, mauvaise lisibilité et faible réutilisabilité des programmes. Cependant, ceci ne limite pas en théorie l'expressivité de Sub$_C$, car un codage judicieux des autres types à partir des types prédéfinis (entiers et listes) est toujours possible (*cf.* les Sections 6.4.1 et 6.4.2 où des arbres et des graphes sont codés en Sub$_C$).

Enfin, le cas des pointeurs et des autres mécanismes susceptibles d'entraîner des effets de bord est discuté dans le cadre du système *SOS*Sub$_C^{ext}$ (*cf.* Chapitre 5).

La grammaire EBNF du langage Sub$_C$ est donnée dans la Figure 4.1, pour les constructions du langage, et dans la Figure 4.2, pour les expressions du langage. Un programme est donc une collection de fonctions, ou de procédures, puisque le type de retour `void` est admis, même si dans ce dernier cas aucune équation ne sera produite. Notre méthode doit pouvoir s'appliquer à des programmes incomplets, *e.g.*, des bibliothèques de fonctions, aussi, aucune fonction particulière ne dénote le point d'entrée dans le programme. La définition précise de chacune des constructions du langage est donnée de façon incrémentale dans la suite de ce chapitre.

[7] Pour l'anecdote, ce titre fameux n'est d'ailleurs pas l'original de E. W. Dijkstra, qui avait plus modérément intitulé son article *A case against the goto statement*, mais aurait été donné par le journal où a paru l'article (dont l'éditeur était alors Niklaus Wirth) selon les notes de E. W. Dijkstra (2001).

[8] Au moins dans le titre de l'article cité : *Global variable considered harmful*.

program ::= function*

function ::= returnType ident *'('* (*'void'* | argumentList)? *')'* (*';'* | functionBody)

argumentList ::= argumentDeclaration (*','* argumentDeclaration)*

argumentDeclaration ::= varType ident

functionBody ::= *'{'* variableDeclaration* statementList returnStatement? *';'* *'}'*

variableDeclaration ::= varType variableInitialisation (*','* variableInitialisation)* *';'*

variableInitialisation ::= ident (*'='* expression)?

statementList ::= statement*

statement ::= assignStatement *';'*
| whileStatement *';'*
| branchStatement *';'*
| *'{'* statementList *'}'*

assignStatement ::= ident *'='* expression

returnStatement ::= *'return'* expression

whileStatement ::= *'while'* *'('* condition *')'* statement

branchStatement ::= *'if'* *'('* condition *')'* statement (*'else'* statement)?

returnType ::= *'void'* | varType

varType ::= *'int'* | *'list'*

FIG. 4.1 – Grammaire EBNF des constructions de $Sub_{\mathcal{C}}$

condition ::= andExpression (**'||'** andExpression)*

andExpression ::= equalExpression (**'&&'** equalExpression)*

equalExpression ::= relExpression ((**'=='** | **'!='**) relExpression)*

relExpression ::= expression (relOperator expression)*

expression ::= mulExpression (**'+'** | '−') mulExpression)*

mulExpression ::= unExpression (('*' | '/') unExpression)*

unExpression ::= unOperator? factor

factor ::= *'('* expression *')'* | definedName | integer | ***'NULL'***

definedName ::= ident (*'('* effectiveArgumentList? *')'*)?
| ***'add'*** *'('* expression *','* definedName *')'*
| ***'element'*** *'('* definedName *')'*
| ***'next'*** *'('* definedName *')'*

effectiveArgumentList ::= expression (*','* expression)*

unOperator ::= *'!'* | '−' | **'+'**

relOperator ::= **'<'** | **'<='** | **'>'** | **'>='**

integer ::= ***un entier***
ident ::= ***un identificateur***

FIG. 4.2 – Grammaire EBNF des expressions de Sub$_C$

Les expressions forment deux ensembles : celui des expressions conditionnelles (appelé condition dans la grammaire) et celui des expressions de listes et des expressions arithmétiques (appelé expression dans la grammaire), qui est un sous-ensemble du premier.

4.2 Axiomatisation

Dans notre approche, nous travaillons sur le code source écrit par des programmeurs dans le langage Sub$_{\mathcal{C}}$. Le code des programmes ne comporte pas d'annotations spécifiques à notre approche. Le système SOSSub$_{\mathcal{C}}$ exprime la sémantique d'un programme P par un ensemble d'équations conditionnelles E_P qui sont la définition de fonctions de transfert f^t des données en entrée D_e de P vers ses données en sortie D_s. Chaque fonction de transfert $f^t_{d_s}$ correspond à une donnée en sortie d_s de P.

La définition de $f^t_{d_s}$ est telle que le résultat, pour d_s, de l'exécution de P sur les données en entrée D_e est le même, à une traduction ϕ des symboles près, que le résultat obtenu de la déduction équationnelle (*i.e.*, la forme normale) dans E_p à partir de $f^t_{d_s}(\phi(D_e))$.

Les équations E_P forment les axiomes d'une théorie de P dans la logique équationnelle. De ce fait, nous bénéficions, à la fois d'une sémantique dénotationnelle du programme, grâce au modèle des algèbres abstraites de la logique équationnelle, et d'une sémantique opérationnelle pour le programme, puisque la théorie de la réécriture fournit une définition opératoire de l'évaluation du programme.

Le rôle de SOSSub$^{\mathcal{R}}_{\mathcal{C}}$ est de réaliser automatiquement l'axiomatisation qui traite le problème suivant :

Axiomatisation
`Entrée` : Un programme Sub$_{\mathcal{C}}$ P. `Sortie` : Les axiomes d'une théorie équationnelle du programme $< \sum_P, E_P >$. Avec $\sum_P$ et E_P, respectivement, la signature et la sémantique de P en logique équationnelle.

Le cœur de SOSSub$^{\mathcal{R}}_{\mathcal{C}}$ est le système de réécriture $SOS_{\mathcal{R}}$: les règles de $SOS_{\mathcal{R}}$ définissent la sémantique du langage Sub$_{\mathcal{C}}$. Le choix d'un système de réécriture pour la spécification de la sémantique d'un langage offre deux avantages :

- la sémantique claire de la réécriture permet de *formaliser* la spécification et aide ainsi à la compréhension et à la vérification de celle-ci ;
- la spécification est *exécutable* puisque la réécriture jouit d'une interprétation opérationnelle.

Nous montrerons, de plus, que le système de réécriture $SOS_{\mathcal{R}}$ possède la propriété de convergence, qui le rend apte à évaluer les programmes Sub$_{\mathcal{C}}$ (*cf.* Section 4.4). La signature Σ_{sos} et les règles du système de réécriture $SOS_{\mathcal{R}}$ se trouvent en Annexe A. Nous détaillons

et explicitons les règles du système dans la Section 4.3. La signature est introduite au fil de la description de l'axiomatisation dans les Sections 4.2.1, 4.2.2 et 4.2.3.

La sémantique du langage $\text{Sub}_{\mathcal{C}}$ est exprimée par les modifications des constructions du langage sur un ensemble d'équations caractéristiques d'un programme : l'*environnement* du programme (*cf.* Section 4.2.1). L'axiomatisation du code source des programmes $\text{Sub}_{\mathcal{C}}$ est fondée sur le concept d'environnement. Elle comprend trois étapes :

1. chaque sous-programme est tout d'abord traduit en un *terme bien formé* de Σ_{sos} : ce terme constitue une entrée convenable pour la réécriture par $SOS_{\mathcal{R}}$ (*cf.* Section 4.2.2) ;

2. ensuite, le terme obtenu à la première étape est normalisé par réécriture avec $SOS_{\mathcal{R}}$: sa forme normale est l'*environnement final* du sous-programme (*cf.* Section 4.2.3) ;

3. enfin, nous extrayons de l'environnement final les informations nécessaires à la formulation des *équations* du sous-programme (*cf.* Section 4.2.4).

Au final, l'évaluation d'un programme selon les règles de la sémantique du langage produit un ensemble d'équations qui, interprété dans la logique équationnelle, constitue la sémantique du programme.

Avant d'approfondir le contenu de chacune de ces étapes, nous introduisons le concept d'environnement de sous-programme pour $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$.

4.2.1 Environnements

L'environnement d'un sous-programme est, en substance, un *état* du sous-programme au cours d'une *évaluation symbolique* des instructions qui le composent. L'évaluation est symbolique parce que faite sur des entrées du sous-programme non définies (*i.e.*, dont la valeur n'est pas connue), et donc encore manipulées sous forme de symboles, comme des abstractions des entrées réelles. C'est-à-dire, par exemple, que la valeur des variables d'un sous-programme $\text{Sub}_{\mathcal{C}}$ s'exprime par une fonction des entrées du sous-programme, et non par une valeur constante.

L'environnement d'un sous-programme contient la valeur (symbolique) de toutes les variables et tous les paramètres du sous-programme pour chaque *chemin d'exécution* dans celui-ci. Il peut contenir, en plus, ce que nous appelons une *fermeture de boucle* (*while closure*) et qui correspond à un nouveau sous-programme qui possède son propre environnement.

Un chemin d'exécution dans un sous-programme est défini comme la suite des instructions exécutées pour des valeurs d'entrée données. Les instructions de contrôle de flot (instructions conditionnelles et itératives) déterminent les différents chemins possibles. Nous verrons par la suite que, dans notre méthode, les boucles sont traitées comme des appels de sous-programme. De ce fait, les instructions de branchement conditionnelles sont les seules à scinder le sous-programme en chemins. Pour nous, ceci a deux conséquences favorables :

;;Types
;;env = les environnements
;;env_elt = les éléments d'environnement
;;cond = les conditions
;;exp = les expressions (exp ⊃ var)
;;var = les variables
;;var_list = les listes de variables
;;id = les identificateurs (id ⊃ var)
;;nat = les entiers naturels

;;Constructeurs des environnements
$C\_Env : env\_elt \times env \rightarrow env$
$C\_Empty\_Env : \rightarrow env$
$C\_Choice : env \times env \rightarrow env$
$C\_Branch : cond \times env \rightarrow env$

;;Constructeurs des éléments d'environnement
$C\_Pair : id \times exp \rightarrow env\_elt$
$C\_While\_Closure : nat \times cond \times env \times var\_list \rightarrow env\_elt$

;;Constructeurs des listes de variables
$C\_L\_Var : var \times var\_list \rightarrow var\_list$
$C\_Empty\_L\_Var : \rightarrow var\_list$

;;Constructeur des fonctions de boucle
$C\_Loop : nat \times var \times env \rightarrow exp$

FIG. 4.3 – Signature des environnements de $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$ dans Σ_{sos}

- Le nombre des chemins possibles est fini : un environnement pourra donc facilement tous les contenir.
- Nous pouvons associer à chaque chemin une condition simple qui caractérise les contraintes que les données d'entrée doivent respecter pour que ce chemin soit exécuté. Cette condition sera formée de la conjonction (ET logique) de toutes les conditions des instructions de branchement rencontrées sur le chemin considéré.

Nous définissons maintenant formellement le concept d'environnement de $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$: un environnement est représenté par un terme de Σ_{sos}. Nous exhibons en Figure 4.3 les symboles de Σ_{sos} qui sont les constructeurs des termes du type des environnements.

Définition 4.1 (environnement). Un *environnement* est un terme bien formé de type *env* sur la signature définie par la Figure 4.3.

Dans l'environnement d'un sous-programme, un chemin d'exécution est représenté par un terme *C_Branch* qui réunit la condition du chemin et son environnement spéci-

env ::= choice | env_elt_list

env_elt_list ::= *'C_Env' '('* env_elt *','* env_elt_list *')'* | *'C_Empty_Env'*

env_elt ::= while_closure | pair

choice ::= *'C_Choice' '('* branch *','* branch *')'*

branch ::= choice | *'C_Branch' '('* cond *','* env *')'*

while_closure ::= *'C_While_Closure' '('* nat *','* cond *','* env *','* var_list *')'*

pair ::= *'C_Pair' '('* var *','* exp *')'*

var_list ::= *'C_L_Var' '('* var *','* var_list *')'* | *'C_Empty_L_Var'*

loop ::= *'C_Loop' '('* nat *','* var *','* env *')'*

exp ::= *une* expression *Sub$_C$ dans laquelle le terme* loop
apparaît comme une alternative supplémentaire de factor
cond ::= *une condition pour* Σ_{sos}
var ::= *une variable*
nat ::= *un entier naturel*

FIG. 4.4 – Grammaire EBNF des environnements de $SOS\mathrm{Sub}_C^{\mathcal{R}}$

fique. Les différents chemins du sous-programme sont regroupés par les termes *C_Choice*. L'environnement spécifique à un chemin peut contenir une liste de paires (*C_Pair*), qui représente l'état des variables dans ce chemin, et des termes *C_While_Closure*, qui sont les fermetures des boucles rencontrées sur ce chemin.

Pour faciliter la compréhension de ce que sont les environnements, nous en proposons une autre description sous la forme d'une grammaire EBNF en Figure 4.4. Cette grammaire définit les environnements lorsqu'ils sont en forme normale.

4.2.2 Les programmes sont des termes

Un programme, afin de pouvoir être réécrit par le système $SOS_{\mathcal{R}}$, doit être un terme de la signature du système, à savoir Σ_{sos}. Nous avons prévu dans Σ_{sos} des symboles de fonctions et des types pour chaque construction du langage Sub_C. Nous les récapitulons dans la Figure 4.5.

Les fonctions et constructeurs usuels des expressions ne sont pas cités dans cette figure, car nous ne leur donnons pas de sémantique particulière dans $SOS_{\mathcal{R}}$. En effet, nous déléguons cette tâche aux systèmes de preuve qui seront utilisés, *in fine*, pour démontrer la correction du programme et dans lesquels ces symboles sont généralement prédéfinis

;;Types
;;stmt = les instructions
;;stmt_list = les listes d'instructions

;;Constructeurs des listes d'instructions
C_L_Stmt : $stmt\_list \times stmt \rightarrow stmt\_list$
C_Empty_L_Stmt : $\rightarrow stmt\_list$

;;Constructeurs des instructions
C_If : $cond \times stmt\_list \times stmt\_list \rightarrow stmt$
C_Assign : $id \times exp \rightarrow stmt$
C_While : $nat \times cond \times stmt\_list \rightarrow stmt$

;;Constructeur des paramètres effectifs
C_EA : $var \rightarrow exp$

;;Constructeurs des conditions
C_And : $cond \times cond \rightarrow cond$
C_Not : $cond \rightarrow cond$

FIG. 4.5 – Signature des constructions du langage $\text{Sub}_{\mathcal{C}}$ dans Σ_{sos}

(*cf.* Chapitres 6 et 7). Les constructeurs *C_And* et *C_Not* ne sont pas confondus avec le **&&** et le ! de $\text{Sub}_{\mathcal{C}}$, afin de permettre une définition éventuellement différente, et leur sémantique ne fait pas non plus partie de $SOS_{\mathcal{R}}$. Les autres symboles servent à représenter les constructions du langage $\text{Sub}_{\mathcal{C}}$ par des termes.

L'objectif de cette **première étape de l'axiomatisation** est le suivant :

Traduction en terme	
`Entrée`	: Un programme $\text{Sub}_{\mathcal{C}}$ P composé de sous-programmes f_i.
`Sortie`	: Des termes $T_P^{f_i}$ sur la signature Σ_{sos}.

L'analyse syntaxique du code source des programmes $\text{Sub}_{\mathcal{C}}$ produit un terme différent pour chaque sous-programme. Le terme est construit avec les symboles de la signature Σ_{sos} en sélectionnant celui qui correspond à la construction du langage $\text{Sub}_{\mathcal{C}}$ analysée. Les correspondances entre constructions $\text{Sub}_{\mathcal{C}}$ et symboles de Σ_{sos} sont définies dans la Figure 4.6. La représentation des listes pour les termes de cette figure, et dans la suite de ce chapitre, est celle de la Note 1.7 (Page 12) : les éléments de la liste sont entre des accolades et séparés par des points . Les constructeurs réellement employés par le système se déduisent du type des éléments, *e.g.*, *C_Env* et *C_Empty_Env* pour les listes de paires des environnements. Nous aurons l'occasion d'illustrer le fonctionnement des règles de correspondance syntaxique dans la section suivante, lorsque nous détaillerons le comportement de $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$ pour chaque construction du langage $\text{Sub}_{\mathcal{C}}$ (*cf.* Section 4.3 et Exemple 4.7).

$$\frac{\texttt{stmts} \longmapsto \textit{stmt\_list} \quad \texttt{type}_t \in \{\textbf{int}, \textbf{list}\}}{\texttt{type}_0\ \texttt{fct\_name}\ (\texttt{type}_1\ \texttt{p}_1, \ldots, \texttt{type}_n\ \texttt{p}_n)\ \{\texttt{stmts}\} \longmapsto \textit{GE(stmt\_list, \{C\_Pair(p}_1\textit{, C\_EA(p}_1\textit{)) } \cdot \ldots \cdot \textit{ C\_Pair(p}_n\textit{, C\_EA(p}_n\textit{))\})}} \text{ (Fct)}$$

$$\frac{\texttt{stmt} \longmapsto \textit{stmt}' \quad \texttt{stmts} \longmapsto \textit{stmt\_list}}{\texttt{stmt; stmts} \longmapsto \{\textit{stmt}' \cdot \textit{stmt\_list}\}} \text{ (Seq)}$$

$$\frac{\textit{nom\_fct} = \text{nom du sous-programme}}{\textbf{return}\ \texttt{exp} \longmapsto \textit{C\_Assign(nom\_fct, exp)}} \text{ (Ret)}$$

$$\frac{}{\texttt{x = y} \longmapsto \textit{C\_Assign(x, y)}} \text{ (Assg)}$$

$$\frac{\texttt{stmts}_1 \longmapsto \textit{stmt\_list}_1 \quad \texttt{stmts}_2 \longmapsto \textit{stmt\_list}_2}{\textbf{if}\ \texttt{(c)}\ \texttt{stmts}_1\ \textbf{else}\ \texttt{stmts}_2 \longmapsto \textit{C\_If(c, stmt\_list}_1\textit{, stmt\_list}_2\textit{)}} \text{ (Cond1)}$$

$$\frac{\texttt{stmts} \longmapsto \textit{stmt\_list}}{\textbf{if}\ \texttt{(c) stmts} \longmapsto \textit{C\_If(c, stmt\_list, \{\})}} \text{ (Cond2)}$$

$$\frac{\texttt{stmts} \longmapsto \textit{stmt\_list} \quad \textit{num} = \text{un numéro de boucle inutilisé}}{\textbf{while}\ \texttt{(cond) stmts} \longmapsto \textit{C\_While(num, cond, stmt\_list)}} \text{ (Iter)}$$

$$\frac{\texttt{type} \in \{\textbf{int}, \textbf{list}\}}{\texttt{type var = val} \longmapsto \textit{C\_Assign(var, val)}} \text{ (VarDecl1)}$$

$$\frac{}{\textbf{int}\ \texttt{var} \longmapsto \textit{C\_Assign(var, 0)}} \text{ (VarDecl2)}$$

$$\frac{}{\textbf{list}\ \texttt{var} \longmapsto \textit{C\_Assign(var, NULL)}} \text{ (VarDecl3)}$$

FIG. 4.6 – Règles de correspondance syntaxique

4.2.3 Les termes sont réécrits en des environnements

Le terme T_P^f obtenu de l'étape précédente pour le sous-programme f associe la séquence d'instructions de f à un environnement initial, dans lequel vont être évaluées les instructions. L'évaluation consiste à appliquer les règles de $SOS_{\mathcal{R}}$ au terme T_P^f.

La **deuxième étape de l'axiomatisation** est donc la suivante :

Normalisation	
`Entrée`	: Un terme T_P^f sur Σ_{sos}.
`Sortie`	: Un terme normalisé, $env_{T_P^f}$, appelé environnement final de f et tel que $T_P^f \downarrow_{SOS_{\mathcal{R}}} env_{T_P^f}$.

Les règles du système de réécriture $SOS_{\mathcal{R}}$ définissent plusieurs opérations dont le rôle est de donner une sémantique aux constructions du langage $\text{Sub}_{\mathcal{C}}$. Ces opérations interprètent les constructions du langage comme des fonctions des environnements vers les environnements. La signature de ces opérations est présentée dans la Figure 4.7.

La définition précise des opérations *GE*, *Comp* et *GL* est donnée par les règles, respectivement, (1–2), (3–10) et (11–12) de $SOS_{\mathcal{R}}$ en Annexe A. Leur sens est expliqué dans la Section 4.3. La plupart des autres opérations décrivent des actions élémentaires sur les environnements. Ainsi :

- *Update_Env*, définie par les règles (13–16), met à jour la valeur d'une variable dans la liste de paires d'un environnement donné ; si la paire correspondant à la variable mise à jour n'existe pas, elle est ajoutée à la liste. Dans le même temps, l'expression qui constitue la valeur de la variable est aussi mise à jour pour refléter la valeur courante des variables dans l'environnement, *i.e.*, les variables que l'expression contient sont remplacées par leur valeur dans l'environnement.

- *Update_Cond*, définie par les règles (17–19), met à jour une condition pour refléter dans la condition la valeur courante des variables dans un environnement donné.

- *GLOV*, définie par les règles (26–28), construit la liste des variables présentes dans la liste de paires d'un environnement donné.

- *Merge_Env* et *Insert_Pair*, définies par les règles (35–36) et (37–40), ajoutent les paires d'un premier environnement à celles du second. Si une variable donnée a déjà une paire dans le second environnement, cette paire est remplacée par celle du premier environnement. Cette opération permet donc de mettre à jour la valeur d'un ensemble de variables (ou d'une seule pour *Insert_Pair*), sans que l'expression qui leur est associée soit, elle aussi, mise à jour (à la différence de *Update_Env*).

- *Merge_L_Var* et *Insert_Var*, définies par les règles (41–42) et (43–45), permettent de réaliser la fusion de deux listes de variables sans doublons.

Les opérations *GIE* et *GLOMV* sont liées à la sémantique des boucles et sont présentées avec celle-ci dans la Section 4.3.3. Quant aux opérations *C_Subst*$_c$(v, e_1, e_2) et

;;Génération de l'environnement d'une liste d'instructions
$GE : stmt\_list \times env \rightarrow env$

;;Opération d'évaluation d'une instruction dans un environnement
$Comp : stmt \times env \rightarrow env$

;;Génération des appels aux fonctions de boucle
$GL : nat \times var\_list \times env \rightarrow env$

;;Insertion et mise à jour d'une paire dans un environnement
$Update\_Env : env\_elt \times env \rightarrow env$

;;Mise à jour d'une condition dans l'environnement courant
$Update\_Cond : cond \times env \rightarrow cond$

;;Génération de l'environnement initial d'une fonction de boucle
$GIE : env \rightarrow env$

;;Génération de la liste des variables d'un environnement
$GLOV : env \rightarrow var\_list$

;;Génération de la liste des variables modifiées d'un environnement
$GLOMV : env \rightarrow var\_list$

;;Fusion de deux environnements
$Merge\_Env : env \times env \rightarrow env$

;;Insertion sans mise à jour d'une paire dans un environnement
$Insert\_Pair : \ env\_elt \times env \rightarrow env$

;;Fusion de deux listes de variables
$Merge\_L\_Var : var\_list \times var\_list \rightarrow var\_list$

;;Insertion d'une variable dans une liste de variables
$Insert\_Var : var \times var\_list \rightarrow var\_list$

;;Opérations de substitution
$C\_Subst_c : var \times cond \times exp \rightarrow cond$
$C\_Subst_e : var \times exp \times exp \rightarrow exp$

FIG. 4.7 – Signature des opérations sur les environnements dans Σ_{sos}

C_Subst$_e$(v, e_1, e_2), elles permettent de remplacer v par e_2 dans, respectivement, la condition ou l'expression e_1. Nous les utiliserons pour remplacer une variable par sa valeur dans une condition ou une expression. Ces opérations de substitution ne sont pas définies dans le système de réécriture $SOS_{\mathcal{R}}$ et sont effectuées une fois la réécriture terminée.

4.2.4 Les environnements sont des équations

Dans $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$, un environnement est la représentation d'un ensemble d'équations caractéristiques d'un programme. Le système de réécriture $SOS_{\mathcal{R}}$ définit la sémantique du langage $\text{Sub}_{\mathcal{C}}$ par des modifications dans un tel environnement. Lors de la seconde étape, en réécrivant par $SOS_{\mathcal{R}}$ le terme T^f_P d'un sous-programme $\text{Sub}_{\mathcal{C}}$ f, nous avons obtenu l'environnement final $env_{T^f_P}$. Ce dernier est donc l'expression de la sémantique du sous-programme f. Pour donner la définition équationnelle de la sémantique de f, la dernière étape consiste à formuler les équations contenues dans l'environnement final. Ainsi, le rôle de la **troisième étape de l'axiomatisation** est le suivant :

Formulation des équations	
Entrée	: Un environnement final $env_{T^f_P}$.
Sortie	: Un ensemble d'équations conditionnelles E_P qui définissent la sémantique du programme P.

Les environnements finaux contiennent trois éléments qui nous serviront à formuler les équations d'un sous-programme f :

- Les *listes de paires* variable-valeur décrivent l'état des variables du sous-programme à la fin de son évaluation symbolique. Parmi toutes les paires, seule celle contenant la valeur de retour du sous-programme produira une équation. Cette paire particulière se reconnaît à ce que sa variable porte le nom du sous-programme (*cf.* Section 4.3.1).

- Un terme de racine *C_Branch* représente un chemin d'exécution dans le sous-programme. Il associe une condition, celle que les données en entrée doivent remplir pour que le chemin considéré soit exécuté, et une liste de paires variable-valeur, qui rend compte de la valeur des variables après l'exécution des instructions du chemin considéré. L'équation formée à partir de la paire contenant la valeur de retour de f comportera donc en plus une condition.

- Les termes de racine *C_While_Closure* représentent des fermetures de boucle. Ils se trouvent dans la liste des paires d'une branche donnée. Ces termes sont à l'origine d'un ensemble d'équations conditionnelles qui définissent une famille de fonctions récursives pour la boucle considérée : une fonction pour chaque variable modifiée dans le corps de la boucle. Ce comportement est défini précisément dans la Section 4.3.3.

Définition 4.2 (éléments générateurs d'équations). Soient f un sous-programme $\text{Sub}_{\mathcal{C}}$ et $P_1, \ldots P_n$ ses paramètres formels. Nous définissons les *éléments générateurs d'équations* de l'environnement final de f et leurs équations associées comme suit :

```
int identité(int x) {
 return x;
}
```

FIG. 4.8 – La fonction identité en $\text{Sub}_{\mathcal{C}}$

C_Env La liste de paires, construite avec les constructeurs *C_Env* et *C_Empty_Env*,

$$\{t_1 \cdot \ldots \cdot \textit{C\_Pair(f, val)} \cdot \ldots \cdot t_m\}$$

produit l'équation

$$f(P_1, \ldots, P_n) = val \ . \tag{4.3}$$

C_Branch Le terme

$$\textit{C\_Branch(Cond, } \{t_1 \cdot \ldots \cdot \textit{C\_Pair(f, val)} \cdot \ldots \cdot t_m\})$$

produit l'équation

$$\text{Cond} \Rightarrow f(P_1, \ldots, P_n) = val \ . \tag{4.4}$$

C_While_Closure Un terme *C_While_Closure* produit l'ensemble des équations conditionnelles défini par l'Équation (4.9) de la Définition 4.8 (Page 76).

La formulation des équations d'un sous-programme consiste alors à parcourir l'environnement final de ce sous-programme et à former les équations correspondant aux éléments générateurs rencontrés. La présence des termes *C_Env* et *C_Choice* à la racine du terme de l'environnement distingue, respectivement, un sous-programme ne comportant qu'un chemin d'exécution, donc inconditionnel, d'un sous-programme avec plusieurs chemins d'exécution.

Définition 4.5 (équations des sous-programmes et boucles $\text{Sub}_{\mathcal{C}}$). Soient f un sous-programme $\text{Sub}_{\mathcal{C}}$ et env_f l'environnement final de f. Les équations formulées pour le sous-programme f sont les suivantes :

1. Chaque occurrence d'un terme de racine *C_While_Closure* dans env_f produit un ensemble d'équations conditionnelles défini par l'Équation (4.9) (Page 76).

2. Si le symbole à la racine de env_f est le constructeur *C_Env*, une équation est formulée sur le modèle de l'Équation (4.3).

3. Sinon, le symbole à la racine de env_f est le constructeur *C_Choice* et une équation sur le modèle de l'Équation (4.4) est formulée pour chaque occurrence d'un terme de racine *C_Branch* dans env_f.

Exemple 4.6 (identité). Nous prenons l'exemple minimal de la fonction `identité` de la Figure 4.8 pour détailler les trois étapes de l'axiomatisation.

1. Lors de la première étape, la fonction `identité` est transformée en un terme sur la signature Σ_{sos}. La fonction `identité` ne contient que l'instruction `return x`. Pour cette instruction, nous construisons un terme qui associe le nom de la fonction avec l'expression de retour de la fonction (règle de correspondance syntaxique (Ret)) : *C_Assign(identité, x)*.

   L'*environnement initial* d'un sous-programme est une liste de paires qui associent paramètres formels et paramètres effectifs du sous-programme (règle (Fct)). Toutefois, les paramètres effectifs d'un sous-programme n'étant pas toujours connus, le sous-programme est « symboliquement évalué » : en l'occurrence, nous représentons par le terme *C_EA(x)* le paramètre effectif du paramètre formel `x`. La fonction `identité` ne prend qu'un paramètre ; nous représentons la liste ne contenant qu'une seule paire par la notation : *{C_Pair(x, C_EA(x))}*.

   Enfin, la fonction *GE* regroupe la séquence d'instructions du corps de la fonction et l'environnement initial de la fonction :

   $$GE(\{C\_Assign(identité, x)\}, \{C\_Pair(x, C\_EA(x))\})$$

2. Le terme précédent est prêt à être normalisé par le système de réécriture $SOS_{\mathcal{R}}$ (*cf.* Annexe A). Les Règles (1) et (2) réécrivent la fonction *GE* pour évaluer la première, et seule, instruction de la séquence d'instructions, nous obtenons :

   $$Comp(C\_Assign(identité, x), \{C\_Pair(x, C\_EA(x))\})$$

   puis les Règles (4), (14) et (15) expriment la sémantique de l'instruction `return` comme l'insertion et la mise à jour de la valeur de retour de la fonction dans l'environnement courant. L'application des règles aboutit à l'environnement suivant :

   $$\{C\_Pair(x, C\_EA(x)) \cdot C\_Pair(identité, C\_Subst_e(x, C\_EA(x), C\_EA(x)))\}$$

   Nous obtenons, après substitution[9], l'environnement final :

   $$\{C\_Pair(x, C\_EA(x)) \cdot C\_Pair(identité, C\_EA(x))\}$$

3. Ainsi, l'environnement final contient deux paires. La première dénote la valeur de `x` à la fin de l'exécution de la fonction. La seconde contient le nom de la fonction analysée et dénote, par conséquent, la valeur de retour de celle-ci. Ces deux valeurs sont égales au paramètre effectif, c'est-à-dire, à la valeur de `x` au moment de l'appel. À la troisième étape de l'axiomatisation, nous ne nous intéressons qu'à la valeur de retour de la fonction. Cet environnement, complété de l'information sur le prototype de la fonction `identité`, n'engendrera donc qu'une seule équation :

   $$\text{identité}(x) = x$$

Cette équation exprime la sémantique de la fonction `identité` en logique équationnelle. ♦

[9] Les substitutions n'affectent pas les termes de racine *C_EA* (*cf.* Section 4.3.1).

4.3 Sémantique du langage Sub$_\mathcal{C}$

Nous avons présenté dans la section précédente les trois étapes de l'axiomatisation. Nous développons dans cette section la sémantique des constructions du langage Sub$_\mathcal{C}$ telle qu'elle est définie par le système de réécriture $SOS_\mathcal{R}$ de l'Annexe A. Le processus de réécriture intervient lors de la deuxième étape ; il réalise l'évaluation symbolique d'un sous-programme, le résultat de laquelle est l'environnement final du sous-programme.

Nous commençons par les constructions dont la sémantique est la plus simple : les constructions de base de la Section 4.3.1. Nous abordons ensuite successivement l'instruction conditionnelle, Section 4.3.2, et l'instruction itérative, Section 4.3.3. Nous serons amenés à faire fréquemment référence aux règles de réécriture : chaque fois les numéros cités renvoient aux règles correspondantes de l'Annexe A (Page 163).

4.3.1 Constructions de base

Le langage Sub$_\mathcal{C}$ est composé du jeu de constructions élémentaires de tout langage de programmation impératif. Notamment, il acquiert son appartenance à ce paradigme des langages de programmation par son instruction d'affectation. Cette dernière est en effet l'instruction par excellence du changement d'état, qui fonde la programmation impérative. Pour cette raison, l'instruction d'affectation débute logiquement la présentation des constructions de base de Sub$_\mathcal{C}$. Nous énonçons ensuite la sémantique des types et expressions du langage, puis, de la séquence d'instructions, avant de clore cette section par la notion de sous-programme.

Affectation

Lorsqu'une instruction d'affectation `x = y` est rencontrée dans un programme, elle est traduite en un terme *C_Assign(x, y)* selon la règle de correspondance syntaxique (Assg) de la Figure 4.6 (Page 61). La sémantique de l'affectation est donnée par les règles de réécriture (3) et (4) que nous rappelons ici :

(3) *Comp(C_Assign(v_var, v_exp), C_Empty_Env)* →
C_Env(C_Pair(v_var, v_exp), C_Empty_Env)

(4) *Comp(C_Assign(v_var, v_exp), C_Env(v_pair, v_env))* →
Update_Env(C_Pair(v_var, v_exp), C_Env(v_pair, v_env))

Le symbole *Comp* dénote la fonction chargée d'évaluer une instruction dans un environnement courant. Deux possibilités sont prises en compte pour l'évaluation d'une affectation :

- règle (3), l'environnement courant est vide (*C_Empty_Env*) : une paire formée de la variable et de sa valeur est ajoutée dans l'environnement ;
- règle (4), l'environnement courant n'est pas vide, il contient une liste de paires : l'appel de la fonction *Update_Env* (*cf.* Section 4.2.3) met à jour, d'une part, l'expression affectée (*y* en l'occurrence), et d'autre part, l'environnement lui-même en modifiant la paire qui contient la valeur de `x`.

La sémantique de l'affectation est donc de modifier la liste de paires d'un environnement, de façon à ce que la paire correspondant à la variable affectée reflète la nouvelle valeur de la variable dans l'environnement courant.

Types et expressions

Les deux types de données dont disposent $\text{Sub}_{\mathcal{C}}$ sont les *entiers* et les *listes*. Les variables d'un programme sont typées statiquement et déclarées en début de programme. La sémantique d'une déclaration de variable est identique à celle d'une affectation : règles (VarDecl1), (VarDecl2) et (VarDecl3) de la Figure 4.6.

Si la variable n'est pas explicitement initialisée, nous lui donnons une valeur par défaut : 0 pour les entiers, règle (VarDecl2), et *NULL* pour les listes, règle (VarDecl3).

Expressions Les expressions, et par ce terme nous incluons ici aussi les conditions, ne sont pas interprétées par $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$. Nous nous contentons d'y mettre à jour la valeur des variables lorsque nécessaire. Nous laissons le soin de la définition des opérations intervenant dans les expressions au système de preuve qui sera utilisé par la suite. Il est bien entendu que les opérations dans les expressions n'ont pas d'effet de bord.

Listes Le langage $\text{Sub}_{\mathcal{C}}$ est muni d'un type pour les listes. Il peut s'agir de listes d'entiers, mais aussi de listes de listes.

Dans $\text{Sub}_{\mathcal{C}}$, le type `list` désigne une *représentation fonctionnelle* des listes, *i.e.*, un type récursif défini à l'aide de deux constructeurs (la liste vide et l'ajout en tête d'un élément). La conséquence de cette représentation est que les listes sont vues par l'affectation comme des valeurs atomiques : si `La` et `Lb` sont des listes, `La = Lb` a pour effet, selon la sémantique de l'affectation, de dupliquer la valeur de `Lb` dans l'environnement courant, afin de former la valeur de la paire de `La` dans ce même environnement.

Il en va de même des expressions de liste que des autres expressions $\text{Sub}_{\mathcal{C}}$: elles ne sont pas interprétées par $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$. Les opérations sur les listes, *i.e.*, `element` et `next`, ne sont pas définies dans $SOS_{\mathcal{R}}$. Ainsi, une expression de liste comme

```
next(add(1, NULL))
```

n'est pas évaluée comme la liste `NULL` et reste sous sa forme initiale.

Séquence d'instructions

En $\text{Sub}_{\mathcal{C}}$, un point-virgule sépare deux instructions dans une séquence d'instructions. La règle (Seq) de la Figure 4.6 fait correspondre à une séquence d'instructions une liste d'instructions sous la forme d'un terme de Σ_{sos}.

La sémantique de la séquence d'instructions est donnée par la fonction *GE* définie par les règles (1) et (2) du système de réécriture $SOS_{\mathcal{R}}$:

(1) *GE(C_L_Stmt(v_l_stmt, v_stmt), v_env)* →
Comp(v_stmt, GE(v_l_stmt, v_env))

(2) *GE(C_Empty_L_Stmt, v_env)* → *v_env*

Ainsi, la signification d'une séquence d'instructions est d'évaluer (fonction *Comp*) la dernière instruction de la séquence dans l'environnement qui résulte de l'évaluation, dans l'environnement courant, des instructions précédentes de la séquence. Lorsque la liste d'instruction est vide, l'environnement courant n'est pas modifié.

Sous-programme

En Sub$_C$, les sous-programmes peuvent avoir une valeur de retour ou pas. Cependant, lorsqu'ils n'en ont pas, $SOS\text{Sub}_C^{\mathcal{R}}$ ne génèrera aucune équation pour eux (*cf.* Section 4.2.4). En effet, les sous-programmes n'ont pas d'effets de bord (*e.g.*, aucune instruction d'entrée/sortie), donc, s'ils n'ont pas non plus de valeur de retour, ils n'ont aucune donnée en sortie.

La règle (Fct) de la Figure 4.6 spécifie le terme de Σ_{sos} construit pour un sous-programme. Ce terme définit la sémantique d'un sous-programme comme l'environnement obtenu de l'évaluation de la séquence d'instructions qui forme le corps du sous-programme dans un environnement initial. L'environnement initial est composé d'une liste de paires établie en fonction des paramètres du sous-programme.

Paramètres Le mode de passage des paramètres est par valeur. Ceci implique que les listes, comme lors d'une affectation, sont dupliquées lorsqu'elles sont passées en paramètre à un sous-programme.

Lorsque l'environnement initial d'un sous-programme est constitué, chacun de ses paramètres ajoute une paire dans l'environnement. Chaque paire associe un paramètre *p* avec un terme *C_EA(p)*. Ce terme dénote la valeur du *paramètre effectif* correspondant à *p* qui aura été passé au moment de l'appel du sous-programme. Nous avons recours à ce terme car l'évaluation des sous-programmes se fait hors de tout contexte d'appel. C'est en ce sens que l'évaluation est symbolique : nous ne manipulons pas la valeur réelle des paramètres effectifs.

L'ajout de ces paires dans l'environnement traduit le fait que les paramètres formels sont considérés comme des variables locales initialisées à la valeur de leur paramètre effectif. Si la valeur d'un paramètre formel peut changer au cours de l'évaluation du sous-programme, par contre, la valeur d'un paramètre effectif est considérée comme une constante par le sous-programme. Pour cette raison, les substitutions, qui font suite aux mises à jour de l'environnement, ne s'appliquent pas aux variables qui apparaissent dans des termes de racine *C_EA*.

Lors de la formulation des équations, une fois que toutes les substitutions ont été accomplies, les termes *C_EA(p)* ne sont plus nécessaires et sont remplacés par des variables de la logique. En effet, dans la logique équationnelle, seule la référence à la valeur du paramètre effectif sera nécessaire pour définir les fonctions de transfert d'un sous-programme Sub$_C$.

Return La valeur de retour d'un sous-programme est désignée par l'expression attachée à l'instruction `return`. Cette instruction ne doit apparaître qu'une fois par sous-programme et elle ne peut être que la toute dernière instruction du sous-programme. Ces contraintes ne sont pas réellement une limitation puisque nous pouvons toujours mettre

sous cette forme, en préservant la sémantique, un sous-programme comportant plusieurs instructions `return`.

Lorsqu'elle est rencontrée dans un sous-programme, l'instruction `return` est traduite en un terme de racine *C_Assign* (*cf.* règle (Ret) de la Figure 4.6) qui est la fonction de Σ_{sos} correspondant aux affectations. Ainsi, la sémantique de l'instruction `return` est la même qu'une affectation de l'expression de retour à une variable un peu spéciale qui est le nom du sous-programme[10]. Nous avons vu en Section 4.2.4 que la paire qui résulte de cette pseudo-affectation sert à formuler les équations du sous-programme.

4.3.2 Instruction conditionnelle

Les instructions conditionnelles du langage $\text{Sub}_{\mathcal{C}}$ sont de la forme `if ... else`. Elles divisent un chemin d'exécution dans un sous-programme en deux nouveaux chemins, même lorsque la partie `else` de l'instruction est omise. Le premier chemin est composé des instructions exécutées lorsque la condition du `if` est vraie, alors que le second comporte les instructions exécutées lorsque la condition est fausse.

Un terme de racine *C_If* est associé à une instruction `if` (*cf.* les règles de correspondance syntaxique (Cond1) et (Cond2) en Page 61). Ce terme regroupe la condition du `if`, la séquence d'instructions de la partie « alors » et celle de la partie « sinon ». La définition de la fonction *C_If* correspond aux règles (5) et (6) de $SOS_{\mathcal{R}}$. Nous rappelons ici ces règles :

(5) *Comp(C_If(v_cond, v_l_stmt1, v_l_stmt2), C_Empty_Env)* →
C_Choice(C_Branch(v_cond, GE(v_l_stmt1, C_Empty_Env)),
C_Branch(C_Not(v_cond), GE(v_l_stmt2, C_Empty_Env)))

(6) *Comp(C_If(v_cond, v_l_stmt1, v_l_stmt2), C_Env(v_pair, v_env))* →
C_Choice(C_Branch(Update_Cond(v_cond, C_Env(v_pair, v_env)),
GE(v_l_stmt1, C_Env(v_pair, v_env))),
C_Branch(C_Not(Update_Cond(v_cond,
C_Env(v_pair, v_env))),
GE(v_l_stmt2, C_Env(v_pair, v_env))))

La sémantique d'une instruction conditionnelle est donc de construire un terme de racine *C_Choice* qui représente une alternative entre deux chemins d'un sous-programme. Chacun des chemins d'un sous-programme est quant à lui symbolisé par un terme de racine *C_Branch* contenant la condition et l'environnement courant associés à ce chemin. La condition du chemin correspondant à la partie `else` d'une instruction `if` est simplement la négation, *C_Not*, de la condition du `if`.

Lorsque l'environnement dans lequel est évaluée l'instruction conditionnelle n'est pas vide, la condition de l'instruction est mise à jour par la fonction *Update_Cond* (*cf.* Section 4.2.3).

D'autre part, les règles de réécriture (9) et (10) signifient que les instructions qui suivent un `if` dans une séquence d'instructions sont exécutées dans les deux chemins d'une

[10] Cette façon de faire n'est pas sans rappeler ce qui existe dans le langage PASCAL.

```
1 int comparer(int x, int y) {
2   int ret;
3
4   if(x > y) ret = −1;
5   else if(x == y) ret = 0;
6   else ret = 1;
7
8   return ret;
9 }
```

FIG. 4.9 – Sous-programme `comparer`

alternative *C_Choice*. Enfin, les règles (20) et (21) permettent de réaliser la conjonction des conditions qui portent sur un même chemin d'exécution. Cette situation découle de la présence de plusieurs instructions conditionnelles sur un même chemin d'exécution. Les conditions sont regroupées dans un terme de racine *C_And*.

Exemple 4.7 (comparaison d'entiers). Nous illustrons la sémantique des instructions conditionnelles avec l'exemple du sous-programme de la Figure 4.9. Lors de l'analyse syntaxique du sous-programme, le terme qui lui correspond est construit par l'application des règles suivantes (le symbole □ dénote un sous-terme qui est le résultat de la suite de l'analyse du sous-programme) :

`int comparer(int x, int y){`	$\overset{(Fct)}{\longmapsto}$	*GE*(□$_1$, *{C_Pair(x, C_EA(x)) · C_Pair(y, C_EA(y))}*)
`int ret;`	$\overset{(VarDecl2)}{\longmapsto}$	t_1 = *C_Assign(ret, 0)*
`if(x > y)`	$\overset{(Cond1)}{\longmapsto}$	t_2 = *C_If(x>y,* □$_2$, □$_3$*)*
`ret = −1;`	$\overset{(Assg) \text{ et } (Seq)}{\longmapsto}$	□$_2$ = *{C_Assign(ret, −1)}*
`else if(x == y)`	$\overset{(Cond1) \text{ et } (Seq)}{\longmapsto}$	□$_3$ = *{C_If(x==y,* □$_4$, □$_5$*)}*
`ret = 0;`	$\overset{(Assg) \text{ et } (Seq)}{\longmapsto}$	□$_4$ = *{C_Assign(ret, 0)}*
`else ret = 1;`	$\overset{(Assg) \text{ et } (Seq)}{\longmapsto}$	□$_5$ = *{C_Assign(ret, 1)}*
`return ret;`	$\overset{(Ret)}{\longmapsto}$	t_3 = *C_Assign(comparer, ret)*
`}`	$\overset{(Seq)}{\longmapsto}$	□$_1$ = { $t_1 \cdot t_2 \cdot t_3$ }

Finalement, le terme obtenu pour le sous-programme à la fin de la première étape de l'axiomatisation est :

GE({C_Assign(ret, 0) ·
C_If(x>y,{C_Assign(ret, −1)},
{C_If(x==y, {C_Assign(ret, 0)}, {C_Assign(ret, 1)})}) ·
C_Assign(comparer, ret)}, {C_Pair(x, C_EA(x)) · C_Pair(y, C_EA(y))})

Lors de la seconde étape de l'axiomatisation, ce terme est normalisé par $SOS_{\mathcal{R}}$. Nous ne détaillons que quelques-unes des règles appliquées dans la séquence de réécriture. Après

plusieurs applications des règles pour la fonction *GE* et l'application de la règle (4) pour le terme *C_Assign(ret, 0)*, nous obtenons le terme suivant :

Comp(C_Assign(comparer, ret),
Comp(C_If(x>y,{C_Assign(ret, −1)},
{C_If(x==y, {C_Assign(ret, 0)}, {C_Assign(ret, 1)})}),
{C_Pair(x, C_EA(x)) · C_Pair(y, C_EA(y)) · C_Pair(ret, 0)}))

Correspondant à l'instruction conditionnelle de la ligne 4, la règle (6) introduit une alternative *C_Choice* composée de deux chemins d'exécution *C_Branch* :

Comp(C_Assign(comparer, ret),
C_Choice(C_Branch(Update_Cond(x>y, env),
GE({C_Assign(ret, −1)}, env)),
C_Branch(C_Not(Update_Cond(x>y, env)),
GE({C_If(x==y,{C_Assign(ret, 0)},
{C_Assign(ret, 1)})}, env))))

avec *env* = *{C_Pair(x, C_EA(x)) · C_Pair(y, C_EA(y)) · C_Pair(ret, 0)}*.

La mise à jour des conditions et l'évaluation de l'affectation de la ligne 4 par les règles correspondantes de $SOS_{\mathcal{R}}$ donne le terme :

Comp(C_Assign(comparer, ret),
C_Choice(C_Branch(C_EA(x)>C_EA(y), env'),
C_Branch(C_Not(C_EA(x)>C_EA(y)),
GE({C_If(x==y,{C_Assign(ret, 0)},
{C_Assign(ret, 1)})}, env))))

avec *env' = {C_Pair(x, C_EA(x)) · C_Pair(y, C_EA(y)) · C_Pair(ret, −1)}*

De nouveau, mais cette fois pour l'instruction conditionnelle de la ligne 5, la règle (6) introduit une alternative. Après mise à jour des conditions et évaluation de affectations, nous obtenons :

Comp(C_Assign(comparer, ret),
C_Choice(C_Branch(C_EA(x)>C_EA(y), env'),
C_Branch(C_Not(C_EA(x)>C_EA(y)),
C_Choice(C_Branch(C_EA(x)==C_EA(y), env''),
C_Branch(C_Not(C_EA(x)==C_EA(y)),
env''')))))

avec

env'' = {C_Pair(x, C_EA(x)) · C_Pair(y, C_EA(y)) · C_Pair(ret, 0)}

et

env''' = {C_Pair(x, C_EA(x)) · C_Pair(y, C_EA(y)) · C_Pair(ret, 1)}.

Les règles (20) et (21) s'appliquent alors à la deuxième occurrence de la fonction *C_Branch* afin de réaliser la conjonction des conditions d'un même chemin d'exécution. Le terme qui en résulte est le suivant :

```
Comp(C_Assign(comparer, ret),
     C_Choice(C_Branch(C_EA(x)>C_EA(y), env'),
              C_Choice(C_Branch(C_And(C_Not(C_EA(x)>C_EA(y)),
                                      C_EA(x)==C_EA(y)),
                                env''),
                       C_Branch(C_And(C_Not(C_EA(x)>C_EA(y)),
                                      C_Not(C_EA(x)==C_EA(y))),
                                env''')))
```

Enfin, les règles (9) et 10 propagent la dernière instruction du sous-programme dans tous les chemins d'exécution. Ce qui conduit, après évaluation de cette dernière instruction à l'environnement final :

```
C_Choice(C_Branch(C_EA(x)>C_EA(y), env1),
         C_Choice(C_Branch(C_And(C_Not(C_EA(x)>C_EA(y)),
                                 C_EA(x)==C_EA(y)),
                           env2),
                  C_Branch(C_And(C_Not(C_EA(x)>C_EA(y)),
                                 C_Not(C_EA(x)==C_EA(y))),
                           env3)))
```

avec

$$
\begin{aligned}
env_1 &= \{C\_Pair(x, C\_EA(x)) \cdot C\_Pair(y, C\_EA(y)) \cdot \\
&\quad\ C\_Pair(ret, -1) \cdot C\_Pair(comparer, -1)\} \\
env_2 &= \{C\_Pair(x, C\_EA(x)) \cdot C\_Pair(y, C\_EA(y)) \cdot \\
&\quad\ C\_Pair(ret, 0) \cdot C\_Pair(comparer, 0)\} \\
env_3 &= \{C\_Pair(x, C\_EA(x)) \cdot C\_Pair(y, C\_EA(y)) \cdot \\
&\quad\ C\_Pair(ret, 1) \cdot C\_Pair(comparer, 1)\}.
\end{aligned}
$$

Pour conclure, lors de la dernière étape de l'axiomatisation, chaque terme de racine *C_Branch* produit une équation conditionnelle (*cf.* Section 4.2.4). La valeur de retour de la fonction définie par l'axiomatisation est lue dans la paire pour laquelle le nom de la variable est le nom du sous-programme. Les équations produites sont les suivantes :

$$
\begin{aligned}
x > y &\Rightarrow \text{comparer}(x, y) = -1 \\
\neg(x > y) \wedge x = y &\Rightarrow \text{comparer}(x, y) = 0 \\
\neg(x > y) \wedge \neg(x = y) &\Rightarrow \text{comparer}(x, y) = 1
\end{aligned}
$$

♦

4.3.3 Instruction itérative

Intuitivement, l'idée qui sous-tend la sémantique de l'instruction itérative `while` est de remplacer l'itération par la récursivité. De fait, le principe d'itération n'existe pas en tant que tel dans la logique équationnelle. En revanche, la récursivité est un principe naturel dans cette logique. Par conséquent, nous traitons les boucles comme des fonctions récursives terminales. Cependant, si nous considérons les boucles comme des fonctions, nous devons nous poser la question des données en entrée et en sortie de ces *fonctions de boucle*.

Principes

Dans la logique équationnelle, une fonction n'a qu'une valeur de retour. Or, dans une boucle, les valeurs de plusieurs variables peuvent être modifiées. Après l'exécution d'une boucle, une seule fonction de boucle ne suffit donc pas pour rendre compte de tous les changements dans l'état des variables d'un sous-programme. Pour surmonter cette difficulté, nous avons recours à une famille de fonctions de boucle. À chaque variable modifiée dans la boucle est attaché un membre de la famille qui renvoie la valeur de cette variable spécifiquement. Chaque fonction de boucle de la famille se comporte comme la projection de l'état des variables à la fin des itérations d'une boucle pour la variable qui lui est attachée.

Le cas des données en entrée des fonctions de boucle est considérablement plus simple. Potentiellement, une boucle peut accéder à la valeur de n'importe quelle variable du sous-programme auquel elle appartient. L'ensemble des variables d'un sous-programme comprend les variables déclarées localement dans le sous-programme, mais aussi ses paramètres. Pour les données en entrée des fonctions de boucle, le choix le plus simple est de prendre acte de cette possibilité d'accès à toutes les variables et de faire correspondre à chaque variable d'un sous-programme un paramètre de la fonction de boucle. Ainsi, la lecture de la valeur d'une variable d'un sous-programme dans le corps de la boucle devient la lecture de la valeur du paramètre correspondant dans la fonction de boucle.

Avec ces définitions, une instruction itérative dans un sous-programme est remplacée par des appels aux fonctions de boucle. Chaque variable modifiée dans la boucle prend pour nouvelle valeur le résultat de la fonction de boucle qui lui correspond.

Formalisation de la sémantique dans $SOS_{\mathcal{R}}$

Lorsqu'une instruction itérative est rencontrée dans un sous-programme, un terme de racine *C_While* est formé (règle de correspondance syntaxique (Iter) de la Figure 4.6 Page 61). Dans $SOS_{\mathcal{R}}$, les règles qui définissent la sémantique des instructions itératives sont les règles (7) et (8). Nous détaillons uniquement la règle (8), l'autre règle étant un cas particulier de celle-ci pour un environnement courant vide. Nous rappelons ici la règle :

(8)*Comp(C_While(v_num, v_cond, v_l_stmt), C_Env(v_pair, v_l_pair))* →
C_Env(C_While_Closure(v_num, v_cond,
GE(v_l_stmt, GIE(C_Env(v_pair, v_l_pair))),
GLOV(C_Env(v_pair, v_l_pair))),
Merge_Env(GL(v_num,
GLOMV(GE(v_l_stmt,
GIE(C_Env(v_pair, v_l_pair)))),
C_Env(v_pair, v_l_pair)),
C_Env(v_pair, v_l_pair)))

Dans cette règle, *v_num* fait référence à un numéro qui identifie uniquement la boucle dans le programme dont elle fait partie.

Les modifications sur un environnement courant (*C_Env(v_pair, v_l_pair)* dans la règle) qui résultent de l'évaluation d'une boucle B dans un sous-programme `f` sont de deux ordres :

1. La liste d'instructions *v_l_stmt*, qui correspond au corps de la boucle B, est évaluée dans un nouvel environnement. Cet environnement est produit par la fonction *GIE*, il consiste en la liste des paires *C_Pair(v, C_EA(v))*, pour toute variable *v* présente dans l'environnement courant de `f`. Nous obtenons ainsi l'équivalent de l'environnement initial d'un sous-programme qui prendrait en paramètre toutes les variables de `f`. Le résultat de la fonction *GE* pour la liste *v_l_stmt* est donc identique à l'environnement final env_B d'un sous-programme qui a pour corps la séquence d'instructions de la boucle B et pour paramètres toutes les variables qui apparaissent dans le sous-programme `f`.

   Une *fermeture de boucle* (symbole *C_While_Closure*) contenant env_B est ajoutée dans l'environnement courant de `f`. Elle comporte aussi la liste de toutes les variables de `f` (produite par la fonction *GLOV*). Ces informations nous serviront à la troisième étape de l'axiomatisation pour formuler les équations définissant les fonctions de boucle de B (voir la section suivante).

2. Le rôle de la fonction *GL* est de générer une liste de paires qui associent chaque variable modifiée dans le corps de la boucle B avec un appel à sa fonction de boucle spécifique. Ces paires remplacent l'ancienne valeur des variables considérées dans l'environnement courant (fonction *Merge_env*). Les appels aux fonctions de boucle sont représentés par une fonction *C_Loop*.

   Pour trouver la liste des variables modifiées par B, la fonction *GLOMV* évalue le corps de B dans un nouvel environnement initial et compare la valeur des variables dans l'environnement final obtenu avec leur valeur dans l'environnement initial.

Nous retrouvons donc dans cette règle la sémantique des instructions itératives que nous avons décrite dans la section précédente : une boucle est remplacée par des appels à des fonctions de boucle. Ces appels deviennent la nouvelle valeur, dans l'environnement courant, des variables modifiées par la boucle.

Fermeture de boucle

Lors de la dernière étape de l'axiomatisation, différents éléments de l'environnement final d'un sous-programme sont utilisés pour formuler les équations qui définissent la sémantique algébrique de ce sous-programme. La Définition 4.2 (Page 64) énumère ces éléments générateurs d'équations. Parmi eux, nous trouvons les fermetures de boucle (*C_While_Closure*).

Une fermeture de boucle génère une famille de fonctions de boucle : une fonction par variable modifiée dans la boucle correspondante. Une fonction de boucle spécifique à une variable v est définie par deux équations :

– Une première équation est un appel récursif à la fonction de boucle, avec pour nouvelle valeur des paramètres celle qui résulte de l'évaluation du corps de la boucle

pour ces paramètres. Elle traduit la sémantique de la répétition d'une séquence d'instructions lorsque la condition de boucle est vraie.

– La seconde équation correspond à l'arrêt de la récurrence, lorsque la condition de boucle est fausse. La valeur renvoyée par la fonction de boucle est alors celle du paramètre correspondant à la variable v.

Nous pouvons maintenant définir plus formellement les équations formulées pour une fermeture de boucle.

Définition 4.8 (équations des fermetures de boucle). Soit la fermeture d'une boucle B en forme normale suivante :

C_While_Closure(num, cond, {C_Pair($\mathtt{v}_1$*,* $\mathtt{e}_1$*)* $\cdot \ldots \cdot$ *C_Pair(*$\mathtt{v}_n$*,* $\mathtt{e}_n$*)}, {*$\mathtt{v}_1 \cdot \ldots \cdot \mathtt{v}_n$*})*

avec *num* un numéro qui identifie uniquement la fermeture dans le programme et *cond* la condition associée à la fermeture.

Les équations formulées pour cette fermeture de boucle sont les suivantes :

$$\bigcup_{i \in V_{mod}} \left\{ \begin{array}{r} cond \Rightarrow \mathrm{loop}^{num}_{v_i}(v_1, \ldots, v_n) = \mathrm{loop}^{num}_{v_i}(e_1, \ldots, e_n) \\ \neg(cond) \Rightarrow \mathrm{loop}^{num}_{v_i}(v_1, \ \ldots, \ v_n) = v_i \end{array} \right\} \tag{4.9}$$

avec V_{mod} l'ensemble des variables $v \in \{v_1, \ldots, v_n\}$ tel que v est une variable modifiée par la boucle B.

Exemple 4.10 (somme des n premiers entiers). Dans la Section 1.2.1, Exemple 1.10 (Page 13), nous avons illustré le déroulement de l'axiomatisation avec le programme de calcul de la somme des n premiers entiers. Nous ne disposions pas encore de toutes les notions nécessaires pour expliquer précisément la formulation des équations. Nous nous proposons dans ce nouvel exemple de détailler ce que nous avions alors tu.

Nous repartons de l'environnement final du programme :

{C_While_Closure(1, n $\neq$ *0, {C_Pair(n, n−1)* $\cdot$ *C_Pair(sum, sum+n)})* $\cdot$
C_Pair(n, C_Loop(1, n, {C_Pair(n, C_EA(n)) $\cdot$ *C_Pair(sum, 0)}))* $\cdot$
C_Pair(sum, C_Loop(1, sum, {C_Pair(n, C_EA(n)) $\cdot$ *C_Pair(sum, 0)}))* $\cdot$
C_Pair(somme, C_Loop(1, sum, {C_Pair(n, C_EA(n)) $\cdot$ *C_Pair(sum, 0)}))}*

Selon la Définition 4.5, les éléments générateurs d'équations de cet environnement sont la fermeture de boucle et la liste de paires qui la suit.

1. Le terme de la fermeture de boucle est :

   C_While_Closure(1, n $\neq$ *0, {C_Pair(n, n−1)* $\cdot$ *C_Pair(sum, sum+n)})*

   Le programme ne comporte qu'une seule boucle : la fermeture a donc reçu le numéro 1. L'environnement *{C_Pair(n, n−1)* $\cdot$ *C_Pair(sum, sum+n)}*, contenu par la fermeture, nous permet de déduire que les deux variables du programme, *n* et *sum*, ont été modifiées par la boucle. En effet, aucune des deux variables n'a conservé sa

valeur initiale, qui pour une variable `v` serait représentée par une paire `C_Pair(v, v)`. Deux fonctions de boucle seront donc générées : loop^1_n, pour la variable `n`, et loop^1_{sum}, pour la variable `sum`.

Les équations générées pour les deux fonctions de boucle sont les suivantes (Définition 4.8) :

$$\begin{aligned} n \neq 0 &\Rightarrow \mathrm{loop}^1_n(n, sum) = \mathrm{loop}^1_n(n-1, sum+n) \\ \neg(n \neq 0) &\Rightarrow \mathrm{loop}^1_n(n, sum) = n \\ n \neq 0 &\Rightarrow \mathrm{loop}^1_{sum}(n, sum) = \mathrm{loop}^1_{sum}(n-1, sum+n) \\ \neg(n \neq 0) &\Rightarrow \mathrm{loop}^1_{sum}(n, sum) = sum \end{aligned}$$

L'équation récursive est la même pour les deux fonctions de boucle, seule la valeur renvoyée sur le cas d'arrêt change : chaque fonction de boucle renvoie la valeur de la variable qui lui est attachée.

2. Dans le sous-programme `somme` contenant la boucle, celle-ci a été remplacée par des appels aux fonctions de boucle : les termes de racine `C_Loop`. La valeur des paramètres effectifs de ces appels est la valeur des variables dans l'environnement courant du sous-programme au moment de l'instruction itérative. Lors de la formulation des équations, un terme `C_Loop(num, v_i, {(v1, e1) · ... · (vn, en)})` devient l'appel de fonction $\mathrm{loop}^{num}_{v_i}(e_1, \ldots, e_n)$.

   Le sous-programme `somme` ne comporte qu'un seul chemin d'exécution (une fois la boucle remplacée par des appels de fonction), son environnement final est donc seulement composé d'une liste de paires. Dans ce cas, l'équation générée est :

$$\mathrm{somme}(n) = \mathrm{loop}^1_{sum}(n, 0)$$

   Par la suite, les équations de la fonction loop^1_n ne sont pas conservées parce que cette fonction ne fait pas partie du graphe d'appel de la fonction somme. ♦

4.4 Convergence du système de réécriture $SOS_{\mathcal{R}}$

Dans la Section 3.2.3, nous avons défini la convergence d'un système de réécriture comme la conjonction des propriétés de terminaison et de confluence. La convergence du système de réécriture $SOS_{\mathcal{R}}$ est la preuve que le processus de réécriture dans $SOS\mathrm{Sub}^{\mathcal{R}}_{\mathcal{C}}$ termine et produit une forme normale unique pour chaque terme de programme. Elle garantit que, pour tout programme $\mathrm{Sub}_{\mathcal{C}}$, $SOS\mathrm{Sub}^{\mathcal{R}}_{\mathcal{C}}$ fournit toujours une unique formulation équationnelle de la sémantique du programme.

Nous détaillons dans cette section la preuve de convergence de $SOS_{\mathcal{R}}$, qui a été effectuée à l'aide de l'atelier de réécriture et démonstrateur de théorèmes RRL [Kapur et Zhang, 1988].

4.4.1 Terminaison

La terminaison de $SOS_{\mathcal{R}}$ nous certifie que tout terme de Σ_{sos} possède au moins une forme normale dans $SOS_{\mathcal{R}}$. Dans le cas de la réécriture conditionnelle, le Théorème 3.33 (Page 48) affirme qu'il suffit que le système soit décroissant pour qu'il termine. Pour montrer que notre système est décroissant, nous devons exhiber un ordre de simplification qui contienne la relation de réécriture $\rightarrow_{SOS_{\mathcal{R}}}$ et tel que la partie gauche des règles conditionnelles de $SOS_{\mathcal{R}}$ soit supérieure, au sens de cet ordre, à tous les termes qui apparaissent dans la condition de la règle.

À cette fin, nous définissons un *ordre lexicographique des chemins* qui, par le Théorème 3.25, est un ordre de simplification. Nous donnons dans la Figure 4.10 l'ordre de précédence $>$ sur les symboles de Σ_{sos} qui induit notre ordre lexicographique des chemins $>_{lpo}$ (*cf.* Définition 3.23).

L'atelier RRL dispose d'un algorithme (commande `makerule`) pour orienter automatiquement un ensemble d'équations en règles. L'orientation des règles respecte l'ordre lexicographique des chemins que l'utilisateur fournit à l'atelier. Nous avons utilisé cet algorithme, avec $>_{lpo}$, sur les règles de $SOS_{\mathcal{R}}$ transformées en équations, *i.e.*, en remplaçant la relation de réécriture par l'égalité. Le résultat produit par l'algorithme de RRL, sans aucune intervention de notre part, est le système de règles de $SOS_{\mathcal{R}}$. Ceci assure que $>_{lpo}$ contient la relation de réécriture $\rightarrow_{SOS_{\mathcal{R}}}$.

Il nous reste alors à montrer que pour les règles conditionnelles du système (*i.e.*, les règles (14), (31), (39) et (44)) la partie gauche de la règle est supérieure aux termes apparaissant dans la condition. La démonstration peut être faite dans RRL en présentant chaque couple de termes dont nous voulons vérifier l'appartenance à $>_{lpo}$ comme une équation. Nous contrôlons alors que l'orientation proposée par RRL est celle que nous souhaitons.

4.4.2 Confluence

Nous venons de démontrer que $SOS_{\mathcal{R}}$ est terminant. Pour sa convergence, il nous reste à montrer qu'il est confluent. La confluence d'un système de réécriture certifie que la forme normale d'un terme pour ce système est unique, si elle existe. Grâce au Théorème 3.35 (Page 48), la confluence d'un système de réécriture conditionnelle décroissant est assurée dès lors que toutes ses *paires critiques* sont joignables.

L'atelier RRL comporte un algorithme de complétion d'un système de réécriture (commande `kb`), dit de Knuth-Bendix [Knuth et Bendix, 1970], dont il suffira pour nous de savoir qu'il génère une nouvelle règle chaque fois qu'une paire critique du système n'est pas joignable. L'algorithme de complétion appliqué à $SOS_{\mathcal{R}}$ ne génère aucune nouvelle règle, prouvant ainsi la confluence de $SOS_{\mathcal{R}}$.

GE et *Comp* sont équivalents.
Pour *GE* le chemin lexicographique est de gauche à droite.
Pour *C_And* le chemin lexicographique est de gauche à droite.
Pour *Update_Env* le chemin lexicographique est de droite à gauche.
Pour *Update_Cond* le chemin lexicographique est de droite à gauche.
Pour *C_Branch* le chemin lexicographique est de droite à gauche.

$Comp > Update\_Env$ $Comp > Update\_Cond$
$Comp > C\_Branch$ $Comp > C\_Not$
$Comp > GLOV$ $Comp > Merge\_Env$
$Comp > GL$ $Comp > GLOMV$
$Comp > GIE$

$Update\_Cond > C\_Subst$

$Update\_Env > C\_Pair$ $Update\_Env > C\_Subst$
$Update\_Env > C\_Env$

$C\_Branch > C\_Choice$ $C\_Branch > C\_And$

$GIE > C\_Env$ $GIE > C\_Pair$
$GIE > C\_EA$

$GLOV > C\_L\_Var$

$GLOMV > C\_EA$ $GLOMV > Merge\_L\_Var$

$Merge\_L\_Var > Insert\_Var$ $Insert\_Var > C\_L\_Var$

$GL > C\_Env$ $GL > C\_Pair$
$GL > C\_Loop$

$Merge\_Env > Insert\_Pair$ $Insert\_Pair > C\_Env$
$Insert\_Pair > C\_While\_Closure$ $Insert\_Pair > C\_Pair$

FIG. 4.10 – Ordre partiel sur les symboles de fonction de Σ_{sos}

Chapitre 5

$SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$

Dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$, la sémantique d'un programme est une théorie de la logique équationnelle dont une axiomatisation est obtenue automatiquement par l'analyse statique du code source du programme. Cette approche originale s'applique à une classe de programmes qui n'emploient que les constructions basiques des langages impératifs, parmi lesquels :

- le séquencement (ou la composition) des instructions ;
- les instructions d'affectation, conditionnelles et itératives ;
- la notion de sous-programme et d'appel par valeur ;
- des types de données entier et liste.

Les effets de bord sont une caractéristique constitutive des langages impératifs. Le plus fondamental est sans nul doute l'affectation. Néanmoins, nous avons vu à la Section 1.1.2, avec l'exemple du programme de tri par fusion, que d'autres effets de bord sont usuellement disponibles dans les langages impératifs afin d'en accroître l'expressivité et de permettre l'écriture d'algorithmes plus efficaces. Nous proposons avec $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ d'étendre le langage $\mathrm{Sub}_{\mathcal{C}}$ dans le but de prendre en compte certains effets de bord absents de $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$, à savoir les *listes mutables* et l'*appel par référence*.

Nous débutons le chapitre par une présentation des principes à l'œuvre dans $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ accompagnée d'une brève comparaison avec les approches existantes. Les sections suivantes exposent la syntaxe et la sémantique du langage $\mathrm{Sub}_{\mathcal{C}}^{ext}$, qui n'est autre que le langage $\mathrm{Sub}_{\mathcal{C}}$ augmenté des listes mutables, Section 5.2, et de l'appel par référence, Section 5.3. Enfin, la dernière section, Section 5.4, décrit l'étape de formulation des équations à partir de la sémantique des sous-programmes $\mathrm{Sub}_{\mathcal{C}}^{ext}$.

5.1 Raisonner sur les effets de bord

Une expression avec *effet de bord* est une expression qui modifie l'état global d'un programme. Certaines de ces expressions ne sont d'ailleurs souvent évaluées que pour leur effet de bord et non pour leur valeur. C'est typiquement le cas de l'affectation dont

la valeur, celle de son membre droit en langage `C`, n'est utilisée que dans certains schémas de programmation spécifiques (*e.g.*, l'initialisation de plusieurs variables à la même valeur : `i = j = 0`). Ordinairement, les langages impératifs offrent, au-delà de l'affectation, d'autres mécanismes produisant des effets de bord. Il s'agit notamment des opérateurs d'incrément et de décrément (*e.g.*, en `C` : les opérateurs `++` et $--$), des instructions d'entrées-sorties, du passage de paramètre par référence, etc.

En langage `C`, de nombreux effets de bord sont réalisés par l'intermédiaire de *pointeurs*. Un pointeur est une variable contenant l'adresse d'une autre variable. L'emploi raisonné des pointeurs conduit généralement à une diminution substantielle de la complexité des algorithmes, en espace mémoire et en temps d'exécution, notamment en évitant d'inutiles et coûteuses recopies de valeurs. En revanche, les pointeurs rendent plus ardue l'analyse statique des programmes (ce point est détaillé dans la Section 5.1.1 qui suit).

Les pointeurs permettent ainsi de construire le type de données récursif des listes simplement chaînées. Un des avantages de cette implantation des listes réside dans la possibilité de modifier la liste sur place, sans recopie, par réarrangement des liens ou modification de l'information d'un maillon ; nous parlerons de liste *mutable* par opposition à un modèle fonctionnel des listes (*cf.* Section 5.2).

En langage `C`, les pointeurs sont aussi utilisés pour réaliser l'équivalent d'un *appel par référence*. Celui-ci est obtenu par la combinaison d'un *appel par valeur*, seul mode de passage des paramètres en `C`, et d'un *pointeur* qui contient l'adresse de la variable passée par référence. L'appel par référence permet à un sous-programme de modifier les *paramètres effectifs* qui lui sont passés.

Exemple 5.1 (passage par adresse en langage C). Dans le programme `C` suivant :

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int i = 0, j = 1;
  swap(&i, &j);
}
```

l'appel de `swap` dans la fonction `main` se fait en passant les adresses des variables `i` et `j`, grâce à l'opérateur d'adresse `&`. Dans la fonction `swap`, ces adresses sont manipulées par l'intermédiaire des pointeurs `a` et `b`. Chaque modification de `*a` et `*b` est en réalité une modification de `i` et `j`, respectivement. Si bien que dans la fonction `main`, après l'appel de `swap`, les valeurs de `i` et `j` ont été échangées : `i` vaut `1` et `j` vaut `0`. ♦

Ce sont ces deux derniers mécanismes, liste mutable et appel par référence, que nous avons décidé d'ajouter au langage $\text{Sub}_{\mathcal{C}}$ pour obtenir le langage $\text{Sub}_{\mathcal{C}}^{ext}$. Ils nous permettent de bénéficier dans le langage de l'efficacité et de la concision d'écriture des pointeurs, tout en conservant un contrôle sur leur emploi : pas de pointeurs pour le type entier, pas d'arithmétique des pointeurs, pas de pointeur de pointeur, etc.

Cette section comporte une première partie sur les obstacles qui s'opposent à l'analyse statique des programmes avec pointeurs et effets de bord. Nous donnons ensuite une vue d'ensemble des travaux qui entreprennent d'apporter une réponse, même partielle, à ce problème. Enfin, nous esquissons les principes de la méthode implantée dans $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ et qui sera détaillée dans les Sections 5.2 et 5.3.

5.1.1 Pointeurs, effets de bord et intraitables alias

Efficaces, les pointeurs sont aussi réputés pour leur propension à induire des programmes erronés et illisibles. En effet, leur emploi requiert de vérifier précautionneusement l'absence de fuite de mémoire (*memory leak*), l'absence de pointeurs sur des adresses invalides (*dangling pointer*), l'absence de déréférencement invalide de pointeurs, l'absence de dépassement de capacité de la mémoire accessible au programme (*heap overflow*) ; cela, bien entendu, en plus des propriétés spécifiques à chaque programme.

De surcroît, raisonner sur des programmes manipulant des pointeurs, ou des structures de données utilisant des pointeurs, est notoirement difficile. Les raisons en sont les possibilités d'allocation dynamique de mémoire, la mise à jour « destructive » des structures et, de façon prépondérante, les *alias*, c'est-à-dire, la capacité d'un objet d'être connu sous plusieurs noms différents parce que plusieurs références pointent sur lui[11]. Par exemple, du fait des alias, l'habitude de considérer comme distincts des objets de noms différents n'est plus valide, et une propriété portant sur une variable v_1 peut n'être plus vraie après une opération sur une variable v_2.

Nous prendrons pour définition des alias celle de W. Landi (1992) :

Définition 5.2 (alias). Un *alias* est créé au cours de l'exécution d'un programme lorsque deux noms ou plus dans le programme (*e.g.*, des variables) désignent la même zone de mémoire.

Nous verrons par la suite qu'en $\mathrm{Sub}_{\mathcal{C}}$, l'ajout des listes mutables et de l'appel par référence fournira deux manières de créer des alias :

- soit explicitement, par l'intermédiaire des variables de type `list` qui seront des pointeurs, *e.g.*, après l'affectation `liste1 = liste2`, les variables `liste1` et `liste2` sont des alias ;
- soit implicitement,
  - quand la même variable de type liste est passée plusieurs fois en paramètre d'un sous-programme, *e.g.*, après l'appel `f(liste1, liste1)`, les deux paramètres formels du sous-programme `f` sont des alias ;

[11] Dans une tentative humoristique de dédouaner les pointeurs de difficultés qui lui semblent de préférence imputables à la faculté humaine de donner plusieurs noms à une même chose, B. Meyer (2003) raconte combien la lecture d'*Anna Karenine* de Tolstoï a pu dérouter plus d'un amateur de littérature russe, lorsqu'il s'agit de se souvenir en page 467 que « Darya Alexandrovna » n'est autre que la « comtesse Oblonsky » de la page 5, aussi appelée « Dounia » page 35 et simplement « Dolly » partout ailleurs.

– à travers l'appel par référence : quand la même variable est passée plusieurs fois selon ce mode lors d'un appel de sous-programme, tous les paramètres formels correspondants sont alias dans le corps du sous-programme appelé (*cf.* Section 5.3 et l'Exemple 5.36 Page 116).

Lorsque nous raisonnons sur un programme, alias et effets de bord sont étroitement liés par la nécessité de « propager » à tous les alias d'un même objet le résultat de chaque effet de bord sur cet objet.

Exemple 5.3. Prenons l'exemple du fragment de programme `C` suivant :

```
int i = 0;
int *j = &i;
i = 1;
suite(*j);
```

Après les initialisations, la variable `j` est un pointeur sur l'entier `i` ; donc, `*j` vaut `0`. L'affectation de `1` à `i` change l'état de la variable `i`, et donc de `*j`. Lorsque nous raisonnons sur le programme, l'affectation de la nouvelle valeur à `i` doit aussi être « propagée » à `*j` qui vaut donc maintenant `1` dans l'appel `suite(*j)`. ♦

Un prérequis à cet objectif est de déceler tous les alias d'un objet en tout point d'un programme. Malheureusement, ce problème est *indécidable* pour les langages, tels $\text{Sub}_{\mathcal{C}}^{ext}$, qui disposent de l'affectation, d'instructions conditionnelles et itératives et de mémoire dynamique.

Plus précisément, l'analyse d'alias dans un programme consiste généralement à déterminer s'il existe une exécution du programme telle que deux noms x et y sont des alias d'un même objet en un point particulier du programme[12]. Le problème de déterminer quels sont les chemins exécutables d'un programme étant lui-même indécidable, il est traditionnel en analyse statique d'ignorer les conditions des instructions de contrôle de flot et donc de considérer que tout chemin dans un programme est exécutable. Malgré cela, W. Landi (1992) a montré que l'analyse d'alias ainsi définie, pour un programme, même réduit à un seul sous-programme non récursif, écrit dans un langage disposant d'allocation de mémoire dynamique, de structures de données récursives et des instructions conditionnelles et itératives, est indécidable[13]. Plus tard, V. T. Chakaravarthy (2003) a étendu ce résultat aux langages comportant uniquement des variables scalaires, *i.e.*, dépourvus de structures et de tableaux[14].

Par conséquent, en pratique, les analyses d'alias automatiques se bornent à approximer les ensembles réels d'alias. Les méthodes existantes cherchent un compromis entre efficacité et précision. De même que la complexité des analyses varie de linéaire à doublement exponentielle, leur précision dépend de nombreux facteurs qui peuvent aider à

[12] Une variante de l'analyse d'alias est de déterminer si deux noms x et y sont alias en un point particulier du programme pour *toutes* les exécutions de celui-ci. Ce problème est connu pour être plus difficile.

[13] La démonstration repose sur une réduction du problème de l'analyse d'alias au « problème de l'arrêt ».

[14] L'auteur réduit le problème de l'analyse d'alias à celui, indécidable, de savoir si un polynôme à plusieurs variables avec coefficients entiers possède des racines entières (ce problème est aussi connu comme le dixième problème de Hilbert).

classer la profusion de méthodes hétéroclites à la terminologie confuse. À titre d'exemple (voir [Hind, 2001] pour un panorama plus complet), les méthodes d'analyse d'alias peuvent tenir compte des instructions de contrôle de flot (*e.g.*, [Choi *et al.*, 1993]), ou ne pas en tenir compte, puis se partager entre celles qui traitent l'affectation comme étant bidirectionnelle (*e.g.*, [Steensgaard, 1996]) et celles pour qui l'affectation est unidirectionnelle (*e.g.*, [Andersen, 1994]), etc.

Souvent utilisée à des fins d'optimisation des programmes par les compilateurs, l'analyse d'alias, pour être valide, doit être conservative (*safe*) du point de vue de la compilation, c'est-à-dire qu'elle ne doit pas conduire à des optimisations erronées. Très souvent, ce critère se traduit concrètement par une approximation qui surestime l'ensemble réel des alias. Cependant, dans notre cas, nous ne pouvons pas nous contenter d'une approximation : la connaissance exacte des alias dans un programme nous est nécessaire pour déterminer précisément quelles sont les variables affectées par un effet de bord. Nous attendons d'une analyse d'alias qu'elle nous indique si, pour une exécution donnée, deux noms dans un programme sont alias en un point particulier du programme. En effet, il serait tout aussi incorrect d'ignorer un effet de bord sur une variable v parce que nous ignorons que v est un alias de l'objet dont l'état a changé, que de modifier la valeur de v parce que nous croyons à tort que cette variable est liée à l'objet modifié. Dans les deux cas, par sous-estimation ou surestimation de l'ensemble réel des alias, la traduction en équations des programmes pourrait être erronée.

5.1.2 Approches existantes : raisonner malgré tout

Face au constat dressé dans la section précédente, il n'est pas étonnant que les programmes manipulant des pointeurs défient l'analyse statique depuis des décennies. La formalisation du raisonnement sur les pointeurs est toujours l'objet d'actives recherches.

Une façon de restreindre quelque peu le problème, tout en gardant un intérêt pratique manifeste, est de concentrer les efforts sur les structures de données avec pointeurs. Néanmoins, même dans ce cadre limité, les structures de données récursives représentent un nombre potentiellement infini d'objets. Par conséquent, l'automatisation des analyses réclame une approximation finie du nombre de ces objets. Deux voies sont principalement explorées dans ce sens, chacune apportant son contingent de situations d'échec :

- Les structures de données peuvent être limitées à une profondeur arbitrairement fixée (*k-limiting*). Cependant, lorsqu'un élément de la structure au-delà de la limite devient un alias, tous les éléments suivants le deviennent aussi et il s'en suit une analyse erronée du programme.

- Plusieurs objets, éventuellement une infinité d'objets, peuvent être fusionnés en un seul. Cette technique conduit à des structures de données dont la précision de la représentation est morcelée. Elle a ainsi tendance à introduire de faux cycles et ne permet pas, par exemple, de distinguer entre une liste linéaire et une liste cyclique.

D'autres approches ont renoncé à l'automatisation pour proposer des méthodes exactes souvent plus expressives dans les propriétés de programmes qu'elles permettent de formuler. Outre l'automatisation et la précision, les approches de la vérification de programmes

manipulant des pointeurs diffèrent par leurs domaines d'application et leur capacité d'extension.

Logique des listes chaînées

Cette logique (*Linked Lists Logic* [Ranise et Zarba, 2005] ou LLL) est confinée au raisonnement sur les programmes manipulant des listes chaînées. Elle permet d'exprimer des propriétés sur la structure des listes (*e.g.*, existence d'un cycle, accessibilité des cellules) ainsi que sur les données contenues dans les listes. La modularité de LLL lui permet d'être étendue par d'autres logiques.

LLL manipule des « configurations de mémoire » (*i.e.*, des fonctions associant des adresses à des cellules de la mémoire) qui apparaissent explicitement dans les formules de la logique. Les auteurs montrent que le problème de la satisfaisabilité d'une formule de la logique est décidable. L'algorithme de décision suggéré est NP-complet.

Cette logique peut être utilisée pour annoter dans le style de la logique de Floyd-Hoare, notamment avec des invariants de boucle, des programmes écrits dans un langage impératif simplifié afin d'en montrer la correction partielle.

Logique des assertions sur pointeurs

A. Møller et M. I. Schwartzbach (2001) utilisent la notation formelle PAL (*Pointer Assertion Logic*) pour annoter un programme impératif. Le programme et ses annotations sont transformés en un automate. La correction du programme est ensuite vérifiée automatiquement avec l'outil PALE (*Pointer Assertion Logic Engine*). Si la vérification échoue, un contre-exemple est exhibé. Toutefois, l'algorithme de décision n'est pas « élémentairement récursif », *i.e.*, aucune puissance d'exponentielles de hauteur finie ne peut en borner la complexité. En conséquence, certains programmes ne peuvent pas être vérifiés en un temps raisonnable.

Le domaine d'application recouvre une intéressante classe de structures de données incluant, entre autres, les listes doublement chaînées et différentes sortes d'arbres. Des propriétés sur les données contenues dans les structures (*e.g.*, la liste est-elle triée ?) peuvent aussi être formulées, au prix de modifications peu naturelles dans les structures de données. Pour pallier les difficultés qu'introduisent les boucles et la récursivité, la méthode requiert la spécification d'invariants détaillés. Ceci rend la méthode inappropriée pour de grands programmes et la réserve pour la vérification de types de données dont l'implantation est critique.

Analyse des formes

Les questions auxquelles l'analyse des formes (*shape analysis*) cherche à répondre concernent principalement le comportement structurel des structures de données (*e.g.*, l'arbre en entrée du programme est-il encore un arbre en sortie ?) et plus généralement les propriétés qui nécessitent de savoir quels éléments d'une structure sont accessibles depuis un élément donné. Subsidiairement, ce type d'analyse peut vérifier certaines propriétés sur les alias et le partage d'éléments entre structure de données allouées dynamiquement.

L'algorithme présenté dans [Sagiv *et al.*, 1998] associe à chaque état d'un programme un graphe des configurations de la mémoire. Pour que l'analyse termine, le nombre de nœuds du graphe doit être fini. En présence de boucles, l'analyse ne réclame pas d'invariant mais effectue une approximation : les cellules de la mémoire qui ne sont pas pointées par des variables sont fusionnées en un même nœud. Cette approximation peut conduire à de faux rejets.

Dans [Sagiv *et al.*, 2002], les auteurs présentent une analyse des formes paramétrable selon la structure de données observée ou le niveau de précision désiré. En principe, la classe de structures de données analysable est très riche, *e.g.*, plus riche que celle de la logique des assertions sur pointeurs. Cependant, chaque nouvelle structure de données exige la définition, et la preuve de correction, d'un ensemble d'actions élémentaires propres à la méthode. Le raisonnement sur les données des structures est astreint au même type de contraintes. L'analyse est fondée sur la représentation de la mémoire par des structures dans une logique à trois états. L'avantage de disposer d'une logique à trois états est de clairement distinguer le vrai et le faux de l'indéterminé.

Logique des effets

Cette branche des recherches pour raisonner sur les programmes impératifs avec effets remonte aux premiers travaux de Ian A. Mason sur les données mutables en Lisp. La logique des effets (*Variable Typed Logic of Effects* ou VTLoE) axiomatise une relation d'équivalence opérationnelle pour des programmes fonctionnels avec des traits impératifs [Honsell *et al.*, 1995; Mason, 1997]. Cette logique, conçue pour montrer la divergence d'un programme ou bien son équivalence avec un autre, permet aussi d'exprimer une riche variété de propriétés. La logique est complète dans sa variante du premier ordre.

La logique, encore en phase de développement, ne cesse d'évoluer et plusieurs problèmes restent ouverts. Parmi ceux-ci, les auteurs posent celui d'inclure des quantificateurs permettant de raisonner de façon moins globale sur la mémoire pour ne désigner que les cellules « locales » au contexte courant.

Raisonnement local

Une partie des difficultés rencontrées par les précédentes méthodes vient de l'obligation de raisonner sur la totalité de la mémoire. Le raisonnement local (appelé aussi *separation logic* [O'Hearn *et al.*, 2001; Reynolds, 2002]) est une extension de la logique de Floyd-Hoare pour raisonner sur les pointeurs. Dans cette dernière, la règle de l'affectation, notamment, n'est plus valide en présence d'alias. Le principal apport de la logique de séparation est l'introduction d'une « conjonction de séparation » qui découpe une formule en plusieurs sous-formules vraies pour des parties disjointes d'une structure de données. Cette solution permet de raisonner « localement » sur des segments indépendants de la mémoire et permet ainsi des preuves souvent succinctes.

La logique de séparation peut représenter tout type de structure de données, l'arithmétique des pointeurs, la libération explicite de mémoire et le partage des variables dans un cadre concurrent. L'expressivité de la logique est néanmoins jugulée par son automatisation. Pour l'instant, seuls des sous-ensembles de la logique, *e.g.*, soit spécifiques

aux listes chaînées [Berdine *et al.*, 2004], soit restreints à des programmes sans boucles pour une classe réduite d'assertions [Berdine *et al.*, 2005], se sont avérés décidables. Par l'intermédiaire d'une traduction vers la logique du premier ordre [Calcagno *et al.*, 2005], le sous-ensemble spécifique aux listes chaînées peut tirer profit des démonstrateurs de théorèmes existants.

Logique d'ordre supérieur

Une famille de travaux, dont un descendant est exposé dans [Mehta et Nipkow, 2003], recourent à une logique d'ordre supérieur pour modéliser la mémoire comme un ensemble de fonctions des adresses vers les valeurs. Cette approche n'est pas limitée à des structures de données particulières et peut s'appliquer à toutes les propriétés d'un programme, *i.e.*, pas uniquement aux propriétés liées aux pointeurs. Le programme à vérifier est annoté dans cette logique et un générateur de conditions de vérification produit un ensemble « d'obligations de preuve » à destination d'un système de preuve générique (*e.g.*, HOL). Le degré d'automatisation de ces preuves est très variable et la question de la satisfaisabilité des formules de la logique est peu abordée.

5.1.3 Relever le défi dans *SOS*$\mathbf{Sub}_{\mathcal{C}}^{ext}$

Avec le langage $\text{Sub}_{\mathcal{C}}^{ext}$, nous avons introduit la possibilité de former des alias dans les programmes. De ce fait, nous sommes confrontés au problème de l'indécidabilité de l'analyse statique des relations d'alias. Pourtant, nous souhaitons définir une axiomatisation *exacte* de la sémantique des programmes $\text{Sub}_{\mathcal{C}}^{ext}$ dans la logique équationnelle. Cette contrainte élimine d'emblée les analyses d'alias fondées sur des approximations : leur niveau de précision limiterait par trop la classe de programmes acceptée ou les propriétés qu'il serait possible de montrer. Les autres approches dont nous avons connaissance contournent le problème de l'indécidabilité d'une analyse statique des alias et de l'espace non borné de recherche en le déplaçant dans la logique constitutive de la méthode, elle-même indécidable. Principalement, deux situations se rencontrent :

- Dans les logiques de type Floyd-Hoare, l'obligation de fournir des invariants permet de raisonner sur les programmes avec boucle et récursivité comme si les programmes n'en comportaient pas[15]. En effet, dans le cas d'une boucle, par exemple, il suffit de prouver que l'invariant de boucle est vrai après *une exécution* du corps de la boucle en supposant qu'il est vrai juste avant la boucle. Une boucle est ainsi ramenée à un fragment de programme sans itération.

- Les autres logiques, *e.g.*, la logique de séparation, la logique des effets, les logiques d'ordre supérieur, sont des logiques non standard comportant des règles d'inférence et des axiomes *ad hoc* pour raisonner sur les pointeurs et la mémoire.

Pour *SOS*$\text{Sub}_{\mathcal{C}}^{ext}$, nous avons dû écarter les solutions mises en œuvre dans les autres méthodes car elles ne correspondent pas aux objectifs que nous nous sommes fixés :

[15] Ceci à condition de ne considérer que la *correction partielle* d'un programme. La *terminaison* du programme doit être montrée en sus.

- *La logique cible de l'axiomatisation des programmes demeure la logique équationnelle.* Sans revenir sur les raisons initiales du choix de la logique équationnelle, exposées en Section 2.5.5, nous soulignons que cette logique ne circonscrit pas les propriétés de programmes qu'il est possible d'exprimer à la question des structures de données, comme peuvent le faire des logiques plus spécialisées, *e.g.*, la logique des listes chaînées. Cependant, à la différence des logiques spécifiques, la logique équationnelle ne possède pas, en propre, les nouveaux mécanismes que nous introduisons dans le langage $\mathrm{Sub}_{\mathcal{C}}^{ext}$. Par exemple, auparavant, la notion d'appel par valeur trouvait un équivalent naturel dans la logique équationnelle en l'application d'un symbole de fonction. L'équivalence ne sera pas aussi directe avec l'appel par référence et les effets de bord sur les listes : nous devrons les simuler en combinant les mécanismes existants, plus fondamentaux, de la logique équationnelle.

- *Les programmes ne sont pas annotés.* L'annotation d'un programme, en raison de la recherche des invariants, est une tâche souvent aussi difficile que la preuve de la correction du programme elle-même. Dans notre approche, nous travaillons directement sur le code source afin d'automatiser la traduction en équations du programme. De plus, ceci permet de mieux distinguer l'activité de programmation de celle de la preuve.

- *La mémoire n'est pas modélisée dans la logique.* La majorité des approches est assortie d'un modèle de la mémoire des programmes qui sert de support au raisonnement sur les pointeurs. Dès lors, la mémoire apparaît explicitement dans les formules et les règles d'inférence de la logique. Bien qu'une modélisation de la mémoire soit envisageable en logique équationnelle, comme l'attestent, par exemple, J. A. Goguen et G. Malcolm (1996) qui édifient leur méthode pour la preuve de programme sur une abstraction de la mémoire (appelée *store* et qui associe des entiers à des expressions de leur langage de programmation), ce procédé complique la théorie dans laquelle les preuves de programme sont conduites. La nécessité de quantifier sur la mémoire obscurcit l'énoncé des équations du programme et de ses propriétés. En outre, les schémas d'induction sur les listes ne sont plus aussi intuitifs que lorsqu'ils se déduisent de la définition usuelle du type de données récursif pour les listes.

- *Les programmes ne sont pas complets.* La plupart du temps, $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ sera appliqué à des collections de sous-programmes plutôt qu'à un programme complet, *i.e.*, un programme avec un sous-programme principal servant de point d'entrée dans le programme (*e.g.*, la fonction *main* en langage `C`). En conséquence, au cours de l'analyse statique, nous ne connaissons pas tous les contextes d'appel des sous-programmes. Le programme de tri par fusion de la Figure 1.6 (Page 19) est un exemple de programme incomplet, pour lequel nous ne savons pas dans quel contexte sera appelé le sous-programme `mergeSort`.

Parmi les contraintes que nous venons d'énumérer, nous renonçons à modéliser la mémoire explicitement dans l'axiomatisation de la théorie des programmes. Pourtant, avec les listes et l'opérateur de création de maillon `add`, le langage $\mathrm{Sub}_{\mathcal{C}}^{ext}$ dispose d'allocation dynamique de mémoire. Or, comme l'affirment S. Horwitz *et al.* (1989), l'analyse statique

d'un programme avec allocation dynamique de mémoire réclame la notion de cellule de la mémoire. En effet, les variables du programme ne suffisent plus à nommer tous les emplacements de la mémoire accessibles au programme. Pour résoudre ce problème, nous adoptons un *schéma de nommage* pour les cellules de la mémoire dynamiquement allouées. Cette technique nous permet de nommer tous les objets accessibles à un programme et de leur associer des variables de la logique pour les désigner dans les équations produites par l'axiomatisation du programme (*cf.* Section 5.2.1). Ainsi, nous serons capables de faire apparaître les maillons des listes au fur et à mesure que le programme y accède (*cf.* Section 5.2.2).

Le choix d'exprimer, sans modèle de la mémoire, les sous-programmes $\mathrm{Sub}_{\mathcal{C}}^{ext}$ comme des fonctions de transfert, des données en entrée vers les données en sortie, implique que nous ne modéliserons pas non plus dans la logique cible les mécanismes de création et de modification des alias. Par conséquent, c'est pendant l'axiomatisation que l'analyse des alias sera conduite. Afin qu'elle soit automatique, nous serons amenés à poser des *prérequis* sur l'existence et la formation des alias dans les programmes $\mathrm{Sub}_{\mathcal{C}}^{ext}$ (l'Annexe B offre une synthèse des prérequis présentés au fil du texte).

D'autre part, nous souhaitons aussi traiter les programmes incomplets. Ceci nous interdit de mener une analyse interprocédurale[16] significative puisque, dans ces conditions, le contexte d'appel d'un sous-programme peut ne pas être connu. Nous avons donc décidé dans $SOS\mathrm{Sub}_{\mathcal{C}}$ d'analyser chaque sous-programme indépendamment des autres. Ce point de vue était valide dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ étant donné que le langage $\mathrm{Sub}_{\mathcal{C}}$ ne comportait que des fonctions avec appel par valeur. Ainsi, les fonctions étaient insensibles au contexte de l'appel, *i.e.*, le résultat était entièrement déterminé par les valeurs des paramètres au moment de l'appel de la fonction. En raison des possibilités d'alias, la situation est changée avec $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$: les alias dans le contexte d'appel peuvent influer sur le comportement d'un sous-programme relativement à des entrées de mêmes valeurs[17] (*cf.* Exemple 5.4). Or, en l'absence du contexte d'appel, les combinaisons d'alias sont trop nombreuses (voire non bornées dans le cas des listes) pour être systématiquement prises en compte lors de l'analyse d'un sous-programme. Pour contourner cette impossibilité et mener une analyse intraprocédurale exacte, nous imposerons des restrictions sur les alias qui portent sur les paramètres des sous-programmes $\mathrm{Sub}_{\mathcal{C}}^{ext}$ avant un appel (*cf.* Sections 5.2.2 et 5.3.2). Nous verrons aussi, Section 5.2.2, que le même type de prérequis sera nécessaire, en présence de boucles, sur les variables d'un sous-programme.

Exemple 5.4 (influence des alias du contexte d'appel). Soit le sous-programme `C` suivant :

```
int modifierTete(LIST La, LIST Lb) {
  La->element = 1;
  Lb->element = 2;
  return La->element;
}
```

[16]Type d'analyse qui recoupe les informations collectées sur l'ensemble des sous-programmes examinés ; par opposition aux analyses *intraprocédurales* qui examinent chaque sous-programme séparément en abstrayant les contextes d'appel, ce qui implique généralement une perte de précision.

[17]La valeur d'une liste se comprend ici comme la séquence des valeurs des éléments qui la composent.

FIG. 5.1 – Influence du contexte d'appel

Le type `LIST` est défini comme à la Figure 1.2 de la Page 7. Les deux premières instructions du sous-programme `modifierTete` affectent 1, respectivement 2, au contenu du premier élément de `La`, respectivement `Lb`. Considérons les deux contextes d'appel de la Figure 5.1, avec une représentation « boîtes et flèches » des pointeurs.

Dans les deux contextes, la valeur des listes est : $La = 0 \cdot \text{NULL}$ et $Lb = 0 \cdot \text{NULL}$. Avec le contexte (a), après l'appel `modifierTete(La, Lb)`, les nouvelles valeurs sont : $La = 1 \cdot \text{NULL}$ et $Lb = 2 \cdot \text{NULL}$. La valeur renvoyée par le sous-programme est donc 1.

Par contre, avec le contexte (b), après le même appel `modifierTete(La, Lb)`, les nouvelles valeurs sont : $La = 2 \cdot \text{NULL}$ et $Lb = 2 \cdot \text{NULL}$. La fonction renvoie donc la valeur 2.

La différence de comportement du sous-programme sur des entrées de « même valeur » s'explique par l'existence d'un alias entre `La` et `Lb` dans le contexte (b). Sans cette information, la valeur de retour du sous-programme ne peut être déterminée. C'est précisément cette information qui manque lorsque le programme à analyser est incomplet. ♦

Cette présentation de $SOS\text{Sub}_{\mathcal{C}}^{ext}$ ne serait pas complète sans mentionner l'abandon de la réécriture dans le système $SOS\text{Sub}_{\mathcal{C}}^{ext}$. La définition du système n'est plus donnée sous la forme d'un système de réécriture. Le plus grand nombre et la plus grande complexité des structures de données et algorithmes nécessaires à l'expression de $SOS\text{Sub}_{\mathcal{C}}^{ext}$ auraient rendu le système de réécriture peu compréhensible et peu pratique à manipuler ; sans même mentionner les performances d'exécution médiocres que nous aurions obtenues en implantant $SOS\text{Sub}_{\mathcal{C}}^{ext}$ sous cette forme.

Suite à l'abandon de la réécriture, les étapes de transformation des programmes $\text{Sub}_{\mathcal{C}}$ en terme et de réécriture des termes sont fusionnées dans la nouvelle implantation de $SOS\text{Sub}_{\mathcal{C}}^{ext}$. L'étape de génération des équations à partir des environnements des sous-programmes subsiste.

5.2 Listes mutables

Dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$, le type liste se comporte similairement aux listes des langages de programmation fonctionnels. Bien que les listes soient des séquences d'éléments, elles constituent un type « atomique » : *i.e.*, l'ensemble de la séquence est une unique valeur. Par exemple, le langage ne permet pas de modifier la valeur d'un élément particulier sans construire une nouvelle liste, ou encore, l'affectation d'une liste à une autre implique la recopie de tous les éléments de la liste affectée, il en va de même lorsqu'une liste est passée en paramètre d'une fonction.

Les objets mutables d'un programme sont les objets qui peuvent changer d'état au cours du programme. Nous avons souligné les bénéfices qu'apporte l'introduction d'un type de liste mutable dans un langage de programmation (*cf.* Section 5.1) et nous avons illustré par l'exemple du tri par fusion l'emploi de ce type de données (*cf.* Section 1.1.2). Mise à part leur indéniable efficacité, ce type d'objets mutables est communément disponible dans les langages impératifs et présente à ce titre un intérêt pour notre méthode.

Nous présentons dans cette section les ajouts faits au langage $\text{Sub}_{\mathcal{C}}$ afin d'inclure les listes mutables. Nous commençons par les aspects syntaxiques à la Section 5.2.1. Nous poursuivons avec la sémantique des nouvelles constructions du langage en Section 5.2.2 où nous détaillons aussi les prérequis qu'elles imposent. Enfin, la Section 5.4 décrit la dernière étape de l'axiomatisation des programmes écrits en $\text{Sub}_{\mathcal{C}}^{ext}$: la formulation des équations.

5.2.1 Représentation des listes mutables

Nous introduisons avec *SOS*$\text{Sub}_{\mathcal{C}}^{ext}$ les listes mutables. Il s'agit d'une structure de données récursive avec pointeur suivant le modèle du type `LIST` (en langage `C`) de la Figure 1.2 (Page 7). Ainsi, une liste est un pointeur, nous dirons aussi une référence, sur un maillon de liste. Un maillon est composé d'un champ `element` et d'un champ `next`. Le champ `element` accepte indifféremment un objet de type liste ou de type entier. Le champ `next` est une liste. Un maillon n'est pas en soi un type du langage : seuls ses champs `element` et `next` sont accessibles par l'intermédiaire d'une référence sur un maillon, *i.e.*, un objet de type liste.

Le langage $\text{Sub}_{\mathcal{C}}^{ext}$

Le langage $\text{Sub}_{\mathcal{C}}^{ext}$ est une extension du langage $\text{Sub}_{\mathcal{C}}$ dans laquelle le type `list` est modifié afin de correspondre aux listes mutables des langages impératifs et dans laquelle nous disposons d'un passage des paramètres par référence (cet ajout au langage $\text{Sub}_{\mathcal{C}}$ est traité séparément dans la Section 5.3). Excepté ces extensions, le langage est inchangé.

Nous retrouvons les trois opérations spécifiques aux listes du langage $\text{Sub}_{\mathcal{C}}$, mais avec une sémantique différente (nous supposons que `L` est une liste) :

- Comme attendu, `element(L)` renvoie la valeur du champ `element` du maillon référencé par `L`, c'est-à-dire la *tête* de la liste `L`. Notons que dans le cas où la tête est elle-même une liste, c'est une référence qui est renvoyée.
- Sur le même principe, `next(L)` renvoie la valeur du champ `next` du maillon référencé par `L`, c'est-à-dire la *queue* de la liste `L`. Cette fois encore, la valeur renvoyée est une référence sur une liste.
- Enfin, `add(elt, L)` permet d'allouer dynamiquement un nouveau maillon et renvoie une référence sur celui-ci. Les champs `element` et `next` du nouveau maillon sont initialisés, respectivement, à `elt` et `L`.

Nous poursuivons l'inventaire avec les deux nouveaux opérateurs de mutation sur les listes :

– L'opérateur `->element` est appliqué à un objet de type liste en membre gauche d'une affectation. Il permet d'affecter une nouvelle valeur au champ `element` du maillon référencé par la liste à laquelle il s'applique. Par exemple, pour modifier la valeur de l'élément en tête de la liste `L`, nous écrirons : `L->element = valeur`. Autre exemple, `L->element->element` permet d'accéder à l'élément en tête de la liste `L->element` qui est elle-même le premier élément de la liste `L` (`L` est donc une liste de listes).

– De même, l'opérateur `->next` permet d'affecter une nouvelle valeur au champ `next` du maillon référencé par la liste à laquelle il s'applique. Par exemple, pour modifier la liste en queue de `L`, nous écrirons : `L->next = valeur`.

Bien sûr, les deux opérateurs peuvent être combinés pour modifier le champ souhaité d'un maillon quelconque d'une liste, *e.g.*, `L->next->element` permet de changer la valeur du deuxième élément de la liste `L`.

Enfin, nous complétons par l'affectation entre listes qui est maintenant l'affectation d'une référence à un pointeur : il n'y a donc plus de recopie de la liste affectée comme c'était le cas dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$. Le membre gauche d'une affectation entre listes est toujours une variable du programme ; éventuellement, une combinaison des opérateurs de mutation a pu être appliquée à la variable. Ainsi, une affectation telle que `f() = L`, où `f` est une fonction renvoyant une liste, n'est pas correcte.

Note 5.5. D'une manière générale, nous imposons que les listes renvoyées par des sous-programmes soient toujours affectées avant d'être utilisées. Notamment, nous n'acceptons pas que la valeur de retour d'un sous-programme, lorsqu'elle est de type liste, soit directement passée en paramètre d'un autre sous-programme, comme dans `g(f())`, avec `g` un sous-programme ou un opérateur du langage prenant une liste en paramètre. Ce comportement n'est pas restrictif puisqu'il est toujours possible d'employer une variable locale comme intermédiaire : *e.g.*, `L = f(); g(L)`.

Schéma de nommage

Avec l'introduction des pointeurs pour les listes mutables et en présence d'allocation dynamique (avec l'opérateur `add`), toutes les valeurs accessibles à un programme ne sont plus désignées *uniquement* par les variables du programme. Afin de raisonner sur les cellules mémoires créées dynamiquement et d'être en mesure de les comparer pour détecter les alias potentiels, nous donnons un nom à ces objets *anonymes*. Le rôle du schéma de nommage est double :

– nommer de manière unique les éléments des listes manipulés par un sous-programme (voir l'Exemple 5.6 cas 1) ;

– nommer de manière unique toutes les références accessibles à un sous-programme (voir l'Exemple 5.6 cas 2).

Exemple 5.6 (nommer les anonymes). Soit le sous-programme `anonymes` suivant :

```
1  list anonymes(list L) {
2    list La;
3    int i;
4
5    i = element(L);
6    L = add(1, add(2, NULL));
7    La = next(L);
8    L = NULL;
9    ⋮
10 }
```

Le sous-programme `anonymes` illustre deux situations dans lesquelles apparaissent des objets anonymes.

1. En ligne 5 du sous-programme, la variable `i` prend la valeur de l'élément en tête de la liste `L`. Or, cette liste étant passée en paramètre de `anonymes`, nous ne connaissons donc pas la valeur affectée. Pour poursuivre l'analyse, nous ne pouvons pas conserver l'information `i = element(L)` telle quelle. En effet, cette assertion serait déjà fausse à la ligne suivante puisque la valeur de `L` est modifiée. Notre solution consiste à donner un nom, unique et invariable, aux éléments des listes passées en paramètre. Dans cet exemple, nous aurions nommé e_{1_L} l'élément considéré, et ce, du début à la fin du sous-programme. L'analyse se poursuit alors avec $i = e_{1_L}$. Le nom e_{1_L} est construit à partir du nom de la liste (ici, `L`) et du rang de l'élément dans la liste (ici, il s'agit du premier élément de la liste `L`).

2. En ligne 6 du sous-programme, deux nouveaux maillons sont créés. Ces objets n'ont pas de nom propre et doivent, pour l'instant, être manipulés par l'intermédiaire de la variable `L`. Lorsqu'à la ligne suivante, `La` pointe sur la queue de `L`, entre autres, nous pouvons envisager de :

   – donner pour valeur à `La` celle du maillon pointé (*e.g.*, $La = (2, \text{NULL})$), mais nous perdons alors la relation qui existe entre `L` et `La` ;

   – garder l'information sous la forme d'une expression de pointeur telle qu'elle apparaît dans le sous-programme ($La = \text{next}(L)$) : il faudra alors la mettre à jour si `L` est modifiée (en effet, à la ligne 8 par exemple, l'expression ne serait plus correcte) et manipuler des expressions de pointeur de plus en plus complexes dans la suite de l'analyse.

   Notre solution est de faire apparaître explicitement au cours de l'analyse les références sur les maillons ainsi créés. Dans cet exemple, nous aurions nommé les références sur les maillons : $\text{ref}_1 = (2, \text{NULL})$ et $\text{ref}_2 = (1, \text{ref}_1)$. Cela permet d'associer à `La` un nom plutôt qu'une expression de pointeur (*i.e.*, $La = \text{ref}_1$) et de garder la relation d'alias qui lie `next(L)` et `La` (en effet, $L = \text{ref}_2 = (1, \text{ref}_1)$ et donc $\text{next}(L) = \text{ref}_1 = La$). ♦

Plusieurs schémas de nommage des objets anonymes ont déjà été proposés, provenant notamment des travaux sur l'analyse des alias ou l'analyse des formes [Larus et Hilfinger,

1988; Choi *et al.*, 1993]. Chaque fois, le schéma est étroitement lié à l'analyse à laquelle il sert de support. L'objectif est souvent de pouvoir modéliser n'importe quel type de structure de données dynamiquement allouée. De ce fait, leurs schémas de nommage sont conçus pour accompagner la représentation statique de toutes les configurations mémoire produites par un programme donné. Comme dans le cas de l'analyse d'alias, se pose le problème des structures de données potentiellement non bornées, ce qui complique la sémantique des schémas de nommage.

Pour ce qui nous concerne, deux particularités de notre méthode contribuent favorablement à l'élaboration d'un schéma de nommage simple :

- Dans $\text{Sub}_{\mathcal{C}}^{ext}$, la seule structure de données allouée dynamiquement est le type des listes mutables. Le schéma de nommage sera donc restreint à la représentation de ce type. Néanmoins, les listes sont suffisamment expressives, à défaut d'être les plus appropriées, pour coder toutes les autres structures de données (*e.g*, arbres, graphes).

- L'impossibilité de borner la taille des structures de données manipulées par les programmes provient des mécanismes d'itération et de récursivité des langages de programmation. Or, nous disposons des mêmes mécanismes (l'itération étant vue comme une forme particulière de récursivité) dans la logique équationnelle. Alors que les autres méthodes cherchent à représenter les structures de données *en extension*, *i.e.*, en énumérant tous leurs éléments, la récursivité nous libère de cette obligation.

Nous donnons maintenant la définition de notre schéma de nommage pour les éléments des listes et les références allouées dynamiquement.

Éléments nommés de liste

Définition 5.7 (rang). Soient l une liste non vide dans laquelle tous les éléments sont identifiés de manière unique et e un élément de la liste l. Le *rang* de e dans la liste l est $\text{rang}(e, l, 1)$, où rang est récursivement défini comme suit :

- $\text{rang}(e, e \cdot l, r) = r$;

- si e_1 est un élément de type entier tel que e et e_1 ne soient pas les mêmes éléments, alors $\text{rang}(e, e_1 \cdot l, r) = \text{rang}(e, l, r + 1)$;

- si e_1 est une liste telle que e soit différent de e_1 et e ne soit pas un élément de e_1, alors $\text{rang}(e, e_1 \cdot l, r) = \text{rang}(e, l, r + 1)$;

- si e_1 est une liste telle que e soit différent de e_1 et e soit un élément de e_1, alors $\text{rang}(e, e_1 \cdot l, r) = r : \text{rang}(e, e_1, 1)$.

De la définition, il découle que le rang d'un élément est unique. Pour identifier les éléments d'une liste L, nous leur donnons un nom qui dépend de leur rang dans L.

Définition 5.8 (éléments nommés d'une liste). Soit L une liste. Nous nommons e_{r_L} l'élément de rang r dans la liste L. Nous nommons l_{r_L} la sous-liste de L dont le premier élément a pour rang r. Nous appelons e_{r_L} et l_{r_L} des *éléments nommés* de liste et Elt_{nom} l'ensemble de tous les éléments nommés d'un sous-programme.

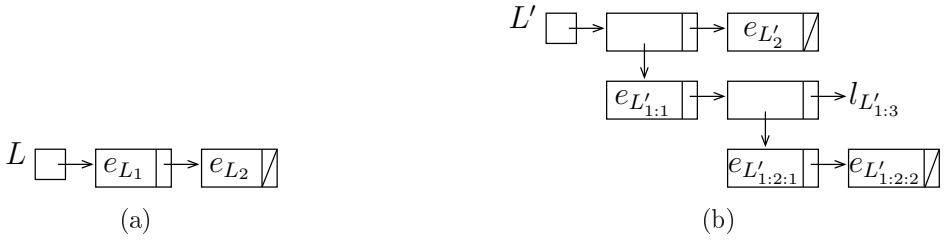


FIG. 5.2 – Éléments nommés

Ce schéma de nommage permet de représenter toutes les listes acycliques en énumérant tous les éléments nommés de la liste selon l'ordre lexicographique de leur rang.

Définition 5.9 (liste nommée). Une *liste nommée* est récursivement définie à partir de la notion d'élément nommé comme suit :

$$\text{liste nommée} ::= \text{NULL} \mid \text{élément nommé} \mid \text{élément nommé} \cdot \text{liste nommée}$$

Exemple 5.10 (éléments et listes nommés). Nous donnons en Figure 5.2 deux exemples de listes, L et L', représentées dans le format boîtes et flèches. Boîtes et flèches sont étiquetées par les éléments nommés qui leur correspondent.

Dans (a), la valeur de la liste L sous une forme fonctionnelle est : $L = e_{1_L} \cdot e_{2_L} \cdot \text{NULL}$. Il s'agit donc d'une liste comportant deux éléments, e_{1_L} et e_{2_L}, qui sont soit des entiers, soit des listes. L'élément nommé l_{3_L} correspondrait ici à la liste vide du dernier maillon de L.

Dans (b), la valeur de la liste L' sous une forme fonctionnelle est :

$$L' = (e_{1:1_{L'}} \cdot (e_{1:2:1_{L'}} \cdot e_{1:2:2_{L'}} \cdot \text{NULL}) \cdot l_{1:3_{L'}}) \cdot e_{2_{L'}} \cdot \text{NULL}$$

Par commodité, nous avons mis entre parenthèses les sous-listes. Le nom des éléments serait cependant suffisant pour déduire cette information. L'élément nommé $l_{1:3_{L'}}$ représente une sous-liste de L', éventuellement vide. Les deux éléments de L' qui sont aussi des listes sont :

$$e_{1_{L'}} = l_{1:1_{L'}} = e_{1:1_{L'}} \cdot (e_{1:2:1_{L'}} \cdot e_{1:2:2_{L'}} \cdot \text{NULL}) \cdot l_{1:3_{L'}}$$

et

$$e_{1:2_{L'}} = l_{1:2:1_{L'}} = e_{1:2:1_{L'}} \cdot e_{1:2:2_{L'}} \cdot \text{NULL}\,.$$

♦

Références nommées de maillon de liste Un maillon de liste est un couple comportant un élément de liste et une référence sur le maillon suivant. Chaque fois qu'un nouveau maillon sera accessible au programme, notamment suite à une allocation dynamique par l'opérateur add, nous lui associerons une référence nommée.

Définition 5.11 (référence nommée et maillon de liste). Une *référence nommée* est un nom construit sur le modèle de ref_n, avec $n \in \mathbb{N}^*$. *Références de liste* et *maillons* sont

définis comme suit :

$$\begin{aligned}\text{référence de liste} ::= &\ \textsc{null} \mid \text{référence nommée}\\ \text{maillon} ::= &\ \text{référence de liste} \cdot \text{référence de liste}\\ \mid &\ \text{entier} \cdot \text{référence de liste}\\ \mid &\ \textsc{null}\\ \mid &\ \text{élément nommé}\end{aligned}$$

Exemple 5.12 (références nommées et maillons). La représentation d'une liste par un chaînage de maillons fournit un modèle de la mémoire dynamique manipulée par les programmes. Dans ce modèle, la mémoire est un ensemble de cellules identifiées par leur adresse. Les maillons des listes sont les cellules de la mémoire. Chaque maillon est identifié par une référence nommée. La mémoire dynamique d'un programme sera donc pour nous un ensemble de couples référence-maillon.

Nous considérons à nouveau les deux listes de la Figure 5.2 et nous donnons leur représentation sous forme de références nommées de maillon :

(a) $\{(\mathrm{ref}_1, \mathrm{ref}_2 \cdot \mathrm{ref}_3), (\mathrm{ref}_2, e_{1_L}), (\mathrm{ref}_3, \mathrm{ref}_4 \cdot \textsc{null}), (\mathrm{ref}_4, e_{2_L})\}$ avec $L = \mathrm{ref}_1$;

(b) $\{(\mathrm{ref}_1, \mathrm{ref}_2 \cdot \mathrm{ref}_3), (\mathrm{ref}_2, \mathrm{ref}_4 \cdot \mathrm{ref}_5), (\mathrm{ref}_3, \mathrm{ref}_6 \cdot \textsc{null}), (\mathrm{ref}_4, e_{1:1_{L'}}), (\mathrm{ref}_5, \mathrm{ref}_7 \cdot \mathrm{ref}_8),$
$(\mathrm{ref}_6, e_{2_{L'}}), (\mathrm{ref}_7, \mathrm{ref}_9 \cdot \mathrm{ref}_{10}), (\mathrm{ref}_8, l_{1:3_{L'}}), (\mathrm{ref}_9, e_{1:2:1_{L'}}), (\mathrm{ref}_{10}, \mathrm{ref}_{11} \cdot \textsc{null}),$
$(\mathrm{ref}_{11}, e_{1:2:2_{L'}})\}$ avec $L' = \mathrm{ref}_1$.

Les ensembles de couples ci-dessus représentent des mémoires contenant les listes L et L', respectivement. Les références nommées sont ici notées ref_1 jusqu'à ref_{11}. ♦

Nous disposons ainsi, avec les listes nommées et les références nommées de maillon de liste, de deux représentations des listes. La première, dite des listes nommées, sera utile pour manipuler la valeur d'une liste sous forme fonctionnelle (*cf.* Exemple 5.10). La seconde, dite des références nommées, permet de modéliser les listes telles que les programmes les manipulent dans la mémoire : un ensemble d'adresses, *i.e*, les références nommées, associées à des cellules de la mémoire, *i.e.*, les maillons de liste (*cf.* Exemple 5.12). Une opération de reconstitution de la valeur d'une liste nous permet de passer de la représentation par références nommées à la représentation par listes nommées (*cf.* Définition 5.20 Page 100).

5.2.2 Sémantique de $\mathrm{Sub}_{\mathcal{C}}^{ext}$: listes mutables

Dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$, la sémantique de $\mathrm{Sub}_{\mathcal{C}}$ est définie par les modifications qu'entraînent les constructions du langage dans un environnement. Ce dernier est une représentation, compacte et appropriée à la réécriture, d'un ensemble d'équations. Pour $\mathrm{Sub}_{\mathcal{C}}^{ext}$, nous enrichissons la notion d'environnement afin de donner un sens aux nouvelles constructions. En effet, celles-ci nécessitent de pouvoir représenter et manipuler la notion de référence sur un maillon de liste. La sémantique des constructions communes à $\mathrm{Sub}_{\mathcal{C}}$ et $\mathrm{Sub}_{\mathcal{C}}^{ext}$ ne change pas et se transpose aisément dans les environnements étendus. Nous ne nous attarderons

donc que sur les constructions particulières à $\text{Sub}_{\mathcal{C}}^{ext}$. Nous traitons dans cette section des listes mutables. Les appels par référence sont étudiés dans la Section 5.3.

Nous commençons par donner la définition des environnements qui serviront à exprimer la sémantique du langage $\text{Sub}_{\mathcal{C}}^{ext}$. Nous définissons aussi les opérations fondamentales sur ces environnements étendus. Dans la Section 5.2.2, nous donnons la sémantique des constructions ayant trait aux listes mutables dans des programmes dépourvus de boucles et de récursivité. Nous verrons alors quels aménagements sont nécessaires pour réintroduire ces mécanismes.

Environnements étendus

Les environnements étendus que nous définissons pour $\text{Sub}_{\mathcal{C}}^{ext}$ ajoutent principalement trois ensembles à celui des paires variable-valeur qui était déjà disponible dans les environnements de $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$.

Définition 5.13 (environnement étendu). Nous notons :

- $\mathcal{V}_p$ l'ensemble des paramètres formels d'un sous-programme ;
- $\mathcal{V}_{loc}$ l'ensemble des variables locales d'un sous-programme ;
- $\mathcal{V}_f$ l'ensemble des variables fraîches (*i.e.*, non déjà utilisées) introduites pour servir de constantes ;
- $\mathcal{V}_{ref}$ l'ensemble des références nommées utilisées.

Un *environnement étendu* d'un sous-programme est le sextuplet $< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >$ tel que :

- $\mathcal{E} \subset \{(var, val) \mid var \in \mathcal{V}_{loc} \cup \mathcal{V}_p \text{ et } val \in \mathbb{Z} \cup \mathcal{V}_{ref}\}$
- $\mathcal{M} \subset \{(\text{ref}, m) \mid \text{ref} \in \mathcal{V}_{ref} \text{ et } m \text{ est un maillon}\}$
- $\mathcal{D} \subset \{(var, l) \mid var \in \mathcal{V}_p \cup \mathcal{V}_f \text{ et } l \text{ est une liste nommée}\}$
- $F \subset \{(var, val) \mid var \in \mathcal{V}_f\}$

Nous désignons par Env^{ext} l'ensemble de tous les environnements étendus.

Note 5.14. Pour alléger la notation, nous pourrons omettre de citer $\mathcal{V}_f$ et $\mathcal{V}_{ref}$. Nous écrirons alors un environnement étendu sous la forme du quadruplet $< \mathcal{E}, \mathcal{M}, \mathcal{D}, F >$.

L'ensemble $\mathcal{E}$ est donc l'équivalent de la liste des valeurs des variables d'un sous-programme qui existait dans les environnements de $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$. C'est l'état des variables du sous-programme.

L'ensemble $\mathcal{M}$ est celui des maillons de liste manipulés par un sous-programme. Il contiendra non seulement les maillons alloués dynamiquement, mais aussi les maillons des listes passées en paramètre auxquels accède le sous-programme. Dans $\mathcal{M}$, les listes sont

représentées sous la forme de références nommées de maillon. Cet ensemble permet de suivre l'évolution de la mémoire dynamique d'un sous-programme[18].

L'ensemble $\mathcal{D}$ permettra de représenter les listes construites hors du sous-programme et dont la structure est donc inconnue. Cet ensemble tient à jour la valeur des listes telles qu'elles sont appréhendées par le sous-programme : n'apparaissent que les éléments auxquels le sous-programme a accédé par une opération de lecture ou d'écriture dans la liste. Dans $\mathcal{D}$, les listes sont représentées par des listes nommées.

Enfin, F est l'ensemble qui conserve les valeurs des variables fraîches introduites au cours de l'axiomatisation pour servir de constantes. Elles seront nécessaires pour identifier une liste obtenue suite à un appel de sous-programme (*cf.* Définition 5.31 Page 110).

L'interprétation sémantique du corps d'un sous-programme se déroule dans un *environnement étendu initial* qui résulte de l'analyse du prototype du sous-programme.

Définition 5.15 (environnement étendu initial).
Soit $\{v_1, \ldots, v_n\} = \{v \mid v \in \mathcal{V}_p \wedge type(v) = \texttt{list}\}$. L'*environnement étendu initial* d'un sous-programme est le sextuplet $< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >$ tel que :

- $\mathcal{E} := \{(v_i, \mathrm{ref}_i) \mid i = 1 \cdots n\} \cup \{(v, v^e) \mid v \in \mathcal{V}_p \wedge type(v) = \texttt{int}\}$
- $\mathcal{M} := \{(\mathrm{ref}_i, l_{1_{v_i}}) \mid i = 1 \cdots n\}$
- $\mathcal{D} := \{(v_i, l_{1_{v_i}}) \mid i = 1 \cdots n\}$
- $F := \emptyset$
- $\mathcal{V}_f := \emptyset$
- $\mathcal{V}_{ref} := \{\mathrm{ref}_i \mid i = 1 \cdots n\}$

Note 5.16. Nous utilisons la notation v^e pour représenter la valeur effective de v. Elle est l'équivalent des termes *C_EA* que nous utilisions dans $SOS\mathrm{Sub}^{\mathcal{R}}_{\mathcal{C}}$.

Opérations fondamentales Nous présentons maintenant les opérations fondamentales sur les ensembles qui composent les environnements étendus. Elles nous serviront dans l'expression de la sémantique des constructions de $\mathrm{Sub}^{ext}_{\mathcal{C}}$. Nous commençons par définir une relation d'appartenance dans les ensembles de couple qui ne porte que sur le premier élément du couple (la *clé*).

Définition 5.17 (relation d'appartenance de clé). Soit E un ensemble quelconque de couples $E_1 \times E_2$, avec E_1 et E_2 des ensembles quelconques. Soit $e \in E_1$, nous définissons la relation d'*appartenance de clé*, $\propto$, comme :

$$e \propto E \text{ si et seulement si } \exists! v \in E_2 \text{ tel que } (e, v) \in E$$

Nous notons $\not\propto$ la négation de cette relation.

[18] L'ensemble $\mathcal{M}$ fournit donc un modèle pour la mémoire. Ceci n'est pas en contradiction avec notre objectif qui est de ne pas user d'un modèle de la mémoire *dans* l'axiomatisation des programmes, mais ne nous interdit pas de le faire *pendant* le processus d'axiomatisation.

Nous définissons ensuite les opérations qui permettent d'obtenir la valeur associée à une variable (ou une référence) dans chacun des ensembles d'un environnement étendu, c'est-à-dire que, connaissant le premier élément d'un couple, nous obtenons le second.

Définition 5.18 (opérations de lecture de valeur). Soit $var \in \mathcal{V}_{loc} \cup \mathcal{V}_p$, nous définissons l'opération de *lecture de valeur*, $\mathrm{Val}_{\mathcal{E}}$, telle que :

$$\mathrm{Val}_{\mathcal{E}}(< \mathcal{E}, \mathcal{M}, \mathcal{D}, F >, var) = val \text{ si et seulement si } var \propto \mathcal{E} \text{ et } (var, val) \in \mathcal{E}$$

Nous définissons similairement $\mathrm{Val}_{\mathcal{M}}$, $\mathrm{Val}_{\mathcal{D}}$ et Val_F. Pour $\mathrm{Val}_{\mathcal{M}}$ et $\mathrm{Val}_{\mathcal{E}}$, nous demandons en outre que $\mathrm{Val}_{\mathcal{M}}(env, \mathrm{NULL}) = \mathrm{NULL}$ et $\forall n \in \mathbb{Z}, \mathrm{Val}_{\mathcal{E}}(env, n) = n$ et $\mathrm{Val}_{\mathcal{M}}(env, n) = n$, pour tout $env \in Env^{ext}$.

Nous aurons aussi besoin de mettre à jour le contenu des ensembles constitutifs d'un environnement étendu : soit en ajoutant un nouveau couple, soit en modifiant la valeur d'une variable (ou d'une référence) d'un couple déjà présent.

Définition 5.19 (opération de mise à jour). Soit E un ensemble quelconque de couples tel que $var \not\propto E$. L'opération associative à gauche de *mise à jour*, $\oplus$, est définie comme suit :

- $E \oplus (var, val) := E \cup \{(var, val)\}$;
- $\big(E \cup \{(var, v)\}\big) \oplus (var, val) := E \cup \{(var, val)\}$.

Nous étendons la définition à la mise à jour de plusieurs couples v_i :

$$E \oplus \{v_1, \ldots, v_n\} := E \oplus v_1 \oplus \ldots \oplus v_n$$

Enfin, nous définissons une opération qui permet de reconstituer la valeur d'une liste à partir d'une référence sur le maillon en tête de la liste. Il suffit de reproduire le chaînage des éléments des maillons présents dans $\mathcal{M}$ (l'ensemble des maillons) en suivant les liens de référence en référence.

Définition 5.20 (opération de reconstitution d'une valeur de liste).
Soient $env = < \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >$ et $r \in \mathcal{V}_{ref}$. Nous définissons l'opération de *reconstitution de la valeur d'une liste*, recons, comme suit :

$$\mathrm{recons}(env, r) := \begin{cases} \mathrm{NULL} & \text{si } \mathrm{Val}_{\mathcal{M}}(env, r) = \mathrm{NULL} \\ e & \text{si } \mathrm{Val}_{\mathcal{M}}(env, r) = e \text{ et} \\ & \quad \text{e est un élément nommé} \\ i \cdot \mathrm{recons}(env, \mathrm{ref}) & \text{si } \mathrm{Val}_{\mathcal{M}}(env, r) = i \cdot \mathrm{ref} \text{ et} \\ & \quad \text{i est un entier} \\ \mathrm{recons}(env, g) \cdot \mathrm{recons}(env, d) & \text{si } \mathrm{Val}_{\mathcal{M}}(env, r) = g \cdot d \end{cases}$$

Exemple 5.21. Pour exemplifier l'utilisation de l'opération recons, nous procédons à la reconstitution de la valeur de la liste L' de l'Exemple 5.12(b). Nous supposons pour cet exemple que l'environnement qui contient la représentation sous forme de références nommées de L' est le suivant :

$$\begin{aligned} env = <&\{(L', \mathrm{ref}_1)\},\\ &\{(\mathrm{ref}_1, \mathrm{ref}_2 \cdot \mathrm{ref}_3), (\mathrm{ref}_2, \mathrm{ref}_4 \cdot \mathrm{ref}_5), (\mathrm{ref}_3, \mathrm{ref}_6 \cdot \mathrm{NULL}), (\mathrm{ref}_4, e_{1:1_{L'}}), (\mathrm{ref}_5, \mathrm{ref}_7 \cdot \mathrm{ref}_8),\\ &\ (\mathrm{ref}_6, e_{2_{L'}}), (\mathrm{ref}_7, \mathrm{ref}_9 \cdot \mathrm{ref}_{10}), (\mathrm{ref}_8, l_{1:3_{L'}}), (\mathrm{ref}_9, e_{1:2:1_{L'}}), (\mathrm{ref}_{10}, \mathrm{ref}_{11} \cdot \mathrm{NULL}),\\ &\ (\mathrm{ref}_{11}, e_{1:2:2_{L'}})\},\\ &\emptyset, \emptyset, \emptyset, \{\mathrm{ref}_1, \ldots, \mathrm{ref}_{11}\} > \end{aligned}$$

L'appel de l'opération se déroule donc comme suit :

$$\begin{aligned} \mathrm{recons}(env, \mathrm{ref}_1) &= \mathrm{recons}(env, \mathrm{ref}_2) \cdot \mathrm{recons}(env, \mathrm{ref}_3)\\ &= \mathrm{recons}(env, \mathrm{ref}_4) \cdot \mathrm{recons}(env, \mathrm{ref}_5) \cdot\\ &\quad \mathrm{recons}(env, \mathrm{ref}_6) \cdot \mathrm{recons}(env, \mathrm{NULL})\\ &= e_{1:1_{L'}} \cdot \mathrm{recons}(env, \mathrm{ref}_7) \cdot \mathrm{recons}(env, \mathrm{ref}_8) \cdot e_{2_{L'}} \cdot \mathrm{NULL}\\ &= e_{1:1_{L'}} \cdot \mathrm{recons}(env, \mathrm{ref}_9) \cdot \mathrm{recons}(env, \mathrm{ref}_{10}) \cdot l_{1:3_{L'}} \cdot e_{2_{L'}} \cdot \mathrm{NULL}\\ &= e_{1:1_{L'}} \cdot e_{1:2:1_{L'}} \cdot \mathrm{recons}(env, \mathrm{ref}_{11}) \cdot \mathrm{recons}(env, \mathrm{NULL}) \cdot l_{1:3_{L'}} \cdot\\ &\quad e_{2_{L'}} \cdot \mathrm{NULL}\\ &= e_{1:1_{L'}} \cdot e_{1:2:1_{L'}} \cdot e_{1:2:2_{L'}} \cdot \mathrm{NULL} \cdot l_{1:3_{L'}} \cdot e_{2_{L'}} \cdot \mathrm{NULL} \end{aligned}$$

Nous retrouvons bien, au parenthésage près, la liste nommée de l'Exemple 5.10(b). ♦

Décomposition des listes Nous introduisons ici deux opérations sur les éléments nommés de liste et les listes nommées. Le rôle de ces opérations est de nommer les éléments d'une liste, selon le schéma de nommage de la Section 5.2.1, au fur et à mesure du parcours de cette liste par un sous-programme. L'idée est donc de ne rendre visible que les éléments de la liste réellement accédés par le sous-programme, autrement dit, d'affiner et de limiter, en fonction des besoins de l'analyse, la connaissance que celle-ci possède de la liste. Cette idée d'approximations de plus en plus fines des structures de données paramétrées par les besoins de l'analyse se retrouve dans d'autres travaux, *e.g.*, elle prend le nom de « matérialisation » dans [Sagiv *et al.*, 1998].

Définition 5.22 (opérations de décomposition). Soit L une liste. L'opération de *décomposition d'un élément nommé*, e, de L est définie comme suit :

$$\mathrm{decomp}(e) := \begin{cases} e_{r:1_L} \cdot l_{r:2_L} & \text{si } e = e_{r_L}\\ e_{r_L} \cdot l_{r+1_L} & \text{si } e = l_{r_L} \end{cases}$$

avec $r+1$ étant défini comme l'ajout de un à l'entier le plus à droite du rang r.

Soient Ln une liste nommée et e un élément nommé. Nous définissons la *décomposition dans une liste nommée* comme suit :

$$\mathrm{decomp}_{ln}(e, Ln) := \begin{cases} \mathrm{NULL} & \text{si } Ln = \mathrm{NULL}\\ \mathrm{decomp}(e) & \text{si } Ln = e\\ \mathrm{decomp}(e) \cdot l & \text{si } Ln = e \cdot l\\ e_1 \cdot \mathrm{decomp}_{ln}(e, l) & \text{si } Ln = e_1 \cdot l \text{ et } e \neq e_1 \end{cases}$$

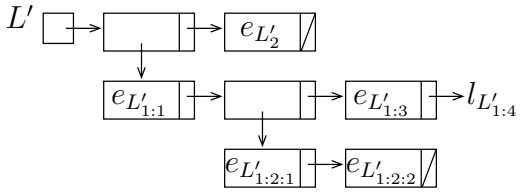


FIG. 5.3 – Exemple de décomposition dans une liste nommée

Exemple 5.23 (décompositions). Nous illustrons la décomposition d'éléments nommés d'une liste L' par deux exemples :

$$\text{decomp}(e_{2_{L'}}) = e_{2:1_{L'}} \cdot l_{2:2_{L'}}$$
$$\text{decomp}(l_{1:3_{L'}}) = e_{1:3_{L'}} \cdot l_{1:4_{L'}}$$

La décomposition dans une liste nommée est simplement le remplacement d'un élément nommé de la liste par sa décomposition. Par exemple, avec la liste de la Figure 5.2(b) :

$$\text{decomp}_{ln}(l_{1:3_{L'}}, e_{1:1_{L'}} \cdot e_{1:2:1_{L'}} \cdot e_{1:2:2_{L'}} \cdot \text{NULL} \cdot l_{1:3_{L'}} \cdot e_{2_{L'}} \cdot \text{NULL}) =$$
$$e_{1:1_{L'}} \cdot e_{1:2:1_{L'}} \cdot e_{1:2:2_{L'}} \cdot \text{NULL} \cdot e_{1:3_{L'}} \cdot l_{1:4_{L'}} \cdot e_{2_{L'}} \cdot \text{NULL}$$

Ce qui donne la nouvelle représentation de la Figure 5.3. ♦

Opérations auxiliaires Les opérations que nous décrivons sous ce nom sont des opérations qui s'expriment à partir des opérations plus primitives définies précédemment. En plus d'aérer l'écriture des règles sémantiques à venir, les opérations auxiliaires classent les transformations de l'environnement en opérations signifiantes.

Les opérations relatives à l'allocation de maillon interviennent lors d'une allocation dynamique explicite (allouer) ou implicite (allouer$_{elt}$ et allouer$_{next}$). Une allocation explicite correspond à l'emploi de l'opérateur **add** de Sub$^{ext}_{\mathcal{C}}$, alors qu'une allocation implicite intervient lorsque le parcours d'une liste fait apparaître de nouveaux éléments de la liste par décomposition. Le propre de ces opérations est d'ajouter de nouveaux maillons dans l'ensemble des maillons $\mathcal{M}$.

Lorsque l'allocation intervient dans le cadre de la décomposition d'une liste existante, elle fait apparaître une nouvelle paire d'éléments nommés. Il faut alors aussi mettre à jour la liste nommée correspondante conservée dans $\mathcal{D}$. La différence entre allouer$_{elt}$ et allouer$_{next}$ tient uniquement en ce que la première renvoie la référence sur la tête de la paire et la seconde sur la queue de la paire.

Définition 5.24 (opérations d'allocation de maillon).
Soient $env =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} > \in Env^{ext}$, $v \in \mathbb{Z} \cup \mathcal{V}_{ref} \cup \{\text{NULL}\}$ et $\text{ref} \in \mathcal{V}_{ref}$. Soient ref' et ref'' des références nommées fraîches (*i.e.*, telles que $\text{ref}', \text{ref}'' \notin \mathcal{V}_{ref} \cup \mathcal{V}_f \cup \mathcal{V}_{loc} \cup \mathcal{V}_p$). Nous définissons les opérations d'allocation suivantes :

$$\text{allouer}(env, v, \text{ref}) := (< \mathcal{E}, \mathcal{M} \oplus (\text{ref}', v \cdot \text{ref}), \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} \cup \{\text{ref}'\} >, \text{ref}')$$

Soit $\mathrm{Val}_{\mathcal{M}}(env, \mathrm{ref}) = el_{r_L} \in Elt_{nom}$, avec $el \in \{\mathrm{e}, \mathrm{l}\}$. Soit $\mathrm{decomp}(el_{r_L}) = g \cdot d$.

$$\begin{aligned}\mathrm{allouer}_{elt}(env, \mathrm{ref}) := (<&\mathcal{E}, \mathcal{M} \oplus (\mathrm{ref}', g) \oplus (\mathrm{ref}'', d) \oplus (\mathrm{ref}, \mathrm{ref}' \cdot \mathrm{ref}''),\\ &\mathcal{D} \oplus (L, \mathrm{decomp}_{ln}(el_{L_r}, \mathrm{Val}_{\mathcal{D}}(env, L))),\\ &F, \mathcal{V}_f, \mathcal{V}_{ref} \cup \{\mathrm{ref}', \mathrm{ref}''\} >, \mathrm{ref}')\end{aligned}$$

$$\begin{aligned}\mathrm{allouer}_{next}(env, \mathrm{ref}) := (<&\mathcal{E}, \mathcal{M} \oplus (\mathrm{ref}', g) \oplus (\mathrm{ref}'', d) \oplus (\mathrm{ref}, \mathrm{ref}' \cdot \mathrm{ref}''),\\ &\mathcal{D} \oplus (L, \mathrm{decomp}_{ln}(el_{r_L}, \mathrm{Val}_{\mathcal{D}}(env, L))),\\ &F, \mathcal{V}_f, \mathcal{V}_{ref} \cup \{\mathrm{ref}', \mathrm{ref}''\} >, \mathrm{ref}'')\end{aligned}$$

Nous sommes à présent prêts pour définir les opérations qui permettent de parcourir une liste. L'opération $\mathrm{element}_m$ accède à la tête d'une liste. Si la liste est suffisamment décomposée, il suffit de renvoyer l'élément correspondant, sinon il faut la décomposer pour faire apparaître l'élément recherché. L'opération next_m se comporte de la même façon mais pour la queue d'une liste.

Définition 5.25 (opération de lecture de l'élément d'un maillon).
Soient $env =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} > \in Env^{ext}$ et $r \in \mathcal{V}_{ref}$.

$$\mathrm{element}_m(env, r) := \begin{cases}(env, v) & \text{si } \mathrm{Val}_{\mathcal{M}}(env, r) = v \cdot \mathrm{ref}\\ \mathrm{allouer}_{elt}(env, r) & \text{si } \mathrm{Val}_{\mathcal{M}}(env, r) \in Elt_{nom}\end{cases}$$

Définition 5.26 (opération de lecture du suivant d'un maillon).
Soient $env =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} > \in Env^{ext}$ et $r \in \mathcal{V}_{ref}$.

$$\mathrm{next}_m(env, r) := \begin{cases}(env, \mathrm{ref}) & \text{si } \mathrm{Val}_{\mathcal{M}}(env, r) = v \cdot \mathrm{ref}\\ \mathrm{allouer}_{next}(env, r) & \text{si } \mathrm{Val}_{\mathcal{M}}(env, r) \in Elt_{nom}\end{cases}$$

Sous-programmes sans boucles ni récursivité

Nous débutons la description de la sémantique des constructions nouvelles de $\mathrm{Sub}_{\mathcal{C}}^{ext}$ en nous plaçant dans le contexte d'un sous-programme non récursif et ne comportant pas de boucle. Cette simplification permet de dévoiler progressivement les *prérequis* à l'axiomatisation par $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$. Le récapitulatif de tous les prérequis est donné en Annexe B. Les mécanismes ici omis réintègrent le langage dès la section suivante.

Prérequis En l'absence de boucles et de récursivité, l'analyse des alias dans un sous-programme $\mathrm{Sub}_{\mathcal{C}}^{ext}$ sans paramètres est décidable : nous pouvons savoir en tout point du sous-programme quels objets du sous-programme ont des alias. En effet, le nombre de tels objets et les opérations sur ceux-ci sont finis. Cependant, si le sous-programme prend des paramètres de type liste, nous ne connaissons pas les relations d'alias préexistantes entre ces structures de données puisque, dans le cas général, nous ignorons le contexte d'appel précis du sous-programme. Les deux raisons de cette méconnaissance sont, d'une part, l'analyse d'un programme incomplet, *i.e.*, le sous-programme fait partie d'une bibliothèque et est vouée à être appelée dans différents programmes, et d'autre part, l'indécidabilité

de l'analyse d'alias, qui interdit de déterminer précisément le contexte d'appel lorsque l'appelante dispose, elle, de boucles et de récursivité.

Nous avons montré sur l'Exemple 5.4 que le comportement d'un sous-programme vis-à-vis des valeurs de ses entrées peut varier selon les relations d'alias qui lient les paramètres du sous-programme. Nous avons aussi discuté dans la Section 5.1.3 de la difficulté en pratique d'énumérer toutes ces relations. Par conséquent, nous imposons en prérequis de l'axiomatisation que les listes passées en paramètre des sous-programmes $\mathrm{Sub}_{\mathcal{C}}^{ext}$ soient exemptes de toute relation d'alias sur une quelconque de leurs sous-listes, à l'exception éventuellement de la liste entière, et que la même liste ne soit pas passée à deux paramètres différents. Dans le cas contraire, les alias ne doivent pas être utilisés de façon ambiguë dans le sous-programme appelé. Nous entendons par *usage ambigu*, l'application d'un effet de bord par l'intermédiaire d'un alias, y compris l'affectation, suivie de la lecture de la valeur d'un autre alias du même objet (ceci comprend l'application d'un opérateur de mutation). En particulier, ce prérequis implique qu'une liste passée en paramètre est acyclique.

Un autre prérequis porte sur la création d'alias dans le corps des sous-programmes. En effet, notre approche considère les sous-programmes comme des fonctions de transfert des valeurs en entrée vers les valeurs en sortie. Ces valeurs n'expriment pas les relations d'alias qui pouvaient lier les variables associées à ces valeurs. Cette information étant perdue, le comportement de la fonction équationnelle ne traduit pas correctement celui du sous-programme dans tous les contextes d'appel de celle-ci. Pour empêcher ce phénomène, nous devons imposer que les sous-programmes ne créent pas des alias qui existent encore une fois que le sous-programme termine. Dans le cas contraire, il faut montrer que les appels au sous-programme n'utilisent pas de façon ambiguë les alias créés par celle-ci.

Nous verrons cependant dans la Section 5.2.2 que notre méthode n'est pas seule en cause dans cette restriction : elle nous aurait été imposée de toute façon par l'indécidabilité des relations d'alias en présence de boucles et de récursivité.

Sémantique Afin de faciliter la lecture, nous introduisons une notation commode pour l'interprétation sémantique des nouvelles constructions du langage $\mathrm{Sub}_{\mathcal{C}}^{ext}$. Nous adopterons conjointement une présentation sous forme de séquents.

Définition 5.27 (interprétation sémantique). Soient $\mathbb{C}$ l'ensemble des constructions du langage $\mathrm{Sub}_{\mathcal{C}}^{ext}$ et $V = \mathbb{Z} \cup \mathcal{V}_{ref}$ l'ensemble des valeurs possibles pour les variables d'un sous-programme. La sémantique des constructions du langage $\mathrm{Sub}_{\mathcal{C}}^{ext}$ est exprimée comme une fonction : $Env^{ext} \times \mathbb{C} \rightarrow Env^{ext} \times V$, notée $env[c] \rhd (env', v)$, avec $env, env' \in Env^{ext}$, $c \in \mathbb{C}$ et $v \in V$.

L'interprétation sémantique des constructions du langage spécifiques à $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ est donnée par la Figure 5.4. Les constructions du langage qui n'apparaissent pas dans ces figures se comportent comme dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ en faisant correspondre la liste des valeurs des variables des environnements de $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ à l'ensemble $\mathcal{E}$ des environnements étendus $< \mathcal{E}, \mathcal{M}, \mathcal{D}, F >$. Il nous suffira de rappeler que nous associons à chaque chemin d'exécution d'un sous-programme une condition et un environnement. La condition d'un chemin d'exécution est la conjonction des conditions des instructions de branchement. En

$$\frac{env[\mathtt{L}] \rhd (< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >, r) \quad r \in \mathcal{V}_{ref}}{env[\mathtt{next(L)}] \rhd \text{next}_m(< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >, r)} \text{(next)}$$

$$\frac{env[\mathtt{L}] \rhd (< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >, r) \quad r \in \mathcal{V}_{ref}}{env[\mathtt{element(L)}] \rhd \text{element}_m(< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >, r)} \text{(element)}$$

$$\frac{env[\mathtt{v}] \rhd (env', v') \quad env'[\mathtt{L}](< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >, r) \quad r \in \mathcal{V}_{ref}}{env[\mathtt{add(v,L)}] \rhd \text{allouer}(< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >, v', r)} \text{(add)}$$

$$\frac{}{env[\mathtt{L->next}] \rhd env[\mathtt{next(L)}]} (\rightarrow\text{next})$$

$$\frac{}{env[\mathtt{L->element}] \rhd env[\mathtt{element(L)}]} (\rightarrow\text{element})$$

(a) opérations spécifiques aux listes

$$\frac{\mathtt{v} \in \mathcal{V}_{loc} \cup \mathcal{V}_p \cup \mathbb{Z} \cup \{\text{NULL}\}}{env[\mathtt{v}] \rhd (env, \text{Val}_{\mathcal{E}}(env, \mathtt{v}))} (\text{Val}_{\mathcal{E}}) \qquad \frac{\begin{array}{c} env[\mathtt{x}] \rhd (env', v) \\ env'[\mathtt{y}] \rhd (env'', v') \quad \mathtt{op} \text{ un opérateur} \end{array}}{env[\mathtt{x\ op\ y}] \rhd (env'', v\ \mathtt{op}\ v')} \text{(op\_exp)}$$

$$\frac{env[\mathtt{val}] \rhd (< \mathcal{E}, \mathcal{M}, \mathcal{D}, F >, v') \quad \mathtt{var} \in \mathcal{V}_{loc} \cup \mathcal{V}_p \wedge \text{type}(\mathtt{var}) = \mathtt{int}}{env[\mathtt{var\ =\ val}] \rhd (< \mathcal{E} \oplus (\mathtt{var}, \text{Val}_{\mathcal{M}}(< \mathcal{E}, \mathcal{M}, \mathcal{D}, F >, v')), \mathcal{M}, \mathcal{D}, F >, v')} (=\_\text{int})$$

$$\frac{env[\mathtt{val}] \rhd (< \mathcal{E}, \mathcal{M}, \mathcal{D}, F >, v') \quad \mathtt{var} \in \mathcal{V}_{loc} \cup \mathcal{V}_p \wedge \text{type}(\mathtt{var}) = \mathtt{list}}{env[\mathtt{var\ =\ val}] \rhd (< \mathcal{E} \oplus (\mathtt{var}, v'), \mathcal{M}, \mathcal{D}, F >, v')} (=\_\text{list})$$

$$\frac{env[\mathtt{val}] \rhd (env', v') \quad env'[\mathtt{var}] \rhd (< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >, r) \quad \begin{array}{c} r \in \mathcal{V}_{ref} \wedge \\ \mathtt{var} \notin \mathcal{V}_{loc} \cup \mathcal{V}_p \end{array}}{env[\mathtt{var\ =\ val}] \rhd (< \mathcal{E}, \mathcal{M} \oplus (r, v'), \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >, v')} (=\_\text{ref})$$

(b) affectation

$$\frac{env =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} > \quad r \notin \mathcal{V}_{ref}}{env[\mathtt{list\ L}] \rhd (< \mathcal{E} \oplus (\mathtt{L}, r), \mathcal{M} \cup (r, \text{NULL}), \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} \cup \{r\} >, r)} \text{(decl\_list)}$$

$$\frac{env[\mathtt{p}_1] \rhd (env_1, p_1) \ldots env_{n-1}[\mathtt{p}_n] \rhd (env_n, p_n)}{env[\mathtt{f(p}_1, \ldots, \mathtt{p}_n)] \rhd \text{appel}(env_n, \mathtt{f}, < p_1, \ldots, p_n >)} \text{(appel)}$$

(c) constructions de base pour les listes

FIG. 5.4 – Sémantique des constructions spécifiques à $\text{Sub}_{\mathcal{C}}^{ext}$

l'absence de boucles, les chemins d'exécution dans un sous-programme sont en nombre fini.

Les règles de la Figure 5.4(a) concernent les opérateurs de liste. Les règles (next) et (element), lorsqu'elles ne se réduisent pas simplement à renvoyer le champ correspondant d'un maillon, sont des appels aux opérations d'allocation et de décomposition de maillons. Ces dernières ont la responsabilité de reconstituer dans l'environnement le chaînage des éléments d'une liste auxquels le programme essaye d'accéder. Les éléments nommés qui sont créés à cet effet respectent le schéma de nommage défini dans la Section 5.2.1.

L'opération d'allocation dynamique de la règle (add) se traduit directement par l'allocation d'un nouveau maillon dans l'environnement.

La représentation choisie pour les listes dans l'environnement permet d'exprimer très naturellement les effets de bord dans les règles (→next) et (→element) comme la recherche de la référence à modifier. La modification elle-même se fait par la règle (=_ref) décrite ci-après, car la syntaxe de $\text{Sub}_{\mathcal{C}}^{ext}$ impose que ces opérateurs soient en membre gauche d'une affectation.

Les règles de la Figure 5.4(b) définissent le comportement de l'affectation. Le cas où le membre gauche de l'affectation est une variable de type liste (=_list) ne présente pas de nouveauté, si ce n'est que v', résultat de l'évaluation du membre droit, sera une référence.

La règle (=_int) est plus intéressante du fait de son appel à l'opération $\text{Val}_{\mathcal{M}}$. Celui-ci s'explique par le traitement des listes de listes. Lorsque, par exemple, les opérateurs `element` et `next` parcourent une liste et la décomposent pour faire apparaître de nouveaux maillons avec allouer_{elt} et allouer_{next}, les maillons créés sont du type $\text{ref}_g \cdot \text{ref}_d$, plutôt que $entier \cdot \text{ref}_d$, car nous ne pouvons pas savoir si la tête du maillon est un entier ou une liste. Or, s'il s'agit d'une liste, l'utilisation d'une référence est obligatoire pour correctement représenter d'éventuelles relations d'alias et simplifier la sémantique des opérateurs d'effet de bord sur les listes. De plus, s'il s'agit d'un entier, nous pouvons simplement gérer ce niveau d'indirection en faisant un appel à $\text{Val}_{\mathcal{M}}$ au moment d'une affectation.

Enfin, la règle (=_ref) est celle qui réalise les effets de bord sur les listes. En effet, l'application des opérateurs `->element` et `->next` fournit la référence du maillon à modifier en membre gauche d'une affectation. Cette règle se charge de la modification du maillon correspondant.

Exemple 5.28 (règles sémantiques). Nous illustrons le fonctionnement des règles de cette section sur l'exemple du sous-programme suivant :

```
1 void f(list L, int i) {
2   L = add(1,L);
3   L->next->element = 2;
4   i = element(next(L));
5   ⋮
6 }
```

L'environnement initial de ce sous-programme est :

$$env =< \{(L, \text{ref}_1), (i, i^e)\}, \{(\text{ref}_1, l_{1_L})\}, \{(L, l_{1_L})\}, \emptyset, \emptyset, \{\text{ref}_1\} >$$

La règle (=_list) commande d'abord l'évaluation de l'opération `add(1, L)`. Celle-ci se traduit par les règles (add) et ($\text{Val}_{\mathcal{E}}$) en l'appel de allouer$(env, 1, \text{ref}_1)$ qui ajoute un nouveau maillon et produit la paire :

$$(< \{(L, \text{ref}_1), (i, i^e)\}, \{(\text{ref}_1, l_{1_L}), (\text{ref}_2, 1 \cdot \text{ref}_1)\}, \{(L, l_{1_L})\}, \emptyset, \emptyset, \{\text{ref}_1, \text{ref}_2\} >, \text{ref}_2)$$

La fin de la règle (=_list) peut alors être effectuée, nous obtenons :

$$env =< \{(L, \text{ref}_2), (i, i^e)\}, \{(\text{ref}_1, l_{1_L}), (\text{ref}_2, 1 \cdot \text{ref}_1)\}, \{(L, l_{1_L})\}, \emptyset, \emptyset, \{\text{ref}_1, \text{ref}_2\} >$$

L'instruction de la ligne 3 correspond à la règle (=_ref). L'évaluation débute par la règle ($\text{Val}_{\mathcal{E}}$) puisque le membre droit de l'affectation est prioritaire. Pour le membre gauche, c'est d'abord la règle (→element), puis (→next). Ceci revient à évaluer :

`element(next(L))`

La règle (next) s'applique alors en premier et produit l'appel $\text{next}_m(env, \text{ref}_2)$. La liste étant suffisamment décomposée, l'environnement n'est pas modifié et la règle (element) produit l'appel $\text{element}_m(env, \text{ref}_1)$. Cette fois, la valeur associée à ref_1 ne fait pas apparaître la tête du maillon et cela entraîne l'appel $\text{allouer}_{elt}(env, \text{ref}_1)$. Par cette opération, deux nouveaux maillons sont créés et la liste L est décomposée en deux éléments nommés :

$$\begin{aligned}(< &\{(L, \text{ref}_2), (i, i^e)\},\\ &\{(\text{ref}_1, \text{ref}_3 \cdot \text{ref}_4), (\text{ref}_2, 1 \cdot \text{ref}_1), (\text{ref}_3, e_{1_L}), (\text{ref}_4, l_{2_L})\},\\ &\{(L, e_{1_L} \cdot l_{2_L})\}, \emptyset, \emptyset, \{\text{ref}_1, \text{ref}_2, \text{ref}_3, \text{ref}_3\} >, \text{ref}_3)\end{aligned}$$

Cette paire permet de terminer la règle (=_ref) et nous obtenons le nouvel environnement où la valeur de ref_3 change :

$$\begin{aligned}env =< &\{(L, \text{ref}_2), (i, i^e)\},\\ &\{(\text{ref}_1, \text{ref}_3 \cdot \text{ref}_4), (\text{ref}_2, 1 \cdot \text{ref}_1), (\text{ref}_3, 2), (\text{ref}_4, l_{2_L})\},\\ &\{(L, e_{1_L} \cdot l_{2_L})\}, \emptyset, \emptyset, \{\text{ref}_1, \text{ref}_2, \text{ref}_3, \text{ref}_3\} >\end{aligned}$$

Enfin, l'instruction de la ligne 4 correspond à la règle (=_int). Nous nous retrouvons de nouveau à devoir évaluer `element(next(L))`, mais cette fois, la liste L est suffisamment décomposée dans l'environnement courant pour ne pas avoir à créer de nouveaux éléments nommés. L'évaluation du membre droit de l'affectation produit donc la paire : (env, ref_3). La règle (=_ref) termine correctement en « déréférençant » ref_3 par l'opération $\text{Val}_{\mathcal{M}}(env, \text{ref}_3)$. Notons que si le membre droit s'était évalué en un entier, $\text{Val}_{\mathcal{M}}$ aurait renvoyé cet entier, par définition de l'opération $\text{Val}_{\mathcal{M}}$. L'environnement à la ligne 5 voit la valeur de i changer et est finalement :

$$\begin{aligned}< &\{(L, \text{ref}_2), (i, 2)\},\\ &\{(\text{ref}_1, \text{ref}_3 \cdot \text{ref}_4), (\text{ref}_2, 1 \cdot \text{ref}_1), (\text{ref}_3, 2), (\text{ref}_4, l_{2_L})\},\\ &\{(L, e_{1_L} \cdot l_{2_L})\}, \emptyset, \emptyset, \{\text{ref}_1, \text{ref}_2, \text{ref}_3, \text{ref}_3\} >\end{aligned}$$

♦

Nous terminons la présentation des règles sémantiques avec celles de la Figure 5.4(c). Elles concernent des constructions déjà présentes dans $\text{Sub}_{\mathcal{C}}$ mais dont la sémantique a évolué dans $SOS\text{Sub}_{\mathcal{C}}^{ext}$. Il s'agit d'abord de la règle (decl_list) qui va modifier l'environnement initial d'un sous-programme en lui ajoutant un nouveau maillon de valeur NULL pour chaque déclaration de variable de type liste.

Beaucoup plus intéressante est la règle (appel) pour les appels de sous-programme. Dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$, un sous-programme qui a une valeur de retour entraîne la définition d'une fonction de la logique équationnelle, avec le cas particulier des boucles qui produisent des familles de fonctions récursives. Dans $SOS\text{Sub}_{\mathcal{C}}^{ext}$, les sous-programmes peuvent avoir des effets de bord : il est permis de modifier, à l'aide des opérateurs de mutation, les maillons d'une liste passée en paramètre (comme dans les Exemples 5.29 et 5.30). Par conséquent, dans $SOS\text{Sub}_{\mathcal{C}}^{ext}$, la génération d'une seule fonction équationnelle n'est plus suffisante pour rendre compte de l'ensemble des valeurs que modifie un sous-programme s'il a des effets de bord sur diverses listes passées en paramètre.

Exemple 5.29 (Sous-programme avec effet de bord). Soient les sous-programmes $\text{Sub}_{\mathcal{C}}^{ext}$ suivants :

```
void f() {
  list L;

  L = add(1, NULL);
  g(L);
}

void g(list Lg) {
  Lg->element = 2;
  Lg = NULL;
}
```

Dans le sous-programme `f`, juste avant l'appel `g(L)`, nous avons $L = 1 \cdot \text{NULL}$. Le sous-programme `g` utilise un opérateur de mutation pour modifier la valeur de la tête de la liste passée en paramètre. Ainsi, dans `f`, après l'appel `g(L)`, la liste `L` vaut $2 \cdot \text{NULL}$ car par l'intermédiaire de `Lg`, le sous-programme `g` a accès aux maillons de `L`. Notons cependant que l'affectation `Lg = NULL` n'a pas d'incidence sur `L` car le passage du paramètre est par valeur. ♦

Le principe de la solution est le même que pour les boucles dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$: produire une famille de fonctions pour tous les objets susceptibles d'être modifiés. Ainsi sur l'Exemple 5.29, nous définirions une fonction $g_{Lg}(t \cdot q) = 2 \cdot q$. Dans `f`, nous saurions alors qu'après l'appel `g(L)`, `L` vaut $g_{Lg}(L)$.

Cependant, la génération d'une seule fonction par paramètre de type liste n'est pas suffisante lorsque la modification porte sur la structure même de la liste, avec l'opération de mutation →next (*cf.* Exemple 5.30). La généralisation de ce procédé conduirait à dénoter par une fonction équationnelle distincte la nouvelle valeur de chaque maillon des listes passées en paramètre d'un sous-programme. Ceci s'apparente à modéliser complètement la mémoire, ce que nous souhaitons éviter.

Heureusement, le prérequis qu'aucun alias ne porte sur une liste passée en paramètre, hormis éventuellement sur la tête de la liste, nous dispense de cette généralisation. En effet, si la structure de la liste change de telle façon que certains de ses maillons ne sont plus accessibles à partir de la tête de la liste, alors ces mêmes maillons ne sont plus accessibles au programme dans son entier puisque le prérequis impose qu'aucun alias ne désigne ces maillons. De ce fait, nous pouvons omettre de rendre compte de leur nouvelle valeur. Cette omission est valide attendu que la classe des propriétés de programmes que vise notre méthode ne concerne pas l'usage de la mémoire. La situation serait différente si nous nous intéressions, par exemple, à la mémoire consommée par le programme.

Exemple 5.30 (sous-programme modifiant la structure d'une liste). Prenons l'exemple des sous-programmes suivants :

```
void f() {
  list La, Lb;

  La = add(1, add(2, NULL));
  Lb = next(La);
  g(La);
}

void g(list Lg) {
  Lg->next->element = 3;
  Lg->next = add(4, NULL);
}
```

Dans le sous-programme `f`, avant l'appel `g(La)`, $La = 1 \cdot 2 \cdot \text{NULL}$ et `Lb` est un alias pour la queue de `La` ($Lb = 2 \cdot \text{NULL}$).

Suite à l'appel `g(La)`, dans `g`, la première instruction donne à `La` la valeur $1 \cdot 3 \cdot \text{NULL}$; `Lb` reste un alias pour la queue de `La`, sa valeur est donc $3 \cdot \text{NULL}$.

À la ligne suivante, par l'intermédiaire de `Lg`, la structure de la liste `La` change. L'instruction a deux implications : la valeur de `La` est maintenant $1 \cdot 4 \cdot \text{NULL}$, mais surtout `Lb` n'est plus un alias pour la queue de `La` car `Lb` continue de pointer sur le même maillon de valeur $3 \cdot \text{NULL}$ qui ne fait plus partie du chaînage de `La`.

En langage `C`, du fait des alias, l'appel `g(La)` produit deux listes distinctes à partir d'une seule passée en paramètre. Pour traduire cet état de fait, nous serions amenés à associer deux fonctions équationnelles à `g`, chacune rapportant la valeur d'une des deux listes produites. Toutefois, l'impossibilité de toujours connaître les relations d'alias rend la généralisation de cette approche inapplicable. ♦

Nous sommes maintenant en mesure d'expliciter l'opération appel d'appel de sous-programme de la règle (appel). Le traitement des sous-programmes par $SOS\text{Sub}_{\mathcal{C}}^{ext}$ consiste à générer la définition équationnelle d'une fonction par paramètre de type liste (en plus de la fonction pour la valeur de retour, si le sous-programme en a une). Ces fonctions rendent compte des effets de bord effectués sur les listes passées en paramètre du sous-programme et serviront à mettre à jour la valeur de la référence idoine dans l'appelante.

Il est nécessaire de différencier les appels de sous-programme selon que la valeur de retour du sous-programme est de type entier ou de type liste. En effet, dans ce dernier cas,

le traitement doit être le même que pour les paramètres de type liste : affectation de la valeur renvoyée par la fonction équationnelle correspondante à une variable fraîche, cette dernière permet l'introduction d'un élément nommé qui servira à représenter d'éventuelles décompositions ultérieures sur cette liste.

Définition 5.31 (opérations d'appel de sous-programme). Soient f le nom d'un sous-programme et $< p_1, \ldots, p_n >$ la séquence des valeurs des paramètres effectifs d'un appel du sous-programme f. Soient $env =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} > \in Env^{ext}$, $\{P_1, \ldots, P_m\}$ l'ensemble des paramètres formels de f de type liste et p_{j_i}, pour $i = 1 \ldots m$, tel que p_{j_i} soit la valeur dans l'environnement env du paramètre effectif correspondant au paramètre formel P_i. Enfin, soit $< p'_1, \ldots, p'_n >$ la séquence $< p_1, \ldots, p_n >$ où les p_{j_i} ont été remplacés par recons(env, p_{j_i}).

$$\text{appel}(env, f, < p_1, \ldots, p_n >) := \begin{cases} \text{appel}_{int}(env, f, < p_1, \ldots, p_n >) & \text{si le type de retour de } f \text{ est } \texttt{int} \text{ ou } \texttt{void} \\ \text{appel}_{list}(env, f, < p_1, \ldots, p_n >) & \text{si le type de retour de } f \text{ est } \texttt{list} \end{cases}$$

$$\begin{aligned} \text{appel}_{int}(env, f, < p_1, \ldots, p_n >) := \\ \big(< \mathcal{E}, \mathcal{M} \oplus \{(p_{j_i}, l_{1_{v_i}}) \mid i = 1 \ldots m\}, \\ \mathcal{D} \cup \{(v_i, l_{1_{v_i}}) \mid i = 1 \ldots m\}, \\ F \cup \{(v_i, f_{P_i}(< p'_1, \ldots, p'_n >)) \mid i = 1 \ldots m\}, \\ \mathcal{V}_f \cup \{v_i \mid i = 1 \ldots m\}, \mathcal{V}_{ref} >, \\ f_{P_0}(< p'_1, \ldots, p'_n >)\big) \end{aligned}$$

$$\begin{aligned} \text{appel}_{list}(env, f, < p_1, \ldots, p_n >) := \\ \big(< \mathcal{E}, \mathcal{M} \oplus \{(p_{j_i}, l_{1_{v_i}}) \mid i = 0 \ldots m\}, \\ \mathcal{D} \cup \{(v_i, l_{1_{v_i}}) \mid i = 0 \ldots m\}, \\ F \cup \{(v_i, f_{P_i}(< p'_1, \ldots, p'_n >)) \mid i = 0 \ldots m\} \\ \mathcal{V}_f \cup \{v_i \mid i = 0 \ldots m\}, \mathcal{V}_{ref} \cup \{p_{j_0}\} >, \\ p_{j_0}\big) \end{aligned}$$

Avec $v_0, v_1, \ldots, v_n$ et p_{j_0} des variables fraîches. De plus, f_{P_i} dénote la fonction équationnelle générée pour le paramètre de type liste P_i et f_{P_0} dénote la fonction équationnelle générée pour la valeur de retour du sous-programme ou bien la fonction nulle si le type de retour est `void`.

Exemple 5.32 (appel de sous-programme). Nous illustrons le fonctionnement de la règle (appel) par l'exemple des sous-programmes $\text{Sub}^{ext}_{\mathcal{C}}$ suivants :

```
1 list g(list Lg) { ... }
2
3 void f(list L) {
4   list La;
5
6   La = g(L);
7   L = next(L);
8   ⋮
9 }
```

Le système $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ va produire pour le sous-programme `g` les définitions de deux fonctions équationnelles : g pour la valeur de retour et g_{Lg} pour les effets de bord sur la liste passée en paramètre.

L'environnement initial du sous-programme `f`, après application de la règle (decl_list) pour la déclaration de `La`, est :

$$env =< \{(L, \mathrm{ref}_1), (La, \mathrm{ref}_2)\}, \{(\mathrm{ref}_1, l_{1_L}), (\mathrm{ref}_2, \mathrm{NULL})\}, \{(L, l_{1_L})\}, \emptyset, \emptyset, \{\mathrm{ref}_1, \mathrm{ref}_2\} >$$

Lors de l'affectation de la ligne 6, la règle (=_list) entraîne l'évaluation par la règle (appel) de `g(L)`. Cette dernière se réduit en $\mathrm{appel}_{list}(env, \mathsf{g}, < \mathrm{ref}_1 >)$, où ref_1 est donc la valeur effective du paramètre formel `Lg`. Cette opération produit la paire suivante :

$$\begin{aligned}(<&\{(L, \mathrm{ref}_1), (La, \mathrm{ref}_2)\},\\ &\{(\mathrm{ref}_1, l_{1_{v_1}}), (\mathrm{ref}_2, \mathrm{NULL}), (\mathrm{ref}_3, l_{1_{v_0}})\},\\ &\{(L, l_{1_L}), (v_0, l_{1_{v_0}}), (v_1, l_{1_{v_1}})\},\\ &\{(v_0, g(< l_{1_L} >)), (v_1, g_{Lg}(< l_{1_L} >))\},\\ &\{v_0, v_1\}, \{\mathrm{ref}_1, \mathrm{ref}_2, \mathrm{ref}_3\} >, \mathrm{ref}_3)\end{aligned}$$

Nous voyons que dans cette paire, suite à l'appel de `g`, la valeur de la variable `L` n'a pas changé, c'est toujours la même référence ref_1, mais le maillon associé à cette référence, lui, a été modifié pour refléter un possible effet de bord dû au sous-programme `g`.

La règle (=_list) termine de mettre à jour la valeur de `La` et donne l'environnement :

$$\begin{aligned}env =<&\{(L, \mathrm{ref}_1), (La, \mathrm{ref}_3)\},\\ &\{(\mathrm{ref}_1, l_{1_{v_1}}), (\mathrm{ref}_2, \mathrm{NULL}), (\mathrm{ref}_3, l_{1_{v_0}})\},\\ &\{(L, l_{1_L}), (v_0, l_{1_{v_0}}), (v_1, l_{1_{v_1}})\},\\ &\{(v_0, g(< l_{1_L} >)), (v_1, g_{Lg}(< l_{1_L} >))\},\\ &\{v_0, v_1\}, \{\mathrm{ref}_1, \mathrm{ref}_2, \mathrm{ref}_3\} >\end{aligned}$$

Nous ne détaillons pas complètement la dernière instruction de `f` à la ligne 7. Notre intention est uniquement de faire remarquer que l'ajout des listes nommées dans l'environnement, en l'occurrence $l_{1_{v_0}}$ et $l_{1_{v_1}}$, conjointement à celui des constantes v_0 et v_1, permet de décomposer, si besoin est, les listes modifiées par un appel de sous-programme. Dans notre exemple, `next(La)` produirait dans l'ensemble correspondant à $\mathcal{D}$ une paire $(v_0, e_{1_{v_0}} \cdot l_{2_{v_0}})$, alors que par ailleurs nous conservons le fait que $v_0 = g(< l_{1_L} >)$ dans l'ensemble correspondant à F. ♦

Sous-programmes avec boucles et récursivité

Les principales opérations de la méthode ont été décrites dans les sections précédentes. Nous considérons dans cette section le cas particulier des boucles qui exige d'examiner les conséquences des mécanismes d'itération et de récursivité sur l'analyse des alias.

Prérequis Des résultats très forts d'indécidabilité sont connus sur les analyses d'alias en présence de boucles et de structures de données dynamiques : même dans un langage qui ne disposerait pas de sous-programmes, l'analyse est indécidable (*cf.* Section 5.1.1). Concrètement, cela se traduit pour nous par la perte des relations sur les alias au cours d'une boucle ou dans un appel à un sous-programme récursif.

En conséquence, pour que l'analyse des alias que nous effectuons pendant l'axiomatisation reste correcte après une boucle ou un appel à un sous-programme récursif, nous imposons en prérequis de l'axiomatisation que les boucles ne créent aucune relation d'alias entre les listes du sous-programme telle que la relation continue à exister à la fin du corps de la boucle. Dans le cas contraire, les alias ne doivent pas être utilisés de façon ambiguë dans le corps de la boucle et dans le reste du sous-programme contenant la boucle.

De même, nous imposons que les sous-programmes ne créent aucune relation d'alias entre les listes du sous-programme telle que la relation continue à exister à la fin du corps du sous-programme[19]. Dans le cas contraire, les alias ne doivent pas être utilisés de façon ambiguë après un appel du sous-programme.

Le comportement d'une boucle, comme celui d'un sous-programme (*cf.* l'Exemple 5.4), est soumis à l'influence des relations d'alias qui existent au moment où débute la boucle. Par conséquent, pour des raisons similaires à celles que nous avons développées dans le cadre des sous-programmes sans boucles ni récursivité, nous imposons en prérequis de l'axiomatisation qu'aucune relation d'alias ne porte sur les listes modifiées par une boucle avant la boucle. Dans le cas contraire, les alias ne doivent pas être utilisés de façon ambiguë dans la boucle et la suite du sous-programme.

Sémantique La seule construction non encore explicitée qu'ajoute la réintroduction des mécanismes d'itération et de récursivité est celle que permet l'instruction `while`. L'interprétation sémantique de cette instruction dans $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ est la même que dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$, à savoir qu'une boucle donne lieu à la définition d'une famille de fonctions récursives loop_v, une pour chaque variable v affectée dans le corps de la boucle. Ces fonctions prennent en paramètre toutes les variables locales, ainsi que les paramètres, du sous-programme contenant la boucle. Dans ce sous-programme, la valeur de la variable v devient un appel à la fonction loop_v.

Dans $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$, toutefois, cette famille de fonctions est complétée par une seconde famille de fonctions récursives loop_v^r, une pour chaque variable v du sous-programme de type liste. Chaque fonction loop_v^r renvoie la valeur de la liste référencée par v et rend ainsi compte des mutations subies par cette liste. De plus, la valeur de v n'est pas

[19] Ce prérequis a déjà été posé, lorsque nous avons exclu les boucles, avec d'autres justifications inhérentes à notre approche. Ces justifications s'appliquent aussi aux boucles que nous traitons comme des fonctions récursives terminales.

$$\frac{\forall v \in \mathcal{V}_{loc} \cup \mathcal{V}_p \text{ modifiée dans } \texttt{corps}, \text{ la fonction } \text{loop}_v \text{ a été générée} \quad \forall v \in \{u \mid u \in \mathcal{V}_{loc} \cup \mathcal{V}_p \wedge \text{type}(u) = \texttt{list}\}, \text{ la fonction } \text{loop}^r_v \text{ a été générée}}{env[\texttt{while(cond) corps}] \rhd \text{appel}_{loop}(env)} \text{ (boucle)}$$

FIG. 5.5 – Sémantique de l'instruction d'itération

directement l'appel à la fonction loop^r_v correspondante. Nous introduisons une constante pour mémoriser cette valeur ainsi qu'une nouvelle liste nommée pour la représenter dans la suite du programme : ceci, comme à chaque fois avec les listes, afin de permettre d'éventuelles décompositions de la liste dans la suite du sous-programme.

La génération des deux familles de fonctions loop est décrite dans la Section 5.4.3 avec les autres éléments de la traduction en équations. L'interprétation sémantique de l'instruction `while` est donnée dans la Figure 5.5. L'opération sur les environnements qui lui est associée se comporte essentiellement comme les opérations d'appel de sous-programme de la Définition 5.31. En effet, les boucles sont traduites comme des fonctions récursives et remplacées par des appels à ces fonctions. La particularité des boucles est, comme dans $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$, qu'elles peuvent modifier la valeur des variables du sous-programme (variables locales et paramètres). Ceci se traduit par la mise à jour des couples dans l'ensemble $\mathcal{E}$.

Définition 5.33 (opération d'appel de fonction de boucle). Soit une boucle B dans un sous-programme `f`. Soit $env =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} >\in Env^{ext}$. Soient V_{proc}, l'ensemble des variables locales et paramètres de `f`, et $V_{mod} \subset V_{proc}$, l'ensemble des variables affectées dans le corps de B. Pour toute variable $var \in V_{mod}$, loop_{var} est la fonction générée pour la boucle et la variable var considérées.

Soit $V_{list} = \{var_1, \ldots, var_m\} = \{var \mid var \in V_{proc} \wedge \text{type}(var) = \texttt{list}\}$. Pour $i = 1 \ldots m$, $\text{loop}^r_{var_i}$ est la fonction générée pour la boucle considérée et la liste référencée par la variable var_i.

Enfin, à chaque variable $var \in V_{proc}$ nous associons une valeur p_j telle que $p_j = \text{recons}(env, \text{Val}_{\mathcal{E}}(env, var))$, si $var \in V_{list}$, et $p_j = \text{Val}_{\mathcal{E}}(env, var)$, sinon.

La définition de l'opération d'*appel pour fonctions de boucle*, appel_{loop}, est la suivante :

$$\begin{aligned}
\text{appel}_{loop}(env) := \big(&<\mathcal{E} \oplus \{(var, \text{loop}_{var}(< p_1, \ldots, p_n >)) \mid var \in V_{mod} \setminus V_{list}\} \\
&\quad \oplus \{(var_i, \text{ref}_{i+m}) \mid i = 1 \ldots m\}, \\
&\mathcal{M} \oplus \{(\text{Val}_{\mathcal{E}}(env, var_i), l_{1_{v_i}}) \mid i = 1 \ldots m\} \\
&\quad \cup \{(\text{ref}_i, l_{1_{v_i}}) \mid i = m+1 \ldots 2m\}, \\
&\mathcal{D} \cup \{(v_i, l_{1_{v_i}}) \mid i = 1 \ldots 2m\}, \\
&F \cup \{(v_i, \text{loop}^r_{var_i}(< p_1, \ldots, p_n >)) \mid i = 1 \ldots m\} \\
&\quad \cup \{(v_{i+m}, \text{loop}_{var_i}(< p_1, \ldots, p_n >)) \mid i = 1 \ldots m\}, \\
&\mathcal{V}_f \cup \{v_i \mid i = 1 \ldots 2m\}, \mathcal{V}_{ref} \cup \{\text{ref}_{m+1}, \ldots, \text{ref}_{2m}\} >, 0\big)
\end{aligned}$$

avec $v_0, \ldots, v_{2m}, \text{ref}_{m+1}, \ldots, \text{ref}_{2m}$ des variables fraîches et n le cardinal de V_{proc}.

Exemple 5.34. Nous illustrons le fonctionnement de la règle (boucle) sur l'exemple d'un sous-programme `f` qui change la valeur de tous les éléments d'une liste passée en

paramètre. La liste en paramètre de `f` aura effectivement changé dans le sous-programme appelant.

```
1 void f(list L) {
2
3   while(L != NULL) {
4     L->element = 0;
5     L = next(L);
6   }
7 }
```

L'environnement initial de ce sous-programme est :

$$env =< \{(L, \mathrm{ref}_1)\}, \{(\mathrm{ref}_1, l_{1_L})\}, \{(L, l_{1_L})\}, \emptyset, \emptyset, \{\mathrm{ref}_1\} >$$

L'analyse de la boucle aura généré par ailleurs les définitions des fonctions suivantes : loop_L et loop_L^r. La fonction loop_L donne la valeur de `L`, c'est-à-dire, une référence sur un maillon, après la boucle. Quant à la fonction loop_L^r, elle indique le nouveau chaînage des maillons que la boucle a créé à partir du maillon que référençait `L` avant la boucle. La règle (boucle) se réduit à $\mathrm{appel}_{loop}(env)$ et produit l'environnement suivant :

$$\begin{aligned}<&\{(L, \mathrm{ref}_2)\}, \{(\mathrm{ref}_1, l_{1_{v_1}}), (\mathrm{ref}_2, l_{1_{v_2}})\}, \{(L, l_{1_L}), (v_1, l_{1_{v_1}}), (v_2, l_{1_{v_2}})\},\\ &\{(v_1, \mathrm{loop}_L^r(l_{1_L})), (v_2, \mathrm{loop}_L(l_{1_L}))\}, \{v_1, v_2\}, \{\mathrm{ref}_1, \mathrm{ref}_2\} >\end{aligned}$$

Le second élément du couple renvoyé par appel_{loop}, 0 en l'occurrence, est sans importance car il n'est jamais utilisé. ♦

5.3 Appel par référence

L'*appel par référence* est un mode de passage des paramètres dans lequel l'*adresse* des paramètres effectifs, au lieu de leur valeur, est passée à un sous-programme. Ce mode de passage autorise ainsi un sous-programme à avoir un effet de bord sur les paramètres effectifs du sous-programme appelant, auxquels il accède par l'intermédiaire de leur adresse.

Pour certains langages, *e.g.*, FORTRAN, l'appel par référence est le seul mode de passage disponible ; pour d'autres, *e.g.*, Pascal, C++, c'est un des modes disponibles généralement dénoté par des constructions syntaxiques particulières ; enfin, pour la dernière catégorie de langages, *e.g.*, C, ce mode de passage n'existe pas en soi, mais peut parfois être simulé par le truchement d'autres mécanismes (notamment les pointeurs).

5.3.1 Représentation de l'appel par référence

Dans $\mathrm{Sub}_{\mathcal{C}}$, le mode de passage par défaut est par valeur. Nous souhaitons que $\mathrm{Sub}_{\mathcal{C}}^{ext}$ dispose de l'appel par référence. Pourtant, pour être exact, nous disposons déjà avec les listes mutables d'une forme d'appel par référence. Effectivement, la valeur des variables de type liste est une référence : par son intermédiaire, un sous-programme a la possibilité de modifier la structure de données visible par l'appelant. Néanmoins, la référence elle-même ne peut être modifiée, pour pointer sur une autre liste, que dans le sous-programme

```
void swap(int &a, int &b) {
 int tmp;

 tmp = a;
 a = b;
 b = tmp;
}
```

FIG. 5.6 – Sous-programme `swap`

appelé. La modification n'a pas d'incidence dans l'appelant, car dans les faits, $\text{Sub}_{\mathcal{C}}$ n'offre qu'une combinaison de passage par valeur et de pointeur similaire à celle qui existe en langage `C`.

Nous ajoutons avec $\text{Sub}_{\mathcal{C}}^{ext}$ un mode de passage par référence des paramètres de type entier ou liste, dénoté par une perluète (symbole **&**) placée avant le nom du paramètre concerné dans le prototype d'un sous-programme. Bien que ce symbole soit le même que l'opérateur d'adresse du langage `C`, leurs rôles ne doivent pas être confondus : il n'existe pas d'opérateur d'adresse en $\text{Sub}_{\mathcal{C}}^{ext}$. De plus, nous interdisons syntaxiquement que soit passée par référence la valeur de retour d'un sous-programme ou des opérateurs **`element`**, **`next`**, ou **`add`** uniquement, de sorte que les paramètres effectifs passés par référence seront toujours des variables.

Exemple 5.35 (permutation de valeur). Nous présentons en Figure 5.6 l'exemple classique d'un sous-programme `swap` qui permute la valeur de ses deux paramètres. Ce sous-programme est écrit en $\text{Sub}_{\mathcal{C}}^{ext}$. Soit un sous-programme `p` qui appelle `swap` :

```
int p() {
  int x = 1;
  int y = 2;

  swap(x, y);
  ⋮
}
```

Après l'appel `swap(x, y)`, les valeurs de `x` et `y` ont été modifiées dans `p` : `x` vaut 2 et `y` vaut 1. ♦

5.3.2 Sémantique de $\text{Sub}_{\mathcal{C}}^{ext}$: appel par référence

La sémantique du mode de passage des paramètres par référence dans $\text{Sub}_{\mathcal{C}}^{ext}$ s'exprime à deux niveaux :

- au niveau des sous-programmes disposant de paramètres passés par référence ;
- au niveau de l'appel d'un tel sous-programme.

Avant de détailler chacun des niveaux, nous discutons de l'influence de ce mécanisme sur les relations d'alias et par conséquent sur les prérequis de l'axiomatisation.

Prérequis

L'introduction du mode de passage par référence offre une possibilité supplémentaire de créer des alias en $\text{Sub}_{\mathcal{C}}^{ext}$. En effet, si la même référence est passée à plusieurs paramètres, les paramètres formels correspondants deviennent des alias du même objet dans le sous-programme appelé. Comme l'Exemple 5.36 le met en lumière, la situation est semblable à celle rencontrée lorsque nous passons des paramètres de type liste qui sont alias d'un même objet : le comportement du sous-programme en est affecté. Néanmoins, cette fois, les alias portent sur les références elles-mêmes, ce qui ne limite plus aux opérations de mutation sur maillon les instructions sensibles aux alias.

Exemple 5.36 (alias et paramètres par référence). Nous exemplifions la création d'alias par le passage de paramètres par référence avec les sous-programmes suivants :

```
1  int p(int &a, int &b) {
2    a = 0;
3    return b;
4  }
5
6  void p2(void) {
7    int i, j, x, y;
8
9    x = 1; y = 2;
10   i = p(x, y);
11
12   x = 1; y = 2;
13   j = p(x, x);
14 }
```

Le sous-programme `p` met son premier paramètre à 0 et renvoie la valeur de son second. À la ligne 10, l'appel `p(x, y)` produit bien le comportement attendu dans `p2` : `x` vaut 0 et `i` vaut 2 (la valeur de `y` au moment de l'appel).

En revanche, à la ligne 13, l'appel `p(x, x)` entraîne que `a` et `b` sont alias dans `p`. Ainsi, l'instruction `a = 0` a pour effet de positionner `b` à 0 aussi. En conclusion, `j` reçoit la valeur 0 au lieu de la valeur 1 attendue. ♦

Nous avons abordé les contraintes imposées dans $SOS\text{Sub}_{\mathcal{C}}^{ext}$ par le passage en paramètre de références à l'occasion des listes mutables (*cf.* Section 5.2.2). Une des raisons alors avancées était que les listes, en tant que structures de données allouées dynamiquement et en présence de récursivité, empêchent l'analyse exacte des relations d'alias.

En revanche, en $\text{Sub}_{\mathcal{C}}^{ext}$, la notion de référence sur les entiers n'existe pas. Cette caractéristique a l'avantage de rendre décidable le problème de déterminer statiquement les alias créés par le passage par référence de paramètres de type entier. Une telle analyse nécessite tout de même de connaître les appels des sous-programmes prenant des paramètres par référence. Or, en ce qui concerne $SOS\text{Sub}_{\mathcal{C}}^{ext}$, l'analyse doit pouvoir se faire sans aucun contexte d'appel.

Nous imposons donc comme prérequis de l'axiomatisation que les paramètres formels d'un sous-programme ne soient pas alias l'un de l'autre, même lorsqu'ils sont de type entier.

Sémantique

Le passage de paramètres par référence est vu comme la possibilité pour un sous-programme `f` de modifier la valeur des paramètres effectifs de son sous-programme appelant `Proc`. Nous traduisons cela dans l'appelant en affectant au paramètre effectif la valeur, à la fin de `f`, du paramètre formel de `f` correspondant. Dans ce but, le sous-programme appelé est représenté dans la logique équationnelle par une famille de fonctions renvoyant chacune la valeur de l'un des paramètres passé par référence. Ce procédé est semblable à celui que nous employons pour les boucles et les sous-programmes prenant des listes en paramètre.

Sous-programme appelé Sous l'hypothèse que les paramètres formels ne sont pas des alias les uns des autres, l'axiomatisation d'un sous-programme prenant des paramètres par référence se déroule de la même façon que pour les autres sous-programmes, jusqu'au moment de la formulation en équations de l'environnement final. Lors de cette étape, une fonction équationnelle supplémentaire est générée pour chaque paramètre passé par référence. La valeur renvoyée par ces fonctions est celle des paramètres correspondants dans l'environnement final de l'appelé. Ces fonctions rendent ainsi compte des modifications survenues sur les valeurs des paramètres effectifs.

Sous-programme appelant Dans `Proc`, l'appel de `f` entraîne, en plus de la sémantique d'un appel de sous-programme comme décrite par la Définition 5.31 (Page 110), une série de modifications de la valeur des paramètres effectifs passés par référence à `f`. Les paramètres effectifs sont toujours des variables de `Proc` (variables locales et paramètres).

Soient P un paramètre formel de `f` passé par référence et f_P^r la fonction équationnelle générée pour dénoter la valeur de ce paramètre après l'exécution de `f`. Soit `p` le paramètre effectif de `Proc` correspondant au paramètre P de `f`. L'effet de bord de `f` dans l'environnement de `Proc` pour `p` est la mise à jour de `p` à la valeur de retour de f_P^r.

Dans le cas d'un paramètre P de type liste passé par référence, deux mises à jour seront donc opérées :

1. celle effectuée pour tout paramètre de type liste (*cf.* Définition 5.31) qui rend compte des modifications sur la liste qui était pointée par son paramètre effectif `p` avant l'appel ;

2. celle spécifique aux paramètres passés par référence.

Nous reformulons l'opération d'appel de sous-programme précédente (Page 110) pour prendre en compte les effets de bord dus au passage par référence de paramètres. La nouvelle règle d'interprétation sémantique se trouve en Figure 5.7. Un exemple d'utilisation de cette opération est donné dans la section suivante (*cf.* Exemple 5.44).

Définition 5.37 (opérations d'appel de sous-programme avec passage par référence). Soient $env =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F, \mathcal{V}_f, \mathcal{V}_{ref} > \in Env^{ext}$, f le nom d'un sous-programme tel que, sans perte de généralité, $< P_1, \ldots, P_q >$ soit la séquence des paramètres formels de f, avec :

$$\frac{env[\mathsf{p}_1] \rhd (env_1, p_1) \ldots env_{n-1}[\mathsf{p}_n] \rhd (env_n, p_n)}{env[\mathsf{f(p}_1, \ldots, \mathsf{p}_n)] \rhd \text{appel}(env_n, \mathsf{f}, < p_1, \ldots, p_n >, < \mathsf{p}_1, \ldots, \mathsf{p}_n >)} \text{(appel\_ref)}$$

FIG. 5.7 – Sémantique des appels de sous-programme avec passage par référence

– $< P_1, \ldots, P_m >$ les paramètres de type entier non passés par référence,

– $< P_{m+1}, \ldots, P_n >$ les paramètres de type entier passés par référence,

– $< P_{n+1}, \ldots, P_o >$ les paramètres de type liste non passés par référence et,

– $< P_{o+1}, \ldots, P_q >$ les paramètres de type liste passés par référence.

Pour $i = 1 \ldots q$, soient p_i le paramètre effectif correspondant à P_i et p_i la valeur de ce paramètre dans l'environnement *env*. Soit $< p'_1, \ldots, p'_q >$ la séquence $< p_1, \ldots, p_q >$ où les p_i ont été remplacés par recons(env, p_i) pour $i = n+1 \ldots q$ (*i.e.*, pour les paramètres de type liste).

Enfin, soient :

– $f^r_{p_i}$, pour $i = m+1 \ldots n, o+1 \ldots q$, la fonction générée pour le paramètre passé par référence P_i ;

– f_{p_i}, pour $i = n+1 \ldots q$, la fonction générée pour le paramètre de type liste P_i.

– f_{p_0}, la fonction générée pour la valeur de retour du sous-programme f ou bien la fonction nulle si le type de retour est `void`.

Les opérations d'*appel de sous-programme avec passage par référence* sont définies comme suit :

$$\text{appel}(env, f, < p_1, \ldots, p_q >, < \mathsf{p}_1, \ldots, \mathsf{p}_q >) := \begin{cases} \text{appel}_{int}(env, f, < p_1, \ldots, p_q >, < \mathsf{p}_1, \ldots, \mathsf{p}_q >) & \text{si le type de retour de } f \text{ est } \texttt{int} \text{ ou } \texttt{void} \\ \text{appel}_{list}(env, f, < p_1, \ldots, p_q >, < \mathsf{p}_1, \ldots, \mathsf{p}_q >) & \text{si le type de retour de } f \text{ est } \texttt{list} \end{cases}$$

$$
\begin{array}{l}
\text{appel}_{int}(env, f, < p_1, \ldots, p_q >, < \mathsf{p}_1, \ldots, \mathsf{p}_q >) := \\
\quad \big(< \mathcal{E} \oplus \{(\mathsf{p}_i, \text{ref}_{q+i}) \mid i = o+1 \ldots q\} \\
\qquad \oplus \{(\mathsf{p}_i, f^r_{P_i}(< p'_1, \ldots, p'_q >)) \mid i = m+1 \ldots n\}, \\
\qquad (\mathcal{M} \oplus \{(p_i, l_{1_{v_i}}) \mid i = n+1 \ldots q\}) \cup \{(\text{ref}_{q+i}, l_{1_{v_{q+i}}}) \mid i = o+1 \ldots q\}, \\
\qquad \mathcal{D} \cup \{(v_i, l_{1_{v_i}}) \mid i = n+1 \ldots q\} \cup \{(v_{q+i}, l_{1_{v_{q+i}}}) \mid i = o+1 \ldots q\}, \\
\qquad F \cup \{(v_i, f_{P_i}(< p'_1, \ldots, p'_q >)) \mid i = n+1 \ldots q\} \\
\qquad\quad \cup \{(v_{q+i}, f^r_{P_i}(< p'_1, \ldots, p'_q >)) \mid i = o+1 \ldots q\}, \\
\qquad \mathcal{V}_f \cup \{v_i \mid i = n+1 \ldots q\} \cup \{v_{q+i} \mid i = o+1 \ldots q\}, \\
\qquad \mathcal{V}_{ref} \cup \{\text{ref}_{q+i} \mid i = o+1 \ldots q\} >, \\
\quad f_{P_0}(< p'_1, \ldots, p'_q >) \big)
\end{array}
$$

$$
\begin{array}{l}
\text{appel}_{list}(env, f, < p_1, \ldots, p_q >, < \mathsf{p}_1, \ldots, \mathsf{p}_q >) := \\
\quad \big(< \mathcal{E} \oplus \{(\mathsf{p}_i, \text{ref}_{q+i}) \mid i = o+1 \ldots q\} \\
\qquad \oplus \{(\mathsf{p}_i, f^r_{P_i}(< p'_1, \ldots, p'_q >)) \mid i = m+1 \ldots n\}, \\
\qquad (\mathcal{M} \oplus \{(p_i, l_{1_{v_i}}) \mid i = n+1 \ldots q\}) \\
\qquad \cup \{(\text{ref}_{q+i}, l_{1_{v_{q+i}}}) \mid i = o+1 \ldots q\} \\
\qquad \cup \{(\text{ref}_0, l_{1_{v_0}})\}, \\
\qquad \mathcal{D} \cup \{(v_i, l_{1_{v_i}}) \mid i = n+1 \ldots q\} \\
\qquad\quad \cup \{(v_{q+i}, l_{1_{v_{q+i}}}) \mid i = o+1 \ldots q\} \\
\qquad\quad \cup \{(v_0, l_{1_{v_0}})\}, \\
\qquad F \cup \{(v_i, f_{P_i}(< p'_1, \ldots, p'_q >)) \mid i = n+1 \ldots q\} \\
\qquad\quad \cup \{(v_{q+i}, f^r_{P_i}(< p'_1, \ldots, p'_q >)) \mid i = o+1 \ldots q\} \\
\qquad\quad \cup \{(v_0, f_{P_0}(< p'_1, \ldots, p'_q >))\}, \\
\qquad \mathcal{V}_f \cup \{v_i \mid i = n+1 \ldots q\} \cup \{v_{q+i} \mid i = o+1 \ldots q\} \cup \{v_0\}, \\
\qquad \mathcal{V}_{ref} \cup \{\text{ref}_{q+i} \mid i = o+1 \ldots q\} \cup \{\text{ref}_0\} >, \\
\quad \text{ref}_0 \big)
\end{array}
$$

Avec v_0, v_i, pour $i = n+1 \ldots q, q+o+1 \ldots 2q$ et ref_0 des variables fraîches.

Exemple 5.38 (appel du sous-programme `swap`). Nous illustrons le fonctionnement de cette opération sur l'exemple de l'appel du sous-programme `swap` (Figure 5.6) dans le sous-programme `p` de l'Exemple 5.35 que nous rappelons ici :

```
int p() {
  int x = 1;
  int y = 2;

  swap(x, y);
  ⋮
}
```

Le sous-programme `p` comporte deux variables locales de type entier, son environnement initial est donc le suivant :

$$< \{(x, 1), (y, 2)\}, \emptyset, \emptyset, \emptyset >$$

L'axiomatisation du sous-programme `swap` produit la définition d'une fonction pour chaque paramètre passé par référence, à savoir : swap_a^r et swap_b^r. Chaque fonction renvoie la valeur à la fin de l'évaluation de `swap` du paramètre qui lui est associé. Dans le cas de `swap`, nous aurons (*cf.* Section 5.4) :

$$\text{swap}_a^r(a, b) = b$$
$$\text{swap}_b^r(a, b) = a$$

Lors de l'évaluation de l'appel de `swap` dans `p`, la règle (appel_ref) de la Figure 5.7 est appliquée. Puisque le type de retour de `swap` est `void`, l'environnement de `p` est modifié par l'opération appel_{int}. Les variables `x` et `y`, étant passées par référence, sont modifiées par l'appel de `swap` : leur nouvelle valeur est celle renvoyée par la fonction produite pour le paramètre formel de `swap` correspondant à la variable, à savoir : swap_a^r pour `x` et swap_b^r pour `y`. L'environnement obtenu est donc le suivant :

$$< \{(x, \text{swap}_a^r(1, 2)), (y, \text{swap}_b^r(1, 2))\}, \emptyset, \emptyset, \emptyset > \quad \blacklozenge$$

Exemple 5.39 (appel avec un paramètre de type liste passé par référence). Nous donnons encore un exemple d'emploi des opérations d'appel de sous-programme avec passage par référence dans lequel nous envisageons le passage d'un paramètre de type liste. Soit le programme suivant :

```
void ajouterTete(list &L, int tete) {
  L = add(tete, L);
}

list creerListe(int i) {
  list La;

  ajouterTete(La, i);
  return La;
}
```

Le sous-programme `creerListe` comporte un paramètre de type entier et une variable locale de type liste, son environnement initial est donc le suivant :

$$< \{(i, i^e), (La, \text{ref}_1)\}, \{(\text{ref}_1, \text{NULL})\}, \emptyset, \emptyset, \emptyset, \{\text{ref}_1\} >$$

Nous verrons en Section 5.4 que le sous-programme `ajouterTete` donne lieu à la définition équationnelle de deux fonctions, une première pour le paramètre de type liste et une seconde parce que ce paramètre est passé par référence :

$$\text{ajouterTete}_L(L, tete) = L$$
$$\text{ajouterTete}_L^r(L, tete) = tete \cdot L$$

L'appel de `ajouterTete` dans `creerListe` entraîne l'exécution de l'opération appel_{int}, qui modifie l'environnement du dernier sous-programme de deux façons :

1. Comme c'était déjà le cas avant d'introduire l'appel par référence (*cf.* Définition 5.31 Page 110), l'appel d'un sous-programme avec un paramètre de type liste modifie l'environnement pour rendre compte des effets de bord qui ont pu affecter la liste passée en paramètre. Dans notre exemple, l'environnement de `creerListe` devient donc :

$$<\{(i, i^e), (La, \mathrm{ref}_1)\}, \{(\mathrm{ref}_1, l_{1_{v_1}})\}, \{(v_1, l_{1_{v_1}})\},$$
$$\{(v_1, \mathrm{ajouterTete}_L(\textsc{null}, i^e))\}, \{v_1\}, \{\mathrm{ref}_1\} >$$

2. La nouveauté avec le passage par référence est que la valeur du paramètre effectif est modifiée par l'appel d'un sous-programme. Puisque dans notre exemple le paramètre effectif est une variable de type liste, en l'occurrence `La`, nous introduisons une constante (v_2) pour la nouvelle valeur de `La` et nous lui associons une liste nommée ($l_{1_{v_2}}$) qui nous servira à représenter la structure de la liste que référence dorénavant `La`. L'environnement de `creerListe` après l'appel de `ajouterTete` est donc finalement :

$$<\{(i, i^e), (La, \mathrm{ref}_2)\}, \{(\mathrm{ref}_1, l_{1_{v_1}}), (\mathrm{ref}_2, l_{1_{v_2}})\}, \{(v_1, l_{1_{v_1}}), (v_2, l_{1_{v_2}})\},$$
$$\{(v_1, \mathrm{ajouterTete}_L(\textsc{null}, i^e)), (v_2, \mathrm{ajouterTete}'_L(\textsc{null}, i^e))\}, \{v_1, v_2\}, \{\mathrm{ref}_1, \mathrm{ref}_2\} >$$

♦

5.4 Formulation en équations

L'objectif de l'axiomatisation est de donner une définition équationnelle de chaque sous-programme d'un programme sous la forme d'une fonction de transfert des valeurs en entrée du sous-programme vers les valeurs en sortie du sous-programme. Les *valeurs* auxquelles nous nous intéressons comprennent les listes, en tant que séquence d'entiers ou de listes, et les entiers.

Avec $SOS\mathrm{Sub}^{ext}_{\mathcal{C}}$, la validité de l'axiomatisation est conditionnée par des prérequis sur l'existence et la formation des alias dans les programmes $\mathrm{Sub}^{ext}_{\mathcal{C}}$ (les prérequis sont récapitulés dans l'Annexe B). Malheureusement, des résultats d'indécidabilité et la nature même du problème que prétend résoudre notre système (analyse statique de programmes incomplets) s'opposent à l'automatisation de la vérification des prérequis. Par conséquent, la démonstration qu'un programme $\mathrm{Sub}^{ext}_{\mathcal{C}}$ satisfait les prérequis est majoritairement accomplie à la main, éventuellement interactivement dans un système de preuve, selon une approche différente de la nôtre.

Cette section décrit la dernière étape de l'axiomatisation. Cette étape consiste à formuler les équations correspondant aux sous-programmes $\mathrm{Sub}^{ext}_{\mathcal{C}}$ à partir des environnements étendus de celles-ci. Les environnements sont le produit de l'évaluation symbolique des sous-programmes par la fonction d'interprétation sémantique définie dans les Sections 5.2 et 5.3.

Nous donnons tout d'abord les principes de la formulation des équations dans la Section 5.4.1. Nous détaillons ensuite, Section 5.4.2, la définition des fonctions associées aux sous-programmes, et, Section 5.4.3, la définition de celles associées aux boucles.

5.4.1 Principes

La formulation des équations logiques à partir des environnements étendus se fait selon les mêmes principes dans $SOS\text{Sub}_{\mathcal{C}}^{ext}$ que dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$. Chaque sous-programme est découpé en chemins d'exécution. Chaque chemin d'exécution est associé à une condition Cond_{ce}. Cette condition est la conjonction des conditions des instructions de branchement du chemin d'exécution, *i.e.*, les instructions `if`, évaluées dans l'environnement correspondant au point du sous-programme auquel elles apparaissent. L'évaluation des conditions peut d'ailleurs avoir un effet de bord sur l'environnement puisque les opérateurs `element` et `next` sont autorisés dans les conditions, ainsi que les appels de sous-programme.

Les boucles étant remplacées par des appels de fonction, le nombre des chemins d'exécution dans un sous-programme est fini. Les instructions d'un chemin d'exécution sont évaluées par la fonction d'interprétation sémantique dans l'environnement initial du sous-programme. Cette évaluation produit un environnement étendu final env_{ce} pour le chemin d'exécution. De plus, les listes nommées de la condition Cond_{ce} sont mises à jour au fur et à mesure des décompositions dans l'évaluation du chemin.

Dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$, chaque couple ($\text{Cond}_{ce}$, env_{ce}) donne lieu à la définition d'une équation logique qui lie la condition du chemin d'exécution et la valeur de retour du sous-programme $\text{Sub}_{\mathcal{C}}$ lue dans l'environnement. Dans $SOS\text{Sub}_{\mathcal{C}}^{ext}$, pour chaque chemin d'exécution, une famille d'équations est générée sur le modèle des familles d'équations pour les boucles dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$ (*cf.* Section 5.4.2). Les boucles dans $SOS\text{Sub}_{\mathcal{C}}^{ext}$ sont traitées de la même manière que dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$, à la différence qu'une famille d'équations supplémentaire est générée pour les variables de type liste du sous-programme (*cf.* Section 5.4.3).

5.4.2 Équations pour les sous-programmes

Un sous-programme $\text{Sub}_{\mathcal{C}}^{ext}$ entraîne la définition de plusieurs fonctions équationnelles. Le nombre de ces fonctions dépend du type et du mode de passage des paramètres du sous-programme ainsi que de l'existence ou non d'une valeur de retour. Chaque chemin d'exécution dans le sous-programme produit exactement une équation conditionnelle pour chacune des fonctions équationnelles en cours de définition.

La formulation des équations conditionnelles d'un sous-programme f pour un chemin d'exécution (Cond_{ce}, env_{ce}) se fait en deux temps : d'abord la construction de la condition commune à toutes les équations, puis la génération de l'équation complète pour chaque fonction équationnelle.

Construction de la condition

Nous donnons dans cette section la définition des conditions formées pour les équations produites par $SOS\text{Sub}_{\mathcal{C}}^{ext}$.

Définition 5.40 (condition des équations de $SOS\text{Sub}_{\mathcal{C}}^{ext}$). Soit un chemin d'exécution (Cond_{ce}, env_{ce}) d'un sous-programme, avec $env_{ce} =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F >$. La condition, Cond_e, de toutes les équations formulées pour ce chemin est :

$$\text{Cond}_e := \text{Cond}_{ce} \bigwedge_{l \propto \mathcal{D}} l = \text{Val}_{\mathcal{D}}(l) \bigwedge_{c \propto F} c = \text{Val}_F(c)$$

La première série de contraintes ajoutée à Cond_{ce} indique quelle a été la décomposition de la variable de type liste l dans le sous-programme pour ce chemin d'exécution. Ainsi, les fonctions équationnelles produites ne pourront pas être appliquées à des listes qui ne comporteraient pas au moins autant d'éléments que ceux qui seront accédés dans les chemins concernés du sous-programme.

Exemple 5.41. Soit le sous-programme suivant :

```
void f(list &L) {
  L = next(L);
}
```

Ce sous-programme ne comporte qu'un chemin d'exécution dans lequel la liste `L` est décomposée en $e_{1_L} \cdot l_{2_L}$. La condition des équations générées pour `f` comporteront donc la contrainte : $L = e_{1_L} \cdot l_{2_L}$. Ainsi, un appel `f(NULL)` ne serait pas défini. ⧫

La seconde série de contraintes ajoutée à Cond_{ce} indique la valeur des constantes qui ont été définies pour représenter la valeur des listes après une boucle ou un appel de sous-programme.

Exemple 5.42. Soit le sous-programme suivant :

```
void f(list &L) {
  L = g();
  L = next(L);
}
```

Dans ce sous-programme, l'appel `g()` introduit une constante $c = g()$. Nous introduisons aussi la liste nommée correspondant à c : $c = l_{1_c}$. Dans l'environnement, la liste L prend la valeur de l_{1_c}. Lors de l'instruction suivante, l_{1_c} est décomposée en $c = e_{1_c} \cdot l_{2_c}$. Nous trouverons donc dans la condition des équations générées pour `f` la contrainte :

$$c = g() \wedge c = e_{1_c} \cdot l_{2_c}$$

La signification de cette contrainte est que l'appel de la fonction équationnelle g renvoie une liste qui comporte au moins deux éléments. ⧫

Formulation des équations complètes

Chaque sous-programme $\text{Sub}_{\mathcal{C}}^{ext}$ `f` peut engendrer la définition de trois types de fonctions équationnelles :

- la fonction f qui correspond à la valeur de retour de `f`, si elle existe ;
- les fonctions f_p, pour tout paramètre p de `f` de type liste, qui correspondent aux effets de bord survenus sur les têtes des listes référencées par les paramètres p au moment de l'appel de `f` ;
- les fonctions f_p^r, pour tout paramètre p de `f` passé par référence, qui correspondent aux valeurs des paramètres p à la fin de `f` ;

Définition 5.43 (équations des sous-programmes $\mathbf{Sub}_{\mathcal{C}}^{ext}$). Soient un chemin d'exécution d'un sous-programme `f`, ($\text{Cond}_{ce}, env_{ce}$), avec $env_{ce} =< \mathcal{E}, \mathcal{M}, \mathcal{D}, F >$, et Cond_e la condition construite pour les équations de ce chemin selon la Définition 5.40. Soit $< P_1, \ldots, P_n >$ la séquence des paramètres formels du sous-programme.

Les équations formulées pour ce chemin du sous-programme `f` sont les suivantes :

- $\text{Cond}_e \Rightarrow f(P_1, \ldots, P_n) = ret$, avec ret la valeur de l'expression de retour de `f` dans env_{ce}, si `f` a une valeur de retour ;
- $\text{Cond}_e \Rightarrow f_{P_i}(P_1, \ldots, P_n) = \text{recons}(env_{ce}, \text{ref}_{P_i})$, pour $1 \leq i \leq n$ tel que $\text{type}(P_i) =$ `list`, avec ref_{P_i} la référence nommée associé à P_i dans l'environnement initial de `f` ;
- $\text{Cond}_e \Rightarrow f^r_{P_i}(P_1, \ldots, P_n) = \text{recons}(env_{ce}, \text{Val}_{\mathcal{E}}(env_{ce}, P_i))$, pour $1 \leq i \leq n$ tel que P_i soit passé par référence.

Exemple 5.44 (split). Nous développons dans cet exemple l'axiomatisation du sous-programme `split` du programme de tri `MergeSort` de la Figure 1.6 (Page 19). La satisfaction des prérequis de l'axiomatisation pour le programme `MergeSort` est discutée dans la Section 7.1. Pour faciliter la lecture de l'exemple, nous en redonnons le code source :

```
1  list split(list L)
2  {
3    list pSC;
4
5    if(L == NULL) return NULL;
6    else if(next(L) == NULL) return NULL;
7    else {
8      pSC = next(L);
9      L->next = next(pSC);
10     pSC->next = split(next(pSC));
11     return pSC;
12   }
13 }
```

L'environnement initial du sous-programme, après l'application de la règle (decl_list) pour la déclaration de `pSC`, est :

$$env_{ini} =< \{(L, \text{ref}_1), (pSC, \text{ref}_2)\}, \{(\text{ref}_1, l_{1_L}), (\text{ref}_2, \text{NULL})\}, \\ \{(L, l_{1_L})\}, \emptyset, \emptyset, \{\text{ref}_1, \text{ref}_2\} >$$

La déclaration du sous-programme `split` annonce une valeur de retour de type liste et un paramètre de type liste. Son axiomatisation génèrera donc deux fonctions équationnelles, respectivement : *split* et $split_L$. Le sous-programme comporte trois chemins d'exécution :

1. Le premier correspond à la ligne 5 et a pour condition associée $l_{1_L} = \text{NULL}$. L'environnement final de ce chemin est le même que l'environnement initial. La valeur

de l'expression de retour est NULL et pour la liste initialement référencée par `L` : recons(env_{ini}, ref_1) = l_{1_L}. La formulation des deux équations de ce chemin est donc :

$$L = \text{NULL} \wedge L = l_{1_L} \Rightarrow split(L) = \text{NULL}$$
$$L = \text{NULL} \wedge L = l_{1_L} \Rightarrow split_L(L) = l_{1_L}$$

2. Le deuxième chemin d'exécution correspond à la ligne 5 du programme. Du fait de l'opérateur `next` dans la condition du `if`, l'environnement inital est devenu :

$$\begin{aligned} env_f = <&\{(L, \text{ref}_1), (pSC, \text{ref}_2)\},\\ &\{(\text{ref}_1, \text{ref}_3 \cdot \text{ref}_4), (\text{ref}_2, \text{NULL}), (\text{ref}_3, e_{1_L}), (\text{ref}_4, l_{2_L})\},\\ &\{(l, e_{1_L} \cdot l_{2_L})\}, \emptyset, \emptyset, \{\text{ref}_1, \text{ref}_2, \text{ref}_3, \text{ref}_4\} > \end{aligned}$$

Sa condition associée est $e_{1_L} \cdot l_{2_L} \neq \text{NULL} \wedge l_{2_L} = \text{NULL}$. La valeur de l'expression de retour est NULL et pour la liste initialement référencée par `L` : recons(env_f, ref_1) = $e_{1_L} \cdot l_{2_L}$. La formulation des deux équations de ce chemin est donc :

$$e_{1_L} \cdot l_{2_L} \neq \text{NULL} \wedge l_{2_L} = \text{NULL} \wedge L = e_{1_L} \cdot l_{2_L} \Rightarrow split(L) = \text{NULL}$$
$$e_{1_L} \cdot l_{2_L} \neq \text{NULL} \wedge l_{2_L} = \text{NULL} \wedge L = e_{1_L} \cdot l_{2_L} \Rightarrow split_L(L) = e_{1_L} \cdot l_{2_L}$$

3. Enfin, le troisième chemin contient les instructions qui vont de la ligne 8 à la ligne 11. Après la ligne 9, l'environnement initial est devenu :

$$\begin{aligned} <&\{(L, \text{ref}_1), (pSC, \text{ref}_4)\},\\ &\{(\text{ref}_1, \text{ref}_3 \cdot \text{ref}_6), (\text{ref}_2, \text{NULL}), (\text{ref}_3, e_{1_L}), (\text{ref}_4, \text{ref}_5 \cdot \text{ref}_6), (\text{ref}_5, e_{2_L}), (\text{ref}_6, l_{3_L})\},\\ &\{(l, e_{1_L} \cdot e_{2_L} \cdot l_{3_L})\}, \emptyset, \emptyset, \{\text{ref}_1, \ldots, \text{ref}_6\} > \end{aligned}$$

L'environnement final, suite à l'appel récursif de `split`, est :

$$\begin{aligned} env_f = <&\{(L, \text{ref}_1), (pSC, \text{ref}_4)\}, \{(\text{ref}_1, \text{ref}_3 \cdot \text{ref}_6), (\text{ref}_2, \text{NULL}),\\ &(\text{ref}_3, e_{1_L}), (\text{ref}_4, \text{ref}_5 \cdot \text{ref}_7), (\text{ref}_5, e_{2_L}), (\text{ref}_6, l_{1_{v_1}}), (\text{ref}_6, l_{1_{v_1}})\},\\ &\{(l, e_{1_L} \cdot e_{2_L} \cdot l_{3_L}), (v_1, l_{1_{v_1}}), (v_2, l_{1_{v_2}})\},\\ &\{(v_1, split_L(l_{3_L}), (v_2, split(l_{3_L})\}, \{v_1, v_2\}, \{\text{ref}_1, \ldots, \text{ref}_7\} > \end{aligned}$$

La condition associée à cet environnement est $e_{1_L} \cdot e_{2_L} \cdot l_{3_L} \neq \text{NULL} \wedge e_{2_L} \cdot l_{3_L} \neq \text{NULL}$. La valeur de l'expression de retour est recons(env_f, ref_4) = $e_{2_L} \cdot l_{1_{v_2}}$ et pour la liste initialement référencée par `L` : recons(env_f, ref_1) = $e_{1_L} \cdot l_{1_{v_1}}$. La formulation des deux équations de ce chemin est donc :

$$\begin{aligned} &e_{1_L} \cdot e_{2_L} \cdot l_{3_L} \neq \text{NULL} \wedge e_{2_L} \cdot l_{3_L} \neq \text{NULL} \wedge\\ &L = e_{1_L} \cdot e_{2_L} \cdot l_{3_L} \wedge v_1 = l_{1_{v_1}} \wedge v_2 = l_{1_{v_2}} \wedge\\ &\qquad v_1 = split_L(l_{3_L}) \wedge v_2 = split(l_{3_L}) \Rightarrow split(L) = e_{2_L} \cdot l_{1_{v_2}} \end{aligned}$$

$$\begin{aligned} &e_{1_L} \cdot e_{2_L} \cdot l_{3_L} \neq \text{NULL} \wedge e_{2_L} \cdot l_{3_L} \neq \text{NULL} \wedge\\ &L = e_{1_L} \cdot e_{2_L} \cdot l_{3_L} \wedge v_1 = l_{1_{v_1}} \wedge v_2 = l_{1_{v_2}} \wedge\\ &\qquad v_1 = split_L(l_{3_L}) \wedge v_2 = split(l_{3_L}) \Rightarrow split_L(L) = e_{1_L} \cdot l_{1_{v_1}} \end{aligned}$$

Les équations finales de `split` en Figure 1.9 (Page 23) s'obtiennent simplement en substituant aux noms des variables leur valeur dans la condition et en supprimant les égalités trivialement vraies. Ceci peut être fait automatiquement. ♦

5.4.3 Équations pour les boucles

Dans *SOS*$\text{Sub}_{\mathcal{C}}^{ext}$, une boucle est interprétée comme un sous-programme récursif terminal. Les variables modifiées dans le corps de la boucle sont passées par référence. Un nom de sous-programme unique dans tout le programme est associé à chaque boucle. Le corps d'un sous-programme associé à une boucle donnée est constitué du corps de la boucle suivi d'un appel récursif au sous-programme. Les équations générées pour une boucle sont ainsi celles générées pour le sous-programme récursif associé à la boucle, comme décrit dans la section précédente.

Définition 5.45 (équations des boucles $\text{Sub}_{\mathcal{C}}^{ext}$). Soit une boucle B de condition Cond_B et de corps `corps` dans un sous-programme `f`. Soient V_{proc}, l'ensemble des variables locales et paramètres de `f`, et $V_{mod} \subset V_{proc}$, l'ensemble des variables affectées dans `corps`. Soient $\{P_1, \ldots, P_m\} = V_{mod}$, $\{P_{m+1}, \ldots, P_n\} = V_{proc} \setminus V_{mod}$ et t_i le type $\text{Sub}_{\mathcal{C}}^{ext}$ de P_i pour $i = 1 \ldots n$.

Soit loop un nom de sous-programme non utilisé dans le programme. Les équations produites pour la boucle B sont celles obtenues par la Définition 5.43 pour le sous-programme $\text{Sub}_{\mathcal{C}}^{ext}$ suivant :

```
void loop(t1 &P1,..., tm &Pm, tm+1 Pm+1,..., tn Pn) {
  if(CondB) {
    corps;
    loop(P1,..., Pn);
  }
}
```

Exemple 5.46. Nous exemplifions la définition des équations formulées pour les boucles par le sous-programme suivant :

```
void f(list L) {
  list La;
  int i;

  while(L != NULL) {
    L->element = 0;
    L = next(L);
    i = i + 1;
  }
  ⋮
}
```

Les variables modifiées dans le corps de la boucle sont L et i ; elles seront donc passées par référence dans le sous-programme loop associé à la boucle. Voici ce sous-programme :

```
void loop(list &L, int &i, list La) {
  if(L != NULL) {
    L->element = 0;
    L = next(L);
    i = i + 1;
```

```
    loop(L, i, La);
  }
}
```

Compte tenu de la Définition 5.43 des équations produites pour les sous-programmes $\text{Sub}_{\mathcal{C}}^{ext}$, la famille des fonctions générées pour la boucle comprend : loop_L^r, loop_L, loop_i^r et loop_{La}. Le sous-programme loop ne comporte que deux chemins d'exécution dont les conditions sont $L \neq \text{NULL}$ et $\neg(L \neq \text{NULL})$, respectivement. Les fonctions de cette famille seront donc définies par deux équations, une pour chaque chemin d'exécution.

Pour les variables modifiées dans le corps de la boucle, nous obtiendrons les équations suivantes :

$$\begin{aligned} L \neq \text{NULL} \wedge L = e_{1_L} \cdot l_{2_L} &\Rightarrow \text{loop}_L^r(L, i, La) = \text{loop}_L^r(l_{2_L}, i+1, La) \\ \neg(L \neq \text{NULL}) &\Rightarrow \text{loop}_L^r(L, i, La) = L \end{aligned}$$

$$\begin{aligned} L \neq \text{NULL} \wedge L = e_{1_L} \cdot l_{2_L} &\Rightarrow \text{loop}_i^r(L, i, La) = \text{loop}_i^r(l_{2_L}, i+1, La) \\ \neg(L \neq \text{NULL}) &\Rightarrow \text{loop}_i^r(L, i, La) = i \end{aligned}$$

Enfin, pour les variables de type liste du sous-programme `f`, nous obtiendrons les équations suivantes, qui rendent compte des effets de bord de la boucle sur les maillons des listes :

$$\begin{aligned} L \neq \text{NULL} \wedge L = e_{1_L} \cdot l_{2_L} &\Rightarrow \text{loop}_L(L, i, La) = 0 \cdot \text{loop}_L(l_{2_L}, i+1, La) \\ \neg(L \neq \text{NULL}) &\Rightarrow \text{loop}_L(L, i, La) = L \end{aligned}$$

$$\begin{aligned} L \neq \text{NULL} \wedge L = e_{1_L} \cdot l_{2_L} &\Rightarrow \text{loop}_{La}(L, i, La) = \text{loop}_{La}(l_{2_L}, i+1, La) \\ \neg(L \neq \text{NULL}) &\Rightarrow \text{loop}_{La}(L, i, La) = La \end{aligned}$$

♦

Troisième partie

Expérimentations sur machine

Chapitre 6

Preuve de propriétés de programmes et $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$

Dans ce chapitre, nous rapportons une série d'expérimentations menées avec notre système $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$. Nous avons écrit en $\mathrm{Sub}_{\mathcal{C}}$ plusieurs algorithmes typiques du domaine de la vérification de programme et de la démonstration automatique. Nous avons alors utilisé $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ pour produire leur sémantique algébrique. Ceci permet d'évaluer la qualité des équations produites, qui sont proches de ce que nous aurions pu obtenir manuellement.

Pour certains des programmes, nous présentons de surcroît la preuve de plusieurs de leurs propriétés. Les démonstrations des propriétés sont effectuées dans des systèmes de preuve, PVS et RRL, à partir de la sémantique algébrique des programmes générée par $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$. Ceci permet d'évaluer plus globalement notre approche et de montrer l'effectivité de la preuve de propriétés de programmes avec la sémantique des programmes que produit notre système.

6.1 Plus grand commun diviseur

Nous touchons finalement au but que nous nous étions fixé dans les premières pages de ce manuscrit, Section 1.1.1, où nous avions introduit le programme du calcul du pgcd comme l'incarnation de la problématique de la preuve de programme. Nous montrons ici comment notre approche propose une réponse aux questions alors soulevées.

Nous rappelons par commodité le programme $\mathrm{Sub}_{\mathcal{C}}$ du *pgcd* en Figure 6.1(a) et les équations produites par $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ en Figure 6.1(b). Les preuves des propriétés que nous avions énoncées pour le programme `pgcd` ont été conduites dans le système de preuve PVS. Nous donnons en Figure 6.1(c) la théorie PVS contenant l'axiomatisation $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ du programme `pgcd` (les trois axiomes de la théorie). L'Annexe C.1 comporte le détail des démonstrations conduites dans PVS pour référence.

Les théorèmes *th1*, *th2* et *th3* constituent la spécification du programme `pgcd`. La démonstration que ces théorèmes sont des conséquences logiques des axiomes du programme assure la correction du programme. Les preuves des différents théorèmes et lemmes se font en raisonnant par cas sur la valeur du deuxième argument de la fonction pgcd, de façon à

```
int pgcd(int a, int b)
{
  int r;

  while(b != 0)
  {
    r = b;
    b = a % b;
    a = r;
  }
  return a;
}
```

(a) programme `pgcd`

$$b \neq 0 \Rightarrow \mathrm{loop}_a(a,b,r) = \mathrm{loop}_a(b, a \bmod b, b)$$
$$\mathrm{loop}_a(a,0,r) = a$$
$$\mathrm{pgcd}(a,b) = \mathrm{loop}_a(a,b,0)$$

(b) axiomatisation par $SOS\mathrm{Sub}^{\mathcal{R}}_{\mathcal{C}}$ de `pgcd`

```
pgcd: THEORY
 BEGIN
  a, b, r: VAR nat

  pgcd: [nat, nat → nat]
  loop: [nat, nat, nat → nat]
  pgcd1: AXIOM b ≠ 0 ⇒ loop_a(a, b, r) = loop_a(b, rem(b)(a), b)
  pgcd2: AXIOM b = 0 ⇒ loop_a(a, b, r) = a
  pgcd3: AXIOM pgcd(a, b) = loop_a(a, b, 0)

  lemma1: LEMMA b ≠ 0 ⇒ rem(b)(a) = rem(b)(a − b)
  th1: THEOREM a > b ⇒ pgcd(a, b) = pgcd(a − b, b)

  lemma2: LEMMA loop_a(a, b, r) = loop_a(a, b, 0)
  th2: THEOREM pgcd(a, b) = pgcd(b, a)

  th3: THEOREM pgcd(a, a) = a
 END pgcd
```

(c) théorie PVS du programme `pgcd`

FIG. 6.1 – Du programme `pgcd` aux preuves de ses propriétés

```
list inverser(list L) {
 list W = NULL;
 while (L != NULL) {
  W = add(element(L), W);
  L = next(L);
 }
 return W;
}
```

(a) le programme `inverser`

$$L = \text{NULL} \Rightarrow \text{loop}_W(L, W) = W$$
$$L \neq \text{NULL} \Rightarrow \text{loop}_W(L, W) = \text{loop}_W(\text{next}(L), \text{add}(\text{element}(L), W))$$
$$\text{inverser}(L) = \text{loop}_W(L, \text{NULL})$$

(b) axiomatisation par $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$ de `inverser`

FIG. 6.2 – Axiomatisation du programme `inverser`

pouvoir réécrire les termes de l'égalité à prouver avec l'un ou l'autre des axiomes **pgcd1** et **pgcd2**. Deux lemmes ont été introduits pour faciliter la démonstration, dont un qui porte sur la fonction *rem*. Cette fonction est l'équivalent dans PVS de l'opérateur modulo du langage $\text{Sub}_{\mathcal{C}}$ ($rem(b)(a) =$ `a % b`).

La fonction *rem* est une des nombreuses fonctions définies dans le *prélude* du système PVS. Ce prélude, composé aussi de définitions de types, de prédicats et de théorèmes usuels, fonde le socle commun à toutes les théories développées ensuite par l'utilisateur. La plupart des systèmes de preuve proposent de telles bibliothèques logiques standard. C'est pourquoi nous avons omis de spécifier, dans $SOS\text{Sub}_{\mathcal{C}}$ même, les opérateurs des expressions et conditions du langage $\text{Sub}_{\mathcal{C}}$. Notons, toutefois, que si nous désirions figer la sémantique de ces opérateurs et la rendre indépendante de fluctuations éventuelles d'un système de preuve à l'autre, nous pourrions très simplement fournir notre propre bibliothèque de définitions en « prélude » des axiomes produits par $SOS\text{Sub}_{\mathcal{C}}$.

6.2 Inversion des éléments dans une liste

Nous illustrons maintenant notre approche avec un exemple de programme manipulant une liste. La Figure 6.2(a) contient un programme $\text{Sub}_{\mathcal{C}}$ supposé renvoyer une liste dont les éléments sont dans l'ordre inverse des éléments de la liste passée en paramètre. Une des propriétés que doit posséder un tel programme est la suivante :

$$\text{inverser}(\text{inverser}(L)) = L$$

Afin de démontrer cette propriété pour le programme `inverser`, nous l'axiomatisons avec $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$. Nous obtenons les équations de la Figure 6.2(b). Cette fois, nous utilisons

le démonstrateur de théorèmes RRL (version 4.1). Ce système est capable de réaliser automatiquement des preuves par récurrence, en revanche, il n'offre pas de bibliothèque de théories prédéfinies. Nous devons donc définir nous-mêmes le type liste, les opérations sur les listes et la fonction de concaténation de listes, `append`, qui nous sera utile au cours de la démonstration. Ces définitions ont la forme suivante dans RRL :

```
[C_Null : list]
[C_Cons : univ, list -> list]

[Cdr : list -> list]
Cdr(C_Cons(v_e, v_L)) := v_L

[Car : list -> univ]
Car(C_Cons(v_e, v_L)) := v_e

[Append : list, list -> list]
Append(C_Null, v_L) := v_L
Append(v_L, C_Null) := v_L
Append(C_Cons(v_La, v_Lb), v_Lc) := C_Cons(v_La, Append(v_Lb, v_Lc))
Append(Append(v_La, v_Lb), v_Lc) == Append(v_La, Append(v_Lb, v_Lc))
```

Ensuite, nous ajoutons les équations de $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$ pour le programme `inverser`, en Figure 6.2(b), et nous essayons d'orienter toutes les équations en règles de façon à ce qu'elles servent de procédure de décision pour la théorie qu'elles définissent. Nous avons vu en Section 4.4.1 comment réaliser cette étape dans RRL et nous obtenons le système de réécriture suivant :

```
[1] CDR(C_CONS(V_E, V_L)) ---> V_L  [DEF, 1]
[2] CAR(C_CONS(V_E, V_L)) ---> V_E  [DEF, 2]
[3] APPEND(C_NULL, V_Y) ---> V_Y  [DEF, 3]
[4] APPEND(V_X, C_NULL) ---> V_X  [DEF, 4]
[5] APPEND(C_CONS(V_X, V_Y), V_Z) --->
      C_CONS(V_X, APPEND(V_Y, V_Z))  [DEF, 5]
[6] LOOPW(C_NULL, V_W) ---> V_W  [DEF, 6]
[7] LOOPW(C_CONS(V_E, V_L), V_W) --->
      LOOPW(V_L, C_CONS(V_E, V_W))  [DEF, 7]
[8] REVERSE(V_L) ---> LOOPW(V_L, C_NULL)  [DEF, 8]
[9] APPEND(APPEND(V_LA, V_LB), V_LC) --->
      APPEND(V_LA, APPEND(V_LB, V_LC))  [USER, 9]
```

Les opérations `Car` et `Cdr` ne sont que partiellement définies mais le cas absent, lorsque la liste est vide, correspond à un cas d'erreur. Aussi, il n'a pas lieu d'intervenir dans la preuve. D'ailleurs, si le cas était rencontré, la preuve ne pourrait pas aboutir et nous aurions une bonne indication de l'origine de l'erreur dans le programme.

Nous choisissons ensuite de mener la démonstration par une méthode de récurrence explicite (*cover set method*). Cette méthode n'exige pas que le système de réécriture

soit convergent ou que les définitions soient complètes. Pour mener à bien la preuve automatique de notre propriété, nous recourons à deux lemmes. Le premier rend apparent le rôle d'accumulateur de la variable `W` dans la boucle du programme :

```
prove
Loopw(v_La, C_Cons(v_e, v_Lb)) ==
  Append(Loopw(v_La, C_Null), C_Cons(v_e, v_Lb))
...
restart
hypothesis
hypothesis
continue
...
  [USER, 10] is an inductive theorem in the current system.
```

Les démonstrations dans RRL sont considérablement automatisées, cependant, pour ce lemme, le système échoue à appliquer l'hypothèse de récurrence et nous devons intervenir à deux occasions pour commander au système de réécrire les termes de la preuve en cours avec l'hypothèse de récurrence. La cause de ce comportement semble être une généralisation insuffisante de l'hypothèse de récurrence par le système.

Le second lemme établit la distributivité de `Loopw` sur `Append`. Cette fois, nous n'intervenons pas dans la preuve :

```
prove
Loopw(Append(v_La, v_Lb), C_Null) ==
  Append(Loopw(v_Lb, C_Null), Loopw(v_La, C_Null))
...
  [USER, 11] is an inductive theorem in the current system.
```

Pour terminer, nous prouvons sans autre intervention la propriété désirée du programme `inverser` :

```
prove
inverser(inverser(v_L)) == v_L
...
  [USER, 12] is an inductive theorem in the current system.
```

L'axiomatisation du programme `inverser` produite par $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ a ainsi permis de montrer une propriété générique du programme nécessitant une preuve par récurrence. La démonstration, guidée par deux lemmes, ne réclame que très peu d'interventions de la part de l'utilisateur.

6.3 Tri par insertion

Nous continuons à explorer la preuve de propriétés de programmes avec $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ sur un exemple plus complexe. La Figure 6.3(a) présente la version $\mathrm{Sub}_{\mathcal{C}}$ d'un programme de tri par insertion. Ce programme comporte deux fonctions :

- La fonction `ins` prend en entrée un entier `e` et une liste triée `L`. La fonction renvoie la liste `L` augmentée de `e` triée.

- La fonction `ISort` prend en entrée une liste `L` et renvoie une version triée de `L`. Elle procède en insérant à sa place, par un appel à la fonction `ins`, le premier élément de `L` dans la queue récursivement triée de la liste `L`.

L'axiomatisation de ces fonctions par $SOS\text{Sub}^{\mathcal{R}}_{\mathcal{C}}$ génère les équations des Figures 6.3(b) et 6.3(c).

Nous souhaitons prouver que la fonction `ISort` permet effectivement de trier une liste d'entiers. Cela nécessite de montrer deux propriétés du programme :

1. la liste renvoyée par le programme `ISort` est une *permutation* de la liste passée en paramètre, *i.e.*, les deux listes contiennent exactement les mêmes éléments, éventuellement dans un ordre différent ;

2. la liste renvoyée par le programme `ISort` n'est pas n'importe quelle permutation : elle est ordonnée.

Nous utilisons de nouveau le système de preuve PVS, qui prédéfinit avantageusement un type des listes paramétrique et un prédicat *member* pour l'appartenance à une liste. Nous déclarons donc un type des listes d'entiers et nous utilisons les équations générées par notre système pour définir la théorie du programme de tri par insertion, comme illustré par la Figure 6.4. L'Annexe C.2 comporte le détail des démonstrations conduites dans PVS pour référence.

6.3.1 Propriété de permutation

Nous nous attachons d'abord à prouver que $isort(L)$ est une permutation de la liste L. Pour cela, nous définissons la notion de permutation : nous dirons qu'une liste L' est une permutation de L, si, après avoir retiré de L' tous les éléments de L, L' est vide. Nous ajoutons donc les deux fonctions de la Figure 6.5 à la théorie du tri par insertion.

Nous introduisons aussi deux lemmes qui portent sur la fonction *ins*. Le premier, *ins_member*, garantit qu'après l'insertion par la fonction *ins* d'un élément dans une liste, cet élément appartient effectivement à la liste. Le second lemme nous dit que l'insertion d'un élément dans une liste avec *ins*, suivie du retrait de l'élément laisse la liste inchangée.

Les étapes importantes de la preuve de ces lemmes sont l'emploi de la récurrence structurelle sur les listes et un raisonnement par cas afin de pouvoir appliquer les axiomes **ins_le** ou **ins_gt** de la fonction *ins*. Le théorème *isort_perm* peut alors être démontré par récurrence sur l.

6.3.2 Propriété de liste triée

Il nous reste maintenant à prouver que la liste $isort(L)$ est triée, c'est-à-dire, que ses éléments sont ordonnés. Nous définissons cette notion par le prédicat *sorted* de la Figure 6.6. Nous donnons aussi quatre propriétés de *sorted* sous la forme des lemmes

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

list ins (int e, list L) {
  list ret = NULL;
  if (L == NULL)
    ret = add(e, NULL);
  else if (e <= element(L))
    ret = add(e, L);
  else {
    ret = ins(e, next(L));
    ret = add(element(L), ret);
  }
  return ret;
}
```

(a) le programme de tri par insertion

$$L = \text{NULL} \Rightarrow \text{ISort}(L) = \text{NULL}$$
$$L \neq \text{NULL} \Rightarrow \text{ISort}(L) = \text{ins}(\text{element}(L), \text{ISort}(\text{next}(L)))$$

(b) axiomatisation par $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$ de `ISort`

$$L = \text{NULL} \Rightarrow \text{ins}(e, L) = \text{add}(e, \text{NULL})$$
$$L \neq \text{NULL} \wedge e \leq \text{element}(L) \Rightarrow \text{ins}(e, L) = \text{add}(e, L)$$
$$L \neq \text{NULL} \wedge e > \text{element}(L) \Rightarrow \text{ins}(e, L) = \text{add}(\text{element}(L), \text{ins}(e, \text{next}(L)))$$

(c) axiomatisation par $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$ de `ins`

FIG. 6.3 – Axiomatisation du programme de tri par insertion

```
insertion_sort: THEORY
 BEGIN

  lists: TYPE = list[nat]

  x, y: VAR nat
  l, l1, l2: VAR lists

  ins: [nat, lists → lists]
  ins_empty: AXIOM ins(x, null) = cons(x, null)
  ins_le: AXIOM x ≤ y ⇒ ins(x, cons(y, l)) = cons(x, cons(y, l))
  ins_gt: AXIOM ¬ (x ≤ y) ⇒ ins(x, cons(y, l)) = cons(y, ins(x, l))

  isort: [lists → lists]
  isort_empty: AXIOM isort(null) = null
  isort_rec: AXIOM isort(cons(x, l)) = ins(x, isort(l))

 END insertion_sort
```

FIG. 6.4 – Théorie PVS du programme de tri par insertion

```
  del: [nat, lists → lists]
  del_empty: AXIOM del(x, null) = null
  del_eq: AXIOM x = y ⇒ del(x, cons(y, l)) = l
  del_diff: AXIOM x ≠ y ⇒ del(x, cons(y, l)) = cons(y, del(x, l))

  perm: [lists, lists → bool]
  perm_empty1: AXIOM perm(null, cons(x, l)) = FALSE
  perm_empty2: AXIOM perm(null, null) = TRUE
  perm_not: AXIOM ¬ (member(x, l2)) ⇒ perm(cons(x, l1), l2) = FALSE
  perm_rec: AXIOM
    member(x, l2) ⇒ perm(cons(x, l1), l2) = perm(l1, del(x, l2))

  ins_member: LEMMA ins(x, l1) = l2 ⇒ member(x, l2)
  del_ins: LEMMA del(x, ins(x, l)) = l

  isort_perm: THEOREM perm(l, isort(l))
```

FIG. 6.5 – Propriété de permutation de `ISort`

```
sorted: [lists → bool]
sorted_empty: AXIOM sorted(null) = TRUE
sorted_one: AXIOM sorted(cons(x, null)) = TRUE
sorted_le: AXIOM
  x ≤ y ⇒ sorted(cons(x, cons(y, l))) = sorted(cons(y, l))
sorted_gt: AXIOM ¬ (x ≤ y) ⇒ sorted(cons(x, cons(y, l))) = FALSE

lemma_sorted1 : LEMMA sorted(cons(x, l)) ⇒ sorted(l)
lemma_sorted2 : LEMMA
  sorted(cons(x, cons(y, l))) ⇒ x ≤ y ∧ sorted(cons(y, l))
lemma_sorted3 : LEMMA sorted(cons(x, cons(y, l))) ⇒ sorted(cons(x, l))
lemma_sorted4 : LEMMA sorted(cons(x, l)) ∧ member(y, l) ⇒ x ≤ y

ins_not_empty: LEMMA ∃ (y: nat, l2: lists): ins(x, l1) = cons(y, l2)
ins_member2 : LEMMA x ≠ y ∧ ins(x, l1) = cons(y, l2) ⇒ member(y, l1)
lemma_ins: LEMMA sorted(l) ⇒ sorted(ins(x, l))

isort_sorted: THEOREM sorted(isort(l))
```

FIG. 6.6 – Propriété de liste triée de `ISort`

lemma_sorted1 à *lemma_sorted4*. Bien que ces propriétés se déduisent des axiomes de *sorted*, nous aurions pu nous contenter de les ajouter à la définition du prédicat. Par exemple, le quatrième lemme énonce seulement que le premier élément d'une liste triée dans l'ordre croissant de ses éléments est aussi le plus petit des éléments de la liste.

Viennent ensuite trois nouveaux lemmes précisant des propriétés de la fonction *ins*. Parmi eux, *lemma_ins* est le plus important : il affirme que l'insertion par la fonction *ins* d'un élément dans une liste ordonnée produit une liste ordonnée. Le théorème final, *isort_sorted*, découle directement de ce dernier lemme.

6.4 Autres exemples

Nous présentons dans cette section deux exemples supplémentaires de programmes $\text{Sub}_{\mathcal{C}}$ axiomatisés par $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$. Les exemples choisis sont des algorithmes classiques mettant en œuvre des types de données structurées, à savoir les arbres et les graphes, plus complexes que ceux vus jusqu'à présent. Nous utilisons les listes du langage $\text{Sub}_{\mathcal{C}}$ pour coder ces nouveaux types de données. Afin de rendre les équations plus lisibles, les conditions des équations ont été partiellement évaluées et les composantes triviales éliminées.

6.4.1 Arbres binaires de recherche

La Figure 6.7(a) contient un programme $\text{Sub}_{\mathcal{C}}$ qui indique si un élément `e` donné appartient à un arbre binaire de recherche `tree` donné. La fonction `bs` retourne 1 si

l'élément `e` se trouve dans l'arbre `tree` et 0 sinon. Nous modélisons les nœuds d'un arbre binaire par une liste à trois éléments :

1. un entier pour la valeur du nœud ;
2. un nœud pour le fils gauche ;
3. un autre nœud pour le fils droit.

Un accesseur est défini pour chacun de ces éléments (fonctions `root`, `lc` et `rc`, respectivement). Les équations produites par l'axiomatisation du programme sont dans la Figure 6.7(b).

6.4.2 Parcours en profondeur d'abord

Enfin, nous terminons ce chapitre avec l'exemple du parcours en profondeur d'abord d'un graphe. Le programme $\text{Sub}_{\mathcal{C}}$ implantant cet algorithme est dans les Figures 6.8(a) et 6.9(a) (Pages 142 et 143). Les sommets du graphe sont simplement représentés par des entiers. Le graphe lui-même est défini par les listes d'adjacence de ses sommets. Toutes les listes d'adjacence sont regroupées dans une liste de listes telle que le $v^{\text{ième}}$ élément de cette liste correspond à la liste de tous les sommets connectés au sommet v.

La fonction `member` fait cette fois partie du programme et non pas de sa spécification comme dans l'exemple précédent du tri par insertion. La fonction renvoie 1 si un élément `e` donné est dans une liste `L` donnée, et 0 sinon. Le rôle de la fonction `adj` est de renvoyer la liste d'adjacence d'un sommet `v` donné dans la liste des listes d'adjacence `adjlist`.

Le point d'entrée du programme de parcours en profondeur d'abord est la fonction `dfs`. Cette fonction est appelée avec la liste de tous les sommets, une liste des sommets visités initialement vide et la liste des listes d'adjacence du graphe. Cette fonction lance un parcours en profondeur d'abord pour tous les sommets non déjà visités de la liste des sommets. Le parcours est effectué par des appels mutuellement récursifs à la fonction `depth` qui s'assure que tous les adjacents d'un sommet donné seront visités. La valeur de retour de ces deux fonctions est artificiellement introduite pour que le système génère effectivement leur sémantique, car les sous-programmes sans valeur de retour ne produisent pas d'équations dans $SOS\text{Sub}_{\mathcal{C}}^{\mathcal{R}}$.

Les équations produites pour le programme de parcours en profondeur d'abord sont données dans les Figures 6.8(b) et 6.9(b).

```
int root (list tree) { return element(tree); }

list lc (list tree) { return element(next(tree)); }

list rc (list tree) { return element(next(next(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;
}
```

(a) le programme de recherche dichotomique

$$\begin{aligned} \mathrm{root}(t) &= \mathrm{element}(t) \\ \mathrm{lc}(t) &= \mathrm{element}(\mathrm{next}(t)) \\ \mathrm{rc}(t) &= \mathrm{element}(\mathrm{next}(\mathrm{next}(t))) \end{aligned}$$

$$\begin{aligned} \mathrm{bs}(e, \mathrm{NULL}) &= 0 \\ \mathrm{bs}(e, e \cdot L) &= 1 \\ e < r \Rightarrow \mathrm{bs}(e, r \cdot L) &= \mathrm{bs}(e, \mathrm{lc}(r \cdot L)) \\ e > r \Rightarrow \mathrm{bs}(e, r \cdot L) &= \mathrm{bs}(e, \mathrm{rc}(r \cdot L)) \end{aligned}$$

(b) axiomatisation par $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ de `bs`

FIG. 6.7 – Axiomatisation du programme de recherche dichotomique

```
int depth (list vertices, list adjacents, list marked,
        list adjlist) {
  int ret, v;
  if (adjacents == NULL)
    ret = dfs(next(vertices), marked, adjlist);
  else {
    v = element(adjacents);
    if (member(v, marked) == 1)
      ret = depth(vertices, next(adjacents),
              marked, adjlist);
    else
      ret = dfs(add(v, vertices), marked, adjlist);
  }
  return ret;
}

int dfs (list vertices, list marked, list adjlist) {
  int ret, v;
  if (vertices == NULL) ret = 1;
  else {
    v = element(vertices);
    ret = depth(vertices, adj(v, adjlist),
            add(v, marked), adjlist);
  }
  return ret;
}
```

(a) les fonctions $\mathrm{Sub}_{\mathcal{C}}$ `depth` et `dfs`

$$\mathrm{depth}(v \cdot L, \mathrm{NULL}, M, A) = \mathrm{dfs}(L, M, A)$$
$$\mathrm{member}(v, M) = 1 \Rightarrow \mathrm{depth}(L, v \cdot L', M, A) = \mathrm{depth}(L, L', M, A)$$
$$\mathrm{member}(v, M) = 0 \Rightarrow \mathrm{depth}(L, v \cdot L', M, A) = \mathrm{dfs}(v \cdot L', M, A)$$

$$\mathrm{dfs}(\mathrm{NULL}, M, A) = 1$$
$$\mathrm{dfs}(v \cdot L, M, A) = \mathrm{depth}(v \cdot L, \mathrm{adj}(v, A), v \cdot M, A)$$

(b) axiomatisation par $SOS\mathrm{Sub}^{\mathcal{R}}_{\mathcal{C}}$ de `depth` et `dfs`

FIG. 6.8 – Axiomatisation du programme de parcours en profondeur d'abord (partie 1)

```
int member (int e, list L) {
  int ret;
  if (L == NULL)
    ret = 0;
  else if (e == element(L))
    ret = 1;
  else
    ret = member(e, next(L));
  return ret;
}

list adj (int v, list adjlist) {
  list ret;
  if (v == 1)
    ret = element(adjlist);
  else
    ret = adj(v−1, next(adjlist));
  return ret;
}
```

(a) les fonctions $\mathrm{Sub}_{\mathcal{C}}$ `member` et `adj`

$$\begin{aligned} \mathrm{member}(e, \mathrm{NULL}) &= 0 \\ \mathrm{member}(e, e \cdot L) &= 1 \\ e \neq c \Rightarrow \mathrm{member}(e, c \cdot L) &= \mathrm{member}(e, L) \end{aligned}$$

$$\begin{aligned} \mathrm{adj}(1, L' \cdot L) &= L' \\ \mathrm{adj}(n+1, L' \cdot L) &= \mathrm{adj}(n, L) \end{aligned}$$

(b) axiomatisation par $SOS\mathrm{Sub}^{\mathcal{R}}_{\mathcal{C}}$ de `member` et `adj`

FIG. 6.9 – Axiomatisation du programme de parcours en profondeur d'abord (partie 2)

Chapitre 7

Preuve de propriétés de programmes et $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$

Ce chapitre fait pendant au précédent en proposant l'exemple du tri par fusion pour illustrer les aspects spécifiques du système $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$. Nous présentons aussi une version « en place » de l'algorithme d'inversion de l'ordre des éléments dans une liste. Nous verrons à cette occasion, par comparaison avec le résultat obtenu pour la version avec duplication du chapitre précédent, que l'axiomatisation par $SOS\mathrm{Sub}_{\mathcal{C}}$ parvient à capturer le principe commun des deux versions de l'algorithme.

Avec $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$, l'axiomatisation des programmes est assujettie à la satisfaction par les programmes de *prérequis*. Il s'agit notamment de montrer avec les exemples développés ici que la classe de programmes définie par les prérequis est suffisamment riche pour contenir des instances intéressant la vérification de programme.

7.1 Tri par fusion

Nous reprenons l'exemple du programme de tri par fusion qui avait motivé, en Section 1.1.2, les extensions au système $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ que comporte le système $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$. Le code source du programme est celui de la Figure 1.6 (Page 19). Les équations de l'axiomatisation de ce programme par $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ sont données par la figure 1.9 (Page 23).

Notre ambition est de démontrer pour le programme `MergeSort` les mêmes propriétés que pour le tri par fusion :

- la liste renvoyée par le sous-programme `mergeSort` est une permutation de la liste qu'il prend en paramètre : $perm(l, mergesort(l)) = \text{TRUE}$;
- la liste renvoyée par le sous-programme `mergeSort` est une liste triée par ordre croissant des éléments : $sorted(mergesort(l)) = \text{TRUE}$.

Toutefois, avant de vérifier ces propriétés, nous devons nous assurer que l'axiomatisation du programme par $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ est valide. Nous commencerons donc par montrer que les prérequis de l'axiomatisation ne sont pas violés par le programme `mergeSort`.

De même que dans le chapitre précédent, nous ne développons pas la preuve de chaque lemme ou théorème démontré avec le système de preuve PVS (le détail est tout de même fourni pour référence en Annexe C.3). Ils sont en effet au nombre de quarante et un. Nous essayons plutôt d'esquisser le schéma des démonstrations des propriétés du programme. À cette fin, nous ne présenterons que les lemmes les plus importants.

7.1.1 Vérification des prérequis

Nous faisons dans cette section la démonstration que le programme `MergeSort` satisfait les prérequis de l'axiomatisation par $SOS\text{Sub}_{\mathcal{C}}^{ext}$. Nous citons les prérequis en suivant leur numérotation dans l'Annexe B.

Le programme ne comporte pas d'instructions d'itération, nous n'avons donc à nous préoccuper que des prérequis n° 1, 2, et 3. La preuve de ces prérequis nécessite que nous introduisions les deux lemmes suivants :

Lemme 7.1. *Soit L une liste d'entiers acyclique. Soient L_r la liste renvoyée par un appel* `split(L)` *et L_m la liste L après le même appel. L_r et L_m sont deux listes distinctes.*

Démonstration. Nous procédons par récurrence sur la longueur de la liste L. L'hypothèse de récurrence est que pour toute liste L' dont la longueur est inférieure à celle de L, L'_r et L'_m ne partagent aucun maillon. Nous continuons par analyse de cas pour la liste L. Par définition du type des listes :

- Soit $L =$ NULL. Alors, la valeur renvoyée par `split(L)` est NULL et le lemme est trivialement vrai.
- Soit $L = e_1 \cdot l_2$, avec e_1 un entier et l_2 une liste. À nouveau, par analyse de cas pour l_2 :
  - Soit $L = e_1 \cdot$ NULL. Alors, la valeur renvoyée par `split(L)` est NULL et le lemme est trivialement vrai.
  - Soit $L = e_1 \cdot e_2 \cdot l_3$, avec e_2 un entier et l_3 une liste. Cette situation correspond au dernier `else` du sous-programme `split`. Juste avant l'appel récursif à `split`, nous avons : $pSecondCell = e_2 \cdot l_3$ et $L = e_1 \cdot l_3$. L'appel est donc : `split(`l_3`)`. Après l'appel et l'affectation qui s'ensuit, nous avons : $pSecondCell = e_2 \cdot l_{3_r}$ et $L = e_1 \cdot l_{3_m}$. La longueur de la liste l_3 est strictement inférieure à celle de L par construction, donc l'hypothèse de récurrence s'applique et nous avons que l_{3_r} et l_{3_m} ne partagent aucun maillon. Par conséquent, *pSecondCell* et L sont aussi deux listes distinctes. La liste renvoyée par `split` est *pSecondCell*, donc, quand `split` termine, $L_r = pSecondCell$ et $L_m = L$ sont deux listes distinctes. □

Lemme 7.2. *Dans le cadre du programme* `MergeSort`*, le sous-programme* `merge` *est toujours appelé avec des listes distinctes en paramètre.*

Démonstration. Dans le cadre de la preuve de propriété du sous-programme `mergeSort`, les appels de `merge` sont tous connus. Nous pouvons donc vérifier que le lemme est vrai

pour chacun de ces appels. Il y a trois contextes d'appel, dont deux sont des appels récursifs. Pour les appels récursifs, nous supposons que les paramètres formels sont des listes distinctes et nous montrons alors que les appels suivants se font avec des listes distinctes. Nous montrons ensuite que le seul appel non récursif de `merge` est fait avec des listes distinctes.

- Dans le cas des appels récursifs de `merge`, nous avons posé comme hypothèse que `La` et `Lb` étaient distinctes. Par conséquent, au moment des appels récursifs, nous avons `next(La)` distincte de `Lb`, dans un cas, et `La` distincte de `next(Lb)`, dans l'autre cas.
- Le seul appel non récursif de `merge` est dans le sous-programme `mergeSort`. Par le Lemme 7.1, `secondList` et `L` sont distinctes après l'appel de `split` dans `mergeSort`, par suite, `La` et `Lb` sont distinctes au moment de l'appel de `merge`. □

Nous montrons maintenant la validité de l'axiomatisation du programme `MergeSort` :

Théorème 7.3. *Le programme* `MergeSort` *satisfait les prérequis de l'axiomatisation par* SOSSub_C^{ext}.

Démonstration. Nous montrons les trois premiers prérequis :

1. Seul le sous-programme `merge` est susceptible de violer le prérequis n° 1. En effet, les autres sous-programmes ne prennent qu'un seul paramètre. Le Lemme 7.2 nous assure cependant que ce ne sera jamais le cas, puisque les deux paramètres sont toujours des listes distinctes.
2. Nous détaillons le prérequis n° 2 pour chaque sous-programme :

   mergeSort Par le Lemme 7.1, les appels récursifs de `mergeSort` se font tous deux sur des listes sans alias.

   merge Par le Lemme 7.2, le cas des appels récursifs est traité. En revanche, lors de l'appel de `merge` dans `mergeSort`, si `La` et `Lb` sont distinctes, `L` et `La`, d'une part, et, `secondList` et `Lb`, d'autre part, peuvent partager des maillons. Cependant, il n'y a pas d'usage ambigu de ces alias car `mergeSort` termine immédiatement après l'appel de `merge`.

   split L'appel de `split` dans `mergeSort` vérifie le prérequis trivialement. En revanche, lors de l'appel récursif de `split`, `next(pSecondCell)` est un alias de `next(L)`. Cependant, cet alias porte sur la tête de la liste passée en paramètre de `split` et satisfait donc le prérequis n° 2.
3. Nous détaillons le prérequis n° 3 pour chaque sous-programme :

   mergeSort La liste renvoyée par le sous-programme `mergeSort` peut être en relation d'alias avec la liste passée en paramètre : il suffit de considérer le cas où `next(L) == NULL`. Cependant, il n'y a dans le programme `MergeSort` aucune utilisation ambiguë de ces alias car il n'y a aucun appel de `mergeSort`.

merge De même, des relations d'alias existent entre les paramètres et la liste renvoyée dans le sous-programme `merge`. Cependant, il n'y a aucune utilisation ambiguë de ces alias après les appels de `merge` dans le programme `MergeSort`.

split Le Lemme 7.1 correspond exactement à ce prérequis pour `split`. □

Puisque la validité de l'axiomatisation est vérifiée, nous pouvons maintenant faire la preuve de la correction du programme `MergeSort`.

7.1.2 Propriété de permutation

Nous commençons par présenter en Figure 7.1, la théorie PVS pour le programme `MergeSort`. Nous ne définissons pas de nouveau le prédicat de permutation *perm* qui est le même qu'à la Figure 6.5 du chapitre précédent.

L'idée centrale qui va permettre de prouver la correction du programme `MergeSort` est que le sous-programme `split` réalise une *partition* de la liste passée en paramètre. À cette fin, nous donnons la définition de la partition d'une liste, dans la Figure 7.2, sous la forme d'un prédicat $part(l_1, l_2, l)$ qui est vrai si et seulement si les listes l_1 et l_2 forment une partition de la liste l.

La liste passée en paramètre du sous-programme `split` est modifiée par effet de bord de l'appel de `split`. Nous prouvons que cette liste ainsi modifiée forme avec la liste renvoyée par `split` une partition de la liste initialement passée en paramètre (*split_lemma*). La preuve de ce lemme est faite par récurrence sur la longueur de la liste l, puis par cas sur le type des listes.

Nous montrons ensuite, avec *merge_lemma*, que l'opération de fusion réalisée par le sous-programme `merge` s'apparente à une concaténation de listes particulière. La preuve de ce lemme est faite par récurrence sur la somme des longueurs des listes l_1 et l_2, puis par cas sur la relation d'ordre qui existe entre les premiers éléments des deux listes.

Nous introduisons alors deux lemmes qui relient les notions de permutation et de partition : ce sont les lemmes *perm_part_append* et *perm_part*. Ils nous permettent, avec *split_lemma* et *merge_lemma*, de démontrer la propriété de permutation du sous-programme `mergeSort` : *mergesort_perm*. La preuve de ce théorème est faite par récurrence sur la longueur de la liste l.

7.1.3 Propriété de liste triée

Le prédicat *sorted* pour les listes triées a été défini à la Figure 6.6. La démonstration du théorème *mergesort_sorted*, en Figure 7.3 découle principalement du lemme *merge_sorted*. En effet, c'est dans le sous-programme `merge` que s'opère concrètement le tri par comparaison des éléments. La démonstration de cette propriété de `merge` se fait de nouveau par récurrence sur la somme des longueurs de listes l_1 et l_2.

```
merge_sort: THEORY
 BEGIN
  IMPORTING list

  lists: TYPE = list[nat]

  x, y: VAR nat
  l, l1, l2, l3, l4: VAR lists

  split: [lists → lists]
  split_empty: AXIOM split(null) = null
  split_un: AXIOM split(cons(x, null)) = null
  split_rec: AXIOM split(cons(x, cons(y, l))) = cons(y, split(l))

  splitl: [lists → lists]
  splitl_empty: AXIOM splitl(null) = null
  splitl_un: AXIOM splitl(cons(x, null)) = cons(x, null)
  splitl_rec: AXIOM splitl(cons(x, cons(y, l))) = cons(x, splitl(l))

  merge: [lists, lists → lists]
  merge_empty1: AXIOM merge(null, l) = l
  merge_empty2: AXIOM ¬ null?(l) ⇒ merge(l, null) = l
  merge_inf: AXIOM
    x ≤ y ⇒
     merge(cons(x, l1), cons(y, l2)) = cons(x, merge(l1, cons(y, l2)))
  merge_sup: AXIOM
    x > y ⇒
     merge(cons(x, l1), cons(y, l2)) = cons(y, merge(cons(x, l1), l2))

  mergesort: [lists → lists]
  mergesort_empty: AXIOM mergesort(null) = null
  mergesort_un: AXIOM mergesort(cons(x, null)) = cons(x, null)
  mergesort_rec: AXIOM
    ¬ l = null ⇒
     mergesort(cons(x, l)) =
      merge(mergesort(splitl(cons(x, l))), mergesort(split(cons(x, l))))

 END merge_sort
```

FIG. 7.1 – Théorie PVS du programme de tri par fusion

```
part: [lists, lists, lists → bool]
part_empty: AXIOM part(null, null, null) = TRUE
part_false1: AXIOM part(cons(x, l1), l2, null) = FALSE
part_false2: AXIOM part(l1, cons(x, l2), null) = FALSE
part_false3: AXIOM
  ¬ (member(x, l1)) ∧ ¬ (member(x, l2)) ⇒
   part(l1, l2, cons(x, l)) = FALSE
part_rec1: AXIOM
  member(x, l1) ⇒ part(l1, l2, cons(x, l)) = part(del(x, l1), l2, l)
part_rec2: AXIOM
  member(x, l2) ⇒ part(l1, l2, cons(x, l)) = part(l1, del(x, l2), l)

split_lemma: LEMMA part(splitl(l), split(l), l)

merge_lemma: LEMMA perm(merge(l1, l2), append(l1, l2))


perm_part_append: LEMMA part(l1, l2, l) ⇒ perm(l, append(l1, l2))
perm_part: LEMMA
  perm(l1, l2) ∧ perm(l3, l4) ∧ part(l1, l3, l) ⇒ part(l2, l4, l)


mergesort_perm: THEOREM perm(l, mergesort(l))
```

FIG. 7.2 – Propriété de permutation de `mergeSort`

```
merge_sorted: LEMMA sorted(l1) ∧ sorted(l2) ⇒ sorted(merge(l1, l2))

mergesort_sorted: THEOREM sorted(mergesort(l))
```

FIG. 7.3 – Propriété de liste triée de `mergeSort`

```
void inverserIter(list &L) {
  list pre, ptr;

  while(L != NULL) {
    ptr = L;
    L = next(L);
    ptr->next = pre;
    pre = ptr;
    ptr = NULL;
  }
  L = pre;
}
```

(a) sous-programme `inverserIter`

$$\begin{aligned}
\text{inverserIter}^r_L(L) &= \text{loop}^r_{pre}(L, \text{NULL}, \text{NULL}) \\
\text{loop}^r_{pre}(e \cdot L, ptr, pre) &= \text{loop}^r_{pre}(L, \text{NULL}, e \cdot pre) \\
\text{loop}^r_{pre}(\text{NULL}, ptr, pre) &= pre
\end{aligned}$$

$$\begin{aligned}
\text{inverserIter}_L(L) &= \text{loop}_L(L, \text{NULL}, \text{NULL}) \\
\text{loop}_L(e \cdot L, ptr, pre) &= e \cdot pre \\
\text{loop}_L(\text{NULL}, ptr, pre) &= \text{NULL}
\end{aligned}$$

(b) axiomatisation par $SOS\text{Sub}^{ext}_{\mathcal{C}}$ de `inverserIter`

FIG. 7.4 – Axiomatisation de l'inversion en place d'une liste

7.2 Inversion en place des éléments dans une liste

Nous présentons dans cette section un second exemple d'axiomatisation avec le système $SOS\text{Sub}^{ext}_{\mathcal{C}}$ pour un sous-programme d'inversion de l'ordre des éléments dans une liste. Contrairement à l'exemple développé dans la Section 6.2, le sous-programme de la Figure 7.4(a) inverse l'ordre des éléments d'une liste « en place », *i.e.*, sans dupliquer aucun maillon de la liste et avec une quantité finie de mémoire supplémentaire (en l'occurrence, deux variables locales). En effet, le sous-programme `inverserIter` procède uniquement par réarrangement des liens entre les maillons, et ce, sans récursivité.

7.2.1 Vérification des prérequis

Pour que l'axiomatisation par $SOS\text{Sub}^{ext}_{\mathcal{C}}$ du sous-programme `inverserIter` soit valide, nous devons montrer que le sous-programme satisfait les prérequis de l'Annexe B. Nous prenons pour hypothèse que la liste passée en paramètre du sous-programme est une liste d'entiers acyclique. La vérification des prérequis est beaucoup plus simple que pour le programme `MergeSort`.

Nous montrons tout d'abord un lemme qui nous permettra d'affirmer que l'instruction d'itération de `inverserIter` ne crée pas d'alias.

Lemme 7.4. *Si aucune relation d'alias n'existe avant le corps de la boucle du sous-programme* `inverserIter`*, alors aucune relation d'alias n'existe après une exécution du corps de cette boucle.*

Démonstration. Par hypothèse, il n'y a initialement aucune relation d'alias. Dans le corps de la boucle, après les deux premières affectations, `L` et `ptr->next` sont devenus des alias. Cependant, l'affectation qui suit (`ptr->next = pre`) supprime cette relation et en crée une nouvelle entre `ptr->next` et `pre`. Cette relation est à nouveau supprimée par l'affectation `pre = ptr`. À ce point de la boucle, la seule relation d'alias qui existe lie donc `pre` et `ptr`. Enfin, la dernière instruction de la boucle supprime toute relation d'alias entre les listes du sous-programme en affectant la liste vide à `ptr`. □

Théorème 7.5. *L'axiomatisation du sous-programme* `inverserIter` *par* SOS$Sub_{\mathcal{C}}^{ext}$ *est valide.*

Démonstration. Nous montrons que `inverserIter` vérifie les cinq prérequis de l'Annexe B :

1. Le prérequis n° 1 est trivialement vrai puisque le sous-programme `inverserIter` ne prend qu'un seul paramètre.

2. Le prérequis n° 2 est trivialement vrai puisque le programme à axiomatiser ne comporte aucun appel de `inverserIter`.

3. De nouveau, le prérequis n° 3 est trivialement vrai puisque `inverserIter` ne prend qu'un seul paramètre.

4. Avant l'instruction d'itération, aucune relation d'alias ne lie les variables de type liste du sous-programme : en effet, `pre` et `ptr` sont initialisées à `NULL`. Le prérequis n° 4 est donc trivialement vrai.

5. Nous avons vu pour le prérequis précédent qu'aucune relation d'alias n'existe avant l'instruction d'itération. Par le Lemme 7.4, nous déduisons donc que le prérequis n° 5 est vérifié. □

7.2.2 Équations

Les équations de l'axiomatisation du sous-programme `inverserIter` sont données dans la Figure 7.4(b). Deux fonctions sont définies : $\text{inverserIter}^r_L(L)$ et $\text{inverserIter}_L(L)$.

La fonction $\text{inverserIter}_L(L)$ renvoie la valeur de la liste qui était référencée par `L` avant l'appel de `inverserIter`. Les maillons de cette liste sont modifiés par un effet de bord de l'appel du sous-programme `inverserIter`.

La fonction $\text{inverserIter}^r_L(L)$, quant à elle, renvoie la valeur de la liste qui est référencée par `L` après l'appel de `inverserIter`. En effet, s'agissant d'un paramètre passé par référence, le paramètre effectif est modifié. Nous trouvons pour cette valeur des équations

très semblables à celles de la version avec duplication de maillons du programme d'inversion des éléments d'une liste (*cf.* Figure 6.2(b)). La propriété démontrée pour cette précédente version dans la Section 6.2 pourrait donc l'être tout aussi facilement pour la version en place.

Enfin, les fonctions loop^r_{pre} et loop_L font partie de la famille de fonctions définies par l'axiomatisation pour l'instruction d'itération. Ne figurent ici que les fonctions nécessaires à la définition équationnelle de `inverserIter`.

La comparaison des équations obtenues pour les versions avec duplication des maillons et en place met en relief l'abstraction des algorithmes que réalise l'axiomatisation par $SOS\text{Sub}_{\mathcal{C}}$. Ainsi, les principes de l'algorithme sont clairement exposés : une itération avec une variable jouant le rôle d'*accumulateur* dans lequel se construit le résultat. En revanche, les détails de l'implantation sont masqués. Par exemple, il ne serait pas possible de montrer à l'aide des équations produites par l'axiomatisation que l'inversion de la liste se fait effectivement en place et qu'il n'y a donc pas de duplication de maillons.

Conclusion

Le domaine de la vérification des programmes informatiques est le théâtre de l'affrontement de deux paradigmes : validation par jeux de tests et méthodes formelles y sont régulièrement opposées. Confrontés à cette alternative, les programmeurs, dans leur imposante majorité, optent pour les tests. Force est de constater qu'en dépit d'un récent essor, l'emploi des méthodes formelles dans le monde industriel est toujours confiné à des usages spécifiques et rares. Cette situation est principalement due au coût de l'intégration des méthodes formelles dans le métier d'une entreprise : méthodologie inadaptée aux contraintes du développement logiciel tel que pratiqué dans l'industrie, mise en œuvre difficile nécessitant une expertise, etc.

Pourtant, l'alternative est artificielle : les deux paradigmes ne se recouvrent pas. Les tests ont la prérogative de la *validation fonctionnelle* d'un système logiciel, notamment en s'appuyant sur un ensemble de cas d'utilisations typiques. En revanche, le privilège de garantir la *correction* d'un programme revient aux méthodes formelles. Dans l'idéal, l'un complète l'autre. La réalisation de cet objectif commande que les méthodes formelles se rapprochent de leurs utilisateurs potentiels.

La thèse défendue dans ce manuscrit est que la logique équationnelle est un support adéquat pour la vérification de la correction de programmes impératifs. La logique équationnelle, avec la clarté de sa sémantique et la simplicité de sa syntaxe, avec les nombreux algorithmes qui automatisent les raisonnements possibles dans cette logique, avec la quantité d'outils qui l'intègrent déjà, peut prétendre à vaincre les réticences des programmeurs et à surmonter les obstacles au déploiement d'une méthode formelle dans l'industrie. Encore faut-il que cette méthode fondée sur la logique équationnelle ne remette pas en cause des cycles de développement bien établis. Nous proposons, à cet effet, une approche de la vérification des programmes qui ne requiert aucune mise en œuvre particulière au cours du développement des programmes et qui s'applique directement au code source des programmes sans aucune annotation. Notre approche peut donc être utilisée dans le cadre de la maintenance d'applications existantes. De plus, nous nous sommes intéressé aux langages de programmation impératifs car, bien que notoirement plus difficiles à analyser statiquement, ils sont les plus répandus en pratique.

Dans notre approche, la preuve qu'un programme est correct au regard de ses spécifications se déroule en trois temps. Tout d'abord, la spécification du programme est formalisée par des propriétés dans une logique cohérente avec la logique équationnelle, *e.g.*, la logique du premier ordre. Ensuite, l'analyse automatique du code source du programme extrait la sémantique algébrique du programme, c'est-à-dire, un ensemble d'équations condition-

nelles qui définissent une théorie du programme dans la logique équationnelle. Dans un dernier temps, un des multiples systèmes de preuve disponibles pour la logique équationnelle est utilisé pour démontrer que les propriétés de la spécification sont des théorèmes dans la théorie du programme.

Synthèse des travaux accomplis

Notre contribution dans cette thèse a consisté à concevoir et développer un système, nommé $SOS\mathrm{Sub}_{\mathcal{C}}$, pour l'analyse automatique du code source de programmes impératifs, dans le but de formuler la sémantique des programmes dans la logique équationnelle. Cette analyse est appelée l'axiomatisation des programmes.

Nous avons défini un langage impératif pour servir de support à notre approche. Ce langage comporte tous les principaux attributs du paradigme impératif, *e.g.*, affectation, séquence, itération, conditionnelle, sous-programme.

Nous avons défini la sémantique du langage en décrivant l'interprétation des constructions de ce langage par des modifications dans un ensemble d'équations caractéristiques des programmes. La formulation automatique des équations de la sémantique des programmes repose essentiellement sur l'évaluation symbolique des programmes conformément à la sémantique du langage.

Nous avons développé deux versions du système $SOS\mathrm{Sub}_{\mathcal{C}}$ qui diffèrent par la classe des programmes à laquelle chaque version s'applique et par la méthode employée pour l'évaluation symbolique.

- $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ est une version du système dans laquelle le langage impératif, nommé $\mathrm{Sub}_{\mathcal{C}}$, ne comporte pas d'autre effet de bord que l'affectation. En outre, le type liste de $\mathrm{Sub}_{\mathcal{C}}$ correspond à un modèle fonctionnel des listes. La pierre angulaire de $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$ est un système de réécriture dont la fonction est double : il formalise la sémantique du langage et opère l'évaluation symbolique des programmes.

  Nous avons prouvé que le système de réécriture est terminant et confluent. Ces propriétés garantissent la terminaison de notre analyse des programmes ainsi que l'unicité de son résultat. De plus, la démonstration de ces propriétés a été menée automatiquement à l'aide d'un démonstrateur de théorèmes.

- $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ considère une extension du langage $\mathrm{Sub}_{\mathcal{C}}$, nommée $\mathrm{Sub}_{\mathcal{C}}^{ext}$, dans laquelle les listes sont mutables et le mode de passage des paramètres est par valeur ou par référence. L'introduction de ces concepts dans le langage oblige à considérer une plus grande variété d'effets de bord que dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$.

  La méthode proposée pour l'axiomatisation ne repose plus sur un système de réécriture. Il nous a en effet semblé que les avantages du formalisme de la réécriture n'étaient plus aussi évidents dans $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ que dans $SOS\mathrm{Sub}_{\mathcal{C}}^{\mathcal{R}}$, principalement du fait de la sémantique plus complexe du langage.

  Le traitement des listes se fait par des approximations de plus en plus fines de leur structure, selon les besoins de l'analyse, en suivant un schéma de nommage des éléments de la liste.

Bien que le langage dispose d'allocation dynamique de mémoire et d'un type des références sur des éléments de la mémoire, nous avons défini un ensemble de prérequis qui permettent l'axiomatisation automatique des programmes, sans avoir recours à un modèle de la mémoire dans la théorie du programme. Cette thèse est, à notre connaissance, la première à traiter la sémantique des programmes comportant des données structurées mutables dans la logique équationnelle, sans lui adjoindre un modèle de la mémoire.

Ces caractéristiques confèrent à notre analyse une qualité de « raisonnement local » sur les structures de données qui explique la tournure naturelle des équations formulées par notre système.

Enfin, nous avons réalisé un ensemble d'expérimentations sur machine, avec chacune des deux versions du système, pour valider fonctionnellement notre approche de la vérification des programmes. En effet, nous avons illustré, sur un ensemble de programmes non triviaux, les trois temps de la vérification suggérés par notre approche : nous prouvons, à l'aide de systèmes de preuve largement diffusés, des propriétés génériques de ces programmes. La preuve est réalisée à partir de l'axiomatisation de ces programmes par notre système.

Toutefois, l'approche que nous proposons connaît plusieurs limitations :

- En premier lieu, les propriétés de programmes qui peuvent être montrées avec notre approche concernent uniquement les relations qui existent entre les données en entrée et en sortie des programmes. Par exemple, il n'est pas possible de montrer des propriétés concernant l'utilisation de la mémoire en particulier, ou la consommation des ressources en général (si ce n'est une approximation du temps d'exécution). Effectivement, la sémantique que nous donnons à un programme est un ensemble de fonctions de transfert des données en entrée vers les données en sortie. Ceci masque en partie les détails des calculs qui conduisent des entrées aux sorties d'un programme.

- L'automatisation de l'axiomatisation avec $SOS\mathrm{Sub}_{\mathcal{C}}^{ext}$ est conditionnée par la vérification que les programmes respectent des prérequis sur la création et l'utilisation des alias. Trois facteurs contribuent à cette restriction sur la classe des programmes que nous pouvons traiter :

  1. Notre méthode permet de traiter des programmes incomplets, de façon à certifier, par exemple, des bibliothèques de sous-programmes. De ce fait, nous sommes dans l'ignorance des relations d'alias qui préexistent à l'appel d'un sous-programme.
  2. Quand bien même nous disposerions de programmes complets, des résultats d'indécidabilité de l'analyse statique des alias interdisent de connaître précisément toutes les relations d'alias qui existent dans un programme.
  3. Les programmes sont interprétés par des fonctions de transfert des valeurs en entrée et en sortie des programmes, sans modèle de la mémoire. Par conséquent,

un effet de bord sur un objet du programme doit être propagé à tous ses alias pour modifier leur valeur. Ce qui impliquerait de connaître tous les alias de l'objet.

En plus de restreindre la classe des programmes qui peut être axiomatisée par notre méthode, les prérequis ne peuvent pas toujours être vérifiés automatiquement dans le cas général, du fait de l'indécidabilité de l'analyse des alias. Ceci ne rend pas notre méthode caduque. En effet, sous la forme de ces prérequis ou sous une autre forme, toute méthode de vérification de programme en présence d'alias est confrontée au problème de démontrer des propriétés sur les alias du programme. Nous pensons que les prérequis énoncés correspondent à des pratiques de programmation raisonnables et que leur vérification est aisée dans la plupart des cas. Nous pensons en outre que les avantages de pouvoir raisonner ensuite dans la logique équationnelle surpassent les inconvénients qui résultent de devoir faire la preuve des prérequis par un autre moyen.

Perspectives

Les travaux décrits dans ce manuscrit ont montré qu'une riche classe de programmes impératifs pouvaient être axiomatisée dans la logique équationnelle, afin de vérifier des propriétés pertinentes de ces programmes. Nous avons montré que l'axiomatisation des programmes est automatique, moyennant la preuve, éventuellement manuelle, que les programmes respectent certains prérequis. Toutefois, plusieurs prolongements de ces travaux peuvent être envisagés.

L'analyse des alias effectuée au cours de l'axiomatisation est le principal frein à son automatisation complète. Elle est aussi la direction de recherche la plus riche en pistes à explorer. Par exemple, l'analyse que nous effectuons est intraprocédurale car la vision que nous avons des programmes est plutôt celle d'une bibliothèque de sous-programmes. Cependant, dans certains cas où les contextes d'appel d'un sous-programme sont connus, une analyse interprocédurale peut suffire pour vérifier automatiquement les prérequis. D'une manière générale, les prérequis peuvent être affinés afin d'éviter de devoir les vérifier pour un ensemble de cas d'usages des alias compatibles avec l'axiomatisation et identifiables statiquement.

Une autre direction de recherche consiste à intégrer les restrictions sur la formation des alias au niveau du langage lui-même. Il convient pour cela de limiter volontairement les possibilités de formation des alias dans le langage de programmation, *e.g.*, en agissant sur les constructions disponibles ou le typage des expressions, afin que l'analyse d'alias reste décidable. La difficulté de ce projet est évidemment de trouver le juste équilibre entre automatisation et expressivité. Nous pourrons à cette fin nous inspirer des travaux sur les langages de programmation concurrente, pour lesquels les problèmes d'alias incontrôlés sur les ressources d'un programme sont encore plus aigus.

Dans la direction opposée, nous pouvons étudier comment étendre notre approche à un langage plus riche. Notamment, il serait intéressant de proposer d'autres types de

données structurées, *e.g.*, les tableaux, d'inclure les listes cycliques ou encore de considérer la libération explicite de la mémoire.

Enfin, il serait intéressant d'accroître la portée de notre travail en montrant l'équivalence de la sémantique de notre langage avec celle du langage C, pour la partie en commun. En effet, le langage C est un standard de fait auprès des programmeurs et nous nous en sommes inspiré pour concevoir le langage $\text{Sub}_{\mathcal{C}}$. Notre approche serait ainsi plus proche des préoccupations des programmeurs.

Annexes

Annexe A

Règles du système de réécriture $SOS_{\mathcal{R}}$

Cette annexe contient toutes les règles du système de réécriture $SOS_{\mathcal{R}}$. La signification de ces règles a été discutée dans le cadre de la sémantique du langage $\text{Sub}_{\mathcal{C}}$ en Section 4.3.

Note A.1.

1. Les opérateurs des expressions du langage $\text{Sub}_{\mathcal{C}}$ (*e.g.*, `add`, `+`, etc.) n'apparaissent pas dans ces règles car ils ne sont pas interprétés par notre système au cours de l'axiomatisation des programmes. De plus, ces opérateurs sont en général prédéfinis dans les systèmes de preuve.

2. La syntaxe choisie pour la présentation des règles est celle du système RRL. Compte tenu des contraintes de cette syntaxe et dans le but d'une notation homogène, nous avons préfixé le nom des variables dans les règles par `V_` et le nom des constructeurs par `C_`. Tous les autres symboles représentent des fonctions si leur première lettre est une majuscule, ou des types sinon. Le type des symboles est donné à titre indicatif car il n'est pas pris en compte par le processus de réécriture. Dans les faits, $SOS_{\mathcal{R}}$ n'est pas typé.

```
;;;; types
;; stmt : les instructions
;; stmt_list : les listes d'instructions
;; var : les variables
;; var_list : les listes de variables
;; id : les identificateurs
;; exp : les expressions
;; cond : les expressions conditionnelles
;; nat : les entiers naturels
;; env : les environnements
;; env_elt : les éléments d'environnement

;; Constructeurs des listes d'instructions
[ C_L_Stmt : stmt_list, stmt →stmt_list ]
[ C_Empty_L_Stmt : stmt_list ]
```

*;; **Constructeurs des instructions***
[C_If : cond, stmt_list, stmt_list →stmt]
[C_Assign : id, exp →stmt]
[C_While : nat, cond, stmt_list →stmt]

*;; **Constructeurs des environnements***
[C_Env : env_elt, env →env]
[C_Empty_Env : env]
[C_Choice : env, env →env]
[C_Branch : cond, env →env]

*;; **Constructeurs des éléments d'environnement***
[C_Pair : id, exp →env_elt]
[C_While_Closure : nat, cond, env, var_list →env_elt]

*;; **Constructeurs des listes de variables***
[C_L_Var : var, var_list →var_list]
[C_Empty_L_Var : var_list]

*;; **Constructeurs des conditions***
[C_And : cond, cond →cond]
[C_Not : cond →cond]

*;; **Constructeurs des expressions***
[C_EA : var →exp]
[C_Loop : nat, var, env →exp]
[C_Subst : var, cond, exp →cond]

*;; **une expression est une expression conditionnelle***
[exp < cond]

*;; **une variable est une expression***
[var < exp]

*;; **une variable est un identificateur***
[var < id]

*;; **Opération d'évaluation d'une liste d'instructions dans un environnement***
[GE : stmt_list, env →env]

(1) *GE(C_L_Stmt(v_l_stmt, v_stmt), v_env) →*
Comp(v_stmt, GE(v_l_stmt, v_env))

(2) *GE(C_Empty_L_Stmt, v_env) →v_env*

*;; **Opération d'évaluation d'une instruction dans un environnement***
[Comp : stmt, env →env]

(3) *Comp(C_Assign(v_var, v_exp), C_Empty_Env) →*
C_Env(C_Pair(v_var, v_exp), C_Empty_Env)

(4) *Comp(C_Assign(v_var, v_exp), C_Env(v_pair, v_env)) →*
Update_Env(C_Pair(v_var, v_exp), C_Env(v_pair, v_env))

(5) *Comp(C_If(v_cond, v_l_stmt1, v_l_stmt2), C_Empty_Env) →*
C_Choice(C_Branch(v_cond, GE(v_l_stmt1, C_Empty_Env)),
C_Branch(C_Not(v_cond), GE(v_l_stmt2, C_Empty_Env)))

(6) *Comp(C_If(v_cond, v_l_stmt1, v_l_stmt2), C_Env(v_pair, v_env)) →*
C_Choice(C_Branch(Update_Cond(v_cond, C_Env(v_pair, v_env)),
GE(v_l_stmt1, C_Env(v_pair, v_env))),
C_Branch(C_Not(Update_Cond(v_cond, C_Env(v_pair, v_env))),
GE(v_l_stmt2, C_Env(v_pair, v_env))))

(7) *Comp(C_While(v_num, v_cond, v_l_stmt), C_Empty_Env) →*
C_Env(C_While_Closure(v_num, v_cond, GE(v_l_stmt, C_Empty_Env),
C_Empty_L_Var),
GL(v_num, GLOMV(GE(v_l_stmt, C_Empty_Env)),
C_Empty_Env))

(8) *Comp(C_While(v_num, v_cond, v_l_stmt), C_Env(v_pair, v_l_pair)) →*
C_Env(C_While_Closure(v_num, v_cond,
GE(v_l_stmt, GIE(C_Env(v_pair, v_l_pair))),
GLOV(C_Env(v_pair, v_l_pair))),
Merge_Env(GL(v_num,
GLOMV(GE(v_l_stmt,
GIE(C_Env(v_pair, v_l_pair)))),
C_Env(v_pair, v_l_pair)),
C_Env(v_pair, v_l_pair)))

(9) *Comp(stmt, C_Choice(v_exp1, v_exp2)) →*
C_Choice(Comp(stmt, v_exp1), Comp(stmt, v_exp2))

(10) *Comp(stmt, C_Branch(v_cond, v_env)) →*
C_Branch(v_cond, Comp(stmt, v_env))

;; Génération des appels aux fonctions de boucle
[GL : nat, var_list, env →env]

(11) *GL(v_num, C_L_Var(v_var, v_l_var), v_env) →*
C_Env(C_Pair(v_var, C_Loop(v_num, v_var, v_env)),
GL(v_num, v_l_var, v_env))

(12) *GL(v_num, C_Empty_L_Var, v_env) →C_Empty_Env*

;; Insertion et mise à jour d'une paire dans un environnement
[Update_Env : env_elt, env →env]

(13) *Update_Env(C_Pair(v_var, v_exp1),*
C_Env(C_Pair(v_var, v_exp2), v_env)) →
Update_Env(C_Pair(v_var, C_Subst(v_var, v_exp1, v_exp2)), v_env)

(14) *Update_Env(C_Pair(v_var1, v_exp1),*
C_Env(C_Pair(v_var2, v_exp2), v_env)) →
C_Env(C_Pair(v_var2, v_exp2),
Update_Env(C_Pair(v_var1,
C_Subst(v_var2, v_exp1, v_exp2)), v_env))
if *not equal(v_var1, v_var2)*

(15) *Update_Env(C_Pair(v_var, v_exp), C_Empty_Env) →*
C_Env(C_Pair(v_var, v_exp), C_Empty_Env)

(16) *Update_Env(C_Pair(v_var, v_exp1),*
C_Env(C_While_Closure(v_num, v_cond,
v_envc, v_l_var), v_env)) →
C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var),
Update_Env(C_Pair(v_var, v_exp1), v_env))

;; Mise à jour d'une condition dans un environnement
[Update_Cond : cond, env →cond]

(17) *Update_Cond(v_cond, C_Env(C_Pair(v_var, v_exp), v_env)) →*
Update_Cond(C_Subst(v_var, v_cond, v_exp), v_env)

(18) *Update_Cond(v_cond, C_Empty_Env) →v_cond*

(19) *Update_Cond(v_cond,*
C_Env(C_While_Closure(v_num, v_condc, v_envc, v_l_var), v_env)) →
Update_Cond(v_cond, v_env)

(20) *C_Branch(v_cond, C_Choice(v_env1, v_env2)) →*
C_Choice(C_Branch(v_cond, v_env1), C_Branch(v_cond, v_env2))

(21) *C_Branch(v_cond1, C_Branch(v_cond2, v_env)) →*
C_Branch(C_And(v_cond1, v_cond2), v_env)

(22) *C_And(C_And(v_cond1, v_cond2), v_cond3) →*
C_And(v_cond1, C_And(v_cond2, v_cond3))

;; Génération de l'environnement initial d'une fonction de boucle
[GIE : env →env]

(23) *GIE(C_Empty_Env) →C_Empty_Env*

(24) *GIE(C_Env(C_Pair(v_var, v_exp), v_env)) →*
C_Env(C_Pair(v_var, C_EA(v_var)), GIE(v_env))

(25) *GIE(C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var), v_env)) →*
GIE(v_env)

;; Génération de la liste des variables d'un environnement
[GLOV : env →var_list]

(26) *GLOV(C_Empty_Env) →C_Empty_L_Var*

(27) *GLOV(C_Env(C_Pair(v_var, v_exp), v_env)) →*
C_L_Var(v_var, GLOV(v_env))

(28) *GLOV(C_Env(C_While_Closure(v_num, v_cond,*
v_envc, v_l_var), v_env)) →
GLOV(v_env)

;; Génération de la liste des variables modifiées d'un environnement
[GLOMV : env →var_list]

(29) *GLOMV(C_Empty_Env) →C_Empty_L_Var*

(30) *GLOMV(C_Env(C_Pair(v_var, C_EA(v_var)), v_env)) →GLOMV(v_env)*

(31) *GLOMV(C_Env(C_Pair(v_var, v_exp), v_env)) →*
C_L_Var(v_var, GLOMV(v_env))
if *not equal(C_EA(v_var), v_exp)*

(32) *GLOMV(C_Env(C_While_Closure(v_num, v_cond,*
v_envc, v_l_var), v_env)) →
GLOMV(v_env)

(33) *GLOMV(C_Branch(v_cond, v_env)) →GLOMV(v_env)*

(34) *GLOMV(C_Choice(v_env1, v_env2)) →*
Merge_L_Var(GLOMV(v_env1), GLOMV(v_env2))

;; Fusion de deux environnements
[Merge_Env : env, env →env]

(35) *Merge_Env(C_Env(v_pair, v_l_pair), v_env) →*
Insert_Pair(v_pair, Merge_Env(v_l_pair, v_env))

(36) *Merge_Env(C_Empty_Env, v_env) →v_env*

;; Insertion sans mise à jour d'une paire dans un environnement
[Insert_Pair : env_elt, env →env]

(37) *Insert_Pair(C_Pair(v_var, v_exp),*
C_Env(C_While_Closure(v_num, v_cond,
v_envc, v_l_var), v_env)) →
C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var),
Insert_Pair(C_Pair(v_var, v_exp), v_env))

(38) *Insert_Pair(C_Pair(v_var, v_exp1),*
C_Env(C_Pair(v_var, v_exp2), v_env)) →
C_Env(C_Pair(v_var, v_exp1), v_env)

(39) *Insert_Pair(C_Pair(v_var1, v_exp1),*
C_Env(C_Pair(v_var2, v_exp2), v_env)) →
C_Env(C_Pair(v_var2, v_exp2),
Insert_Pair(C_Pair(v_var1, v_exp1), v_env))
if *not equal(v_var1, v_var2)*

(40) *Insert_Pair(C_Pair(v_var, v_exp), C_Empty_Env) →*
C_Env(C_Pair(v_var, v_exp), C_Empty_Env)

;; Fusion de deux listes de variables
[Merge_L_Var : var_list, var_list →var_list]

(41) *Merge_L_Var(C_L_Var(v_var, v_l_var1), v_l_var2) →*
Insert_Var(v_var, Merge_L_Var(v_l_var1, v_l_var2))

(42) *Merge_L_Var(C_Empty_L_Var, v_l_var) →v_l_var*

;; Insertion d'une variable dans une liste de variables
[Insert_Var : var, var_list →var_list]

(43) *Insert_Var(v_var, C_L_Var(v_var, v_l_var)) →C_L_Var(v_var, v_l_var)*

(44) *Insert_Var(v_var1, C_L_Var(v_var2, v_l_var)) →*
C_L_Var(v_var2, Insert_Var(v_var1, v_l_var))
if *not equal(v_var1, v_var2)*

(45) *Insert_Var(v_var, C_Empty_L_Var) →C_L_Var(v_var, C_Empty_L_Var)*

Annexe B

Prérequis de $SOS\text{Sub}_{\mathcal{C}}^{ext}$

Nous récapitulons dans cette annexe, les cinq prérequis que doivent vérifier les sous-programmes écrits dans le langage $\text{Sub}_{\mathcal{C}}^{ext}$ afin de garantir la validité de leur axiomatisation par $SOS\text{Sub}_{\mathcal{C}}^{ext}$.

L'origine de ces prérequis réside, d'une part, dans le fait que notre méthode ne restitue aucune information sur les objets manipulés par les programmes autre que leur valeur, et d'autre part, dans le fait que le langage $\text{Sub}_{\mathcal{C}}^{ext}$ est suffisamment expressif pour que la valeur de ces objets dépende de relations d'alias dont nous ne pouvons décider l'existence statiquement. Il est donc entendu que ces prérequis ne peuvent être vérifiés automatiquement dans le cas général, cependant, des méthodes d'analyse d'alias automatiques peuvent y parvenir dans des cas particuliers.

L'objectif de ces prérequis est de garantir que chaque fois que dans notre méthode nous réduisons le contenu informatif d'un objet du programme (*i.e.*, essentiellement les listes pour nous) à sa valeur, cette perte d'information n'entraîne pas d'imprécisions pour la suite de l'analyse. Le lieu de ces approximations est la fonction de transfert, des *valeurs en entrée* vers *les valeurs en sortie*, à laquelle nous assimilons les sous-programmes et les instructions d'itérations (boucles) du langage impératif source. Les prérequis nous assurent donc que, en entrée et en sortie des sous-programmes et des instructions d'itérations, les données du programme sont réductibles à leur valeur ou alors que la perte d'information qui en découle n'a pas d'incidence sur la suite de l'analyse.

Les cinq prérequis sur les programmes $\text{Sub}_{\mathcal{C}}^{ext}$ énoncés au cours du Chapitre 5 sont les suivants :

1. L'appel d'un sous-programme ne crée pas de relations d'alias entre ses paramètres formels.

2. Aucune relation d'alias ne porte sur les listes passées en paramètre d'un sous-programme, à l'exception éventuellement d'un alias sur la tête de la liste. Dans le cas contraire, les alias qui sont alors définis ne doivent pas être utilisés de façon ambiguë dans le sous-programme appelé et dans la suite du programme.

3. Aucune relation d'alias ne lie les paramètres d'un sous-programme entre eux, ou les paramètres et la valeur de retour, lorsque le sous-programme termine. Dans le cas

contraire, il faut montrer qu'après un appel au sous-programme les alias ne sont pas utilisés de façon ambiguë.

4. Aucune relation d'alias ne porte sur les listes modifiées par une instruction d'itération avant cette instruction. Dans le cas contraire, les alias ne doivent pas être utilisés de façon ambiguë dans l'instruction d'itération et dans la suite du sous-programme contenant l'instruction.

5. Les boucles ne créent aucune relation d'alias entre les listes du sous-programme qui les contient, telle que la relation continue à exister à la fin du corps de la boucle. Dans le cas contraire, les alias ne doivent pas être utilisés de façon ambiguë dans le corps de la boucle et dans le reste du sous-programme qui la contient.

Nous entendons par usage *ambigu*, l'application d'un effet de bord par l'intermédiaire d'un alias, y compris l'affectation, suivie de la lecture de la valeur d'un autre alias du même objet (ceci comprend l'application d'un opérateur de mutation).

Annexe C

Détails des preuves machine

Nous donnons dans cette annexe les preuves des lemmes et des théorèmes des Chapitres 6 et 7, lorsque ces preuves ont été conduites dans le système PVS. Le détail des preuves consiste en la trace des interactions avec le système. La syntaxe est celle du système PVS dans sa version 3.1 pour Linux. Enfin, nous ne fournissons pas les démonstrations des contraintes sur les types des variables que le système a lui-même générées et vérifiées.

C.1 Plus grand commun diviseur

lemma1

```
(""
 (skolem!)
 (case "b!1 = 0")
 (("1" (replace -1) (simplify) (propax))
  ("2"
   (use same_remainder)
   (("1" (simplify) (rewrite divides_reflexive) (simplify) (propax))
    ("2"
     (typepred "b!1")
     (expand ">=")
     (expand "<=")
     (prop)
     (("1" (rewrite >)) ("2" (rewrite -1 2)))))))))
```

th1

```
(""
 (skolem!)
 (flatten)
 (case "b!1 = 0")
 (("1" (replace -1) (simplify) (propax))
  ("2"
   (rewrite pgcd3)
   (rewrite pgcd3)
   (rewrite pgcd1)
   (rewrite pgcd1 2 ("b" "b!1"))
   (rewrite lemma1))))
```

lemma2

```
(""
 (induct b)
 (("1" (skolem!) (rewrite pgcd2) (rewrite pgcd2))
  ("2"
   (skolem!)
   (prop)
   (skolem!)
   (rewrite pgcd1)
   (rewrite pgcd1 1 ("a" "a!1")))))
```

th2

```
(""
 (skolem!)
 (case "b!1=0")
 (("1"
   (replace -1)
   (rewrite pgcd3)
   (rewrite pgcd2)
   (case "a!1=0")
   (("1" (replace -1) (rewrite pgcd3) (rewrite pgcd2))
    ("2"
     (rewrite pgcd3)
     (rewrite pgcd1)
     (rewrite rem_zero)
     (rewrite pgcd2))))
  ("2"
   (case "a!1 = 0")
   (("1"
     (replace -1)
     (rewrite pgcd3)
     (rewrite pgcd3)
     (rewrite pgcd1)
     (rewrite rem_zero)
     (rewrite pgcd2)
     (rewrite pgcd2))
    ("2"
     (case "a!1 < b!1")
     (("1"
       (rewrite pgcd3)
       (rewrite pgcd1)
       (rewrite rem_mod2)
       (rewrite pgcd3)
       (rewrite lemma2))
      ("2"
       (rewrite pgcd3)
       (rewrite pgcd3)
       (case "a!1 = b!1")
       (("1" (replace -1) (propax))
        ("2"
         (case "a!1 > b!1")
         (("1"
           (rewrite pgcd1 5 ("a" "b!1"))
           (rewrite rem_mod2)
           (rewrite lemma2 5 ("r" "a!1")))
          ("2" (grind)))))))))))
```

th3

```
(""
 (skolem!)
 (case "a!1=0")
 (("1" (replace -1) (rewrite pgcd3) (rewrite pgcd2))
  ("2"
```

```
   (rewrite pgcd3)
   (rewrite pgcd1)
   (rewrite rem_self)
   (rewrite pgcd2))))
```

C.2 Tri par insertion

ins_member

```
(""
 (induct "l1")
 (("1"
   (skolem 1 ("l2" "x"))
   (rewrite "ins_empty")
   (flatten)
   (replace -1 (1) rl)
   (grind))
  ("2"
   (skolem 1 ("y" "l1"))
   (flatten)
   (skolem 1 ("l2" "x"))
   (flatten)
   (case "x <= y")
   (("1" (rewrite "ins_le") (replace -3 (1) rl) (grind))
    ("2"
     (rewrite "ins_gt")
     (inst -1 "ins(x, l1)" "x")
     (replace -2 (2) rl)
     (grind))))))
```

del_ins

```
(""
 (induct "l")
 (("1" (skolem 1 ("x")) (rewrite "ins_empty") (rewrite "del_eq"))
  ("2"
   (skolem 1 ("y" "l2"))
   (flatten)
   (skolem 1 ("x"))
   (case "x <= y")
   (("1" (rewrite "ins_le") (rewrite "del_eq"))
    ("2" (rewrite "ins_gt") (rewrite "del_diff") (grind))))))
```

isort_perm

```
(""
 (induct "l")
 (("1" (rewrite "isort_empty") (rewrite "perm_empty2"))
  ("2"
   (skolem 1 ("y" "l2"))
   (flatten)
   (rewrite "isort_rec")
   (rewrite "perm_rec")
   (("1" (rewrite "del_ins")) ("2" (use "ins_member"))))))
```

lemma_sorted1

```
(""
 (induct "l")
 (("1" (grind) (rewrite "sorted_empty"))
  ("2"
   (skolem!)
```

```
   (grind)
   (use "excluded_middle" ("A" "x!1 <= cons1_var!1"))
   (split -1)
   (("1" (rewrite "sorted_le")) ("2" (rewrite "sorted_gt"))))))
```

lemma_sorted2

```
(""
 (induct "l")
 (("1"
   (skolem 1 ("x" "y"))
   (flatten)
   (use "excluded_middle" ("A" "x <= y"))
   (split -1)
   (("1" (rewrite "sorted_le") (assert)) ("2" (rewrite "sorted_gt"))))
  ("2"
   (skolem!)
   (flatten)
   (skolem!)
   (flatten)
   (use "excluded_middle" ("A" "x!1 <= y!1"))
   (split -1)
   (("1" (rewrite "sorted_le") (prop)) ("2" (rewrite "sorted_gt"))))))
```

lemma_sorted3

```
(""
 (induct "l")
 (("1" (skolem!) (rewrite "sorted_one"))
  ("2"
   (skolem!)
   (flatten)
   (skolem!)
   (flatten)
   (case "x!1 <= cons1_var!1")
   (("1"
     (rewrite "sorted_le" 1)
     (use "lemma_sorted1")
     (assert)
     (use "lemma_sorted1")
     (assert))
    ("2"
     (use "lemma_sorted2")
     (prop)
     (use "lemma_sorted2")
     (assert))))))
```

lemma_sorted4

```
(""
 (induct "l")
 (("1" (grind))
  ("2"
   (skolem!)
   (flatten)
   (skolem!)
   (flatten)
   (use "excluded_middle" ("A" "y!1 = cons1_var!1"))
   (split -1)
   (("1" (use "lemma_sorted2") (assert))
    ("2"
     (rewrite "member")
     (split -3)
     (("1" (propax))
      ("2" (use "lemma_sorted3") (assert) (inst? -3) (assert))))))))
```

ins_not_empty

```
(""
 (skolem 1 (_ "x"))
 (induct "l1")
 (("1" (rewrite "ins_empty") (instantiate 1 ("x" "null")) (propax))
  ("2"
   (skolem 1 ("z" "l3"))
   (flatten)
   (case "x <= z")
   (("1"
     (rewrite "ins_le")
     (instantiate 1 ("x" "cons(z, l3)"))
     (propax))
    ("2"
     (rewrite "ins_gt")
     (skolem -1 ("n" "l"))
     (replace -1)
     (instantiate 2 ("z" "cons(n, l)"))
     (propax))))))
```

ins_member2

```
(""
 (skolem 1 (_ "l2" "x" "y"))
 (induct "l1")
 (("1" (flatten) (rewrite "ins_empty") (decompose-equality))
  ("2"
   (skolem 1 ("z" "l3"))
   (flatten)
   (case "x <= z")
   (("1" (rewrite "ins_le") (decompose-equality))
    ("2" (rewrite "ins_gt") (decompose-equality) (grind))))))
```

lemma_ins

```
(""
 (auto-rewrite ("ins_empty"
                "ins_le"
                "ins_gt"
                "sorted_empty"
                "sorted_one"
                "sorted_le"
                "sorted_gt"))
 (induct "l")
 (("1" (grind))
  ("2"
   (skolem 1 ("y" "l2"))
   (flatten)
   (skolem 1 "x")
   (flatten)
   (case "x <= y")
   (("1" (grind))
    ("2"
     (assert)
     (inst -1 "x")
     (use "lemma_sorted1")
     (assert)
     (case "exists (z: nat, l3: lists): ins(x, l2) = cons(z, l3)")
     (("1"
       (skolem -1 ("z" "l3"))
       (replace -1)
       (case "z=x")
       (("1" (grind))
        ("2"
         (use "ins_member2")
```

$append\_lemma$: LEMMA $cons(x,\ append(l_1,\ l_2)) = append(cons(x,\ l_1),\ l_2)$
$member\_append$: LEMMA $member(x,\ l_1) \Rightarrow member(x,\ append(l_1,\ l_2))$

FIG. C.1 – Lemmes de *append*

```
        (assert)
        (use "lemma_sorted4" ("x" "y" "y" "z" "l" "l2"))
        (assert)
        (rewrite "sorted_le"))))
     ("2" (use "ins_not_empty")))))))))
```

isort_sorted

```
(""
 (auto-rewrite ("ins_empty"
                "ins_le"
                "ins_gt"
                "sorted_empty"
                "sorted_one"
                "sorted_le"
                "sorted_gt"
                "isort_empty"
                "isort_rec"))
 (induct "l")
 (("1" (grind))
  ("2" (skolem 1 ("x" "l")) (grind) (rewrite "lemma_ins"))))
```

C.3 Tri par fusion

Nous partageons cette annexe en groupant les lemmes selon la fonction à laquelle ils se rapportent. Nous donnons aussi le contenu des lemmes qui ne sont pas déjà présentés dans la Section 7.1. Les prédicats *del*, *perm* et *sorted* ont la même définition que pour le tri par insertion de la Section 6.3. Les fonctions *append*, *member* et *length* sont prédéfinies dans le fichier de prélude du système PVS.

C.3.1 append

Les preuves qui suivent se rapportent aux lemmes de la Figure C.1.

append_lemma

```
("" (grind))
```

member_append

```
(""
 (induct l1)
 (("1" (grind))
  ("2"
   (skosimp)
   (skosimp)
   (case "x!1=cons1_var!1")
   (("1" (expand append) (grind)) ("2" (expand member -2) (grind))))))
```

del_lemma: LEMMA $del(x,\ del(y,\ l)) = del(y,\ del(x,\ l))$
del_length: LEMMA $length(del(x,\ l)) \leq length(l)$
del_member: LEMMA $member(x,\ l) \wedge x \neq y \Rightarrow member(x,\ del(y,\ l))$
$del\_member2$: LEMMA $member(x,\ del(y,\ l)) \Rightarrow member(x,\ l)$
del_append: LEMMA
$member(x,\ l_1) \Rightarrow del(x,\ append(l_1,\ l_2)) = append(del(x,\ l_1),\ l_2)$

FIG. C.2 – Lemmes de *del*

C.3.2 del

Les preuves qui suivent se rapportent aux lemmes de la Figure C.2.

del_lemma

```
("" 
 (induct l)
 (("1"
   (skosimp)
   (rewrite del_empty)
   (rewrite del_empty)
   (rewrite del_empty))
  ("2"
   (skosimp)
   (skosimp)
   (case "y!1 = cons1_var!1")
   (("1"
     (case "x!1 = y!1")
     (("1" (replace -1) (propax))
      ("2"
       (replace -1)
       (rewrite del_eq)
       (rewrite del_diff)
       (rewrite del_eq))))
    ("2"
     (case "x!1 = cons1_var!1")
     (("1"
       (replace -1)
       (rewrite del_diff)
       (rewrite del_eq)
       (rewrite del_eq))
      ("2"
       (rewrite del_diff)
       (rewrite del_diff)
       (rewrite del_diff)
       (rewrite del_diff)
       (grind))))))))
```

del_length

```
("" 
 (induct l)
 (("1" (skosimp) (rewrite del_empty) (grind))
  ("2"
   (skosimp)
   (skosimp)
   (case "x!1 = cons1_var!1")
   (("1" (replace -1) (rewrite del_eq) (grind))
    ("2" (rewrite del_diff) (grind))))))
```

del_member

```
(""
 (induct l)
 (("1" (skosimp) (grind))
  ("2"
   (skosimp)
   (skosimp)
   (case "x!1 = cons1_var!1")
   (("1" (replace -1) (rewrite del_diff) (grind))
    ("2"
     (expand member -2)
     (prop)
     (case "y!1 = cons1_var!1")
     (("1" (replace -1) (rewrite del_eq))
      ("2" (rewrite del_diff) (grind))))))))
```

del_member2

```
(""
 (induct l)
 (("1" (skosimp) (rewrite del_empty))
  ("2"
   (skosimp)
   (skosimp)
   (case "y!1=cons1_var!1")
   (("1"
     (replace -1)
     (rewrite del_eq)
     (case "x!1 = cons1_var!1")
     (("1" (grind)) ("2" (grind))))
    ("2"
     (rewrite del_diff)
     (case "x!1 = cons1_var!1")
     (("1" (grind)) ("2" (expand member -2) (grind)))))))))
```

del_append

```
(""
 (induct l1)
 (("1" (grind))
  ("2"
   (skosimp)
   (skosimp)
   (case "x!1=cons1_var!1")
   (("1"
     (replace -1)
     (rewrite del_eq)
     (expand append 1 1)
     (rewrite del_eq))
    ("2"
     (rewrite del_diff)
     (expand append 2 2)
     (expand append 2 1)
     (rewrite del_diff)
     (apply-extensionality 2)
     (inst? -1)
     (grind))))))
```

C.3.3 perm

Les preuves qui suivent se rapportent aux lemmes de la Figure C.3.

perm_reflexive: LEMMA $perm(l,\ l) = \text{TRUE}$
perm_null: LEMMA $perm(null,\ l) \Rightarrow l = null$
perm_lemma1: LEMMA $perm(l_1,\ l_2) = perm(cons(x,\ l_1),\ cons(x,\ l_2))$
perm_lemma3: LEMMA $member(x,\ l_1) \wedge perm(l_1,\ l_2) \Rightarrow member(x,\ l_2)$
perm_del: LEMMA $perm(l_1,\ l_2) \Rightarrow perm(del(x,\ l_1),\ del(x,\ l_2))$
perm_transitivity: LEMMA $perm(l,\ l_1) \wedge perm(l_1,\ l_2) \Rightarrow perm(l,\ l_2)$
perm_del2: LEMMA
$member(x,\ l_1) \Rightarrow perm(del(x,\ l_1),\ l_2) = perm(l_1,\ cons(x,\ l_2))$
perm_commut: LEMMA $perm(l_1,\ l_2) \Rightarrow perm(l_2,\ l_1)$
perm_commutativity: LEMMA $perm(l_1,\ l_2) = perm(l_2,\ l_1)$
perm_lemma2: LEMMA $member(x,\ l) \Rightarrow perm(cons(x,\ del(x,\ l)),\ l)$
perm_append: LEMMA $perm(l,\ l_1) \Rightarrow perm(append(l,\ l_2),\ append(l_1,\ l_2))$
perm_append1: LEMMA
$perm(append(cons(x,\ l_1),\ l_2),\ append(l_1,\ cons(x,\ l_2)))$
perm_append2: LEMMA $perm(append(l_1,\ l_2),\ append(l_2,\ l_1))$

FIG. C.3 – Lemmes de *perm*

perm_reflexive

```
(""
 (induct l)
 (("1" (rewrite perm_empty2))
  ("2"
   (skosimp)
   (rewrite perm_rec)
   (("1" (rewrite del_eq)) ("2" (grind))))))
```

perm_null

```
(""
 (induct l)
 (skosimp)
 (prop)
 (("1" (replace -1) (rewrite perm_empty1)) ("2" (rewrite perm_empty1))))
```

perm_lemma1

```
(""
 (skolem!)
 (flatten)
 (rewrite perm_rec)
 (("1" (rewrite del_eq)) ("2" (grind))))
```

perm_lemma3

```
(""
 (induct l1)
 (("1" (skosimp) (use perm_null) (grind))
  ("2"
   (skosimp)
   (induct l2)
   (("1" (skosimp) (rewrite perm_not -2) (grind))
    ("2"
     (skosimp)
```

```
    (skosimp)
    (case "x!1 = cons1_var!1")
    (("1" (replace -1 -4 :dir rl) (rewrite perm_not -4))
     ("2"
      (expand member -2)
      (prop)
      (copy -3)
      (hide -1)
      (rewrite perm_rec -3)
      (("1"
        (inst -4 "del(cons1_var!1, cons(cons1_var!2, cons2_var!2))"
         "x!1")
        (prop)
        (case "cons1_var!1 = cons1_var!2")
        (("1" (replace -1) (rewrite del_eq) (expand member) (grind))
         ("2"
          (rewrite del_diff)
          (case "x!1 = cons1_var!2")
          (("1" (replace -1) (expand member) (propax))
           ("2"
            (expand member)
            (split)
            (("1" (propax))
             ("2" (flatten) (use del_member2) (grind))))))))
       ("2" (rewrite perm_not -3)))))))))
```

perm_del

```
(""
 (measure-induct+ "length(l1) + length(l2)" ("l1" "l2"))
 (skosimp)
 (case "null?(x!1) OR cons?(x!1)")
 (("1"
   (split)
   (("1"
     (case "x!1 = null")
     (("1"
       (replace -1)
       (use perm_null ("l" "x!2"))
       (prop)
       (replace -1)
       (rewrite del_empty))
      ("2" (grind))))
    ("2"
     (case "(exists (x:nat),(l:lists):x!1=cons(x,l))")
     (("1"
       (skosimp)
       (replace -1)
       (case "null?(x!2) OR cons?(x!2)")
       (("1"
         (split)
         (("1"
           (case "x!2 = null")
           (("1" (replace -1) (rewrite perm_not -6) (grind))
            ("2" (grind))))
          ("2"
           (case "(exists (x:nat),(l:lists):x!2=cons(x,l))")
           (("1"
             (skosimp)
             (replace -1)
             (hide -1 -2 -3 -4)
             (case "x!3 = x!4")
             (("1"
               (replace -1)
               (rewrite del_eq)
               (case "x!4 = x!5")
               (("1"
```

```
          (replace -1)
          (rewrite del_eq)
          (use perm_lemma1)
          (grind))
         ("2"
          (rewrite del_diff)
          (rewrite perm_rec -3)
          (("1" (rewrite del_diff -3))
           ("2" (rewrite perm_not -3))))))
       ("2"
        (case "x!3 = x!5")
        (("1"
          (replace -1)
          (rewrite del_eq 2 ("y" "x!5"))
          (rewrite del_diff)
          (rewrite perm_rec 2)
          (("1"
            (rewrite perm_rec -3)
            (("1"
              (rewrite del_diff -3)
              (inst -2 "l!1" "cons(x!5, del(x!4, l!2))" "x!5")
              (prop)
              (("1" (rewrite del_eq -1))
               ("2" (grind) (use del_length) (grind))))
             ("2" (rewrite perm_not -3))))
           ("2" (rewrite perm_not -3) (grind))))
         ("2"
          (rewrite del_diff)
          (rewrite del_diff)
          (copy -2)
          (hide -1)
          (rewrite perm_rec -2)
          (("1"
            (rewrite perm_rec 3)
            (("1"
              (case "x!4 = x!5")
              (("1"
                (replace -1)
                (rewrite del_eq)
                (rewrite del_eq)
                (inst -2 "l!1" "l!2" "x!3")
                (grind))
               ("2"
                (rewrite del_diff)
                (rewrite del_diff)
                (inst -1 "l!1" "cons(x!5, del(x!4, l!2))" "x!3")
                (grind)
                (("1" (rewrite del_diff) (rewrite del_lemma))
                 ("2" (use del_length) (grind))))))
             ("2"
              (rewrite perm_rec -2 :dir rl)
              (("1"
                (rewrite perm_not -2)
                (expand member)
                (split)
                (("1" (prop))
                 ("2" (flatten) (use del_member) (grind))))
               ("2" (reveal -1) (rewrite perm_not -1))))))
           ("2" (rewrite perm_not -2))))))))
    ("2"
     (inst 1 "car(x!2)" "cdr(x!2)")
     (("1"
       (apply-extensionality)
       (("1" (typepred "x!2") (grind))
        ("2" (typepred "x!2") (grind))))
      ("2" (typepred "x!2") (grind))
      ("3" (typepred "x!2") (grind))))))))
("2" (grind))))
```

```
     ("2"
      (inst 1 "car(x!1)" "cdr(x!1)")
      (("1"
        (apply-extensionality)
        (("1" (typepred x!1) (grind)) ("2" (typepred x!1) (grind))))
       ("2" (typepred x!1) (grind)) ("3" (typepred x!1) (grind))))))))
  ("2" (grind))))
```

perm_transitivity

```
(""
 (induct l)
 (("1" (skosimp) (use perm_null) (prop) (replace -1) (propax))
  ("2"
   (skosimp)
   (induct l1)
   (("1" (skosimp) (use perm_null) (prop) (replace -1) (propax))
    ("2"
     (skosimp)
     (induct l2)
     (("1" (flatten) (rewrite perm_not -2) (grind))
      ("2"
       (skosimp)
       (case "member(cons1_var!1, cons(cons1_var!2, cons2_var!2))")
       (("1"
         (hide -2)
         (use perm_lemma3
              ("x"
               "cons1_var!1"
               "l1"
               "cons(cons1_var!2, cons2_var!2)"
               "l2"
               "cons(cons1_var!3, cons2_var!3)"))
         (prop)
         (rewrite perm_rec 1)
         (rewrite perm_rec -3)
         (use perm_del
              ("x"
               "cons1_var!1"
               "l1"
               "cons(cons1_var!2, cons2_var!2)"
               "l2"
               "cons(cons1_var!3, cons2_var!3)"))
         (prop)
         (inst -7 "del(cons1_var!1, cons(cons1_var!2, cons2_var!2))"
          "del(cons1_var!1, cons(cons1_var!3, cons2_var!3))")
         (grind))
        ("2" (rewrite perm_not -2)))))))))))
```

perm_del2

```
(""
 (induct l1)
 (("1" (grind))
  ("2"
   (skosimp)
   (skosimp)
   (case "x!1=cons1_var!1")
   (("1"
     (replace -1)
     (rewrite del_eq)
     (rewrite perm_rec)
     (("1" (rewrite del_eq)) ("2" (expand member 1) (propax))))
    ("2"
     (expand member -2)
     (split)
```

```
    (("1" (propax))
     ("2"
      (inst -2 "del(cons1_var!1, l2!1)" "x!1")
      (prop)
      (rewrite del_diff 2)
      (iff)
      (split)
      (("1"
        (flatten)
        (lemma perm_del)
        (inst -1 "cons(cons1_var!1, del(x!1, cons2_var!1))" "l2!1"
         "cons1_var!1")
        (prop)
        (("1"
          (rewrite perm_rec 1)
          (("1" (rewrite del_diff 1))
           ("2" (rewrite perm_not -4) (expand member 1) (propax))))
         ("2" (copy -2) (rewrite perm_rec -3) (rewrite perm_not -1))))
       ("2"
        (flatten)
        (copy -1)
        (hide -1)
        (rewrite perm_rec -1)
        (("1"
          (prop)
          (("1"
            (rewrite perm_rec 1)
            (reveal -1)
            (rewrite perm_not -1)
            (expand member -1)
            (propax))
           ("2" (rewrite del_diff -1))))
         ("2" (rewrite perm_not -1)))))))))))
```

perm_commut

```
(""
 (induct l1)
 (("1"
   (induct l2)
   (("1" (grind))
    ("2"
     (skosimp)
     (rewrite perm_not 1)
     (("1" (rewrite perm_empty1)) ("2" (grind))))))
  ("2"
   (skosimp)
   (skosimp)
   (copy -2)
   (hide -1)
   (rewrite perm_rec -2)
   (("1"
     (inst? -1)
     (prop)
     (lemma perm_del2)
     (inst -1 "l2!1" "cons2_var!1" "cons1_var!1")
     (grind)
     (reveal -3)
     (rewrite perm_not -1))
    ("2" (rewrite perm_not -2))))))
```

perm_commutativity

```
(""
 (induct l1)
 (("1"
```

```
   (induct l2)
   (skosimp)
   (rewrite perm_empty1)
   (rewrite perm_not)
   (grind))
  ("2"
   (skosimp)
   (skosimp)
   (iff)
   (split)
   (("1"
     (flatten)
     (lemma perm_commut)
     (inst -1 "cons(cons1_var!1, cons2_var!1)" "l2!1")
     (grind))
    ("2"
     (flatten)
     (lemma perm_commut)
     (inst -1 "l2!1" "cons(cons1_var!1, cons2_var!1)")
     (grind))))))
```

perm_lemma2

```
(""
 (induct l)
 (("1" (grind))
  ("2"
   (skosimp)
   (skosimp)
   (case "x!1 = cons1_var!1")
   (("1" (replace -1) (rewrite del_eq) (rewrite perm_reflexive))
    ("2"
     (expand member -2)
     (prop)
     (inst? -2)
     (prop)
     (rewrite del_diff)
     (use perm_lemma1 ("x" "cons1_var!1"))
     (replace -1 -2)
     (case "perm(cons(cons1_var!1, cons(x!1, del(x!1, cons2_var!1))),
                 cons(x!1, cons(cons1_var!1, del(x!1, cons2_var!1))))")
     (("1"
       (rewrite perm_commutativity -1)
       (use perm_transitivity
            ("l"
             "cons(x!1, cons(cons1_var!1, del(x!1, cons2_var!1)))"
             "l1"
             "cons(cons1_var!1, cons(x!1, del(x!1, cons2_var!1)))"
             "l2"
             "cons(cons1_var!1, cons2_var!1)"))
       (prop))
      ("2"
       (rewrite perm_rec 1)
       (("1"
         (rewrite del_diff 1)
         (rewrite del_eq 1)
         (rewrite perm_reflexive 1))
        ("2" (grind)))))))))))
```

perm_append

```
(""
 (induct l)
 (("1"
   (induct l1)
   (("1" (skosimp) (expand append) (rewrite perm_reflexive 1))
```

```
   ("2" (skosimp) (skosimp) (rewrite perm_empty1 -2))))
 ("2"
  (skosimp)
  (skosimp)
  (expand append 1 1)
  (copy -2)
  (hide -1)
  (rewrite perm_rec)
  (("1"
    (rewrite perm_rec)
    (("1"
      (inst -1 "del(cons1_var!1, l1!1)" "l2!1")
      (prop)
      (lemma del_append)
      (inst -1 "l1!1" "l2!1" "cons1_var!1")
      (prop)
      (("1" (replace -1 1) (propax))
       ("2" (reveal -3) (rewrite perm_not -1))))
     ("2"
      (reveal -1)
      (rewrite perm_not -1)
      (lemma member_append)
      (inst -1 "l1!1" "l2!1" "cons1_var!1")
      (grind))))
   ("2" (rewrite perm_not -2))))))
```

perm_append1

```
(""
 (induct l1)
 (("1" (grind) (rewrite perm_reflexive))
  ("2"
   (skosimp)
   (skosimp)
   (inst -1 "l2!1" " x!1")
   (lemma perm_lemma1)
   (inst -1 "append(cons(x!1, cons2_var!1), l2!1)"
    "append(cons2_var!1, cons(x!1, l2!1))" "cons1_var!1")
   (replace -1 -2)
   (hide -1)
   (rewrite append_lemma -1)
   (rewrite append_lemma -1)
   (case "perm(cons(cons1_var!1, cons(x!1, cons2_var!1)),
               cons(x!1, cons(cons1_var!1, cons2_var!1)))")
   (("1"
     (lemma perm_append)
     (inst -1 "cons(cons1_var!1, cons(x!1, cons2_var!1))"
      "cons(x!1, cons(cons1_var!1, cons2_var!1))" "l2!1")
     (prop)
     (rewrite perm_commutativity -1)
     (lemma perm_transitivity)
     (inst -1 "append(cons(x!1, cons(cons1_var!1, cons2_var!1)), l2!1)"
      "append(cons(cons1_var!1, cons(x!1, cons2_var!1)), l2!1)"
      "append(cons(cons1_var!1, cons2_var!1), cons(x!1, l2!1))")
     (prop))
    ("2"
     (rewrite perm_rec 1)
     (("1"
       (case "x!1=cons1_var!1")
       (("1" (replace -1) (propax))
        ("2"
         (rewrite del_diff)
         (rewrite del_eq)
         (rewrite perm_reflexive))))
      ("2" (grind)))))))
```

$part\_member$: LEMMA $member(x,\ l_2) \wedge part(l_1,\ l_2,\ l_3) \Rightarrow member(x,\ l_3)$
$part\_lemma1$: LEMMA
  $member(x,\ l_2) \Rightarrow (part(l_1,\ l_2,\ l_3) \Rightarrow part(l_1,\ del(x,\ l_2),\ del(x,\ l_3)))$
$perm\_part\_null1$: LEMMA $part(null,\ l_1,\ l_2) = perm(l_1,\ l_2)$
$perm\_part\_null2$: LEMMA $part(l_1,\ null,\ l_2) = perm(l_1,\ l_2)$
$part\_null$: LEMMA $part(l_1,\ l_2,\ null) \Rightarrow l_1 = null \wedge l_2 = null$
$perm\_part\_append$: LEMMA $part(l_1,\ l_2,\ l) \Rightarrow perm(l,\ append(l_1,\ l_2))$
$perm\_part$: LEMMA
  $perm(l_1,\ l_2) \wedge perm(l_3,\ l_4) \wedge part(l_1,\ l_3,\ l) \Rightarrow part(l_2,\ l_4,\ l)$

FIG. C.4 – Lemmes de *part*

perm_append2

```
(""
 (induct l1)
 (("1"
   (skosimp)
   (grind)
   (rewrite append_null)
   (rewrite perm_reflexive))
  ("2"
   (skosimp)
   (skosimp)
   (expand append 1 1)
   (inst -1 "l2!1")
   (lemma perm_commut)
   (inst -1 "append(cons2_var!1, l2!1)" "append(l2!1, cons2_var!1)")
   (prop)
   (use perm_lemma1 ("x" "cons1_var!1"))
   (replace -1 -2)
   (hide -1 -3)
   (rewrite append_lemma -1)
   (rewrite perm_commutativity -1)
   (lemma perm_append1)
   (inst -1 "l2!1" "cons2_var!1" "cons1_var!1")
   (lemma perm_transitivity)
   (inst -1 "cons(cons1_var!1, append(cons2_var!1, l2!1))"
    "append(cons(cons1_var!1, l2!1), cons2_var!1)"
    "append(l2!1, cons(cons1_var!1, cons2_var!1))")
   (prop))))
```

C.3.4 part

Les preuves qui suivent se rapportent aux lemmes de la Figure C.4.

part_member

```
(""
 (induct l3)
 (("1"
   (induct l2)
   (("1" (skosimp)) ("2" (skosimp) (skosimp) (rewrite part_false2))))
  ("2"
   (skosimp)
   (skosimp)
   (case "x!1=cons1_var!1")
   (("1" (grind))
```

```
   ("2"
    (case "member(cons1_var!1, l1!1) OR member(cons1_var!1, l2!1)")
    (("1"
      (split)
      (("1"
        (rewrite part_rec1)
        (inst -2 "del(cons1_var!1, l1!1)" "l2!1" "x!1")
        (prop)
        (grind))
       ("2"
        (rewrite part_rec2)
        (inst -2 "l1!1" "del(cons1_var!1, l2!1)" "x!1")
        (grind)
        (rewrite del_member))))
     ("2" (flatten) (rewrite part_false3))))))))
```

part_lemma1

```
(""
 (induct l3)
 (("1"
   (induct l2)
   (("1" (skosimp) (grind))
    ("2" (skosimp) (skosimp) (rewrite part_false2))))
  ("2"
   (skosimp)
   (skosimp)
   (case "x!1=cons1_var!1")
   (("1" (rewrite del_eq) (replace -1) (rewrite part_rec2))
    ("2"
     (rewrite del_diff)
     (copy -3)
     (hide -1)
     (case "member(cons1_var!1, l2!1) OR member(cons1_var!1, l1!1)")
     (("1"
       (split)
       (("1"
         (rewrite part_rec2)
         (rewrite part_rec2)
         (("1"
           (rewrite del_lemma)
           (inst? -2)
           (split)
           (("1" (propax)) ("2" (propax)) ("3" (rewrite del_member))))
          ("2" (rewrite del_member))))
        ("2"
         (rewrite part_rec1)
         (rewrite part_rec1)
         (inst? -2)
         (split)
         (("1" (propax)) ("2" (propax)) ("3" (propax))))))
      ("2" (rewrite part_false3 -3))))))))
```

perm_part_null1

```
(""
 (induct l2)
 (("1"
   (induct l1)
   (("1" (rewrite part_empty) (rewrite perm_empty2))
    ("2" (skosimp) (rewrite part_false2) (rewrite perm_not) (grind))))
  ("2"
   (skosimp)
   (skosimp)
   (case "member(cons1_var!1, l1!1)")
   (("1"
```

```
    (rewrite part_rec2)
    (inst? -2)
    (replace -2)
    (rewrite perm_del2))
   ("2"
    (rewrite part_false3)
    (("1" (rewrite perm_commutativity) (rewrite perm_not))
     ("2" (grind)))))))
```

perm_part_null2

```
(""
 (induct l2)
 (("1"
   (induct l1)
   (("1" (rewrite part_empty) (rewrite perm_empty2))
    ("2" (skosimp) (rewrite part_false1) (rewrite perm_not) (grind))))
  ("2"
   (skosimp)
   (skosimp)
   (case "member(cons1_var!1, l1!1)")
   (("1"
     (rewrite part_rec1)
     (rewrite perm_commutativity)
     (rewrite perm_rec)
     (inst? -2)
     (rewrite perm_commutativity))
    ("2"
     (rewrite part_false3)
     (("1" (rewrite perm_commutativity) (rewrite perm_not))
      ("2" (grind)))))))
```

part_null

```
(""
 (skosimp)
 (split)
 (("1"
   (case "null?(l1!1) OR cons?(l1!1)")
   (("1"
     (split)
     (("1" (grind))
      ("2"
       (case "l1!1=cons(car(l1!1), cdr(l1!1))")
       (("1" (replace -1) (rewrite part_false1))
        ("2" (apply-extensionality 1)) ("3" (propax))))))
    ("2" (grind))))
  ("2"
   (case "null?(l2!1) OR cons?(l2!1)")
   (("1"
     (split)
     (("1" (grind))
      ("2"
       (case "l2!1=cons(car(l2!1), cdr(l2!1))")
       (("1" (replace -1) (rewrite part_false2))
        ("2" (apply-extensionality 1)) ("3" (propax))))))
    ("2" (grind))))))
```

perm_part_append

```
(""
 (measure-induct+ "length(l)" ("l"))
 (skosimp)
 (case "null?(x!1) OR cons?(x!1)")
```

```
(("1"
  (split)
  (("1"
    (case "x!1 = null")
    (("1"
      (replace -1)
      (use part_null)
      (prop)
      (replace -1)
      (replace -2)
      (grind)
      (rewrite perm_reflexive))
     ("2" (grind))))
   ("2"
    (case "(exists (x:nat),(l:lists):x!1=cons(x,l))")
    (("1"
      (skosimp)
      (replace -1)
      (hide -1 -2)
      (case "member(x!2, l1!1)")
      (("1"
        (rewrite part_rec1 -3)
        (inst -2 "l!1" "del(x!2, l1!1)" "l2!1")
        (prop)
        (("1"
          (use perm_lemma1 ("x" "x!2"))
          (replace -1 -2)
          (use perm_lemma2)
          (prop)
          (use perm_append
               ("l"
                "cons(x!2, del(x!2, l1!1))"
                "l1"
                "l1!1"
                "l2"
                "l2!1"))
          (prop)
          (rewrite append_lemma -3)
          (use perm_transitivity
               ("l"
                "cons(x!2, l!1)"
                "l1"
                "append(cons(x!2, del(x!2, l1!1)), l2!1)"
                "l2"
                "append(l1!1, l2!1)"))
          (prop))
         ("2" (grind))))
       ("2"
        (case "member(x!2, l2!1)")
        (("1"
          (rewrite part_rec2 -3)
          (inst -2 "l!1" "l1!1" "del(x!2, l2!1)")
          (prop)
          (("1"
            (use perm_append2)
            (use perm_transitivity
                 ("l"
                  "l!1"
                  "l1"
                  "append(l1!1, del(x!2, l2!1))"
                  "l2"
                  "append(del(x!2, l2!1), l1!1)"))
            (prop)
            (use perm_lemma1
                 ("x"
                  "x!2"
                  "l1"
                  "l!1"
```

```
                "l2"
                "append(del(x!2, l2!1), l1!1)"))
           (replace -1 -2)
           (rewrite append_lemma -2)
           (hide -1 -3 -5 -6)
           (use perm_lemma2)
           (prop)
           (("1"
             (use perm_append
                  ("l"
                   "cons(x!2, del(x!2, l2!1))"
                   "l1"
                   "l2!1"
                   "l2"
                   "l1!1"))
             (prop)
             (use perm_transitivity
                  ("l"
                   "cons(x!2, l!1)"
                   "l1"
                   "append(cons(x!2, del(x!2, l2!1)), l1!1)"
                   "l2"
                   "append(l2!1, l1!1)"))
             (prop)
             (use perm_append2)
             (use perm_transitivity
                  ("l"
                   "cons(x!2, l!1)"
                   "l1"
                   "append(l2!1, l1!1)"
                   "l2"
                   "append(l1!1, l2!1)"))
             (prop))
            ("2" (reveal -4) (propax))))
          ("2" (grind))))
        ("2" (rewrite part_false3))))))
    ("2" (inst 1 "car(x!1)" "cdr(x!1)") (apply-extensionality))))))
 ("2" (grind))))
```

perm_part

```
(""
 (induct l)
 (("1"
   (induct l1)
   (("1"
     (induct l3)
     (("1"
       (induct l2)
       (("1"
         (induct l4)
         (("1" (grind)) ("2" (skosimp) (rewrite perm_empty1))))
        ("2" (skosimp) (skosimp) (rewrite perm_empty1))))
      ("2" (skosimp) (skosimp) (rewrite part_false2))))
    ("2" (skosimp) (skosimp) (rewrite part_false1))))
  ("2"
   (skosimp)
   (induct l1)
   (("1"
     (induct l3)
     (("1"
       (skosimp)
       (rewrite part_false3)
       (("1" (grind)) ("2" (grind))))
      ("2"
       (skosimp)
       (skosimp)
```

```
    (case "member(cons1_var!1, l4!1)")
    (("1"
      (use perm_part_null1 ("l1" "cons(cons1_var!2, cons2_var!2)"))
      (replace -1 -6)
      (rewrite perm_commutativity -6)
      (use perm_transitivity
           ("l"
            "cons(cons1_var!1, cons2_var!1)"
            "l1"
            "cons(cons1_var!2, cons2_var!2)"
            "l2"
            "l4!1"))
      (prop)
      (case "l2!1 = null")
      (("1"
        (replace -1)
        (lemma perm_part_null1)
        (inst -1 "l4!1" "cons(cons1_var!1, cons2_var!1)")
        (rewrite perm_commutativity -3)
        (grind))
       ("2" (use perm_null ("l" "l2!1")) (prop))))
     ("2"
      (rewrite perm_part_null1 -4)
      (rewrite perm_commutativity -4)
      (use perm_transitivity
           ("l"
            "cons(cons1_var!1, cons2_var!1)"
            "l1"
            "cons(cons1_var!2, cons2_var!2)"
            "l2"
            "l4!1"))
      (prop)
      (use perm_not
           ("x" "cons1_var!1" "l2" "l4!1" "l1" "cons2_var!1"))
      (grind))))))
("2"
 (skosimp)
 (skosimp)
 (hide -1)
 (case "member(cons1_var!1, cons(cons1_var!2, cons2_var!2))")
 (("1"
   (rewrite part_rec1 -4)
   (lemma perm_del)
   (inst -1 "cons(cons1_var!2, cons2_var!2)" "l2!1" "cons1_var!1")
   (prop)
   (lemma perm_lemma3)
   (inst -1 "cons(cons1_var!2, cons2_var!2)" "l2!1" "cons1_var!1")
   (prop)
   (rewrite part_rec1 1)
   (inst -7 "del(cons1_var!1, cons(cons1_var!2, cons2_var!2))"
    "del(cons1_var!1, l2!1)" "l3!1" "l4!1")
   (prop))
  ("2"
   (case "member(cons1_var!1, l3!1)")
   (("1"
     (rewrite part_rec2 -4)
     (lemma perm_del)
     (inst -1 "l3!1" "l4!1" "cons1_var!1")
     (prop)
     (lemma perm_lemma3)
     (inst -1 "l3!1" "l4!1" "cons1_var!1")
     (prop)
     (rewrite part_rec2 2)
     (inst -7 "cons(cons1_var!2, cons2_var!2)" "l2!1"
      "del(cons1_var!1, l3!1)" "del(cons1_var!1, l4!1)")
     (prop))
    ("2" (rewrite part_false3 -3)))))))))
```

split_lemma : LEMMA $part(splitl(l),\ split(l),\ l)$
split_length1 : LEMMA $length(l) \leq 1 \Rightarrow length(split(l)) = 0$
split_length : LEMMA $length(l) > 1 \Rightarrow length(split(l)) < length(l)$
splitl_length1 : LEMMA $length(l) \leq 1 \Rightarrow length(splitl(l)) = length(l)$
splitl_length : LEMMA $length(l) > 1 \Rightarrow length(splitl(l)) < length(l)$

FIG. C.5 – Lemmes de *split*

C.3.5 split

Les preuves qui suivent se rapportent aux lemmes de la Figure C.5.

split_lemma

```
("" 
 (measure-induct+ "length(l)" ("l"))
 (case "null?(x!1) OR cons?(x!1)")
 (("1"
   (split)
   (("1"
     (case "x!1 = null")
     (("1"
       (replace -1 1)
       (rewrite splitl_empty)
       (rewrite split_empty)
       (rewrite part_empty))
      ("2" (assert))))
    ("2"
     (case "(exists (c:nat), (l:lists) : x!1 = cons(c, l))")
     (("1"
       (skosimp)
       (replace -1 1)
       (case "null?(l!1) OR cons?(l!1)")
       (("1"
         (split)
         (("1"
           (case "l!1 = null")
           (("1"
             (replace -1 1)
             (rewrite splitl_un)
             (rewrite split_un)
             (rewrite part_rec1)
             (("1" (rewrite del_eq) (rewrite part_empty))
              ("2" (grind))))
            ("2" (grind))))
          ("2"
           (case "(exists (c:nat),(l:lists) : l!1 = cons(c,l))")
           (("1"
             (skosimp)
             (replace -1)
             (rewrite splitl_rec)
             (rewrite split_rec)
             (rewrite part_rec1)
             (("1"
               (rewrite del_eq)
               (rewrite part_rec2)
               (("1" (rewrite del_eq) (inst? -5) (grind))
                ("2" (grind))))
              ("2" (grind))))
            ("2"
             (inst 1 "car(l!1)" "cdr(l!1)")
             (apply-extensionality 1))))))
```

```
       ("2" (assert) (grind))))
     ("2" (inst 1 "car(x!1)" "cdr(x!1)") (apply-extensionality 1))))))
  ("2" (grind))))
```

split_length1

```
(""
 (induct l)
 (("1" (flatten) (rewrite split_empty) (grind))
  ("2"
   (skosimp)
   (case "null?(cons2_var!1) OR cons?(cons2_var!1)")
   (("1"
     (split)
     (("1"
       (case "cons2_var!1=null")
       (("1" (replace -1) (rewrite split_un) (grind)) ("2" (grind))))
      ("2"
       (case "cons2_var!1=cons(car(cons2_var!1), cdr(cons2_var!1))")
       (("1" (replace -1) (grind)) ("2" (apply-extensionality 1))
        ("3" (propax))))))
    ("2" (grind))))))
```

split_length

```
(""
 (measure-induct+ "length(l)" ("l"))
 (case "null?(x!1) OR cons?(x!1)")
 (("1"
   (split)
   (("1" (case "x!1=null") (("1" (replace -1) (grind)) ("2" (grind))))
    ("2"
     (case "x!1=cons(car(x!1), cdr(x!1))")
     (("1"
       (replace -1)
       (name "lc" "cdr(x!1)")
       (replace -1)
       (case "null?(lc) OR cons?(lc)")
       (("1"
         (split)
         (("1"
           (case "lc=null")
           (("1" (replace -1) (grind)) ("2" (grind))))
          ("2"
           (case "lc=cons(car(lc), cdr(lc))")
           (("1"
             (replace -1)
             (rewrite split_rec)
             (inst? -6)
             (("1"
               (prop)
               (("1" (grind))
                ("2"
                 (case "length(cdr(lc)) <= 1")
                 (("1"
                   (lemma split_length1)
                   (inst -1 "cdr(lc)")
                   (("1" (grind)) ("2" (grind))))
                  ("2" (grind))))
                ("3" (grind))))
              ("2" (typepred lc) (grind))))
            ("2" (apply-extensionality 1)) ("3" (propax))))))
       ("2" (grind))))
     ("2" (apply-extensionality 1)) ("3" (propax))))))
  ("2" (grind))))
```

splitl_length1

```
(""
 (induct l)
 (("1" (flatten) (rewrite splitl_empty))
  ("2"
   (skosimp)
   (case "null?(cons2_var!1) OR cons?(cons2_var!1)")
   (("1"
     (split)
     (("1"
       (case "cons2_var!1=null")
       (("1" (replace -1) (rewrite splitl_un)) ("2" (grind))))
      ("2"
       (case "cons2_var!1=cons(car(cons2_var!1), cdr(cons2_var!1))")
       (("1" (replace -1) (grind)) ("2" (apply-extensionality 1))
        ("3" (propax))))))
    ("2" (grind))))))
```

splitl_length

```
(""
 (measure-induct+ "length(l)" ("l"))
 (case "null?(x!1) OR cons?(x!1)")
 (("1"
   (split)
   (("1"
     (case "x!1=null")
     (("1" (replace -1) (rewrite splitl_empty) (grind)) ("2" (grind))))
    ("2"
     (case "x!1=cons(car(x!1), cdr(x!1))")
     (("1"
       (replace -1)
       (name "lc" "cdr(x!1)")
       (replace -1)
       (case "null?(lc) OR cons?(lc)")
       (("1"
         (split)
         (("1"
           (case "lc=null")
           (("1" (replace -1) (rewrite splitl_un) (grind))
            ("2" (grind))))
          ("2"
           (case "lc=cons(car(lc),cdr(lc))")
           (("1"
             (replace -1)
             (rewrite splitl_rec)
             (inst? -6)
             (("1"
               (prop)
               (("1" (grind))
                ("2"
                 (case "length(cdr(lc)) <= 1")
                 (("1"
                   (lemma splitl_length1)
                   (inst -1 "cdr(lc)")
                   (("1" (grind)) ("2" (grind))))
                  ("2" (grind))))
                ("3" (grind))))
              ("2" (typepred lc) (grind))))
            ("2" (apply-extensionality 1)) ("3" (propax))))))
        ("2" (grind))))
      ("2" (apply-extensionality 1)) ("3" (propax))))))
  ("2" (grind))))
```

$merge\_lemma$: LEMMA $perm(merge(l_1,\ l_2),\ append(l_1,\ l_2))$
$lemma\_sorted1$: LEMMA $sorted(cons(x,\ l)) \Rightarrow sorted(l)$
$lemma\_sorted2$: LEMMA
$sorted(cons(x,\ cons(y,\ l))) \Rightarrow x \leq y \wedge sorted(cons(y,\ l))$
$lemma\_sorted3$: LEMMA $sorted(cons(x,\ cons(y,\ l))) \Rightarrow sorted(cons(x,\ l))$
$lemma\_sorted4$: LEMMA $sorted(cons(x,\ l)) \wedge member(y,\ l) \Rightarrow x \leq y$
$merge\_member$: LEMMA
$merge(l_1,\ l_2) = cons(x,\ l_3) \Rightarrow member(x,\ l_1) \vee member(x,\ l_2)$
$merge\_sorted$: LEMMA $sorted(l_1) \wedge sorted(l_2) \Rightarrow sorted(merge(l_1,\ l_2))$

FIG. C.6 – Lemmes de *merge* et *sorted*

C.3.6 merge

Les preuves qui suivent se rapportent aux lemmes de la Figure C.6.

merge_lemma

```
("" 
 (measure-induct+ "length(l1) + length(l2)" ("l1" "l2"))
 (case "null?(x!1) OR cons?(x!1)")
 (("1"
   (split)
   (("1"
     (case "x!1 = null")
     (("1"
       (replace -1 1)
       (rewrite merge_empty1)
       (expand append)
       (rewrite perm_reflexive))
      ("2" (grind))))
    ("2"
     (case "(exists (c:nat),(l:lists):x!1=cons(c,l))")
     (("1"
       (skosimp)
       (replace -1 1)
       (case "null?(x!2) OR cons?(x!2)")
       (("1"
         (split)
         (("1"
           (case "x!2 = null")
           (("1"
             (replace -1 1)
             (rewrite merge_empty2)
             (rewrite append_null)
             (rewrite perm_reflexive))
            ("2" (grind))))
          ("2"
           (case "(exists (c:nat),(l:lists):x!2=cons(c,l))")
           (("1"
             (skosimp)
             (replace -1 1)
             (case "c!1 <= c!2")
             (("1"
               (rewrite merge_inf)
               (expand append 1)
               (use perm_lemma1)
               (replace -1 1 :dir rl)
               (inst -7 "l!1" "cons(c!2, l!2)")
               (grind))
```

```
            ("2"
             (hide -1 -2 -3 -4 1)
             (inst -1 "cons(c!1, l!1)" "l!2")
             (split)
             (("1"
               (use perm_append2)
               (lemma perm_transitivity)
               (inst -1 "merge(cons(c!1, l!1), l!2)"
                "append(cons(c!1, l!1), l!2)"
                "append(l!2, cons(c!1, l!1))")
               (prop)
               (use perm_lemma1
                    ("x"
                     "c!2"
                     "l1"
                     "merge(cons(c!1, l!1), l!2)"
                     "l2"
                     "append(l!2, cons(c!1, l!1))"))
               (replace -1 -2)
               (case "cons(c!2, append(l!2, cons(c!1, l!1))) = append(cons(c!2, l!2), cons(c!1, l!1))")
               (("1"
                 (replace -1 -3)
                 (lemma perm_append2)
                 (inst -1 "cons(c!2, l!2)" "cons(c!1, l!1)")
                 (use perm_transitivity
                      ("l"
                       "cons(c!2, merge(cons(c!1, l!1), l!2))"
                       "l1"
                       "append(cons(c!2, l!2), cons(c!1, l!1))"
                       "l2"
                       "append(cons(c!1, l!1), cons(c!2, l!2))"))
                 (prop)
                 (reveal 1)
                 (rewrite merge_sup))
                ("2" (expand append 1 2) (propax))))
              ("2" (reveal -2 -4) (grind))))))
          ("2"
           (inst 1 "car(x!2)" "cdr(x!2)")
           (("1"
             (grind)
             (apply-extensionality 1)
             (typepred "x!2")
             (grind))
            ("2" (typepred "x!2") (grind))
            ("3" (typepred "x!2") (grind))))))))
      ("2" (grind))))
     ("2"
      (inst 1 "car(x!1)" "cdr(x!1)")
      (("1"
        (apply-extensionality 1)
        (("1" (typepred "x!1") (grind))
         ("2" (typepred "x!1") (grind))))
       ("2" (typepred "x!1") (grind))
       ("3" (typepred "x!1") (grind))))))
  ("2" (grind))))
```

lemma_sorted1

```
(""
 (induct "l")
 (("1" (grind) (rewrite "sorted_empty"))
  ("2"
   (skolem!)
   (grind)
   (use "excluded_middle" ("A" "x!1 <= cons1_var!1"))
   (split -1)
   (("1" (rewrite "sorted_le")) ("2" (rewrite "sorted_gt"))))))
```

lemma_sorted2

```
(""
 (induct "l")
 (("1"
   (skolem 1 ("x" "y"))
   (flatten)
   (use "excluded_middle" ("A" "x <= y"))
   (split -1)
   (("1" (rewrite "sorted_le") (assert)) ("2" (rewrite "sorted_gt"))))
  ("2"
   (skolem!)
   (flatten)
   (skolem!)
   (flatten)
   (use "excluded_middle" ("A" "x!1 <= y!1"))
   (split -1)
   (("1" (rewrite "sorted_le") (prop)) ("2" (rewrite "sorted_gt"))))))
```

lemma_sorted3

```
(""
 (induct "l")
 (("1" (skolem!) (rewrite "sorted_one"))
  ("2"
   (skolem!)
   (flatten)
   (skolem!)
   (flatten)
   (case "x!1 <= cons1_var!1")
   (("1"
     (rewrite "sorted_le" 1)
     (use "lemma_sorted1")
     (assert)
     (use "lemma_sorted1")
     (assert))
    ("2"
     (use "lemma_sorted2")
     (prop)
     (use "lemma_sorted2")
     (assert))))))
```

lemma_sorted4

```
(""
 (induct "l")
 (("1" (grind))
  ("2"
   (skolem!)
   (flatten)
   (skolem!)
   (flatten)
   (use "excluded_middle" ("A" "y!1 = cons1_var!1"))
   (split -1)
   (("1" (use "lemma_sorted2") (assert))
    ("2"
     (rewrite "member")
     (split -3)
     (("1" (propax))
      ("2" (use "lemma_sorted3") (assert) (inst? -3) (assert)))))))))
```

merge_member

```
(""
 (induct l1)
```

```
(("1" (skosimp) (rewrite merge_empty1) (replace -1) (grind))
 ("2"
  (skosimp)
  (induct l2)
  (("1" (skosimp) (rewrite merge_empty2) (replace -1) (grind))
   ("2"
    (skosimp)
    (skosimp)
    (case "cons1_var!1 <= cons1_var!2")
    (("1"
      (rewrite merge_inf)
      (case "cons1_var!1=x!1")
      (("1" (replace -1) (grind)) ("2" (decompose-equality -3))))
     ("2"
      (rewrite merge_sup)
      (decompose-equality -2)
      (replace -1)
      (grind))))))))
```

merge_sorted

```
(""
 (measure-induct+ "length(l1) + length(l2)" ("l1" "l2"))
 (case "null?(x!1) or cons?(x!1)")
 (("1"
   (split)
   (("1"
     (case "x!1=null")
     (("1" (replace -1) (rewrite merge_empty1)) ("2" (grind))))
    ("2"
     (case "x!1=cons(car(x!1), cdr(x!1))")
     (("1"
       (replace -1)
       (case "null?(x!2) or cons?(x!2)")
       (("1"
         (split)
         (("1"
           (case "x!2=null")
           (("1" (replace -1) (rewrite merge_empty2)) ("2" (grind))))
          ("2"
           (case "x!2=cons(car(x!2), cdr(x!2))")
           (("1"
             (replace -1)
             (name "c1" "car(x!1)")
             (replace -1)
             (name "c2" "car(x!2)")
             (replace -1)
             (case "c1 <= c2 OR c1 > c2")
             (("1"
               (split)
               (("1"
                 (rewrite merge_inf)
                 (("1"
                   (hide -2 -3 -4 -5 -6 -7)
                   (inst -2 "cdr(x!1)" "cons(c2, cdr(x!2))")
                   (("1"
                     (prop)
                     (("1"
                       (name "lm"
                             "merge(cdr(x!1), cons(c2, cdr(x!2)))")
                       (("1"
                         (replace -1)
                         (case "null?(lm) OR cons?(lm)")
                         (("1"
                           (split)
                           (("1"
                             (case "lm=null")
```

```
 (("1"
   (replace -1)
   (rewrite sorted_one)
   (reveal -2 -3 -4 -5 -6 -7)
   (replace -2 :dir rl)
   (typepred "x!1")
   (grind))
  ("2" (grind))))
("2"
 (case "lm=cons(car(lm), cdr(lm))")
 (("1"
   (replace -1)
   (use merge_member)
   (("1"
     (prop)
     (("1"
       (lemma lemma_sorted4)
       (inst -1 "cdr(x!1)" "c1" "car(lm)")
       (("1"
         (prop)
         (rewrite sorted_le 1)
         (reveal -4 -5 -6 -7 -8 -9)
         (replace -2 :dir rl)
         (hide 2)
         (typepred "x!1")
         (grind))
        ("2" (grind))
        ("3"
         (reveal -4 -5 -6 -7 -8)
         (replace -2 :dir rl)
         (hide 2)
         (typepred "x!1")
         (grind))
        ("4"
         (reveal -4 -5 -6 -7 -8)
         (typepred "x!1")
         (grind))
        ("5"
         (reveal -4 -5 -6 -7 -8)
         (typepred "x!1")
         (grind))))
      ("2"
       (lemma lemma_sorted4)
       (reveal -4 -5 -6 -7 -8 -9)
       (case "car(lm)=c2")
       (("1"
         (replace -1)
         (rewrite sorted_le)
         (replace -2 :dir rl)
         (typepred "x!1")
         (grind))
        ("2"
         (expand member -8)
         (prop)
         (inst
          -8
          "cdr(x!2)"
          "c2"
          "car(lm)")
         (("1"
           (prop)
           (rewrite sorted_le)
           (replace -3 :dir rl)
           (typepred "x!1")
           (grind))
          ("2" (grind))
          ("3"
           (reveal -4)
```

```
                        (replace -1 :dir rl)
                        (typepred "x!2")
                        (grind))
                       ("4"
                        (reveal -4)
                        (typepred "x!2")
                        (grind))
                       ("5"
                        (typepred "x!2")
                        (grind))))))))
                 ("2"
                  (reveal -3 -4 -5 -6 -7 -8)
                  (grind))
                 ("3"
                  (reveal -3 -4 -5 -6 -7 -8)
                  (typepred "x!2")
                  (grind))
                 ("4"
                  (reveal -3 -4 -5 -6 -7 -8)
                  (typepred "x!1")
                  (grind))
                 ("5"
                  (reveal -3 -4 -5 -6 -7 -8)
                  (typepred "x!1")
                  (grind))))
               ("2" (apply-extensionality 1))
               ("3" (propax))))))
           ("2" (grind))))
        ("2"
         (reveal -3 -4 -5 -6 -7 -8)
         (typepred "x!1")
         (grind))))
      ("2"
       (use lemma_sorted1)
       (("1" (prop))
        ("2"
         (reveal -3 -4 -5 -6 -7 -8)
         (replace -2 :dir rl)
         (typepred "x!1")
         (grind))
        ("3"
         (reveal -3 -4 -5 -6 -7 -8)
         (typepred "x!1")
         (grind))
        ("4"
         (reveal -3 -4 -5 -6 -7 -8)
         (typepred "x!1")
         (grind))))
      ("3" (grind))))
    ("2"
     (reveal -3 -4 -5 -6 -7 -8)
     (typepred "x!2")
     (grind))
    ("3"
     (reveal -3 -4 -5 -6 -7 -8)
     (typepred "x!1")
     (grind))
    ("4"
     (reveal -3 -4 -5 -6 -7 -8)
     (typepred "x!1")
     (grind))))
  ("2" (replace -2 :dir rl) (typepred "x!2") (grind))
  ("3" (replace -3 :dir rl) (typepred "x!1") (grind))
  ("4" (typepred "x!2") (grind))
  ("5" (typepred "x!1") (grind))))
("2"
 (rewrite merge_sup)
 (("1"
```

```
(inst? -8)
(("1"
  (prop)
  (("1"
    (name "lm"
          "merge(cons(c1, cdr(x!1)), cdr(x!2))")
    (("1"
      (replace -1)
      (case "null?(lm) OR cons?(lm)")
      (("1"
        (split)
        (("1"
          (case "lm=null")
          (("1"
            (replace -1)
            (rewrite sorted_one)
            (replace -6 :dir rl)
            (typepred "x!2")
            (grind))
           ("2" (grind))))
         ("2"
          (case "lm=cons(car(lm), cdr(lm))")
          (("1"
            (replace -1)
            (use merge_member)
            (("1"
              (split)
              (("1"
                (case "car(lm)=c1")
                (("1"
                  (replace -1)
                  (lemma sorted_le)
                  (inst -1 "cdr(lm)" "c2" "c1")
                  (("1"
                    (prop)
                    (("1"
                      (replace -1 -7 :dir rl)
                      (propax))
                     ("2" (grind))))
                   ("2"
                    (replace -9 :dir rl)
                    (typepred "x!1")
                    (grind))
                   ("3"
                    (replace -8 :dir rl)
                    (typepred "x!2")
                    (grind))
                   ("4" (grind))))
                 ("2"
                  (expand member -1)
                  (split)
                  (("1" (propax))
                   ("2"
                    (lemma lemma_sorted4)
                    (inst
                     -1
                     "cdr(x!1)"
                     "c1"
                     "car(lm)")
                    (("1"
                      (prop)
                      (lemma sorted_le)
                      (inst
                       -1
                       "cdr(lm)"
                       "c2"
                       "car(lm)")
                      (("1"
```

```
                                (prop)
                                (("1"
                                  (replace -1 :dir rl)
                                  (propax))
                                 ("2" (hide 3) (grind))))
                               ("2"
                                (replace -8 :dir rl)
                                (typepred "x!2")
                                (grind))
                               ("3" (grind))))
                              ("2" (grind))
                              ("3"
                               (replace -8 :dir rl)
                               (typepred "x!1")
                               (grind))
                              ("4" (typepred "x!1") (grind))
                              ("5"
                               (typepred "x!1")
                               (grind))))))))
                         ("2"
                          (lemma lemma_sorted4)
                          (inst -1 "cdr(x!2)" "c2" "car(lm)")
                          (("1"
                            (prop)
                            (rewrite sorted_le)
                            (replace -8 :dir rl)
                            (typepred "x!2")
                            (grind))
                           ("2" (grind))
                           ("3"
                            (replace -7 :dir rl)
                            (typepred "x!2")
                            (grind))
                           ("4" (typepred "x!2") (grind))
                           ("5" (typepred "x!2") (grind))))
                         ("3" (propax))))
                       ("2" (grind))
                       ("3" (typepred "x!2") (grind))
                       ("4" (typepred "x!2") (grind))
                       ("5" (typepred "x!1") (grind))))
                     ("2" (apply-extensionality 1))
                     ("3" (propax))))))
                ("2" (grind))))
              ("2" (typepred "x!2") (grind))))
            ("2"
             (lemma lemma_sorted1)
             (inst -1 "cdr(x!2)" "c2")
             (("1" (prop))
              ("2"
               (replace -2 :dir rl)
               (typepred "x!2")
               (grind))
              ("3" (typepred "x!2") (grind))
              ("4" (typepred "x!2") (grind))))
            ("3" (grind))))
          ("2" (typepred "x!2") (grind))
          ("3" (typepred "x!2") (grind))
          ("4" (typepred "x!1") (grind))
          ("5" (typepred "x!1") (grind))
          ("6"
           (replace -3 :dir rl)
           (typepred "x!1")
           (grind))))
        ("2" (replace -2 :dir rl) (typepred "x!2") (grind))
        ("3" (replace -3 :dir rl) (typepred "x!1") (grind))
        ("4" (typepred "x!2") (grind))
        ("5" (typepred "x!1") (grind))))))
("2" (grind)) ("3" (flatten) (typepred "x!2") (grind))
```

mergesort_perm: LEMMA *perm*(*l*, *mergesort*(*l*))
mergesort_sorted: LEMMA *sorted*(*mergesort*(*l*))

FIG. C.7 – Propriétés de *mergesort*

```
            ("4" (typepred "x!1") (grind))
            ("5" (typepred "x!2") (grind))
            ("6" (typepred "x!1") (grind))))
          ("2" (apply-extensionality 1)) ("3" (propax))))))
      ("2" (grind))))
    ("2" (apply-extensionality 1)) ("3" (propax))))))
 ("2" (grind))))
```

C.3.7 mergesort

Les preuves qui suivent se rapportent aux lemmes de la Figure C.7.

mergesort_perm

```
(""
 (measure-induct+ "length(l)" ("l"))
 (case "null?(x!1) OR cons?(x!1)")
 (("1"
   (split)
   (("1"
     (case "x!1 = null")
     (("1"
       (replace -1)
       (rewrite mergesort_empty)
       (rewrite perm_reflexive))
      ("2" (grind))))
    ("2"
     (case "(exists (x:nat),(l:lists):x!1=cons(x,l))")
     (("1"
       (skosimp)
       (replace -1)
       (hide -1)
       (hide -1)
       (case "null?(l!1) OR cons?(l!1)")
       (("1"
         (split)
         (("1"
           (case "l!1 = null")
           (("1"
             (replace -1)
             (rewrite mergesort_un)
             (rewrite perm_reflexive))
            ("2" (grind))))
          ("2"
           (rewrite mergesort_rec)
           (use merge_lemma)
           (copy -3)
           (inst -1 "splitl(cons(x!2, l!1))")
           (prop)
           (("1"
             (inst -4 "split(cons(x!2, l!1))")
             (prop)
             (("1"
               (use split_lemma ("l" "cons(x!2, l!1)"))
               (use perm_part
                    ("l3"
                     "split(cons(x!2, l!1))"
```

```
                "l4"
                "mergesort(split(cons(x!2, l!1)))"
                "l1"
                "splitl(cons(x!2, l!1))"
                "l2"
                "mergesort(splitl(cons(x!2, l!1)))"
                "l"
                "cons(x!2, l!1)"))
           (prop)
           (use perm_part_append
                ("l1"
                 "mergesort(splitl(cons(x!2, l!1)))"
                 "l2"
                 "mergesort(split(cons(x!2, l!1)))"
                 "l"
                 "cons(x!2, l!1)"))
           (prop)
           (rewrite perm_commutativity -6)
           (use perm_transitivity
                ("l"
                 "cons(x!2, l!1)"
                 "l1"
                 "append(mergesort(splitl(cons(x!2, l!1))), mergesort(split(cons(x!2, l!1))))"
                 "l2"
                 "merge(mergesort(splitl(cons(x!2, l!1))), mergesort(split(cons(x!2, l!1))))"))
           (prop))
          ("2" (rewrite split_length) (grind))))
        ("2" (rewrite splitl_length) (grind))))))
    ("2" (grind))))
  ("2" (inst 1 "car(x!1)" "cdr(x!1)") (apply-extensionality))))))
("2" (grind))))
```

mergesort_sorted

```
(""
 (measure-induct+ "length(l)" ("l"))
 (case "null?(x!1) OR cons?(x!1)")
 (("1"
   (split)
   (("1"
     (case "x!1=null")
     (("1"
       (replace -1)
       (rewrite mergesort_empty)
       (rewrite sorted_empty))
      ("2" (grind))))
    ("2"
     (case "x!1=cons(car(x!1), cdr(x!1))")
     (("1"
       (replace -1)
       (name "lc" "cdr(x!1)")
       (replace -1)
       (case "null?(lc) OR cons?(lc)")
       (("1"
         (split)
         (("1"
           (case "lc=null")
           (("1"
             (replace -1)
             (rewrite mergesort_un)
             (rewrite sorted_one))
            ("2" (grind))))
          ("2"
           (case "lc=cons(car(lc), cdr(lc))")
           (("1"
             (replace -1)
             (rewrite mergesort_rec)
```

```
            (copy -6)
            (inst? -1)
            (("1"
              (use splitl_length)
              (prop)
              (("1"
                (inst -8
                 "split(cons(car(x!1), cons(car(lc), cdr(lc))))")
                (("1"
                  (lemma split_length)
                  (inst -1 "x!1")
                  (prop)
                  (("1"
                    (use merge_sorted)
                    (("1" (prop)) ("2" (typepred lc) (grind))
                     ("3" (grind))))
                   ("2" (grind)) ("3" (grind)) ("4" (grind))))
                 ("2" (grind)) ("3" (grind))))
               ("2" (grind)) ("3" (grind)) ("4" (grind))))
             ("2" (grind)) ("3" (grind))))
           ("2" (apply-extensionality 1)) ("3" (propax))))))
       ("2" (grind))))
     ("2" (apply-extensionality 1)) ("3" (propax))))))
("2" (grind))))
```

Bibliographie

[Abrial, 1996]
J.-R. Abrial. *The B-book : assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[Aho et Ullman, 1994]
Alfred V. Aho et Jeffrey D. Ullman. *Foundations of Computer Science : C edition.* W. H. Freeman & Co., 1994.

[Andersen, 1994]
Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Mai 1994.

[Ashcroft et Manna, 1972]
E. Ashcroft et Z. Manna. The translation of GOTO programs into WHILE programs. Dans C. V. Freiman, J. E. Griffith et J. L. Rosenfeld, éditeurs, *Proceedings of IFIP Congress 71*, volume 1, pages 250–255. North-Holland, 1972.

[Baader et Nipkow, 1998]
F. Baader et T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Barnes, 2003]
John Barnes. *High Integrity Software : The SPARK Approach to Safety and Security.* Addison Wesley, 2003.

[Barras *et al.*, 1997]
B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi et B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, Août 1997.

[Berdine *et al.*, 2004]
Josh Berdine, Cristiano Calcagno et Peter W. O'Hearn. A decidable fragment of separation logic. Dans *24th International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 3328 de *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, Décembre 2004.

[Berdine *et al.*, 2005]
Josh Berdine, Cristiano Calcagno et Peter W. O'Hearn. Symbolic execution with separation logic. Dans *Third Asian Symposium on Programming Languages and Systems*, volume 3780 de *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, Novembre 2005.

[Bergstra *et al.*, 1997]
J. A. Bergstra, T. B. Dinesh, J. Field et J. Heering. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programming Languages and Systems*, 19(5) :639–684, Septembre 1997.

[Bertot et Casteran, 2004]
Yves Bertot et Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[Böhm et Jacopini, 1966]
Corrado Böhm et Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communication of the ACM*, 9(5) :366–371, 1966.

[Borovanský *et al.*, 1996]
Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau et Marian Vittek. Elan : A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4 :35–50, 1996.

[Borras *et al.*, 1988]
P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang et V. Pascual. Centaur : the system. *SIGSOFT Software Engineering Notes*, 13(5) :14–24, 1988.

[Bouali, 1998]
Amar Bouali. Xeve, an esterel verification environment. Dans *CAV '98 : Proceedings of the 10th International Conference on Computer Aided Verification*, pages 500–504. Springer-Verlag, 1998.

[Bouhoula *et al.*, 1992]
A. Bouhoula, E. Kounalis et M. Rusinowitch. Spike : An automatic theorem prover. Dans A. Voronkov, éditeur, *Proc. Internat. Conf. on Logic Programming and Automated Reasoning*, volume 624 de *Lecture Notes in Artificial Intelligence*, pages 460–462, St. Petersburgh, Russia, Juillet 1992. Springer-Verlag.

[Bryant, 1986]
Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[Calcagno *et al.*, 2005]
Cristiano Calcagno, Philippa Gardner et Matthew Hague. From separation logic to first-order logic. Dans Vladimiro Sassone, éditeur, *8th International Conference on Foundations of Software Science and Computational Structures*, volume 3441 de *Lecture Notes in Computer Science*, pages 395–409. Springer-Verlag, 2005.

[Campagnolle, 2004]
Laurent Campagnolle. La SNCF élimine son bug. *01net*, Juillet 2004.

[Chakaravarthy, 2003]
Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. *ACM SIGPLAN Notices*, 38(1) :115–125, Janvier 2003.

[Choi *et al.*, 1993]
Jong-Deok Choi, Michael Burke et Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. Dans *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, Janvier 1993. ACM Press.

[Clarke et Wing, 1996]
Edmund M. Clarke et Jeannette M. Wing. Formal methods : state of the art and future directions. *ACM Computing Surveys*, 28(4) :626–643, 1996.

[Colmerauer et Roussel, 1993]
Alain Colmerauer et Philippe Roussel. The birth of Prolog. Dans *HOPL-II : The second ACM SIGPLAN conference on History of programming languages*, pages 37–52, New York, NY, USA, 1993. ACM Press.

[Coquand et Huet, 1988]
Thierry Coquand et Gerard Huet. The calculus of constructions. *Informationa and Computation*, 76(2-3) :95–120, 1988.

[David *et al.*, 2001]
René David, Karim Nour et Christophe Raffalli. *Introduction à la logique, théorie de la démonstration*. Dunod, 2001.

[Denzler et Schweizer, 1996]
C. Denzler et D. Schweizer. Pesca : Programming environnement for safety critical application. Non publié, `http://www.tik.ee.ethz.ch/~pesca`, 1996.

[Dershowitz *et al.*, 1988]
Nachum Dershowitz, Mitsuhiro Okada et G. Sivakumar. Canonical conditional rewrite systems. Dans Ewing L. Lusk et Ross A. Overbeek, éditeurs, *9th International Conference on Automated Deduction*, volume 310 de *Lecture Notes in Computer Science*, pages 538–549. Springer, 1988.

[Dershowitz, 1982]
Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3) :279–301, 1982.

[Didrich *et al.*, 1994]
Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp et Peter Pepper. Opal : design and implementation of an algebraic programming language. Dans *Proceedings of the international conference on Programming languages and system architectures*, pages 228–244. Springer-Verlag, 1994.

[Dijkstra, 1968]
Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3) :147–148, Mars 1968.

[Dijkstra, 1972]
Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10) :859–866, 1972.

[Dijkstra, 1976]
Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dijkstra, 2001]
Edsger W. Dijkstra. What led to "Notes on Structured Programming". Non publié, `http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1308.PDF`, Juin 2001.

[Elseaidy *et al.*, 1997]
Wael M. Elseaidy, Rance Cleaveland et Jr. John W. Baugh. Modeling and verifying active structural control systems. *Science of Computer Programming*, 29(1-2) :99–122, 1997.

[Fédèle et Kounalis, 1999]
C. Fédèle et E. Kounalis. Automatic proofs of properties of simple $C--$ modules. Dans *Proc. 14th IEEE Internat. Conf. on Automated Software Engineering*, pages 283–286, Cocoa Beach, Floride, USA, Octobre 1999.

[Field *et al.*, 1998]
J. Field, J. Heering et T. B. Dinesh. Equations as a uniform framework for partial evaluation and abstract interpretation. *ACM Computing Surveys*, 30(3es) :2, 1998.

[Filliâtre, 1999]
Jean-Christoph. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université de Paris XI, Juillet 1999.

[Floyd, 1967]
R. W. Floyd. Assigning meanings to programs. Dans J. T. Schwartz, éditeur, *Mathematical Aspects of Computer Science : Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32, Providence, 1967. American Mathematical Society.

[Garland *et al.*, 1993]
J. S. Garland, J. V. Guttag et J. J. Horning. An overview of larch. Dans *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 de *Lecture Notes in Computer Science*. Springer-Verlag, Juillet 1993.

[Gibbons, 1998]
Hugh Gibbons. Declarative view of imperative programs. Dans Andrew Butterfield et Sharon Flynn, éditeurs, *Proc. IWFM'98 : 2nd Irish Workshop on Formal Methods*, Workshops in Computing. The British Computer Society, Cork, Ireland, Juillet 1998.

[Goguen et Malcolm, 1996]
Joseph A. Goguen et Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations of Computing. The MIT Press, 1996.

[Gordon, 1987]
M. Gordon. Hol : A proof generating system for higher-order logic. Dans G. Birtwistle et P. A. Subrahmanyam, éditeurs, *VLSI Specification, Verification, and Synthesis*. Kluwer, 1987.

[Guiho et Hennebert, 1990]
G. Guiho et C. Hennebert. Sacem software validation. Dans *ICSE '90 : Proceedings of the 12th international conference on Software engineering*, pages 186–191, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[Gurevich, 1985]
Yuri Gurevich. A new thesis. *Abstracts, American Mathematical Society*, 6(4) :317, Août 1985.

[Gurevich, 1991]
Yuri Gurevich. Evolving algebras : An introductory tutorial. *Bulletin of the European Association for Theoretical Computer Science*, 43 :264–284, Février 1991.

[Guttag et Horning, 1993]
John V. Guttag et James J. Horning. *Larch : languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[Hind, 2001]
Michael Hind. Pointer analysis : haven't we solved this problem yet ? Dans *ACM SIGPLAN–SIGSOFT workshop on Program analysis for software tools and engineering : June 18–19, 2001, Snowbird, Utah, USA : PASTE'01*, pages 54–61. ACM Press, 2001.

[Hoare, 1969]
C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[Homeier et Martin, 1994]
Peter V. Homeier et David F. Martin. Trustworthy tools for trustworthy programs : A verified verification condition generator. Dans *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 269–284, London, UK, 1994. Springer-Verlag.

[Honsell *et al.*, 1995]
Furio Honsell, Ian A. Mason, Scott Smith et Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119(1) :55–90, Mai 1995.

[Horwitz *et al.*, 1989]
S. Horwitz, P. Pfeiffer et T. Reps. Dependence analysis for pointer variables. Dans *PLDI*

'89 : Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, pages 28–40. ACM Press, 1989.

[Hudak et Fasel, 1992]
Paul Hudak et Joseph H. Fasel. A gentle introduction to haskell. *ACM SIGPLAN Notices*, 27(5) :1–52, 1992.

[Igarashi *et al.*, 1975]
S. Igarashi, R. L. London et D. C. Luckham. Automatic program verification I : A logical basis and its implementation. *ACTA Informatica*, 4 :145–182, 1975.

[Jacobs *et al.*, 1998]
Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, U. Hensel et H. Tews. Reasoning about java classes : preliminary report. Dans *OOPSLA '98 : Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 329–340. ACM Press, 1998.

[Jacobs et Poll, 2003]
Bart Jacobs et Erik Poll. Coalgebras and monads in the semantics of java. *Theoretical Computer Science*, 291(3) :329–349, 2003.

[Jones, 1990]
C. B. Jones. *Systematic sofware development using VDM*. International Series In Computer Science. Prentice-Hall, 2^{e} édition, 1990.

[Juellig *et al.*, 1996]
R. Juellig, Y. Srinivas et J. Liu. SPECWARE : An advanced environment for the formal development of complex software systems. *Lecture Notes in Computer Science*, 1101 :551, 1996.

[Jézéquel et Meyer, 1997]
Jean-Marc Jézéquel et Bertrand Meyer. Design by contract : The lessons of ariane. *Computer*, 30(1) :129–130, 1997.

[Kamin et Levy, 1980]
S. Kamin et J.-J. Levy. Two generalizations of the recursive path ordering. Non publié, Department of computer Science, University of Illinois, Urbana, 1980.

[Kapur et Zhang, 1988]
D. Kapur et H. Zhang. RRL : A Rewrite Rule Laboratory. Dans *Proc. 9th Internat. Conf. on Automated Deduction (CADE)*, volume 310 de *Lecture Notes in Computer Science*, pages 768–769, Argonne, Illinois, USA, Mai 1988. Springer.

[Klop, 1992]
J. W. Klop. Term rewriting systems. Dans *Handbook of logic in computer science (vol. 2), background : computational structures*, Osborne Handbooks Of Logic In Computer Science, pages 1–116. Oxford University Press, Inc., New York, NY, USA, 1992.

[Knuth et Bendix, 1970]
D. E. Knuth et P. B. Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[Knuth, 1968]
Donald E. Knuth. *Fundamental Algorithms*, volume I de *The Art of Computer Programming*. Addison-Wesley, 1968.

[Knuth, 1974]
Donald E. Knuth. Computer programming as an art. *Communications of the ACM*, 17(12) :667–673, 1974.

[Landi, 1992]
William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4) :323–337, Décembre 1992.

[Larus et Hilfinger, 1988]
J. R. Larus et P. N. Hilfinger. Detecting conflicts between structure accesses. Dans *PLDI '88 : Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31. ACM Press, 1988.

[Leavens *et al.*, 1999]
Gary T. Leavens, Albert L. Baker et Clyde Ruby. *JML : A Notation for Detailed Design*, chapitre 12, pages 175–188. Behavioral Specifications of Businesses and Systems. Kluwer Academic Publishers, 1999.

[Loeckx et Sieber, 1987]
Jacques Loeckx et Kurt Sieber. *The foundations of program verification*. Wiley-Teubner Series in Computer Science. John Wiley & Sons, Inc., New York, NY, USA, 2^{e} édition, 1987.

[Manna et Pnueli, 1992]
Zohar Manna et Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, Inc., New York, NY, USA, 1992.

[Marché *et al.*, 2004]
Claude Marché, Christine Paulin et Xavier Urbain. The krakatoa tool for JML/Java program certification. *Journal of Logic and Algebraic Programming*, 58(1–2) :89–106, 2004.

[Mason, 1997]
Ian A. Mason. A first order logic of effects. *Theoretical Computer Science*, 185(2) :277–318, 1997.

[McCune, 1994]
W. McCune. *Otter 3.0 Reference Manual and Guide*. Argonne National Laboratory, 1994. Report ANl-94/6.

[McMillan, 1993]
Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Mehta et Nipkow, 2003]
Farhad Mehta et Tobias Nipkow. Proving pointer programs in higher-order logic. Dans Franz Baader, éditeur, *19th International Conference on Automated Deduction*, volume 2741 de *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2003.

[Meyer, 2003]
Bertrand Meyer. Proving pointer program properties, part 1 : Context and overview. *Journal of Object Technology*, 2(2) :87–108, Mars 2003.

[MISRA, 1998]
MISRA, éditeur. *Guidelines for the Use of the C Language in Vehicle Based Software*. MISRA, MISRA Electrical Group, MIRA Ltd, Watling Street, Nuneaton, Warwickshire, CV10 0TU, UK, 1998.

[Moggi, 1991]
Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1) :55–92, Juillet 1991.

[Møller et Schwartzbach, 2001]
Anders Møller et Michael I. Schwartzbach. The pointer assertion logic engine. Dans *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 221–231. ACM Press, 2001.

[Morris, 1973]
F. Lockwood Morris. Advice on structuring compilers and proving them correct. Dans *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 144–152, New York, NY, USA, 1973. ACM Press.

[Moura, 1997]
Bárbara Moura. *Vers une correspondance entre les paradigmes fonctionnels et impératifs*. Thèse de doctorat, Université de Rennes I, Avril 1997.

[O'Hearn *et al.*, 2001]
Peter W. O'Hearn, John C. Reynolds et Hongseok Yang. Local reasoning about programs that alter data structures. Dans *Proceedings of the 15th International Workshop on Computer Science Logic*, volume 2142 de *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001.

[Owre *et al.*, 1992]
S. Owre, J. M. Rushby et N. Shankar. PVS : A prototype verification system. Dans Deepak Kapur, éditeur, *Proc. 11th Internat. Conf. on Automated Deduction (CADE)*, volume 607 de *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, Juin 1992. Springer-Verlag.

[Paulin-Mohring, 1989]
Christine Paulin-Mohring. *Extraction de programmes dans le calcul des constructions.* Thèse de doctorat, Université de Paris VII, Janvier 1989.

[Paulson, 1994]
Lawrence C. Paulson. *Isabelle : a generic theorem prover*, volume 828 de *Lecture notes in Computer Science.* Springer Verlag, 1994.

[Peralta *et al.*, 1998]
Julio C. Peralta, John P. Gallagher et Hüseyin Saglam. Analysis of imperative programs through analysis of constraint logic programs. Dans *SAS '98 : Proceedings of the 5th International Symposium on Static Analysis*, pages 246–261, London, UK, 1998. Springer-Verlag.

[Plaisted, 1993]
David A. Plaisted. Equational reasoning and term rewriting systems. Dans Dov Gabbay, Christopher Hogger et J. A. Robinson, éditeurs, *The Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 1 : Deductive Methodologies*, pages 274–367. Oxford University Press, Oxford, 1993.

[Pnueli, 1981]
Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1) :45–60, 1981.

[Ponsini *et al.*, 2002]
O. Ponsini, C. Fédèle et E. Kounalis. Sos $C--$: A system for interpreting operational semantics of $C--$ programs. Dans M. H. Hamza, éditeur, *Proc. IASTED Internat. Conf. on Applied Informatics*, pages 164–169, Innsbruck, Austria, Février 2002.

[Ponsini *et al.*, 2005]
Olivier Ponsini, Carine Fédèle et Emmanuel Kounalis. Rewriting of imperative programs into logical equations. *Science of Computer Programming*, 56(3) :363–401, 2005.

[Pratt, 1995]
V. R. Pratt. Anatomy of the Pentium Bug. Dans P. D. Mosses, M. Nielsen et M. I. Schwartzbach, éditeurs, *Proceedings of the 6th International Joint Conference CAAP/-FASE on Theory and Practice of Software Development*, number 915 dans Lecture Notes In Computer Science, pages 97–107. Springer Verlag, 1995.

[Ranise et Zarba, 2005]
S. Ranise et C. Zarba. A decidable logic for pointer programs manipulating linked lists. Non publié, `http://www.loria.fr/~ranise/papers.html`, 2005.

[Reynolds, 2002]
John C. Reynolds. Separation logic : A logic for shared mutable data structures. Dans *IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, Juillet 2002.

[RISKS, 2005]
Forum on risks to the public in computers and related systems, 2005. ACM Committee on Computers and Public Policy, Peter G. Neumann, moderator. Liste de diffusion : `news://comp.risks/`, site internet : `http://catless.ncl.ac.uk/Risks/`.

[Russinoff, 1999]
David M. Russinoff. A mechanically checked proof of correctness of the amd k5 floating point square root microcode. *Formal Methods in System Design*, 14(1) :75–125, 1999.

[Sagiv *et al.*, 1998]
Mooly Sagiv, Thomas Reps et Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1) :1–50, Janvier 1998.

[Sagiv *et al.*, 2002]
Mooly Sagiv, Thomas Reps et Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3) :217–298, 2002.

[Scott et Strachey, 1971]
Dana Scott et Christopher Strachey. Towards a mathematical semantics for computer languages. Dans *21st Symp. on Computers and Automata*, pages 19–46. Polytechnic institute of Brooklyn, 1971.

[Spivey, 1992]
Mike Spivey. *The Z Notation : A Reference Manual*. International Series in Computer Science. Prentice Hall, 2^{e} édition, 1992.

[Steensgaard, 1996]
Bjarne Steensgaard. Points-to analysis in almost linear time. Dans *POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41. ACM Press, Janvier 1996.

[van den Brand *et al.*, 2001]
Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser et Joost Visser. The asf+sdf meta-environment : A component-based language development environment. Dans *CC '01 : Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370. Springer-Verlag, 2001.

[Ward, 1989]
Martin Ward. *Proving program refinements and transformations*. PhD thesis, Oxford University, 1989.

[Ward, 1993]
M. Ward. Abstracting a specification from code. *Journal of Software Maintenance : Research and Practice*, 5(2) :101–122, Juin 1993.

[Wing, 1990]
Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9) :8–23, 1990.

[Wulf et Shaw, 1973]
W. Wulf et Mary Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2) :28–34, 1973.

Index