THÈSE

présentée à

L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir

LE GRADE DE DOCTEUR D'ÉTAT (MENTION SCIENCES) **EN INFORMATIQUE**

par

Marie Claude GAUDEL

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE NANCY

Sujet de la thèse :

GÉNÉRATION ET PREUVE DE COMPILATEURS BASÉES SUR UNE SÉMANTIQUE FORMELLE DES LANGAGES DE PROGRAMMATION

D 136 036438 6

) devant la Commission composée de :

		Président
M.	NIVAT	Rapporteurs
M.	SINTZOFF	
J. P.	JOUANNAUD	Examinateurs
B.	LORHO	

1360364386

[T] 1980 GAUDEL, M-C.

THÈSE

présentée à

L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir

LE GRADE DE DOCTEUR D'ÉTAT (MENTION SCIENCES) EN INFORMATIQUE

par

Marie Claude GAUDEL

Sujet de la thèse :

GÉNÉRATION ET PREUVE DE COMPILATEURS BASÉES SUR UNE SÉMANTIQUE FORMELLE DES LANGAGES DE PROGRAMMATION

Soutenue le 10 Mars 1980 devant la Commission composée de :

MM.	C.	PAIR	Président
	M.	NIVAT	Rapporteurs
	M.	SINTZOFF	
	J.P.	JOUANNAUD	Examinateurs
	B.	LORHO	

pour Florence,
Pascale,
Nadia,
et les autres.

Depuis de nombreuses années, j'ai la chance d'effectuer mes recherches sous la direction de C. PAIR, président de l'I.N.P.L. Mes recherches et cette thèse lui doivent beaucoup, tant sur le plan des idées que de l'exposé. Je lui suis extrêmement reconnaissante de suivre d'aussi près, malgré ses nombreuses occupations, les travaux de la parisienne que je suis et c'est un plaisir de pouvoir, en cette occasion, exprimer tout ce que je lui dois.

M. NIVAT, directeur du L.I.T.P. a manifesté depuis fort longtemps de l'intérêt pour ce travail. Sa caution scientifique, ses conseils, ses encouragements et son amitié me sont précieux et je suis tout particulièrement heureuse de le voir figurer dans ce jury.

Lors de nos nombreuses rencontres, à Nancy ou ailleurs, M. SINTZOFF du Philips Research Laboratory à Bruxelles n'a jamais manqué de me faire profiter de ses idées ou remarques, toujours constructives. Qu'il trouve ici l'expression de mes remerciements et de mon amitié.

B. LORHO m'a accueillie dans l'équipe Langages et Traducteurs de l'INRIA (alors IRIA) où j'ai trouvé les moyens de mener mes recherches et d'en développer des applications effectives, ceci grâce à ses travaux antérieurs. Je lui dis ici, très amicalement et très sincèrement, merci.

J.P. JOUANNAUD me fait le plaisir de siéger dans ce jury : ce "nouveau" nancéen a toujours montré de l'intérêt et de la sympathie pour les travaux de l'"ancienne" nancéenne que je suis. Je l'en remercie vivement.

Ph. DESCHAMP et M. MAZAUD de l'INRIA ont participé à cette recherche avec enthousiasme. Je tiens à leur dire tout le plaisir que j'ai à travailler avec eux et à voir, grâce à eux, se développer des applications pratiques de ces idées. Je remercie également mes autres collègues de l'équipe Langages et Traducteurs de l'ambiance sympathique dans laquelle s'effectuent ces recherches.

J'ai toujours trouvé à Nancy des oreilles attentives et des commentaires judicieux. J'exprime ici mes remerciements les plus amicaux à P. LESCANNE, J.P. FINANCE, J.L. REMY, A. QUERE et tous les autres nancéens.

Parmi les personnes qui se sont intéressées à ce travail, je mentionnerais (au risque d'omissions embarrassantes), J.V. GUTTAG du MIT qui m'a beaucoup apporté au cours de ses deux séjours à l'IRIA et R.M. BURSTALL qui m'a donné un sérieux 'coup de main' lors de mon passage à l'Université d'Edimbourg.

La frappe de cette thèse a été réalisée, avec patience et bonne humeur, par M. SUARD aidée de S. LOUBRESSAC. Le résultat de leur difficile travail est remarquable et je les remercie chaleureusement.

Enfin, je voudrais remercier mes parents de leur soutien moral et matériel et dire toute mon affection à Antoinette, Marie-Hélène et Sylvie qui, pendant nos week-ends et nos vacances ont compati à mes soucis pour ces choses si obscures ... et ont toujours su m'aider.

RESUME

Cette thèse propose une solution aux problèmes posés par la production automatique de traducteurs pour les langages de programmation.

En effet, si la construction automatique d'analyseurs syntaxiques est maintenant une technique très au point et largement utilisée, la conception et la réalisation des autres parties des traducteurs demeurent le plus souvent artisanales. Pour résoudre ce problème, il est indispensable de se donner des outils bien adaptés pour la description de la sémantique des langages de programmation.

Ce travail développe une théorie de la sémantique des langages de programmation qui est basée sur la notion de type abstrait algébrique : chaque langage est caractérisé par un tel type abstrait, c'est-à-dire une liste de noms de types, de noms d'opérations et un ensemble d'axiomes qui décrivent les propriétés des opérateurs et des instructions du langage. La signification d'une phrase est un schéma fonctionnel (ou terme) de ce type abstrait.

La traduction d'un langage L1 vers un langage L2 peut alors être considérée comme une représentation du type T1 associé à L1, par le type T2 associé à L2, c'est-à-dire comme un homomorphisme de T1 dans T2. Si cette traduction est correcte, cet homomorphisme doit conserver les axiomes de T1, c'est-à-dire les propriétés de L1.

Cette théorie a abouti au développement d'un générateur de compilateurs qui accepte en entrée les définitions syntaxiques et sémantiques des langages L1 et L2, ainsi que la description des choix de traduction spécifiés comme une représentation de T1 par T2.

Elle permet, en parallèle, d'établir une preuve de correction de la traduction choisie, selon la méthode évoquée ci-dessus.

P L A N

pages

Chapitre I. DEFINITION DES LANGAGES DE PROGRAMMATION ET PRODUCTION DE COMPILATEURS

I.1 Introduction 1

I.2 Contraintes contextuelles, évaluation de la sémantique des programmes 2

I.2.1 *Rappel sur les attributs de Knuth*

I.2.2 *Attributs et spécification de compilateurs*

I.3 Sémantique formelle, production, et preuve de compilateurs 5

I.3.1 *Les systèmes SIS et SEMANOL*

I.3.2 *Preuves de compilateurs*

I.4 Le système PERLUETTE 9

I.5 Plan de la thèse 12

Chapitre II. TYPES ABSTRAITS ALGEBRIQUES

II.1 Définitions et Exemples 13

II.2.1 *Notations*

II.2.2 *Algèbres initiales*

II.2.3 *Démonstrations par induction structurelle*

II.2 Problèmes de consistance et de complétude 20

II.3 Les erreurs 27

II.3.1 *Le problème*

II.3.2 *Spécification des erreurs par restrictions*

II.3.3 *Les E-algèbres*

II.3.4 *Conclusion*

II.4 Preuves de représentation 38

II.4.1 *Fonction d'abstraction*

II.4.2 *Fonction de représentation*

II.5 Conclusion 44

Chapitre III. DEFINITION SEMANTIQUE DE LANGAGES DE PROGRAMMATION A L'AIDE DE TYPES ABSTRAITS ALGEBRIQUES

III.1 Notion d'état et de modification 46

III.2 Primitive de définition des modifications 55

III.3 Le langage SEQFLEX 59

III.3.1 *Expressions*

III.3.2 *Instructions*

III.3.3 *Cas d'erreur*

III.4 Autre définition sémantique de SEQFLEX 68

III.5 Problèmes de consistance et de complétude 70

III.6 Conclusion 71

Chapitre IV. SEMANTIQUE DES PROCEDURES

IV.1 Le langage SIMPROC 74

IV.1.1 *Les expressions*

IV.1.2 *Les instructions*

IV.1.3 *Les fonctions sémantiques*

IV.1.4 *L'appel de procédure*

IV.2 Sémantique d'autres constructions courantes des langages de programmation 96

IV.2.1 *Remarque sur le profil des fonctions sémantiques*

IV.2.2 *Les fonctions*

IV.2.3 *Conclusion*

Chapitre V. SPECIFICATION DES CHOIX D'IMPLANTATION

V.1 Exemple de Structure Objet 104

V.2 Implantation de SEQFLEX 116

V.3 Implantation de SIMPROC 126

Chapitre VI. PREUVES DES CHOIX D'IMPLANTATION

VI.1 Preuve de représentation des Structures d'Information 133

VI.1.1 *Théorème fondamental*

VI.1.2 *La méthode de preuve*

VI.2 Preuves de l'implantation de SEQFLEX 147

VI.2.1 *Types et opérations ne faisant pas intervenir de modifications*

VI.2.2 *Représentation des opérations modifiables*

VI.2.3 *Invariants de représentation*

VI.2.4 *Modifications sans effet sur la représentation*

VI.2.5 *Modifications élémentaires*

VI.2.6 *Compositions de modifications*

CONCLUSION 185

BIBLIOGRAPHIE 187

CHAPITRE I

DEFINITION DES LANGAGES DE PROGRAMMATION ET PRODUCTION DE COMPILATEURS

I.1 - Introduction

Les compilateurs ont, dés les débuts de l'informatique, suscité de nombreux travaux de recherches, qu'il s'agisse de donner des techniques pour les réaliser [Iro 61, Eva 62], des outils d'aide à leur production [Fel 66, FG 68] (appelés, selon les auteurs, "générateurs de compilateurs", "compilateurs de compilateurs", "système d'écriture de traducteurs"), ou des méthodes pour les prouver [McC 61, MP 67, Lon 71].

Très tôt dans ce domaine, il est apparu qu'il était essentiel de se donner des outils de description formelle des langages de programmation [McC 63, NN 66, deB 67, Knu 68, BL 69, SS 71][(1)]. Ces nombreux travaux ont en commun une description bien formalisée de la syntaxe, le reste de la définition du langage étant dirigée par cette syntaxe [(2)], et qualifiée de sémantique. Sous ce dernier vocable, on a souvent réuni l'étude de problèmes aussi différents que : la spécification des contraintes contextuelles (par exemple les règles de déclaration des identificateurs), les outils qui permettent d'évaluer la valeur sémantique d'un programme (autrement dit sa signification) ; l'étude de cette signification et des domaines théoriques

---

(1) Il ne s'agit pas de donner ici un historique complet des recherches en compilation et en sémantique des langages de programmation : la liste des travaux cités est donc loin d'être exhaustive.

(2) Plus exactement il s'agit selon les cas de syntaxe "concrète" ou de syntaxe "abstraite" [McC 63, LLS 68]

auxquels appartiennent ces valeurs sémantiques. Cette confusion est compréhensible dans la mesure où la notion de syntaxe, en tant que grammaire à contexte libre, est bien délimitée. Un consensus s'est formé assez vite autour de son expression en B.N.F. (ou des variantes) et de nombreux outils de traitement automatique de ces grammaires existent aujourd'hui et sont largement utilisés sous le nom de constructeurs syntaxiques. On trouvera une bonne bibliographie dans [AU 77] et la présentation d'un tel système dans [Bou 77, Bou 80]. Ces systèmes permettent d'obtenir automatiquement l'analyseur syntaxique correspondant à une grammaire donnée.

Par contre l'écriture des autres composants des compilateurs se fait, encore aujourd'hui, le plus souvent de manière artisanale. Ceci est probablement dû à l'absence d'outils bien adaptés à la compilation, universellement adoptés, et permettant de décrire les contraintes contextuelles et la sémantique des langages. A ce sujet les auteurs de [MLB 76] parlent de 'Tour de Métababel' et insistent sur le besoin d'une notation rigoureuse et complète pour décrire un langage de programmation, notation qui serait directement acceptable par un générateur de compilateurs.

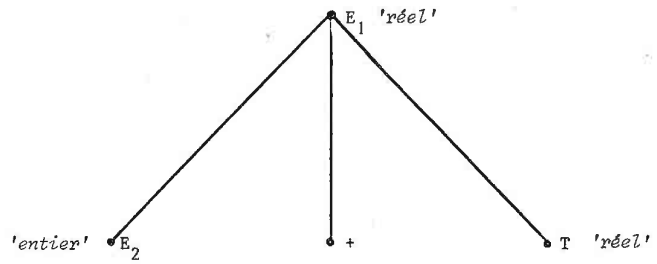
I.2 - Contraintes contextuelles, évaluation de la sémantique des programmes

Parmi les méthodes de description des contraintes contextuelles, les plus connues sont probablement les doubles grammaires [VW 69], les grammaires affixes [Kos 71] et les attributs de Knuth [Knu 68, Lor 74] appelés généralement attributs sémantiques.

On trouvera de bons exposés comparatifs dans [MLB 76] et [Lor 74]. Les attributs de Knuth ont donné lieu à de nombreuses réalisations dont le système DELTA [Lor 76] qui est utilisé dans le générateur de compilateurs présenté dans cette thèse. On va exposer ici leur rôle dans un tel générateur et quels sont les liens entre attributs de Knuth et sémantique formelle des langages de programmation. Précisons cependant que ce qui va être dit est valable pour les grammaires affixes et avec des variantes pour les doubles grammaires : dans le travail présenté ici, les attributs ont été un outil de vérification des contraintes contextuelles et d'évaluation de la sémantique des programmes, au même titre que les autres outils que sont les constructeurs syntaxique et lexicographique.

I.2.1 - Rappels sur les attributs

Une présentation claire et plus complète des attributs de Knuth est donnée dans [Liv 78] : un attribut est une valeur attachée à un noeud d'un arbre syntaxique. Un exemple d'arbre 'attribué' ou 'décoré' est donné dans la figure ci-dessous. Les noms des non-terminaux de la syntaxe sont indiqués en majuscule, les attributs sémantiques sont notés en italique. Ici, ces valeurs sont des chaînes de caractères indiquant des types :



La valeur associée à un noeud peut dépendre des valeurs associées aux noeuds fils : c'est alors un attribut synthétisé et ces valeurs sont calculées en parcourant l'arbre de bas en haut ; si la valeur dépend de la valeur associée au noeud père, c'est un attribut hérité.

Notons type(N) la chaîne de caractères correspondant dans l'exemple ci-dessus à un non-terminal N. En associant à la règle de grammaire :

$$E_1 \rightarrow E_2 + T$$

une définition d'attributs comme :

type(E_1) = <u>si</u> type(E_2) = *'entier'* <u>et</u> type(T) = *'entier'*
<u>alors</u> *'entier'* <u>sinon</u> *'réel'*

on définit le type d'une sous-expression ou d'une expression comme un attribut synthétisé.

Les informations ainsi associées à un noeud peuvent être très diverses : cette méthode permet de décrire le calcul de toutes les caractéristiques 'statiques' d'un texte, qu'elles soient de nature sémantique ou non. En ce sens, le qualificatif 'sémantique' qu'on accole généralement aux attributs est trop restrictif et source de confusion.

I.2.2 - Attributs et spécification de compilateurs

L'application la plus connue des attributs de Knuth est la spécification et la production de compilateurs. Par exemple, un compilateur classique peut se spécifier en utilisant trois définitions d'attributs : la première définit un attribut synthétisé qui construit la table des identificateurs du programme à partir des déclarations ; la deuxième décrit un attribut qui 'hérite' cette table de la racine vers les instructions ; la dernière décrit le code produit sous forme d'un attribut synthétisé (le code correspondant à un non-terminal est une composition du code des noeuds fils et dépend de la table héritée).

En plus de ces attributs principaux [Der 79] on aura besoin, en fonction du langage traduit et des caractéristiques du code engendré, d'un grand nombre d'informations annexes, du genre : type d'une sous-expression, état d'occupation des registres, informations nécessaires à l'allocation des registres, informations nécessaires à l'allocation (statique) en mémoire, etc. On a certainement facilité la tâche du programmeur qui réalise le compilateur, mais on est loin d'une production automatique de ce dernier à partir d'une définition formelle du langage à compiler.

On remarquera cependant que dans une telle description formelle, les attributs sont un bon moyen de spécification des contraintes contextuelles (comme le laisse supposer l'exemple donné en I.2.1) et du calcul de la valeur sémantique d'une phase. De plus ils ont l'avantage d'être directement utilisables puisque des systèmes permettant leur évaluation ont été réalisés. On y revient à la fin de ce chapitre.

I.3 - Sémantique formelle, production et preuves de compilateurs

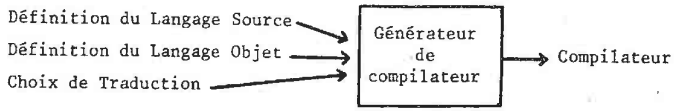
I.3.1 - Les systèmes SIS et SEMANOL

Il existe peu de systèmes acceptant en entrée une définition sémantique des langages de programmation. Nous présentons ici deux des plus connus : le système SIS, développé par P. Mosses [Mos 75, Mos 78] qui est basé sur la sémantique dénotationnelle, et le méta-langage SEMANOL conçu et implanté par E.K. Blum et son équipe [Blu 69, ABB 76] qui est basé sur une sémantique opérationnelle.

Le système SIS a pour données un ensemble d'équations sémantiques à la Scott-Strachey [Ten 76] qui définit un certain langage source : dans ce cadre théorique, la valeur sémantique d'un programme est une fonction (d'un état dans un autre), exprimée sous la forme d'une lambda-expression. Dans le système SIS, les équations sont écrites dans un langage applicatif, traduisible automatiquement dans le langage fonctionnel primitif LAMBDA, proposé par Scott. Le compilateur généré par le système est un traducteur du langage source en LAMBDA (qui est une version "pour ordinateur" du λ-calcul : la syntaxe est plus restrictive et il en existe un interprète). Le langage objet des compilateurs produits est toujours le même : les LAMBDA-expressions dont la taille et la vitesse d'évaluation peuvent poser des problèmes. Ce système est cependant très utile en raison du succès de la sémantique dénotationnelle : il fournit un moyen rapide d'exécuter des programmes d'un langage à partir de la définition dénotationnelle de celui-ci, définition qui existe pour de nombreux langages.

Quoique l'approche théorique de départ soit très différente de celle de SIS, le méta-langage SEMANOL offre des possibilités très voisines. Le programme SEMANOL qui décrit la syntaxe et la sémantique d'un langage, accepte comme donnée tout programme du langage et retourne le résultat de ce programme ainsi que la trace des calculs décrits dans le programme pour obtenir ce résultat. On voit qu'on n'a pas obtenu un traducteur mais plutôt un interprète du langage. La description sémantique est opérationnelle et consiste en un ensemble de définitions des structures de données et de contrôle du langage en terme des structures de données et de contrôle de SEMANOL qui sont, pour citer les auteurs, primitives et de haut niveau.

Ces deux systèmes donnent la possibilité d'exécuter des programmes d'un langage, ceci directement à partir de sa définition formelle. Mais on ne peut pas véritablement les considérer comme des générateurs de compilateurs au sens strict des mots. En fait, si on veut produire des compilateurs, il est indispensable de spécifier formellement, en entrée du système, non seulement le langage source, mais aussi le langage objet (c'est-à-dire le résultat de la traduction) et les choix de traduction. Très schématiquement, on veut un système du type suivant :



Il est souhaitable, sinon indispensable si on veut pouvoir spécifier rigoureusement et facilement les choix de traduction, d'utiliser le même formalisme pour définir le Langage Source et le Langage Objet : la traduction se décrit alors comme une fonction des valeurs sémantiques du langage source dans les valeurs sémantiques du Langage Objet, valeurs qui sont de même nature. De plus, si le formalisme est bien choisi, on peut envisager de prouver que cette fonction conserve bien les propriétés sémantiques du langage source, donc que la traduction est correcte.

I.3.2 - Preuves de compilateurs

La question de la vérification des compilateurs a déjà fait l'objet de nombreuses recherches. Elle reste d'actualité, et l'intérêt du monde industriel pour des compilateurs qui porteraient l'estampille, 'garanti correct' est indéniable. Il est intéressant de donner un historique des travaux dans ce domaine.

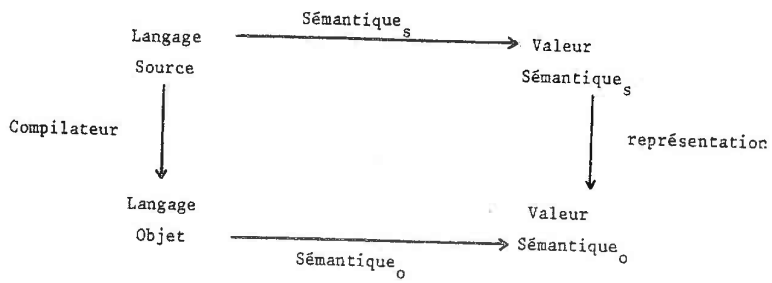
Une des premières preuves d'un algorithme de compilation a été donnée par Mac Carthy et Painter [MP 67] ; l'algorithme considéré était limité à des expressions. Dans [BL 69] Burstall et Landin suggéraient les premiers l'intérêt de méthodes algébriques et reprenaient l'exemple de la compilation des expressions. Milner et Weyhrauch [MW 72] donnent la preuve d'un algorithme de compilation pour un langage plus étoffé, comprenant des affectations

des conditionnelles et des boucles, en insistant sur l'intérêt de l'approche algébrique, que ce soit dans la définition des langages ou dans la structuration de la preuve. Dans ce même papier, ils proposent d'utiliser LCF pour développer automatiquement ce genre de preuves. Le souci d'automatiser ces preuves est d'ailleurs une constante dans tous ces travaux. Mentionnons que dans [Lon 72] London propose la preuve d'un compilateur d'un sous-ensemble de LISP.

Un article qui paraît fondamental dans ce domaine a été présenté par F.L. Morris à POPL en 1973 [Mor 73] ; nous allons le commenter un peu longuement.

Mais avant d'entrer dans des détails techniques, on peut faire deux remarques générales : la première est que tout au long de ces recherches le cadre théorique utilisé est devenu de plus en plus algébrique, et qu'il y a là une remarquable convergence ; la deuxième est que l'article de Lockwood Morris est paru en 1973, qu'il comportait, nous semble-t-il, les bases théoriques d'une méthode d'écriture de compilateurs et de preuve de ceux-ci. Or jusqu'ici, les compilateurs ne sont toujours pas démontrés corrects ...

Dans l'approche présentée par F.L. Morris les langages de programmation sont considérés comme des algèbres hétérogènes dont les ensembles sont les domaines syntaxiques et dont les opérations sont les différentes compositions syntaxiques (par exemple, à partir d'un identificateur et d'une expression on obtient une instruction par l'opération affectation, etc). La compilation se décrit comme un homomorphisme du langage source vers le langage objet. A ces langages, sont associées des fonctions sémantiques qui, à chaque programme associent une signification. La preuve d'un compilateur revient a démontrer que le diagramme de la page suivante commute :



La flèche de droite nommée 'représentation' exprime le fait que la signification du programme source et de sa traduction ne sont pas, en général, identiques mais que la seconde représente "d'une manière correcte" la première. La sémantique utilisée par Lockwood Morris est dénotationnelle. Ce choix présente l'inconvénient de rendre l'expression de la représentation compliquée [Mil 77], et la méthode peu accessible aux programmeurs de compilateurs. Par ailleurs des notations trop mathématiques et une présentation dense rendent cet article difficilement lisible. Ce défaut, ainsi que le fait que ces idées n'étaient pas accompagnées d'une réalisation pratique (générateur de compilateurs, système de preuves) explique sans doute leur peu de retombées pratiques.

Il nous semble plus réaliste d'utiliser pour spécifier et prouver des représentations un formalisme bien adapté et bien connu, qui est celui des types abstraits algébriques : d'où l'idée de base dans cette thèse, qui est d'avoir pour valeurs sémantiques les termes d'un type abstrait, idée qui a débouché sur la conception d'un système, PERLUETTE(1), qui est développé actuellement à l'INRIA.

Signalons que récemment [TWW 79] les idées de F.L. Morris ont été reprises avec une définition algébrique, au sens de [ADJ 75] de la sémantique. La présentation théorique gagne certainement en élégance, mais la lisibilité et les possibilités d'application ne semblent pas améliorées de manière significative.

I.4 - Le système PERLUETTE

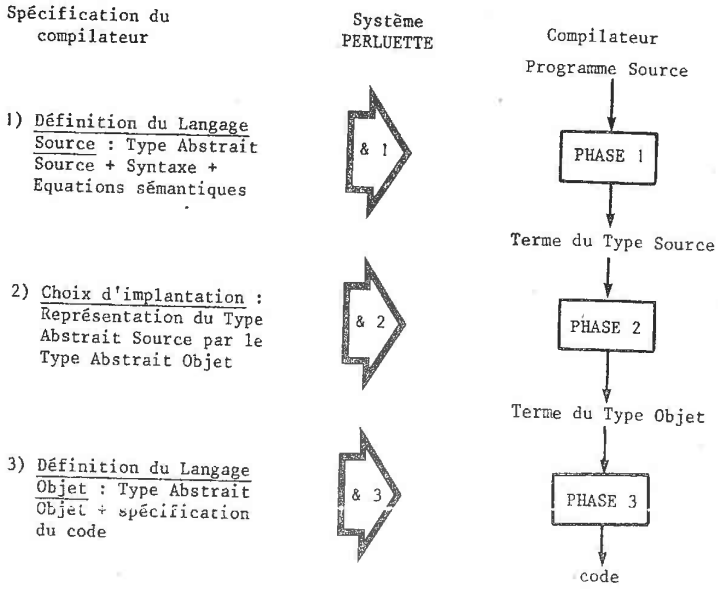
Ce système permet d'obtenir un traducteur à partir de la définition d'un langage source, de celle d'un langage objet et de la description des choix d'implantation. Ces choix d'implantation sont exprimés comme la représentation d'un type abstrait algébrique par un autre. Ils peuvent donc être prouvés corrects et on vérifie ainsi que la traduction choisie préserve

(1) Production Elégante et ReeLlement aUtomatiquE de TraducTEurs.

bien la signification du programme source. Pour l'instant, ces preuves sont faites à la main, mais il est envisageable de les automatiser en utilisant ou en aménageant un système déjà existant comme L.C.F. [GMW 77] ou AFFIRM [Mus 77

Pour définir la sémantique d'un langage de programmation on lui associe un type abstrait algébrique (c'est-à-dire un ensemble d'opérations et d'axiomes ce qui permet de spécifier rigoureusement les propriétés des opérations et des traitements du langage. La signification d'un programme de ce langage peut s'exprimer comme une composition d'opérations du type abstrait, c'est-à-dire un terme. Cette correspondance, Programme → Terme, est décrite au moyen d'équations sémantiques qui donnent la valeur sémantique d'une phrase en fonction des valeurs sémantiques des composants syntaxiques de cette phrase.

Le schéma général du système est le suivant :



Le traducteur produit comporte trois étapes : la première produit un premier texte intermédiaire qui est un terme du type source. Ce terme est réécrit par la deuxième étape sous forme d'un terme du type objet. Enfin la troisième étape produit le code à partir de ce terme.

Dans la version actuelle du système PERLUETTE, les équations sémantiques relatives au Langage Source sont programmées sous forme d'attributs sémantiques synthétisés écrits en LISP. L'évaluation de ces attributs (ainsi que l'analyse syntaxique) est effectuée par le système DELTA précédemment réalisé au Laboria ; les composants de ce système, le constructeur et l'évaluateur ont donc été respectivement intégrés dans la première partie du système (&1) et la première étape des traducteurs (PHASE 1).

Les attributs sont utilisés dans le sens mentionné à la fin de I.2, c'est-à-dire comme outil de spécification dans la définition du Langage Source. Pour éviter des inefficacités l'étape 1 effectue toutes les vérifications statiques des programmes, vérifications également décrites par attributs (cette question est étudiée à la fin de IV.1.3). Le choix de LISP pour l'écriture des attributs permet [Des 80] une réalisation facile et efficace des étapes suivantes : en effet, le résultat de la première étape de traduction, le 'Terme du Type Source' est une forme LISP.

La PHASE2 du traducteur a pour rôle d'évaluer cette forme en utilisant un ensemble de fonctions LISP qui ont été produites à partir de la description des choix d'implantation par &2. Chaque nom d'opération du type Source apparaissant dans le premier texte intermédiaire est considéré alors comme le nom d'une fonction LISP, dont le résultat est sa traduction en terme d'opérations du type objet. Comme on le verra dans les exemples qui suivent le deuxième texte intermédiaire est proche du code à générer. La PHASE 3 du traducteur doit, selon le même principe que la PHASE2 réécrire ce texte comme du code en effectuant des tâches comme l'allocation des registres ou la suppression d'instructions inutiles, si cela n'a pas été fait dans la deuxième étape.

On trouvera une présentation complète de ce système et de son utilisation dans la thèse de Deschamp [Des 80]. Pour ce qui est de la première partie (&1) un manuel d'utilisation est disponible [DM 78].

Plusieurs exemples de compilateurs ont été spécifiés selon cette méthode [Den 76, Gau 77, GDM 78, GP 79, Gau 80]. Les langages sources étudiés comportent des types de données complexes comme les piles et les séquences [Den 76, Gau 77], des procédures et des tableaux à bornes fixes [GDM 78], des blocs et des tableaux à bornes variables [Gau 80]. Ils correspondent donc à des applications réalistes.

Une expérience de preuve automatique du compilateur présenté dans [Gau 77] a été menée à bien, pour les expressions, à l'aide du système de vérification de D. Musser [Mus 77].

I.5 - Plan de la thèse

Cette thèse présente donc une sémantique formelle des langages de programmation qui utilise la notion de type abstrait algébrique. Cette sémantique est utilisée pour spécifier et prouver des compilateurs.

On trouvera dans le chapitre II un rappel sur les types abstraits algébriques, qui a été limité aux notions utilisées par la suite. Dans le chapitre III, l'application des types abstraits à la définition sémantique de langages de programmation est présentée à partir de deux exemples simple. Le chapitre IV est consacré à l'étude de la sémantique des procédures. Le chapitre V présente la sémantique d'un langage de bas niveau et deux exemples de spécification de traducteurs sont donnés : on reprend deux langages sources étudiés respectivement au chapitre III et au chapitre IV. Le dernier chapitre est consacré à la méthode de preuves : les résultats théoriques sur lesquel elle est fondée sont établis et un exemple de preuve de traduction (une de celle donnée au chapitre V) est donné complètement.

CHAPITRE II

TYPES ABSTRAITS ALGEBRIQUES

II.1 - Définitions et exemples

II.1.1 - Notations

Il semble maintenant généralement admis [BG 77, GH 78, GTW 78] de considérer un type abstrait comme une théorie algébrique typée dont la "présentation" est la donnée :

- d'une liste de noms de types (ou sortes) où on distingue un 'type d'intérêt', qui est le type que l'on veut définir ;
- d'une liste de noms d'opérations sur ces types ; à chacun de ces noms est associé un profil (liste des domaines, et co-domaine). Ces symboles permettent de construire des termes (ou schémas fonctionnels) du type abstrait ; le type d'intérêt doit apparaître dans chaque profil ;
- d'une liste d'axiomes (ou lois) qui expriment les relations entre les opérations. Ces axiomes sont des équations entre termes composés de noms d'opérations et de variables. Celles-ci sont universellement quantifiées, de façon implicite.

Exemple 1 : Le type Booléen.
Une présentation possible de ce type est la suivante :

type Bool ;
operations
  ( ) → Bool : vrai, faux ;
  (Bool) → Bool : ¬ ;
  (Bool, Bool) → Bool : ∧, ∨, ⊃, eq ;
axiomes
  soit b, b' ∈ Bool ;
  ¬ vrai = faux ;
  ¬ ¬b = b ;
  ∧ (vrai, b) = b ;

```
  ∧ (faux, b) = faux ;
  ∧ (b, b') = ∧ (b', b) ;
  ∨ (b, b') = ¬∧ (¬b, ¬b') ;
  ⊃ (b, b') = ∨ (¬b, b') ;
  eq(b,b') = ∧ (⊃(b,b'), ⊃ (b',b) ;
fin (1) ;
```

On voit que la partie operations de la présentation définit un langage, celui des "termes de type Bool" et que la partie axiomes définit une ou des relations de congruence sur les termes de ce langage. S'il n'y a pas d'axiomes, tous les termes du type abstrait sont différents. Dans le cas général, où on a des axiomes, ceux-ci définissent des classes d'équivalence dans le langage des termes.

Un deuxième exemple bien connu est le type ensemble d'entiers [HOA 72, GH 78]. On suppose défini le type Entier et les opérations qu'on veut utiliser sont : ajouter, détruire un entier ou tester sa présence dans un ensemble. La présentation de ce type peut s'écrire (on pourrait se donner d'autres axiomes) :

Exemple 2 :

```
type Ens, Bool, Ent ;     N.B : Ens est le nom du type d'intérêt.
operations
  ( ) → Ens : vide ;
  (Ens, Ent) → Ens : ajouter, detruire ;
  (Ens, Ent) → Bool : appartient ;
axiomes
soit s ∈ Ens, i, i' ∈ Ent ;
  appartient(vide, i) = faux ;
  appartient(ajouter(s,i),i') = si eg(i, i') alors
                    vrai sinon appartient(s,i') ;
  detruire(vide,i) = vide ;
```

(1) Dans la suite, pour des raisons de lisibilité, nous utilisons une notation infixée pour les opérations binaires de ce type abstrait, avec les règles de priorités habituelles. De plus, les formules 'p=vrai' seront souvent abrégées en 'p'.


```
  detruire(ajouter(s,i),i') = si eg(i,i') alors
                    detruire(s,i') sinon ajouter(detruire(s,i'),i) ;
fin ;
```

On voit apparaître dans cette présentation des équations conditionnelles. Comme il est noté dans [GTW 78] cela revient à ajouter à tout type T une opération si-alors-sinon vérifiant les axiomes :

```
si-alors-sinon(vrai, t, t') = t
si-alors-sinon(faux, t, t') = t'
```

On peut donc utiliser ce type d'équations sans sortir des théories algébriques habituelles.

Les trois premiers axiomes sont des définitions (récursives ou non) simples. Par contre, le quatrième peut être moins compréhensible pour le lecteur peu familier avec les types abstraits algébriques. Afin de clarifier les choses, considérons le terme :

```
detruire(ajouter(ajouter(ajouter(vide,2),3),2),2)
```

qu'on a parenthésé pour le rendre plus lisible. En appliquant l'axiome 4, ce terme est égal à :

```
= detruire(ajouter(ajouter(vide,2),3),2)
```

car on est dans le cas ou eg(i,i') est vrai.
De même on a :

```
= ajouter(detruire,(ajouter(vide,2),2),3)
= ajouter(vide,3)
```

qui est bien le résultat qu'intuitivement on désirait.

On peut remarquer [GH78] qu'en changeant simplement ce quatrième axiome en :

```
detruire(ajouter(s,i),i') = si eg(i,i') alors
                    s sinon ajouter(detruire(s,i'),i)
```

on obtient un type très différent qui est appelé "Bag" ou "Multiset" dans la littérature Anglo-saxonne et qui correspond à des ensembles où les éléments peuvent avoir plusieurs occurrences. En particulier dans le type Ens,

on a la propriété

appartient(détruire(ajouter(ajouter(vide,i),i),i),i) = faux

alors que dans le type Bag l'expression de gauche est égale à vrai.

On voit que la présentation d'un type abstrait algébrique est d'une syntaxe relativement simple. Le langage de base comporte trois primitives :

. la composition de fonctions (avec parenthèses pour ajouter à la lisibilité) ;

. le signe = ;

. des variables libres.

De plus, on considérera généralement que le type Booléen (tel qu'il est donné dans l'exemple 1) est prédéfini et que pour chaque type on a une opération si-alors-sinon. Enfin, en cas de besoin, on conviendra que certains types : Entier, Caractère, Chaîne de caractères, sont prédéfinis.

Les axiomes sont de la forme

$$g(x^*) = d(x^*)$$

où x^* est une liste de variables libres typées, g est une composition des noms d'opérations du type abstrait, d est une composition de ces noms et de conditionnelles.

Si la syntaxe de telles spécifications est élémentaire, leur sémantique est moins immédiate et demande à être étudiée soigneusement. C'est ce qui va être fait dans le reste de ce chapitre.

II.1.2 - Algèbres initiales

On rappelle ici très brièvement quelques définitions et résultats tirés de [GTW 78], afin de montrer plus formellement ce qu'est le type abstrait défini par une présentation.

Soit Σ une liste de noms d'opérations et leurs profils. On note T_Σ l'ensemble des termes correspondants. (Un terme ne contient pas de variables

Une Σ-algèbre est une algèbre hétérogène telle que : à chaque nom de type apparaissant dans Σ, correspond un ensemble de cette algèbre ; à chaque opération de Σ correspond une opération de cette algèbre, et les profils sont conservés.

Considérons par exemple la liste des opérations du type Bool. Notons la Σ_b. Ici les Σ_b-algèbres sont homogènes puisqu'il n'y a qu'un seul nom de type : Bool. On va donner deux exemples de Σ_b-algèbres :
(attention, on ne tient pas compte des axiomes, pour l'instant)

Exemple 1.1

Soit E l'ensemble T_{Σ_b} ; les opérations de Σ_b permettent de construire des termes de T_{Σ_b} à partir d'autres termes : E muni de ces opérations est un cas particulier de Σ_b-algèbre.

Exemple 1.2

Soit maintenant B = {vrai, faux} et les opérations usuelles sur les booléens notées comme dans Σ_b. B est également une Σ_b-algèbre.

On remarque qu'il est toujours possible de considérer T_Σ comme une Σ-algèbre. De plus, il est connu que T_Σ est une algèbre initiale dans la classe des Σ-algèbres, c'est-à-dire qu'il existe un homomorphisme unique de T_Σ vers toute Σ-algèbre.

Considérons maintenant un ensemble E de Σ-équations. (C'est-à-dire d'équations où apparaissent les opérations de Σ et des variables). Parmi les relations de congruences sur T_Σ, compatibles avec E, on note $\equiv_E$ la plus petite(2).

Soit $T_\Sigma/\equiv_E$ la Σ-algèbre quotient de T_Σ par $\equiv_E$ (on travaille sur les classes d'équivalence de termes au lieu de travailler sur les termes). Il est démontré que c'est une Σ-algèbre, et qu'elle est initiale dans la classe des Σ-algèbres qui vérifient E.

[GTW 78] pose comme définition que le type abstrait algébrique défini par une présentation Σ,E est cette algèbre initiale, qui est notée $T_{\Sigma,E}$.

Si on essaie de se dégager des détails techniques, cette définition revient à dire que : les objets du type abstrait algébrique sont des classes d'équivalence de termes, la relation d'équivalence choisie est "minimale" c'est-à-dire qu'on satisfait les équations E et rien de plus.

---

(2) Intuitivement, cela veut dire qu'un couple (t,t') ne vérifie $\equiv_E$ que si c'est indispensable pour ne pas avoir de contradiction : si $t \equiv t'$ n'est pas une conséquence de E alors on considère que t et t' sont différents. Une contradiction est introduite si, par exemple, on convient pour deux termes du type d'intérêt que $t \equiv t'$, et s'il existe une opération f, dont le résultat n'est pas le type d'intérêt, telle qu'on n'ait pas $f(t) \equiv f(t')$ dans le type résultat.

Reprenons l'exemple des Booléens : appelons E_b les équations données dans l'exemple 1. E_b définit deux classes d'équivalence : V, celle des termes qui sont égaux à vrai et F, celle des termes qui sont égaux à faux. Les opérations $\neg$, $\wedge$, etc sont définies sur ces deux classes d'équivalence et d'après E on a :

$$\neg V = F$$
$$\neg\neg V = V \Longrightarrow \neg F = V$$

etc ...

On peut ainsi montrer que T_{Σ_b}, E_b est isomorphe à l'algèbre bien connu des Booléens et qu'on a bien spécifié le type voulu.

C'est le résultat essentiel apporté par cette formulation très mathématique des types abstraits : une spécification (Σ, E) est correcte si $T_{\Sigma,E}$ est isomorphe à un modèle mathématique connu du type dont on veut donner une spécification. Le problème est qu'on n'a pas toujours un modèle mathématique (dans ce travail en particulier, le modèle est "informatique") auquel se ramener. Dans ce cas, l'utilisation de ce cadre algébrique se justifie moins et d'autres méthodologies peuvent s'avérer suffisantes.

II.1.3 - Démonstrations par induction structurelle

Une méthode pour vérifier, au moins partiellement, la spécification d'un type abstrait est de démontrer les propriétés qu'on attend de ce type à partir de la spécification.

Du fait qu'on a une notation indépendante de toute implémentation ou de tout environnement, les preuves sont généralement simples et suivent toutes le même schéma [SW 75, GH 78] :

Supposons qu'on ait un type T, muni d'opérations à résultat dans T : $f_1, f_2, \ldots f_n$. Soit P une propriété des valeurs de T. Alors si $\forall i = 1, \ldots n$, le fait que P soit vrai pour les opérandes de type T de f_i entraine que P est vrai pour le résultat de f_i, P est vrai pour toutes les termes du type T.

Ce type de raisonnement est appelé "induction structurelle" ou "induction sur le type". On peut l'utiliser dès qu'on a une spécification du type indépendante de son utilisation : on se contente de fournir au

programmeur les noms et caractéristiques des opérations. Ici ce sont des axiomes algébriques, dans d'autres cas ce sont des Pré et Post-conditions, une définition à partir de types prédéfinis, une représentation ...

Exemple 2.1 :

On veut montrer que $\forall s \in$ Ens, $\forall i \in$ Ent on a :

appartient(detruire(s,i),i)=faux

les opérations à résultat dans Ens sont : vide, ajouter, detruire.

a) soit s = vide

appartient(detruire(vide,i),i)=
appartient(vide, i)=faux

b) soit s = ajouter(s',j) tel que
$\forall i$ appartient(detruire(s',i),i)=faux

on a :

appartient(detruire(ajouter(s',j),i),i)=
appartient(si eg(j,i) alors detruire(s',i)
sinon ajouter(detruire(s',i),j)),i)

on fait ce qu'on appelle une "analyse par cas"
si eg(j,i) on a
= appartient(detruire(s',i),i)=faux
par hypothèse de récurrence
sinon eg(i,j)=faux et on a :

= appartient(ajouter(detruire(s',i),j),i)
= si eg(j,i) alors vrai
sinon appartient(detruire(s',i),i)
= appartient(detruire(s',i),i)=faux

par hypothèse de récurrence.

c) soit s = detruire(s',j). On va montrer que s vérifie P(s) avec
P(s) = (s = vide) $\vee$ (s = ajouter(s'',k) $\wedge$ P(s''))
ce qui ramène aux cas précédents.

On travaille également par récurrence : si s' vérifie P

on a :

- s = detruire(vide,j) = vide

ou

- s = detruire(ajouter(s",k),j) = <u>si</u> eg(k,j)
  <u>alors</u> detruire(s",j) <u>sinon</u>
  ajouter(detruire(s",j),k)

si eg(k,j) est vrai, s vérifie la propriété par récurrence, sinon s est bien de la forme ajouter (s',k).

II - Problèmes de consistance et de complétude

S'il est relativement simple de faire des démonstrations sur un type abstrait algébrique donné, il est nettement plus difficile d'établir la spécification d'un tel type. En particulier, une fois qu'on a écrit une spécification, comment répondre aux questions suivantes : n'y a-t-il pas d'axiomes contradictoires (problème de consistance) ? A-t-on donné un nombre suffisant d'axiomes pour décrire toutes les propriétés du type abstrait que l'on voulait spécifier au départ, et dont on n'a, en général, qu'une idée intuitive (problème de complétude) ?

Un exemple d'inconsistance serait, pour l'exemple 2, de trouver un terme s de type Ens et un terme i de type Ent tels qu'on puisse démontrer à la fois que :

appartient(s,i) = vrai
appartient(s,i) = faux.

Un exemple d'incomplétude serait, toujours pour l'exemple 2, de trouver un terme s de type Ens et un terme i de type Ent tels qu'on ne puisse reécrire appartient(s,i) sous forme d'une constante booléenne.

On a vu que si on connait, avant d'écrire les axiomes, un modèle mathématique standard de ce qu'on veut spécifier, une méthode de vérification est de considérer un modèle (l'algèbre initiale) du type abstrait et de montrer qu'il est isomorphe au modèle standard.

Si on n'a pas un tel modèle, il faut vérifier la consistance et la complétude du système d'axiomes. Il est bien connu que ces deux problèmes sont indécidables.

On peut cependant donner des conditions suffisantes pour qu'une axiomatisation soit consistante et complète.

Pour ce qui est de la consistance, une condition suffisante est que si on considère les axiomes comme des règles de reécriture (on les utilise de la gauche vers la droite), ces règles de reécritures sont décroissantes (le nombre d'opérations du type abstrait apparaissant en partie droite est inférieur ou égal au nombre des opérations apparaissant en partie gauche) et vérifient la propriété de Church-Rosser : c'est-à-dire qu'étant donné un terme quelconque, si on lui applique des règles de reécriture tant que c'est possible, on obtient toujours le même terme à la fin, quelles que soient les règles utilisées et leur ordre d'application. Une méthode pour vérifier cette propriété est donnée dans [KB 70].

La notion de complétude, dans le cas de types abstraits "informatique" demande à être examinée avec soin : en mathématiques, une théorie T et par suite le système d'axiomes qui la définit, est dite complète si elle est consistante et si, pour toute formule P sans variable, soit P soit ¬P est un théorème de T [Sch 67]. Cette notion ne convient pas pour les types abstraits algébriques : en effet, elle entraine que toute égalité entre termes doit pouvoir être prouvée soit vraie, soit fausse. Si on reprend l'exemple 2, il ne vérifie pas cette propriété : on considère les deux termes :

t1 = ajouter(ajouter(vide, 1), 2)

et

t2 = ajouter(ajouter(vide, 2), 1)

Il n'y a pas d'axiome du type Ens permettant de reécrire l'un comme l'autre car aucun axiome ne comporte la composition de deux 'ajouter' ; on ne peut donc démontrer l'égalité de ces deux termes. Par ailleurs, pour démontrer leur inégalité il faudrait trouver un entier i tel que :

appartient(t1, i) = vrai
appartient(t2, i) = faux

ou le contraire.

Or, il est immédiat de démontrer, en utilisant les axiomes sur 'appartient', que pour tout i :

appartient(t1, i) = appartient(t2,i)

Par conséquent, on a un système d'axiomes non complet, au sens des mathématiques[3].

Cependant, la spécification donnée pour le type Ens est parfaitement satisfaisante si on se refère à la définition informelle de I.1 : on a défini les propriétés des opérations : ajouter, detruire et appartient "suffisamment". L'égalité entre terme de type Ens n'est pas indispensable dans ce cas.[4]

Guttag et Horning dans [GH 78] définissent un nouveau genre de complétude, la "complétude suffisante". la suite de ce paragraphe expose brièvement les définitions et les résultats donnés dans [GH 78].

Définition 1 :

Soit <T,F,A> une présentation d'un type abstrait t avec :

$T = t_1, \ldots t_m$ est la liste de noms de types,

$F = f_1, \ldots, f_n$ est la liste de noms d'opération,

$A = a_1, \ldots, a_p$ est l'ensemble des axiomes,

$t \in T$ est le type d'intérêt et apparaît au moins une fois dans le profil de chaque f_i.

On appelle opérations externes les opérations de F dont le co-domaine n'est pas t. On note O l'ensemble de ces opérations.

On appelle opérations internes les opérations de F dont le co-domaine est t. On note S l'ensemble de ces opérations.

(3) On a donc plusieurs modèles (algèbres) possibles pour ces types abstraits selon que l'on considère ces termes comme égaux ou non. Rappelons que si on prend comme sémantique du type abstrait l'algèbre initiale, ces deux termes sont considérés comme différents.

(4) Sur la nécessité de l'incomplétude, et sur la complétude et la consistance considérées comme un idéal excessif, on peut consulter M. Sintzoff (communication privée), Lao-Tseu et I. Lakatos : 'Proofs and Refutations', Combridge Univ. Press, 1976.

Intuitivement, on dit que A est une axiomatisation suffisamment complète de t si tout terme de <T,S+O,A>, commençant par une opération de O, peut se réécrire, en utilisant A, sous forme d'un "terme extérieur". Un terme extérieur est un terme qui ne contient pas d'opérations de S+O, par exemple une constante d'un des types $\tau_i \neq t$.

Dans l'exemple 2 on voit qu'on a :

T = Ens, Ent, Bool

S = vide, ajouter, detruire

O = appartient

t = Ens.

La complétude suffisante revient à montrer que $\forall s \in$Ens, $\forall i \in$Ent le terme appartient(s,i) peut être montré égal soit à vrai soit à faux. La démonstration se fait par induction structurelle, en utilisant le fait (montré en I.2) que tout terme de type Ens peut se réécrire soit vide, soit ajouter(s,j).

La définition formelle de la complétude suffisante est un peu compliquée du fait qu'on doit définir précisément la notion de terme extérieur : étant donné la présentation <T,F,A> de la Définition 1, on considère la liste de noms de types :

$$I = T - \{t\}$$

Notons T_I l'ensemble des termes de type $\tau_i \neq t$ où n'apparaissent pas les opérations de F. Il s'agit des termes des types prédéfinis.

Définition 2 :

On dit que A est une axiomatisation suffisamment complète de <t+I, S+O,A> si et seulement si, pour tout terme de la forme

$$f_i(t_1, \ldots, t_{ni})$$

tel que $f_i \in O$, il existe un théorème, démontrable à partir de A, par des reécritures, de la forme :

$$f_i(t_1, \ldots, t_{ni}) = u$$

ou $u \in T_I$

A est <u>consistant</u> si pour tout $f_i(t_1,\dots t_{ni})$, u est unique.

Il est montré que la complétude suffisante d'un ensemble d'axiomes est, en général, indécidable. Cependant, il est possible de donner des conditions suffisantes pour cette propriété si on accepte quelques restrictions sur les systèmes d'axiomes considérés.

Afin de donner une idée de ces restrictions, on va devoir exposer quelques définitions et préciser quelques notations.

Comme dans la partie I de ce chapitre on notera les axiomes

$$g(x^*) = d(x^*)$$

où g est une composition d'opérations de S+O, d'une composition de ces opérations avec éventuellement des conditionnelles, x^* est une liste de variables libres typées.

<u>Définition 3</u> :

Un axiome $g(x^*) = d(x^*)$ est d'imbrication décroissante (ou plus brièvement <u>décroissant</u>) si $IMBR(g(x^*)) > IMBR(d(x^*))$ où IMBR(e) est le degré maximum d'imbrication des opérations de S+O dans l'expression e. Plus précisément :

$IMBR(t) = 0$ si et seulement si t est un terme extérieur;

$IMBR(x) = 0$ pour tout nom x de variable libre ;

$IMBR(f(e_1,\dots, e_n)) = 1+\underset{i=1,\dots,n}{\text{Max}}\ IMBR(e_i)$

si f est une opération de S+O ;)

$IMBR(f(e_1,\dots, e_n)) = \underset{i=1,\dots,n}{\text{Max}}\ IMBR(e_i))$

si f n'est pas une opération de S+O.)

Ce dernier cas s'applique pour les conditionnelles :

$IMBR(\underline{si}\ b\ \underline{alors}\ e_i\ \underline{sinon}\ e_2) = \text{Max}(IMBR(b), IMBR(e_1), IMBR(e_2))$.

Comme le nom du type d'intérêt doit apparaître au moins une fois dans le profil de toute opération de S+O, on peut faire les remarques suivantes :

- On appelle <u>constantes</u> du type t les opérations de S+O qui n'ont pas d'opérande de type t. Elles sont à résultat dans t, et il en existe au moins une dans S+O.

- Les opérations externes ont au moins un opérande du type t. Pour ne pas compliquer inutilement les définitions et théorèmes qui suivent on se donne les restrictions suivantes :
  - . une opération de O n'a qu'un opérande de type t (les résultats qui suivent se généralisent au cas où il y a plusieurs opérandes de type t),
  - . cet opérande est le premier.

<u>Définition 4</u> :

On dit que A est une axiomatisation <u>sure</u> de <T, S+O,A> <u>si</u>

$\forall o\in O$, $\forall s\in S$ il existe un axiome de la forme $o(s(x^*), y^*) = d(x^*, y^*)$ décroissant, ou alors il existe une suite d'axiomes :

$o(s(x^*),y^*) = d_1(x^*, y^*)$,

$d_1(x^*, y^*) = d_2(x^*, y^*)$

...

$d_n(x^*, y^*) = d(x^*, y^*)$

et $o(s(x^*), y^*) = d(x^*, y^*)$ est décroissant.

<u>Théorème</u> si A est sure, alors <T,S+O,A> est suffisamment complet.

La démonstration se fait par récurrence sur la valeur de $IMBR(f(t_1,\dots, t_n))$ où f est une opération de O.

Un terme commençant par une opération de O est forcément de la forme :

$o(s(t_1,\dots t_n), u_1,\dots, u_n)$ $o\in O$, $s\in S$.

Supposons $IMBR(t_i)= 0$ et $IMBR(u_i) = 0$

Par définition de IMBR

$$IMBR(o(s(t_1,\dots,t_n), u_1,\dots u_m)) = 2$$

et par hypothèse sur A on a une réécriture

$o(s(t_1,\dots, t_n), u_1,\dots,u_m) = d(t_1,\dots t_n, u_1,\dots u_m)$

telle que :

$$IMBR(d(t_1,\dots t_n, u_1,\dots u_m))<2.$$

De part les conventions prises précédemment, si le degré d'imbrication d'un tel terme est strictement inférieur à 2, il est nul : en effet, ce terme est de type différent de t ; si une opération de S y apparaît, elle est forcément en opérande d'une opération de O, donc le degré d'imbrication est supérieur ou égal à 2 ; si une opération de O y apparaît, elle a

forcément un opérande de type t, donc commençant par une opération de S.

Donc, tout terme u de type différent de t tel que IMBR(u) < 2 vérifie IMBR(u) = 0. C'est donc un terme extérieur ($u \in T_I$).

On a ainsi montré l'origine de la récurrence : tout terme commençant par une opération de O est de degré d'imbrication supérieur ou égal à 2. Si ce degré d'imbrication est égal à 2, ce terme se reécrit sous forme d'un terme extérieur.

Supposons maintenant que tout terme commençant par une opération de O, de degré d'imbrication inférieur à p puisse se reécrire sous forme d'un terme extérieur, et considérons le terme :

$$o(s(t_1,\ldots t_n), u_1,\ldots, u_m)$$

de degré d'imbrication égal à p.

D'après les propriétés de A on a :

$$o(s(t_1,\ldots, t_n), u_1,\ldots, u_m) = d(t_1,\ldots t_n, u_1,\ldots u_m)$$

avec $IMBR(d(t_1,\ldots t_n, u_1,\ldots u_m)) < p$

Si d commence par une opération de O, on peut appliquer l'hypothèse de récurrence et ce terme se reécrit sous forme d'un terme extérieur.

Si ce n'est pas le cas, on peut appliquer l'hypothèse de récurrence à tous les opérandes commençant par une opération de O (ou opérandes commençant par une opération de O d'opérandes ne commençant pas par une opération de O) de la première opération de d : en effet, par définition de IMBR leur degré d'imbrication est inférieur à p. On peut donc également se ramener à un terme extérieur.

Cependant cette condition est très restrictive.

La plupart des axiomatisations courantes ne sont pas sures : par exemple, celle donnée pour l'exemple 2 ne l'est pas car on n'a pas d'axiome avec en partie gauche :

appartient(detruire(s,i),j)

Cependant, l'axiomatisation est bien suffisamment complète car on a montré que tout terme de la forme "detruire(s,k)" peut se réécrire en utilisant les opérations "vide" et "ajouter".

D'où l'idée assez naturelle de distinguer dans S des opérations convertibles.

Définition 5 :

soit le type abstrait <T,S+O+{f},A> avec f de profil $t \times t_{i1} \times \ldots \times t_{in} \to t$. Si pour toute expression de la forme $f(u,v^*)$ il existe un théorème démontrable par A de la forme $f(u,v^*) = z$ où z ne contient pas d'occurrence de f, on dit que f est convertible à S.

Théorème 6 :

A est suffisamment complet si il existe une partition de S en deux ensembles C (les constructeurs) et E (les extensions) telle que toutes les constantes sont contenues dans C et :

- $\forall c \in C$ et $o \in O$ il existe un axiome $o(c(x^*),y^*) = z$ (où x^* et y^* sont des listes de variables libres) vérifiant les propriétés de la définition 4.
- On peut ordonner les éléments de E : $e_1,\ldots e_m$ de manière à ce que e_1 soit convertible à C, e_2 soit convertible à $C+\{e_1\}$, etc...

Ce théorème permet de vérifier de manière purement syntaxique la complétude suffisante de la plupart des types abstraits usuels. De plus, il fournit une méthodologie pour construire des axiomatisations suffisamment complètes : distinguer les ensembles d'opérations C, E et O ; écrire les axiomes définissant les opérations de E ; écrire la liste des parties gauches de la forme $o(c(x^*),y^*)$ et associer à chacune une partie droite vérifiant les propriétés de la définition 4, cohérente avec le modèle voulu.

De plus, il est montré dans [GH 78] que pour tout type dont les opérations sont récursives primitives, il existe une axiomatisation vérifiant les propriétés du théorème 6, ou alors on peut en trouver une en ajoutant un certain nombre de fonctions auxiliaires dans S ou O.

Les fonctions auxiliaires sont des opérations sans utilité pour l'utilisateur, mais nécessaires à l'écriture des axiomes. Sous le vocable de "hidden functions", leur usage a fait l'objet d'une vigoureuse polémique dont l'intérêt scientifique n'est pas immédiat. La propriété que l'on a énoncée plus haut permet, par contre, de conclure ce paragraphe par un résultat essentiel qui est qu'on a une méthode pour construire une axiomatisation suffisamment complète pour tous les types abstraits "raisonnables".

II.3 - Les erreurs

La spécification des cas d'erreurs est une partie essentielle de la définition d'un type abstrait. Elle doit être aussi formelle et précise que le reste de la spécification, et, de plus, indépendante de la représentation. On présente ici la méthode de spécification par "restrictions" donnée dans

[Gu 78] puis la définition des erreurs abstraites dans les types abstraits proposée par [Go 77]. Dans un premier temps, on montre au moyen d'un exemple que le problème des erreurs dans les types abstraits est loin d'être simple, et qu'on peut, par manque de précaution dans leur définition, introduire des inconsistances dans le type abstrait.

II.3.1 - Le problème

Les types étudiés en I et II, ne comportent pas de cas d'erreurs. Ce n'est pas le cas en général : les entiers débordent, on ne peut pas diviser par zéro ni accéder au sommet d'une pile vide ... Nous donnons ici l'exemple bien connu de la "File d'attente" : le premier élément d'une file vide n'est pas défini, donc toute tentative pour l'utiliser est une erreur. De plus on va considérer qu'essayer d'enlever un élément d'une file vide est également une erreur.

Exemple 3 :

```
type File, Bool, Item ;
operations
   ( ) → File : filevide ;
   (File, Item) → File : ajouter ;
   (File) → File : enlever ;
   (File) → Item : premier ;
   (File) → Bool : vide? ;
axiomes
soit f ∈ File, i ∈ Item ;
   enlever(ajouter(f,i)) = si vide?(f) alors filevide
                           sinon ajouter(enlever(f),i) ;
   premier(ajouter(f,i)) = si vide?(f) alors i
                           sinon premier(f);
   vide?(filevide) = vrai ;
   vide?(ajouter(f,i)) = faux ;
fin ;
```

Cette spécification n'est pas suffisamment complète car on ne sait pas réécrire "premier(filevide)". De plus, "enlever(filevide)" n'a pas été

défini : il est donc impossible de montrer que l'opération "enlever" est une extension, "ajouter" et "filevide" étant les constructeurs.

Une idée naturelle est d'ajouter l'axiome suivant :

premier(filevide) = undef

cela revient à introduire dans le type Item une constante particulière, undef. Malheureusement on ajoute ainsi aux termes du type abstrait File des éléments indésirables comme :

f=ajouter(filevide, premier(filevide))

qui, d'après les axiomes vérifie

premier(f) = undef

vide?(f) = faux.

De plus, les conséquences sur le type Item doivent être précisées. Par exemple, si les Item sont ordonnés, il faut définir la relation d'ordre pour undef, ce qui, on va le voir, n'est pas évident.

De même, si on écrit

enlever(filevide) = undef

on a le même genre de problème : tout d'abord on doit distinguer entre "undef" de type Item et "undef" de type File :

premier(filevide) = undef_i

enlever(filevide) = undef_f

il faut alors définir

vide?(undef_f), premier(undef_f), ajouter(undef_f,i),...

Une attitude raisonnable, semble être de décider que les erreurs se propagent, c'est-à-dire que le résultat de toute opération ayant un undef_t en opérande est la constante undef du type approprié.

D'où les axiomes :

ajouter(undef_f,i) = undef_f ;

ajouter(f,undef_i) = undef_f ;

enlever(undef_f) = undef_f ;

premier(undef_f) = undef_i ;

vide(undef_f) = undef_b ;

...

Malheureusement, on introduit ainsi des inconsistances regrettables ! par exemple on a :

enlever(ajouter(filevide, premier(filevide)) = filevide

d'après les axiomes de l'exemple 3 et

enlever(ajouter(filevide, premier(filevide)) =

enlever(ajouter(filevide, $undef_i$)) = enlever($undef_f$) = $undef_f$

d'après la propagation des erreurs. D'où

filevide = $undef_f$! !

et

vide?($undef_f$) = vrai

Or $undef_f$ = ajouter($undef_f$, i) donc

vide?($undef_f$) = faux

De fait, on a complètement "aplati" le type File et il est possible de montrer que toute file f est égale à $undef_f$.

Pour éviter cela, il faut reformuler tous les axiomes avec des tests sur chaque opérande du terme qu'on définit pour savoir si c'est un "undef". Cela résout le problème du type File, mais pas celui du type Item !

Supposons que le type Item soit le suivant :

```
type Item, Bool ;
operations
   ( ) → Item : min ;
   (Item) → Item : succ ;
   (Item, Item) → Bool : ppe ;
axiomes
soit i,i'∈Item ;
   ppe(min, i) = vrai ;
   ppe(succ(i), min) = faux ;
   ppe(succ(i),succ(i')) = ppe(i,i') ;
fin ;
```

on voit qu'on a

ppe(min,$undef_i$) = vrai d'après les axiomes de Item

ppe(min,$undef_i$) = $undef_b$ d'après la propagation des erreurs

d'où vrai = $undef_b$

de même

ppe(succ($undef_i$,min) = faux

ppe(succ($undef_i$,min) = ppe($undef_i$,min) = $undef_b$

d'où faux = $undef_b$ = vrai ...

Il faut donc également reformuler complètement le type Item et le type Bool.

Tout cela est en contradiction avec les principes essentiels des types abstraits : comme on ne veut pas que la définition d'un type "File d'Items" oblige à redéfinir le type Item ; on souhaite que la spécification des cas normaux ne soit pas alourdie par une quantité de tests dûs aux cas d'erreurs, etc... Bref, on voudrait garder les axiomes "normaux" inchangés et traiter les cas d'erreurs à part, sans doubler ou tripler la longueur de la spécification et de manière à "protéger" les propriétés des types prédéfinis utilisés. C'est ce qui est fait dans les deux approches que nous présentons maintenant.

II.3.2 - Spécification des erreurs par des restrictions

Cette méthode, consiste à compléter la spécification du type abstrait par une quatrième partie où les cas d'erreurs sont décrits. On y trouve deux sortes de définitions : des préconditions, qui limitent l'applicabilité des axiomes, donc l'usage du type abstrait, des cas d'échec qui indiquent les cas où la représentation d'une opération terminera anormalement. Cela peut signifier qu'on s'arrête, qu'on édite un message d'erreur, qu'on se branche à une procédure externe de rattrapage d'erreur, selon la représentation choisie.

L'exemple 3 peut être complété par la partie restriction suivante :

```
restrictions
   Pré(enlever, f) = ¬vide?(f) ;
   Pré(premier,f)  = ¬vide?(f) ;
```

Une précondition restreint les termes que l'utilisateur du type abstrait peut utiliser en ayant l'assurance que les axiomes sont vérifiés. C'est à l'utilisateur de respecter la restriction et, dans les démonstrations sur le

type abstrait, on considère que pour tout terme g(Y), la précondition (si elle existe) Pré(g,Y) est vérifiée par hypothèse.

Avec les restrictions ci-dessus, c'est l'utilisateur du type File qui a charge de prouver ou de tester que tout opérande de "enlever" ou de "premier" n'est pas vide ; par exemple, il écrira :

premier(ajouter(f,i))

ou

si ¬vide?(f) alors premier(f) sinon .....

C'est également l'utilisateur qui doit choisir le traitement en cas d'erreur.

Les restrictions par cas d'échec, au contraire, donnent ces responsabilités à l'implanteur du type abstrait. Afin de compléter l'exemple 3, imaginons qu'on veuille borner la taille des files considérées. A cette fin, on va se donner une opération : "longueur", et on veut avoir un échec si la longueur d'une file dépasse la valeur 100. La spécification complète du type File devient alors :

Exemple 3.1 :

```
type File, Bool, Ent, Item ;
operations
   ( ) → File : filevide ;
   (File, Item) → File : ajouter ;
   (File) → File : enlever ;
   (File) → Item : premier ;
   (File) → Bool : vide ;
   (File) → Ent : longueur ;
axiomes
soit f ∈ File, i ∈ Item ;
   enlever(ajouter(f,i)) = si vide?(f) alors filevide
                           sinon ajouter (enlever(f),i) ;
   premier(ajouter(f,i)) = si vide?(f) alors i sinon premier(f) ;
   vide?(filevide) = vrai ;
   vide?(ajouter(f,i)) = faux ;
   longueur(filevide) = 0 ;
   longueur(ajouter(f,i)) = longueur(f)+1 ;
```



```
restrictions
   Pré(enlever,f) = ¬vide?(f) ;
   Pré(premier,f) = ¬vide?(f) ;
   longueur(f) ≥ 100 ⇒ Echec (ajouter, f, i) ;
fin
```

La dernière ligne de la partie restriction spécifie que toute représentation de l'opération "ajouter" devra comporter un test de la condition d'échec. Le choix du traitement en cas d'erreur est à la charge de l'implanteur. [Gu 78] propose également des cas d'échec optionnels ; qui s'écrivent :

Echec (ajouter, f, i) ⇒ longueur(f) ≥ 100

La signification de cette restriction est que si on a un échec comme résultat du terme "ajouter(f,i)" c'est que la longueur de f est supérieure à 100 : cette longueur peut-être très supérieure à 100, si l'implanteur a choisi de limiter la taille des files à 256 ou 1000. Les cas d'échec optionnel sont un moyen de spécifier des exigences minimales (la taille des files doit pouvoir atteindre 100) tandis que le cas d'échec normal est beaucoup plus restrictif (la taille des files ne peut pas dépasser 100).

L'avantage de cette méthode est qu'elle est d'un usage simple et qu'elle laisse le choix du niveau où les traitements en cas d'erreur sont spécifiés : on a, en quelque sorte, des erreurs de même niveau que le type qui sont les préconditions, avec des traitements spécifiés par l'utilisateur ; les cas d'échec, par contre, peuvent être considérés comme des abstractions d'erreurs dont on spécifie les caractéristiques sans prendre d'option de représentation.

II.3.3 - Les E-algèbres (5)

En I.2, on a donné la sémantique de la présentation d'un type abstrait en terme d'algèbre initiale. On va le faire ici dans le cas où le type à spécifier comporte des erreurs, suivant le formalisme proposé par Goguen dans [Go 77].

---

(5) Le vocable E-algèbre est un néologisme, traduction de l'anglais "error-algebra".

Les idées intuitives sur lesquelles cette théorie est basée sont relativement simples : on veut, comme en I.2, considérer que le type abstrait défini par une présentation est une algèbre dont les objets sont les classes d'équivalence de termes définies par les équations. Parmi ces termes on va avoir des "erreurs" (les $undef_t$ de III.1) que l'on va axiomatiser en respectant certaines règles (pas d'inconsistances, propagation des erreurs). Mais au lieu d'expliciter totalement cette propagation, on la pose en hypothèse : tout terme comportant une opération-erreur, ou pouvant être montré égal à un terme en comportant, est une erreur. Cela revient à définir une infinité de "terme-erreur" par exemple :

$undef_f$, ajouter(filevide, premier(filevide)),

enlever(filevide), ajouter(filevide, $undef_i$),

succ($undef_i$), etc.

Certains sont équivalents entre eux : les classes d'équivalence correspondantes sont des objets-erreurs. Certains, comme succ($undef_i$) ou ajouter($undef_f$,i) ne peuvent être reécrits : on évite ainsi les inconsistances indésirables de III.1 et, comme le fait remarquer Goguen, le terme obtenu permet de localiser l'erreur (si les axiomes ont été faits pour cela). On peut avoir plusieurs opération-erreur pour un même type. Afin de faciliter la lecture des définitions formelles, on donne auparavant l'exemple de la File avec les erreurs spécifiées de cette manière :

Exemple 3.2 :

type File, Bool, Ent, Item ;
operations
  ( ) → File : filevide ;
  (File, Item) → File : ajouter ;
  (File) → File : enlever ;
  (File) → Item : premier ;
  (File) → Bool : vide? ;
  (File) → Ent : longueur ;
E-operations (operation-erreurs)
  ( ) → File : underflow, overflow ;
  ( ) → Item : absent ;

axiomes
soit f ∈ File, i ∈ Item ;
les 6 axiomes donnés en 3.1
E-axiomes (spécification des cas d'erreurs)
  enlever(filevide) = underflow ;
  enlever(underflow) = underflow ;
  ajouter(overflow,i) = overflow ;
  premier(filevide) = absent ;
autres
  ajouter(f,i) = si longueur(f) ≥ 100 alors overflow
                 sinon ajouter(f,i) ;
fin

On pose comme règle que les E-axiomes doivent contenir au moins une occurrence d'une E-opérations.

Notons Σ_f la signature correspondant aux opérations "normales" du type File, et Ξ_f la signature correspondant aux E-opérations. En ajoutant ces opérations, on a augmenté notablement l'ensemble des termes du type File. Il est indispensable de distinguer dans $T_{\Sigma\cup\Xi}$ les termes erronés. Pour cela, on définit le prédicat ERR(t) :

- si $t = \sigma$, avec $\sigma \in \Sigma_f \cup \Xi_f$ (σ est donc une opération 0-aire) ERR(t) est vrai si $\sigma \in \Xi_f$ ou si σ apparait en partie gauche d'un E-axiome.

- si $t = \sigma(t_1, \ldots, t_n)$, ERR(t) est vrai dans les 3 cas suivants :

  . $\sigma \in \Xi_f$
  . $\exists i$ tel que ERR(t_i)
  . pour $i = 1,\ldots, n$, $\exists\, t'_i$ tel que $t_i = t'_i$ peut se déduire des axiomes autres que les E-axiomes, et que $\sigma(t'_1,\ldots,t'_n)$ apparaît en partie gauche d'un E-axiome.

On voit que, de même que les axiomes normaux définissent dans T_Σ des classes d'équivalence, les E-axiomes définissent dans le sous-ensemble de $T_{\Sigma\cup\Xi}$, caractérisé par ERR, des classes d'équivalences. Par définition, le type abstrait défini par une présentation du type de l'exemple 3.2, sera une algèbre sur ces classes d'équivalence, vérifiant les propriétés de propagation des erreurs. Plus précisément c'est l'algèbre initiale dans la classe des $\Sigma \cup \Xi$-algèbres quotientées par la relation de congruence minimale définie par (tous) les axiomes, et vérifiant un certain nombre de propriétés

sur les erreurs : il y a propagation des erreurs ; aux termes tels que ERR(t) correspondent des objets non erronés ...

C'est ce qui est exprimé formellement dans ce qui suit :

Définition 1 :

Soit Σ et Ξ deux signatures disjointes dont les opérations sont définies sur le même ensemble de noms de type : $S = \{s\}$. On appelle OK-opérations les opérations de Σ et E-opérations les opérations de Ξ. Une Σ,Ξ,E-algèbre A est une $\Sigma\cup\Xi$-algèbre telle que :

. A chaque $s\in S$ correspond un ensemble A_s de A partitionné en $K_s \cup E_s$. Les éléments de K_s sont des OK-éléments. Les éléments de E_s sont des E-éléments.

. A chaque $\sigma\in\Sigma$ correspond une opération A_σ de A qui propage les erreurs si un de ses opérandes est un E-élément, le résultat est un E-élément.

. A toute opération $\xi\in\Xi$ correspond une opération de A qui a toujours pour résultat un E-élément.

De même, on définit un Σ,Ξ,E-homomorphisme comme un $\Sigma\cup\Xi$-homomorphisme entre deux Σ,Ξ,E-algèbres, qui, en plus des propriétés habituelles d'un homomorphisme, préserve les erreurs (c'est-à-dire que si $e\in E_s$, $h(e)\in E'$ et on peut alors montrer que $T_{\Sigma\cup\Xi}$ est initiale dans la catégorie des Σ,Ξ,E-algèbres munies de tels homomorphismes, en prenant la convention que tout terme de T_Σ est un OK-élément et que les termes de $T_{\Sigma\cup\Xi} - T_\Sigma$ sont des E-éléments. Soit T_q cette E-algèbre.

Suivant la démarche de I.2, on va maintenant considérer le quotient de cette E-algèbre par la relation de congruence minimale définie par les axiomes. On doit définir ce que c'est qu'une E-algèbre quotient, c'est-à-dire indiquer quelles sont les classes d'équivalences de $T_{\Sigma\cup\Xi}$ qui vont correspondre à des OK-éléments ou à des E-éléments de la E-algèbre quotient.

On notera $P = \langle\Sigma,\Xi, K, E\rangle$ une E-présentation où Σ et Ξ sont deux présentations vérifiant les propriétés de la définition 1, où K est un ensemble de Σ-equations et où E est un ensemble de $\Sigma\cup\Xi$-equations (où toute équation comporte au moins une E-opération). Soit $\equiv_P$ la relation de congruence minimale définie sur $T_{\Sigma\cup\Xi}$ par K et E. Pour faire de $T_{\Sigma\cup\Xi}/\equiv_P$ une E-algèbre on va prendre les conventions suivantes :

Définition 2 :

Soit A une Σ,Ξ,E-algèbre et $\equiv$ une congruence sur A. Considérons $A/\equiv$. On dira qu'une classe d'équivalence [a] est un E-élément de $A/\equiv$ si et seulement si elle contient un E-élément de A. D'où une classe d'équivalence est un OK-élément de $A/\equiv$ seulement si elle ne contient que des OK-éléments de A.

Considérons maintenant $T_q/\equiv_P$ selon cette définition. C'est une E-algèbre. On montre qu'elle est initiale dans la classe des Σ,Ξ,E-algèbres vérifiant K et E et telles que, pour tout terme de T_q on a ERR(t) si et seulement si il correspond à t un E-élément de l'algèbre.

On pose par définition que le type abstrait défini par la présentation P est cette E-algèbre.

II.3.4 - Conclusion

On peut faire un rapprochement entre les E-algèbres de Goguen et les Préconditions de Guttag : dans un cas, on introduit parmi les termes du type abstrait tous les termes erronés (ce qui alourdit considérablement la théorie sous-jacente) ; dans l'autre cas on supprime les termes erronés du type abstrait. Dans le premier cas, l'utilisateur a la responsabilité du traitement des termes erronés, dans le deuxième, il doit vérifier qu'il n'en fait pas usage. D'un point de vue pratique, cela revient probablement au même.

Les cas d'échec de Guttag correspondent au cas où on utilise un terme sans se préoccuper de sa validité parce qu'on sait qu'il va être représenté par une erreur et qu'on n'est pas intéressé par le traitement de cette erreur.

C'est souvent le cas en programmation : les compilateurs testent les débordements de tableau et, si on ne veut pas un rattrapage particulier de ce type d'erreur, cela permet d'éviter d'écrire des tests dans le programme. De ce point de vue, la distinction que fait Guttag entre Précondition et cas d'échec semble permettre des spécifications plus souples et plus précises pour l'utilisateur et l'implémenteur.

Le formalisme des E-algèbres a l'avantage de définir mathématiquement la sémantique des types abstraits algébriques comportant des cas d'erreurs. Il est certainement utilisable pour décrire formellement ce que sont les préconditions et les cas d'échec.

Dans la suite de cette thèse, les cas d'erreurs sont spécifiés par des restrictions, car, en l'état actuel des choses, c'est l'outil le plus raisonnable.

II.4 - Preuves de représentations

On peut spécifier une représentation de plusieurs manières : par une fonction d'abstraction qui va des objets "concrets" vers les objets à représenter ; par une fonction de représentation qui associe à chaque terme du type à représenter un terme du type représentant.

Une des premières approches a été présentée par Hoare dans [Hoa 72]. Elle est plus naturelle si on travaille sur les objets(6) du type ; en effet, à un objet à représenter peuvent correspondre plusieurs représentants. C'est cette méthode qui est utilisée dans [GHM 76] et [Gau 78].

La deuxième approche est envisagée brièvement dans [GTW 78] et semble plus commode si on est intéressé par des reécritures de termes ou des problèmes de traduction, ce qui est notre cas [GP 79].

II.4.1 - Fonction d'abstraction

Durant tout ce paragraphe, nous allons étudier la représentation classique des Files par un tableau et deux entiers qui sont respectivement l'indice du premier élément de la file et l'indice de la première place libre après la file. Pour simplifier l'exemple le nombre d'éléments de la file et la dimension du tableau ne sont pas bornés.

Exemple 4 : il s'agit du type tableau où les indices sont des entiers et les valeurs des Items.

```
type Tableau, Ent, Item, Bool ;
opérations
   ( ) → Tableau : tabvide ;
   (Tableau, Ent) → Item : accès ;
   (Tableau, Ent, Item) → Tableau : assign ;
   (Tableau, Ent) → Bool : défini? ;
axiomes
soit t ∈ Tableau, i, j ∈ Int, it ∈ Item ;
```

(6) qui sont des classes d'équivalences de termes....

Note : pour faciliter la lecture accès(t,i) est écrit t[i].

```
   assign(t,i,it) [j]  = si eg(i,j) alors it sinon t[j] ;
   défini?(tabvide,j) = faux ;
   défini?(assign(t,i,it),j) = si eg(i,j) alors vrai sinon défini?(t,j) ;
restrictions
 ¬défini?(t,i) ⇒ Echec(accès, t, i) ;
fin
```

Pour définir la représentation on se donne la fonction d'abstraction *A* de profil : (Tableau, Ent, Ent) → File qui est caractérisée par les axiomes suivants :

Exemple 5 :

```
   filevide = A(t,i,i) ;
   ajouter(A(t,i,j),it) = A(assign(t,j,it),i,j+1) ;
   enlever(A(t,i,j)) = A(t,i+1, j) ;
   premier(A(t,i,j)) = t [i] ;
   vide?(A(t,i,j)) = eg(i,j) ;
```

On va examiner en quoi consiste cette définition et quelles sont les règles qui doivent être respectées pour qu'on ait ainsi spécifié une représentation correcte.

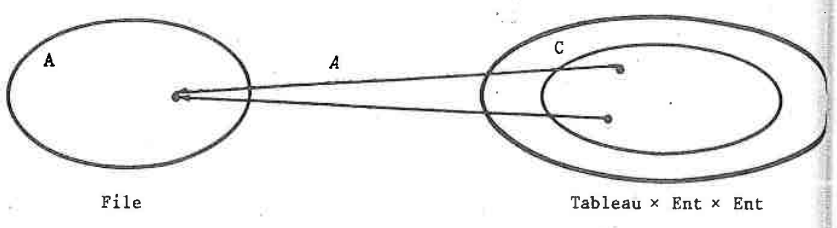
Une propriété indispensable est que la nouvelle opération *A* du type abstrait File ne doit pas introduire d'inconsistance dans ce type (par exemple en "représentant" deux objets différents(7) par des objets identiques). Il faut donc vérifier que le type File ainsi enrichi demeure consistant et suffisamment complet.

On voit ici l'intérêt de ne pas s'obliger à avoir des axiomes complets au sens classique du terme : dans la mesure ou l'égalité entre certains termes est laissée indécidable, comme dans l'exemple 2, on a la possibilité de les représenter par des objets identiques ou non et on a ainsi un plus grand choix de représentations.

Pour que *A* définisse une représentation correcte, elle doit de plus, vérifier d'autres propriétés.

(7) que l'on peut distinguer par une fonction externe.

Soit A l'ensemble des objets à représenter et C l'ensemble des objets utilisés pour la représentation. Ici A est l'ensemble des files, C l'ensemble des triplets <tableau, entier, entier>.



A doit être surjective : tous les objets doivent être représentés. Elle n'est pas forcément injective : un objet peut avoir plusieurs représentations (c'est le cas de filevide). D'autre part, A n'est pas nécessairement définie sur tout le domaine représentant.

Guttag, dans [GHM 76], appelle invariant de représentation les propriétés qui caractérisent le domaine de définition de A. Dans l'exemple des files si $<t,i,j> \in A^{-1}(A)$ on a :

$$i \leq j \text{ et } \forall k \text{ tel que } i \leq k < j \text{ défini?}(t,k) = \text{vrai}$$

Sur ce domaine, A définit la relation suivante :

$$c \approx c' \Leftrightarrow A(c) = A(c')$$

Cette relation est appelée "interprétation de l'égalité". Par exemple, d'après la fonction A de l'exemple 5 on a :

$$\forall t, t' \in \text{Tableau}, \forall i,j \in \text{Ent} \quad <t,i,i> \approx <t',j,j>$$

Pour que la représentation définie par A soit correcte, il faut non seulement que A soit surjective, mais aussi que les propriétés du type à représenter se retrouvent dans le domaine de A : les axiomes doivent être conservés par la représentation.

Pour exprimer (et vérifier) cette propriété on va travailler non plus sur C mais sur les classes d'équivalence de $A^{-1}(A)$ définies par la relation $\approx$. Formellement, une représentation est correcte si tout axiome du type représenté devient un théorème dans $A^{-1}(A)/\approx$, théorème démontrable à partir des axiomes des types représentants, des invariants de représentation et de l'interprétation de l'égalité.

On peut par exemple montrer que l'axiome

premier(ajouter(f,it)) = <u>si</u> vide?(f) <u>alors</u> it
<u>sinon</u> premier(f) ;

est conservé par la représentation de l'exemple 5.

La partie gauche est représentée par :

premier(ajouter(A(t,i,j),it) =
premier(A(assign(t,j,it),i,j +1) =
assign(t,j,it) [i] =
<u>si</u> eg(i,j) <u>alors</u> it <u>sinon</u> t [i] ;

La partie droite devient :

<u>si</u> eg(i,j) <u>alors</u> it <u>sinon</u> premier(A(t,i,j)) =
<u>si</u> eg(i,j) <u>alors</u> it <u>sinon</u> t [i] ;

c.q.f.d.

Les démonstrations de ce genre ne sont pas très difficiles dès l'instant où on a bien précisé les invariants de représentation et l'interprétation de l'égalité.

Elles sont automatisables [Mu 77].

II.4.2 - Fonction de représentation

On peut également spécifier une représentation en associant à tout nom d'opération, f, du type représenté une composition d'opérations des types représentants $\rho(f)$. Ceci doit être fait aussi pour la relation d'égalité qui sera représentée par une relation d'équivalence $\rho(=)$ qui est l'interprétation de l'égalité du paragraphe précédent. ρ s'étend aux termes et aux formules du type représenté de manière très classique :

$$\hat{\rho}(f(u_1,\ldots,u_n)) = \rho(f)\,(\hat{\rho}(u_1),\ldots,\hat{\rho}(u_n))$$

Dans la suite on ne distinguera plus ρ et $\hat{\rho}$. Si le domaine de représentation est le produit cartésien de plusieurs types, on notera $\rho_1(u),\ldots \rho_p(u)$ les p projections de $\rho(u)$.

La spécification de la représentation des files par des triplets <Tableau, Ent, Ent> s'écrit alors :

Exemple 6 :

soit $t = \rho_1(f)$, $i = \rho_2(f)$, $j = \rho_3(f)$;
ρ(filevide) = <tabvide, o, o> ;
ρ(ajouter(f,it)) = <assign(t,j,it), i,j+1> ;
ρ(enlever(f) = <t,i +1,j> ;
ρ(premier(f)) = t [i] ;
ρ(vide?(f)) = eg(i,j) ;
soit $t'=\rho_1(f')$, $i' = \rho_2(f')$, $j' = \rho_3(f')$;
ρ(f = f') = (<t,i,j> ρ(=) <t',i',j'>) =
eg(j-i,j'-i') $\wedge$ (o≤k<j-i $\supset$ eg(t[i+k], t'[i'+k])) ;

L'interprétation de l'égalité est ici complètement spécifiée. On suppose qu'on a une relation d'égalité sur les items.

Pour démontrer que la représentation ci-dessus est correcte on a besoin des invariants de représentation. On peut les démontrer par induction structurelle :

Par exemple on a :

[IR1] $\forall$ f $\in$ File $\rho_2(f) \leq \rho_3(f)$

[IR2] $\forall$ f $\in$ File, $\forall$ it $\in$ Item ρ_2(ajouter(f,it)) < ρ_3(ajouter(f,it))

Démonstration :

[IR1] est vrai pour f = filevide car
ρ(f=filevide) =($\rho_2(f) = \rho_3(f)$) d'après la définition de ρ(=)

Supposons IR1 vrai pour f. Il est vrai pour ajouter(f,it) car ρ_2(ajouter(f,it) = $\rho_2(f)$ et ρ_3(ajouter(f,it) = $\rho_3(f)+1$. Dans ce cas on a même : ρ_2(ajouter(f,it)) < ρ_3(ajouter(f,it))
c'est-à-dire [IR2].

[IR1] est également vrai pour "enlever(f)" :
ρ_2(enlever(f)) = $\rho_2(f)+1$
ρ_3(enlever(f)) = $\rho_3(f)$

Par hypothèse de récurrence, on sait que $\rho_2(f) \leq \rho_3(f)$ et par la précondition sur "enlever", on sait que ¬vide?(f) c'est-à-dire que $\rho_2(f)$ n'est pas égal à $\rho_3(f)$. On a donc $\rho_2(f) < \rho_3(f)$ d'où :
ρ_2(enlever(f)) ≤ ρ_3(enlever(f)).

En utilisant ces deux propriétés, on peut montrer sans grandes difficultés que les axiomes sont conservés. Par exemple :

enlever(ajouter(f,it)) = si vide?(f) alors filevide
sinon ajouter(enlever(f),i)

se reécrit :

ρ(enlever(ajouter(f,it))) ρ(=) si ρ(vide?(f)) alors
ρ(filevide) sinon ρ(ajouter(enlever(f),it))

Soit ρ(f) = <t,i,j>, la partie gauche devient :

<ρ_1(ajouter(f,it)), ρ_2(ajouter(f,it))+1, ρ_3(ajouter(f,it)> =
<assign(t,j,it),i+1,j+1>

La partie droite devient :

si eg(i,j) alors <tabvide,o,o>

sinon <assign(ρ_1(enlever(f)), ρ_3(enlever(f),it), ρ_2(enlever(f),
ρ_3(enlever(f))+1>

En faisant une analyse par cas, on voit que si eg(i,j) est vrai on a

<assign(t,j,it> ,i+1,j+1> ρ(=) <tabvide,o,o>
d'après l'interprétation de l'égalité.

Dans l'autre cas, la partie droite devient
<assign(t,j,it),i+1,j+1>

et la relation ρ(=) est donc satisfaite.

On voit que, quelle que soit l'approche adoptée, la spécification d'une représentation est simple, et les règles de preuves sont élémentaires. La simplicité de ces démonstrations amène à penser qu'il est possible, à partir de la représentation d'un sous-ensemble des opérations, de déduire

la représentation complète systématiquement, voire même automatiquement. Cela semble une démarche plus constructive que de faire une longue preuve "a posteriori". Dans [Gau 78] on trouvera un exemple d'une représentation complète construite à partir de la représentation des constructeurs et de l'interprétation de l'égalité.

II.5 - Conclusion

Ce chapitre est une tentative pour dégager les idées et les résultats essentiels à partir d'une abondante littérature. On s'est volontairement limité aux notions qui seront utilisées par la suite, et celles-ci ont été présentées et étudiées le plus complètement possible. Un domaine important de l'étude des types abstraits, les types paramétrés par d'autres types, comme "File de n'importe quoi" ou "Ens d'un type muni d'une égalité", a été complètement laissé de côté. Ce sujet est abondamment discuté dans les textes cités en bibliographie.

Nous allons étudier dans le chapitre suivant comment nous nous proposons d'utiliser les types abstraits algébriques pour décrire la sémantique des langages de programmation.

CHAPITRE III

DEFINITION SEMANTIQUE DE LANGAGES DE PROGRAMMATION A L'AIDE DE TYPES ABSTRAITS ALGEBRIQUES

Introduction

Comme cela a été dit au chapitre I, l'idée de base de ce travail est de formaliser la compilation comme la représentation d'un type abstrait par un autre.

Ce chapitre présente la sémantique formelle des langages de programmation qui a été définie dans ce but et qui est à la base du système de production de compilateurs et de la méthode de preuve.

On considère que la signification d'un programme source est un terme d'un certain type abstrait algébrique, caractérisant le langage source et qu'on appelle le "Type Source". On verra que si les types abstraits sont un outil commode pour spécifier la sémantique des expressions, dès que l'on aborde la sémantique des instructions, le problème devient plus difficile. Il faut alors introduire un type 'état' et un type 'changement d'état', dont les propriétés sont délicates à spécifier. Nous proposons ici une approche différente à ce problème, bien connu d'ailleurs : Guttag dans [Gu 79] et Guttag et Horning dans [GH 79] suggèrent d'utiliser des "transformations de prédicats" à la Dijkstra, pour décrire les actions. L'idée est de spécifier les propriétés des "ensembles de valeurs" sur lesquelles on travaille, par des axiomes et les changements d'état par des transformations de prédicats. Nous avons développé en parallèle [GDM 78, GP 79] un formalisme très voisin ; plutôt que transformations de prédicats, nous préférons parler de modification d'état, un état étant un ensemble de prédicats valides.

En effet, si les transformations de prédicats sont bien adaptées à la description et à la preuve d'une propriété spécifique d'un programme, il est plus naturel, pour définir la sémantique des instructions d'un langage dans le cadre des types abstraits algébriques, de considérer l'ensemble des prédicats modifiés par une instruction.

III.1 - Notions d'état et de modification

Soit un langage de programmation élémentaire qui permet de calculer des expressions entières et d'affecter des valeurs entières à des identificateurs. De telles affectations peuvent se composer séquentiellement :

```
<PROG> ::= <LISTE D'AFFECT>
<LISTE D'AFFECT> ::= <AFFECTATION> |
            <LISTE D'AFFECT> ; <AFFECTATION>
<AFFECTATION> ::= ID := <EXP>
<EXP> ::= <EXP> + <T> | <T>
 etc ...
```

Pour décrire la sémantique des expressions, on va se donner une définition des entiers avec les quatre opérations classiques. Ce type est donné ici in extenso car il sera souvent utilisé dans la suite. On peut noter que toute l'axiomatisation est basée sur le choix de trois constructeurs : 0, succ, pred ; succ(i) et pred(i) sont notés ici i+1 et i-1.

```
type Ent ;
opérations
   (Ent, Ent) → Ent : add, soust, mult, div ;
   (Ent) → Bool : neg ;
   (Ent, Ent) → Bool : eg, dif ;
axiomes
soit i, i' ∈ Ent ;
   add(i, Ent '0') = i ;
   add(i, add(i', Ent '1')) = add(add(i,i'), Ent '1') ;
   add(i, soust(i', Ent '1')) = soust(add(i,i'), Ent '1') ;
   add(i, j) = add(j,i)
   soust(i, Ent '0') = i ;
   soust(i, add(i', Ent '1')) = soust(soust(i,i'), Ent '1') ;
   soust(i, soust(i, Ent '1')) = add(soust(i,i'), Ent '1') ;
   mult(i, Ent '0') = Ent '0' ;
   mult(i, add(i', Ent '1')) = add(mult(i,i'), i) ;
   mult(i, soust(i', Ent '1')) = soust(mult(i,i'),i) ;
   mult(i,j) = mult(j,i)
```


N.B : "neg" est une opération auxiliaire nécessaire à la définition de la division, qui teste si un entier est négatif.

```
neg(Ent '0') = faux ;
neg(add(i, Ent '1')) = si eg(i, soust(Ent '0', Ent '1'))
                       alors faux sinon neg(i) ;
neg(soust(i, Ent '1')) = si eg(i, Ent '0') alors vrai
                         sinon neg(i) ;
```

N.B : de même l'opération d'égalité est nécessaire pour la définition de "neg".

```
eg(Ent 'C1', Ent 'C2') = eg(C1, C2) ;
eg(i, add(i, Ent '1')) = faux ;
eg(i, soust(i, Ent '1')) = faux ;
eg(i, i') = eg(i', i) ;
eg(add(i, Ent '1'), add(i', Ent '1')) = eg(i, i') ;
eg(soust(i, Ent '1'), soust(i', Ent '1')) = eg(i,i') ;
eg(soust(i, Ent '1'), add(i', Ent '1')) = eg(i,
                        add(add(i', Ent '1'), Ent '1')) ;
dif(i, i') = si eg(i, i')
             alors faux sinon vrai ;
```

N.B : on peut maintenant donner la définition de la division.

```
div(i, i') = si   neg(i) ∧   neg(i') alors
                 si neg(soust(i, i'))
                 alors Ent '0'
                 sinon add(div(soust(i,i'),i'), Ent '1')
             sinon si neg(i) ∧ neg(i') alors
                 div(soust(Ent '0',i), soust(Ent '0', i'))
             sinon si neg(i) ∧   neg(i') alors
                 soust(Ent '0', div(soust(Ent '0',i),i'))
             sinon si    neg(i) ∧ neg(i') alors
                 soust(Ent '0', div(i, soust('Ent'0',i')) ;
restrictions
eg(i', Ent '0') ⇒ Echec (div, i, i') ;
fin
```

N.B : à partir de ce type, il est possible de définir un type "entier borné" par un minimum et un maximum et de spécifier les débordements parmi les cas d'échec.

Les constantes, qui dans le chapitre précédent étaient spécifiées comme des opérations 0-aires, sont ici notées entre guillemets et précédées du nom de leur type. Il serait en effet fastidieux de les énumérer toutes, d'autant plus qu'à ce niveau, la syntaxe des constantes est sans importance : dans la pratique elle sera traitée par le constructeur lexicographique ; sur le plan théorique elle n'est pas significative.

Pour être rigoureux, on devrait donner une syntaxe des constantes entières, et, associer à cette syntaxe une sémantique donnant pour chaque texte de terminal un terme qui lui correspond dans le type Ent :

```
<Cte> → <Digit> | <Cte> <Digit> | - <Cte>
<Digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
V⦃0⦄ = zero
V⦃1⦄ = un
V⦃2⦄ = add(un, un)
V⦃3⦄ = add(V⦃2⦄, un)
...
V⦃Cte Digit⦄ = add(mult(add(V⦃9⦄ ,un),V ⦃Cte⦄),V⦃Digit⦄)
V⦃-Cte⦄ = soust(zero, V⦃Cte⦄)
```

Pour des raisons de lisibilité, d'efficacité et parce que son traitement à la représentation est spécial, on considère que pour chaque type t, la fonction V est prédéfinie, sous la forme d'une opération de nom t, dont l'argument est noté entre guillemets, sans parenthèses. Cette opération est de profil (Chaîne) → t où Chaîne est un type prédéfini "chaîne de caractères" (on pourrait l'appeler aussi "texte de terminaux"). On se donne sur ce type Chaîne une opération d'égalité "eg" (qui a été utilisée dans la définition de l'opération "eg" du type Ent).

Dans le langage que nous considérons, il y a des identificateurs. Le type associé est décrit par :


```
type Ident ;
opérations
   (Ident, Ident) → Bool : eg ;
   (Ident) → Ent : valeur ;
axiomes
   eg(Ident 'C1', Ident 'C2') = eg(C1, C2) ;
   eg(id1, id2) ⊃ eg(valeur(id1, valeur(id2))
fin (1)
```

Le premier axiome indique simplement que deux identificateurs sont égaux s'ils ont le même texte.

Etant donné le type Ent et le type Ident, on a maintenant les outils pour spécifier la sémantique des expressions. Par exemple à l'expression

(I + 3) * J

correspond le terme

mult(add(valeur(Ident 'I'), Ent '3'), valeur(Ident 'J')).

Plus généralement on peut se donner la fonction sémantique suivante :

V : EXP ∪ T ∪ F → termes de type Ent

avec :

```
V⦃EXP + T⦄ = add(V⦃EXP⦄, V⦃T⦄)
V⦃CTE⦄ = Ent 'CTE'
V⦃ID⦄ = valeur(Ident 'ID')
...
```

On peut remarquer que les noms d'opérations ne sont pas interprétés et que tout ce qu'on sait de "add" est ce qui a été spécifié par les axiomes du type Ent.

D'autre part, on n'a pas donné d'axiomes pour l'opération "valeur". En fait, ceux-ci vont être introduits par les affectations. L'instruction :

I := 2

introduit la propriété :

valeur(Ident 'I') = Ent '2'

---

(1) Rappelons que comme au chapitre II, on abrège P=vrai en P, quand P est une expression de type Bool.

et supprime toutes celles qui sont en contradiction ...

Classiquement, la signification d'une instruction est un changement d'état. Ici on change de type abstrait algébrique puisqu'on en modifie les axiomes, et par conséquent les opérations.

Pour rendre compte de cette modification on va définir des opérations qui font passer d'un type abstrait à un autre : dans ce but, on a formalisé la notion d'état d'une manière un peu différente de ce qui est fait habituellement. On trouvera cependant une approche similaire dans les travaux sur les structures d'information [Rem 74, Fin 74, Pai 75].

Définition 1 :

Soit A l'ensemble des axiomes du type abstrait algébrique associé à un langage de programmation. On appelle les axiomes de A les axiomes généraux (Ils sont vérifiés par tous les états que l'on peut obtenir en exécutant des instructions du langage).

Définition 2 :

Les opérations du type abstrait dont la définition est susceptible de changer d'un état à un autre sont appelées opérations modifiables. Les formules (2) ou axiomes qui caractérisent ces opérations pour un état donné sont appelés axiomes spécifiques.

Dans notre exemple :

add(i, Ent '0') = i

est un axiome général ;

valeur(Ident 'I') = Ent '2'

est un axiome spécifique.

Définition 2bis :

Etant donné un type abstrait, on se donne la fonction suivante qui caractérise les termes valides du type abstrait, c'est-à-dire ceux qui vérifient les préconditions :

- Si c est une constante pré(c)=vrai
- Si t est de la forme $f(u_1,u_2,\ldots,u_n)$ on a

  $\text{pré}(t)=\underline{\text{Pré}}(f,u_1,\ldots u_n) \wedge \text{pré}(u_1) \wedge \ldots \text{pré}(u_n)$

et on pose $\underline{\text{Pré}}(f, u_1,\ldots, u_n)$=vrai s'il n'y a pas de précondition sur f.

(2) Une formule est un terme de type Booléen.

Définition 3 :

Soit <T, F, A> la présentation d'un type abstrait algébrique associé à un langage de programmation. Un état est un ensemble S de formules de la forme t = t' (où t et t' sont des termes du type abstrait) qui vérifie les quatre propriétés suivantes :

i) $t = t \in S$ pour tout t terme du type abstrait tel que $\text{pré}(t) \in S$

ii) $t = t' \in S \Rightarrow t' = t \in S$

iii) $t = t' \in S,\ t' = t'' \in S \Rightarrow t = t'' \in S$

iv) pour tout f de $F^{(3)}$, pour tout $t_1, \ldots, t_n$ et $t'_1, \ldots t'_n$ correspondant à l'arité de f, on a :

$\text{pré}(f(t_1,\ldots,t_n)) \wedge \text{pré}(f(t'_1,\ldots,t'_n)) \in S,$

$t_1 = t'_1 \in S,\ldots, t_n = t'_n \in S \Rightarrow f(t_1,\ldots t_n) = f(t'_1,\ldots,t'_N) \in S$

Définition 4 :

Formules directement dérivables d'un ensemble d'axiomes.

Soit <T, F, A> la présentation d'un type abstrait algébrique où $A = \{a_1,\ldots a_n\}$ et où les a_i sont de la forme :

$$g_i(x^*) = d_i(x^*)$$

g_i est une composition des opérations de F, d_i est une composition de conditionnelles et d'opérations de F, x^* est une liste de variables libres typées. On appelle formule directement dérivable de A toute formule qui peut être obtenue en remplaçant dans un a_i toutes les variables de x^* par un terme du type abstrait du type correspondant (vérifiant les préconditions s'il y en a).

Définition 5 :

Etat engendré par un ensemble d'axiomes.

On appelle état engendré par un ensemble d'axiomes A le plus petit état (4) qui contient toutes les formules directement dérivables de A. On note S_A cet état.

(3) On verra dans le chapitre IV que l'introduction de procédures dans les langages considérés amène à modifier légèrement cette définition.

(4) Il s'agit en fait de la plus petite relation de congruence compatible avec les axiomes qui a été présentée au chapitre II, et la sémantique du type abstrait est l'algèbre initiale du paragraphe II.1.2.

Etant donné une formule f, on notera indifféremment

$$f \in S_A \qquad \text{ou} \qquad A \rightarrow f$$

Définition 6 :

Etat satisfaisant un ensemble d'axiomes.

Un état satisfait un ensemble d'axiomes A s'il contient toutes les formules directement dérivables de A.

Selon la convention suivie dans le chapitre II, un terme ou une formule ne comportent pas de variables libres. Un axiome comme :

add(i, Ent '0') = i

n'appartient pas aux états du type abstrait associé au mini-langage. Par contre, la formule :

add(Ent '1', Ent '0') = Ent '1'

appartient à tous les états.

Maintenant qu'on a défini ce qu'est un état, on a les éléments pour décrire la sémantique des instructions.

Soit S l'état courant. La sémantique de l'instruction d'affectation "I := 2" est une transformation de l'état S en l'état S' suivant :

$$S' = \{f \mid f\,[\text{Ent '2'} \,/\, \text{valeur(Ident 'I')}] \in S\}$$

où $f[\lambda/\mu]$ est la formule obtenue en remplaçant dans la formule f toutes les occurrences de μ par λ.

Ceci est la définition classique de l'affectation par une substitution [Flo 67, Hoa 69]. La formule f_1 :

Valeur(Ident 'I') = Ent '2'

devient par substitution :

Ent '2' = Ent '2'

qui est valide dans S. Donc f_1 est valide dans S'.

De même si I avait une autre valeur dans S, les formules qui en étaient la conséquence n'appartiennent plus à S'. Soit par exemple, f_2 :

add(valeur(Ident'I'), Ent '1') = Ent '0'

Après la substitution f_2 devient :

add(Ent '2', Ent '1') = Ent '0'

qui n'est pas valide dans S. Donc f_2 n'est pas valide dans S'.

Revenons aux types abstraits algébriques et à la définition sémantique du mini-langage. On va donner la valeur sémantique des instructions sous forme de termes du type abstrait algébrique "modification".

```
type Modif ;
opérations
   (Ident, Ent) → Modif : affect ;
   (Modif, Modif) → Modif : seq ;
fin
```

La signification d'un programme sera un terme de type Modif. On aura donc la fonction sémantique suivante :

C : PROG ∪ AFFECTATION ∪ LISTE D'AFFECT → termes de type Modif

définie par les équations :

C ⦃ID := EXP⦄ = affect(Ident 'ID', V⦃EXP⦄)

C ⦃LISTE D'AFFECT ; AFFECTATION⦄ = seq(C⦃LISTE D'AFFECT⦄, C⦃AFFECT⦄)

On n'a pas donné d'axiomes pour le type Modif. On pourrait, par exemple, donner un axiome d'associativité pour la composition séquentielle. Cependant, dans l'approche que nous avons choisie, la sémantique des modifications est donnée par des définitions où sont utilisées un certain nombre d'opérations fondamentales sur les états.

La première est l'application d'une modification à un état.

appl(m, S) = S'

Par exemple, on a vu que "affect(Ident 'I', Ent '2')" est la modification correspondant à l'affectation I := 2, et l'état

appl(affect(Ident 'I', Ent '2'), S)

est l'état S' tel qu'il a été défini ci-dessus.

On se donne aussi une opération d'adjonction d'un axiome au jeu d'axiomes caractérisant un état et qui sera notée :

S + a = S'

Définition 7 :

Adjonction d'un axiome à un état.

Etant donné un état S et un axiome a, on note S+a le plus petit état qui contient les formules de S et les formules directement dérivables de {a}.

En utilisant la substitution, l'adjonction et l'opération appl, on peut compléter la sémantique du mini langage par une partie de définition des modifications :

appl(affect(id,i),S) = {f | f[i/valeur(id)] ∈ S}

appl(seq(m,m'),S) = appl(m', appl(m,S))

Le lecteur peut vérifier qu'une conséquence de la deuxième définition est l'associativité de la composition séquentielle mentionnée ci-dessus.

Un type abstrait algébrique, où le type d'intérêt est "Modif", enrichi de définitions de modifications du genre de celles données ci-dessus, permet de décrire commodément la sémantique des langages de programmation. Cet enrichissement de la notion de type abstrait algébrique est très proche de la définition des structures d'informations dans [Rem 74, Pai 75]qui est utilisée en [Fin 74] pour définir la sémantique d'Algol 68. Plus anciennement [Bur 70] avait présenté une approche voisine. Dans la suite, et par abus de langage, on parlera indifféremment de Structure Source ou de Type Source pour désigner le type abstrait, agrémenté de modifications, qui est associé à un langage Source.

Afin de pouvoir faire des preuves de représentation de modification par des modifications, on va se donner un certain nombre de primitives qu'on utilisera pour définir aussi bien les modifications à représenter que les modifications représentantes. Dans [Gau 77] une autre approche avait été choisie qui consistait à se fixer une fois pour toute une modification primitive et des lois de composition standard. La méthode présentée ici permet des définitions plus souples des structures de contrôle d'un langage. Les primitives sont données dans le paragraphe suivant. La spécification et la preuve des représentations sont traitées dans les chapitres V et VI.

III.2 - Primitive de définition des modifications

On a vu sur l'exemple précédent que l'opération appl et la substitution jouaient un rôle essentiel dans la définition des modifications. Plus précisément, appl permet de définir les compositions de modifications et la substitution permet de rendre compte de l'affectation(5). Cette dernière instruction étant présente dans la plupart des langages de programmation, il paraît souhaitable de la modéliser une fois pour toute sous sa forme la plus générale et d'en faire une modification primitive qu'on utilisera par la suite pour définir les modifications des diverses structures sources rencontrées.

En III.1 on voulait ajouter une propriété de la forme :

valeur(id) = i.

Selon les langages et les types abstraits associés considérés, on veut, plus généralement, introduire des formules du type :

$f(\lambda) = \mu$

Malheureusement, la substitution telle qu'elle est donnée en III.1 n'est pas suffisante dans le cas fréquent où on peut avoir des identificateurs équivalents : c'est le cas explicitement en Fortran, mais on a ce problème quand on a des tableaux et des variables indicés ; si i=1 une affectation à a[i+1] doit changer la valeur de a[2] et celle de tout a [t] tel que t=2.

Pour étudier ce problème sur un exemple simple, on ajoute des déclarations comme :

Equivalence I,J

au mini-langage, dont la syntaxe devient :

<PROG> ::= <LISTE DE DECL> ; <LISTE D'AFFECT>

<LISTE DE DECL> ::= <DECL> | <LISTE DE DECL> ; <DECL>

<DECL> ::= Equivalence ID, ID

...

Cela va se traduire dans le type abstrait Ident par le fait que l'opération "eg" devient modifiable, et que l'axiome définissant "eg" sur les Ident en III.1 devient :

eg(C1,C2) ⊃ eg(Ident 'C1', Ident 'C2') ;

d'où l'obligation d'ajouter les axiomes suivants, qui ne se déduisent plus

(5) au sens le plus large de ce mot : changement de la valeur d'une variable, de la procédure en cours, passage de paramètre ...

des propriétés de l'égalité des chaînes :

$$eg(id, id') = eg(id', id) ;$$

$$eg(id, id'') \wedge eg(id'', id') \supset eg(id, id') ;$$

D'autre part, il faut ajouter au type Modif les opérations et définitions suivantes :

<u>operations</u>

(Ident, Ident) → Modif : decl ;

<u>definitions</u>

<u>appl</u>(decl(id, id'), S) = S'

avec S' = {f | f [vrai/eg(id, id')] ∈ S}

Du fait qu'en affectant un identificateur on veut changer la valeur de tout identificateur équivalent, la définition de l'affectation devient légèrement plus compliquée :

<u>appl</u> (affect(id,i), S) = S' avec

S' = {f | f [valeur'/valeur] ∈ S + (valeur'(x) = <u>si</u> eg(x,id) <u>alors</u> i <u>sinon</u> valeur(x))}

On passe par l'intermédiaire d'une opération *valeur'* qui est égale à "valeur" sauf aux points égaux à id. Soit $a_{valeur'}$ l'axiome qui définit cette opération. On voit que si id1 et id ne sont pas équivalents et

si valeur(id1) = i1 ∈ S on a :

valeur'(id1) = i1 ∈ S + $a_{valeur'}$

donc

valeur(id1) = i1 ∈ S'

Par contre si id et id1 sont équivalents on a

valeur(id1) = i ∈ S + $a_{valeur'}$

donc

valeur(id1) = i ∈ S'.

<u>Définition 7</u> :

Soit f le nom d'une opération modifiable, λ un terme (ou un n-uplet de termes) du type (ou des types) correspondant au domaine de f, μ un terme correspondant au co-domaine de f. On appelle "substitution généralisée" une modification notée <u>subst</u>(f(λ),μ).

Par définition

<u>appl</u>(<u>subst</u>(f(λ),μ),S) = {P | P [f'/f] ∈ S + f'(x) = <u>si</u> eg(x),λ) <u>alors</u> μ <u>sinon</u> f(x)}

où f' est un nom de fonction de même arité que f, soumis aux mêmes préconditions, et n'appartenant pas au type abstrait associé au langage(6).

<u>Théorème</u>

Le résultat de l'application à un état, d'une substitution généralisée est un état.

<u>Démonstration</u>

Pour faire cette démonstration, on utilise le lemme suivant :

<u>Lemme</u> : Soit S un état correspondant à un type <T, F ∪ {f'}, A>. L'ensemble de formules {P | P [f'/f] ∈ S} f ∈ F, f' ∉ F est un état correspondant au type <T, F, A>.

Notons S' cet ensemble de formules. Il vérifie les quatre propriétés de la définition 3 :

i) Si pré(t)∈S', c'est que pré(t)[f'/f]∈S. Or f' est soumis aux mêmes préconditions que f donc pré(t[f'/f])∈S. Comme S est un état, d'après i) : t[f'/f]=t[f'/f]∈S, et par définition de S',t=t'∈S'.

ii) t = t' ∈ S' ⇒ t[f'/f] = t'[f'/f] ∈ S

⇒ t'[f'/f] = t[f'/f] ∈ S ⇒ t' = t ∈ S'. Donc S' vérifie la propriété ii).

iii) Du fait que S est un état, on a

t[f'/f] = t'[f'/f] ∈ S, t'[f'/f] = t"[f'/f] ∈ S

⇒ t[f'/f] = t" [f'/f] ∈ S

d'où

t = t' ∈ S', t' = t" ∈ S' = t = t" ∈ S'

(6) Dans [Rem 74] et [Pai 75] ce nom auxiliaire de fonction est utilisé également mais d'une manière différente.

iv) Soit g un nom d'opération de F à un opérande.

- si g n'est pas f, on a

$$g(t)[f'/f] = = g(t[f'/f])$$

où '= =' indique l'identité entre termes.

$t = t' \in S' \Rightarrow t[f'/f] = t'[f'/f] \in S$

$\Rightarrow g(t[f'/f]) = g(t'[f'/f]) \in S$ car S est un état.

Ce qui est identique à :

$$g(t)[f'/f] = g(t')[f'/f] \in S$$

$$g(t) = g(t') \in S'$$

- si g est identique à f on a

$$f(t)[f'/f] = = f'(t[f'/f])$$

$t = t' \in S' \Rightarrow t[f'/f] = t'[f'/f] \in S \Rightarrow$

$f'(t[f'/f]) = f'(t'[f'/f]) \in S$

Ce qui est identique à

$$f(t)[f'/f] = f(t')[f'/f] \in S$$

d'où $f(t) = f(t') \in S'$

Cette dernière démonstration se généralise trivialement au cas où il y a plusieurs opérandes □

La démonstration du théorème est alors immédiate car, par définition de l'adjonction,

$\{S + f'(x) = \underline{si}\ eg(x,\lambda)\ \underline{alors}\ \mu\ \underline{sinon}\ f(x)\}$ est un état □

Pour pouvoir rendre compte de modifications plus complexes qu'une simple affectation on a besoin de substitutions généralisées simultanées : cela revient à ajouter plusieurs axiomes intermédiaires lors du changement d'état. Cela permet, en particulier, de spécifier l'ensemble des substitutions qui doivent être effectuées lors de l'application d'une certaine modification, sans en préciser l'ordre : on a ainsi un plus grand choix de représentation pour cette modification.

Définition 8 :

La substitution généralisée simultanée est notée

$\underline{subst}(\langle f_1(\lambda 1), \ldots, f_n(\lambda n)\rangle, \langle \mu_1, \ldots, \mu_n\rangle)$

soit m cette modification. Elle correspond au changement d'état suivant :

- si pour tout i et j distincts, f_i et f_j sont des noms différents d'opérations on a :

$\underline{appl}(m,S) = \{P \mid P[f'_1/f_1,\ldots,f'_n/f_n] \in S + a_1 + \ldots + a_n\}$

où a_i désigne l'axiome

$$f'_i(x) = \underline{si}\ eg(x,\lambda i)\ \underline{alors}\ \mu_i\ \underline{sinon}\ f_i(x)$$

- s'il existe deux entiers i et j distincts tels que f_i et f_j soient identiques, les axiomes a_i et a_j sont remplacés par l'axiome

$$f'_i(x) = \underline{si}\ eg(x,\lambda_i)\ \underline{alors}\ \mu_i\ \underline{sinon}$$
$$\underline{si}\ eg(x,\lambda_j)\ \underline{alors}\ \mu_j\ \underline{sinon}$$
$$f_i(x)\ ;$$

et on généralise à plus de deux noms d'opération identiques.

A partir de ces modifications primitives on peut décrire tout changement d'état dans une structure d'information. Dans la suite de ce chapitre, nous allons montrer sur un langage un peu plus réaliste comment en définir la sémantique avec ces outils. Dans le chapitre IV, la définition de la sémantique d'un langage comportant des procédures est complètement traitée.

III.3 - Le langage SEQFLEX

Ce langage est une extension du langage source étudié dans [Gau 77]. On y trouve les structures de contrôle usuelles. Il permet de manipuler des entiers et des séquences d'entiers dont la longueur est variable à l'exécution : lors de la déclaration d'un identificateur de séquence, la longueur qui lui est associée prend la valeur zéro. Cette longueur peut être changée par les instructions allonger (qui ajoute un élément à la fin de la séquence) et raccourcir (qui supprime le dernier élément de la séquence). On peut accéder au i^{eme} élément d'une séquence ou en changer la valeur. D'autre part, on peut écrire les expressions entières usuelles, tester si une séquence est vide et si deux

expressions entières sont égales.

La syntaxe de ce langage est la suivante :

```
<P>  → <LD> ; <LI>
<LD> → <LD> ; <D> | <D>
<D>  → entier ID | séquence ID
<LI> → <LI> ; <I> | <I>
<I>  → allonger (ID,<E>) | raccourcir (ID) | ID := <E> |
       ID[<E>] := <E> | si <C> alors <LI> sinon <LI> fsi |
       tantque <C> faire <LI> fait
<E>  → <E> + <T> | <E> - <T> | <T>
<T>  → <T> * <F> | <T> / <F> | <F>
<F>  → (<E>) | ID | ENT | ID[<E>] | longueur (ID)
<C>  → vide (ID) | nonvide (ID) | <E> = <E> | <E> ≠ <E>
```

ENT et ID sont considérés ici comme des terminaux ; ils sont par ailleurs les axiomes de deux grammaires qui engendrent respectivement les constantes entières et les identificateurs du langage. Dans les équations sémantiques ENT et ID sont utilisés pour dénoter le texte d'une constante ou d'un identificateur.

III.3.1 - Les expressions

Un type abstrait associé à SEQFLEX est défini par un certain nombre de noms de type, de noms d'opérations et d'axiomes. Parmi les types, il parait naturel d'avoir : des identificateurs de séquence, des identificateurs entiers, des entiers et des booleens.

Pour ces deux derniers types, on va reprendre les définitions de "Ent" donnée en III.1 et de "Bool" donné en II.1.

Il reste à donner une présentation des types "identificateurs d'entiers" et "identificateurs de séquences" :

```
type Idvar ;
opérations
   (Idvar, Idvar) → Bool : eg ;
   (Idvar) → Ent : valeur mod
axiomes
soit id, id' ∈ Idvar ;
   eg(Idvar 'C1', Idvar 'C2') = eg(C1, C2) ;
   eg(id, id') ⊃ eg(valeur(id), valeur(id')) ;
fin
type Idseq,
opérations
   (Idseq, Idseq) → Bool : eg ;
   (Idseq) → Ent : bornesup mod
   (Idseq, Ent) → Ent : ième mod
axiomes
soit ids, ids'∈Idseq,i,i'∈Ent ;
   eg(Idseq 'C1', Idseq 'C2') = eg(C1, C2) ;
   eg(ids, ids') ⊃ eg(bornesup(ids), bornesup(ids')) ;
   eg(ids, ids') ∧ eg(i,i') ⊃ eg(ième(ids,i), ième(ids',i')) ;
restrictions
   neg(soust(i,Ent '1')) ⇒ Echec(ième, ids, i) ;
   neg(soust(bornesup(ids),i)) ⇒ Echec(ième, ids, i) ;
fin
```


Les opérations "valeur", "bornesup", et "ième" sont modifiables. Les deux dernières restrictions indiquent que dans la spécification de la représentation de 'ième' on devra insérer un test pour savoir si l'indice est bien compris entre les bornes.

On a maintenant les éléments pour décrire la sémantique des expressions de SEQFLEX. La signification d'une expression entière sera un terme de type Ent, celle d'une expression booléenne sera un terme de type Bool. Les fonctions sémantiques sont les suivantes :

$\mathcal{V}$: E ∪ T ∪ F → termes de type Ent

$\mathcal{B}$: C → termes de type Bool.

Elles sont définies par les équations :

$V\llbracket E+T \rrbracket$ = add($V\llbracket E \rrbracket$,$V\llbracket T \rrbracket$) ;
$V\llbracket E-T \rrbracket$ = soust($V\llbracket E \rrbracket$,$V\llbracket T \rrbracket$) ;
$V\llbracket T*F \rrbracket$ = mult($V\llbracket T \rrbracket$,$V\llbracket F \rrbracket$) ;
$V\llbracket T/F \rrbracket$ = div($V\llbracket T \rrbracket$,$V\llbracket F \rrbracket$) ;
$V\llbracket (E) \rrbracket$ = $V\llbracket E \rrbracket$;
$V\llbracket ID \rrbracket$ = Idvar 'ID' ;
$V\llbracket ENT \rrbracket$ = Ent 'ENT' ;
$V\llbracket ID[E] \rrbracket$ = ième(Idseq 'ID', $V\llbracket E \rrbracket$) ;
$V\llbracket$ longueur(ID) $\rrbracket$ = bornesup(Idseq 'ID') ;
$B\llbracket$ vide(ID) $\rrbracket$ = eg(bornesup(Idseq 'ID'), Ent '0') ;
$B\llbracket$ nonvide(ID) $\rrbracket$ = dif(bornesup(Idseq 'ID'), Ent '0') ;
$B\llbracket E_1=E_2 \rrbracket$ = eg($V\llbracket E_1 \rrbracket$, $V\llbracket E_2 \rrbracket$) ;
$B\llbracket E_1 \neq E_2 \rrbracket$ = dif($V\llbracket E_1 \rrbracket$, $V\llbracket E_2 \rrbracket$) ;

On peut remarquer que, du fait que les cas d'erreurs sont spécifiés par des "Echecs" les équations définissant $V\llbracket T/F \rrbracket$ et $V\llbracket ID[E] \rrbracket$ ne comportent pas de tests de ces cas d'erreurs puisque ceux-ci seront traités dans la spécification de la représentation.

On verra en III.3.3 ce que deviennent les équations sémantiques si les erreurs sont spécifiées par des "Préconditions".

On n'a donné ici les équations sémantiques que pour des programmes corrects, c'est-à-dire où tous les identificateurs sont déclarés et utilisés de manière licite. Plus généralement, dans la suite de cette thèse on considérera que toutes les vérifications statiques ont été effectuées : en effet dans le système PERLUETTE, ces vérifications sont programmées à l'aide d'attributs sémantiques qui sont évalués par le système DELTA [Lor 74, DM78].

Dans ce cas particulier, ces vérifications seraient facilement exprimables dans les équations sémantiques: il faudrait enrichir le type abstrait de deux prédicats, "est-déclaré-entier", "est-déclaré-séquence" et les tester à chaque usage d'un identificateur.

III.3.2 - Les instructions

La valeur sémantique d'une instruction est, comme en III.1, un terme de type Modif. La fonction sémantique correspondante C est de profil :

$$P \cup LD \cup D \cup LI \cup I \rightarrow \text{ termes de type Modif.}$$

La spécification du type Modif est la suivante :

type Modif ;
Opérations
(Idseq) → Modif : init ;
(Idseq, Ent) → Modif : ajouter ;
(Idseq) → Modif : retirer ;
(Idseq, Ent, Ent) → Modif : changer-ième ;
(Idvar, Ent) → Modif : affecter ;
(Modif, Modif) → Modif : concat ;
(Bool, Modif, Modif) → Modif : conditionnelle ;
(Bool, Modif) → Modif : itération ;
( ) → Modif : vide ;
définitions
Soit ids ∈ Idseq, idv ∈ Idvar, i,j∈Ent, m,m'∈Modif, b∈Bool ;
init(ids) = subst(bornesup(ids), Ent '0') ;
ajouter(ids, j) = concat(subst(bornesup(ids), add(bornesup(ids), Ent '1')),
(subst(ième(ids, bornesup(ids)), j)) ;
retirer(ids) = subst(bornesup(ids), soust(bornesup(ids), Ent '1')) ;
changer-ième(ids, i, j) = subst(ième(ids, i), j) ;
affecter(idv, i) = subst(valeur(idv, i)) ;
appl(concat(m, m'), S) = appl(m', appl(m, S)) ;
b=vrai ∈ S ⇒ appl(conditionnelle(b, m, m'), S) = appl(m, S) ;
b=faux ∈ S ⇒ appl(conditionnelle(b, m, m'), S) = appl(m', S) ;
b=vrai ∈ S ⇒ appl(itération(b,m), S) = appl(itération(b,m),appl(m, S)) ;
b=faux ∈ S ⇒ appl(itération(b,m), S) = S

N.B : dans le cas où aucune des deux formules b=vrai ou b=faux n'appartient à S, par exemple s'il y a une erreur au cours de l'évaluation de b, cette spécification laisse l'implémentation libre de choisir l'état suivant.

appl(vide, S) = S ;

N.B : les définitions données ici pour la conditionnelle, l'itération, l'affectation etc, doivent être revues si on a des effets de bord (voir Chap. IV), ce qui n'est pas le cas dans ce langage.

```
restrictions
    eg(bornesup(ids), Ent '0') ⇒ Echec(retirer, ids) ;
    neg(soust((i,Ent '1')) ⇒ Echec(changer-ième, ids, i, j) ;
    neg(soust(bornesup(ids), i)) ⇒ Echec(changer-ième, ids, i, j) ;
fin
```

Les équations sémantiques qui définissent C sont :

C⦃LD ; LI⦄ = concat(C⦃LD⦄, C⦃LI⦄) ;
C⦃LD ; D⦄ = concat(C⦃LD⦄, C⦃D⦄) ;
C⦃entier ID⦄ = rien ;

N.B : si on vérifiait la déclaration des identificateurs, on aurait à positionner un prédicat à vrai.

C⦃séquence ID⦄ = init(Idseq 'ID') ;

N.B : la borne supérieure est initialisée à 0.

C⦃LI ; I⦄ = concat(C⦃LI⦄, C⦃I⦄) ;
C⦃allonger(ID, E)⦄ = ajouter(Idseq 'ID', V⦃E⦄) ;
C⦃raccourcir(ID)⦄ = retirer(Idseq 'ID') ;
C⦃ID := E⦄ = affecter(Idvar 'ID', V⦃E⦄) ;
C⦃ID[E_1] := E_2⦄ = changer-ième(Idseq 'ID', V⦃E_1⦄, V⦃E_2⦄) ;
C⦃si C alors LI_1 sinon LI_2 fsi⦄ = conditionnelle(B⦃C⦄, C⦃LI_1⦄, C⦃LI_2⦄) ;
C⦃tant que C faire LI fait⦄ = itération(B⦃C⦄, B⦃LI⦄) ;

La valeur sémantique de tout programme de SEQFLEX est maintenant définie comme un terme de type Modif dont les propriétés sont parfaitement connues par la définition de la Structure Source qui vient d'être donnée. Cette approche est certainement pleine de possibilités : on devrait pouvoir, dans ce cadre, prouver des transformations de programmes, démontrer des propriétés intéressantes de certains programmes ... Nous nous intéresserons ici aux possibilités de spécification et de preuve de représentations, donc de compilateurs.

Ceci explique pourquoi on ne donne pas directement les parties droites des équations définissant C directement en termes de subst : en effet, nous

considérons que subst n'est pas un nom d'opération des types abstraits envisagés, mais un outil primitif de définition de modifications. Lorsqu'on spécifie une implantation d'un langage source, on donne pour chaque modification du Type Source, une représentation en terme de modifications du Type Objet. On vérifie alors, à partir des définitions du Type Objet que les substitutions indiquées dans le Type Source sont bien réalisées par cette représentation (qui peut comporter d'autres traitements auxiliaires, donc d'autres substitutions). D'un point de vue méthodologique, on a intérêt à éviter dans la définition du langage source toute "surspécification" susceptible de limiter les représentations possibles : par exemple, définir une modification comme une composition séquentielle alors que l'ordre d'exécution n'a pas d'importance (d'où l'intérêt des substitutions simultanées). Par contre, on a plusieurs manières de définir un langage source. Dans le paragraphe ci-dessous on verra ce qui se passe si on choisit de spécifier les cas d'erreurs de SQFLEX par des préconditions. Dans le dernier paragraphe de ce chapitre on donne un autre type source susceptible d'être associé à SEQFLEX.

III.3.3 Spécification des cas d'erreurs par des Préconditions

Supposons que les restrictions par cas d'échec du paragraphe précédent soient remplacées par des préconditions. On aura :

```
restrictions
   Pré(ième, ids, i) = neg(soust(Ent '0', i)) ∧ neg(soust(i, bornesup(ids)) ;
   Pré(retirer,  ids) = neg(soust(Ent '0', bornesup(ids)) ;
   Pré(changer-ième, ids, i, j) = Pré(ième, ids, i) ;
fin
```

Ces restrictions spécifient que pour accéder au ième élément ou le modifier on doit avoir testé ou prouvé que i est strictement positif et plus petit ou égal à la borne supérieure. Pour retirer le dernier élément d'une séquence, on doit vérifier que la longueur de celle-ci est strictement positive. Les équations sémantiques sont donc légèrement modifiées dans certains cas puisqu'il faut introduire des tests. Par exemple, dans la définition de C on a deux équations différentes :

$C\llbracket$raccourcir(ID)$\rrbracket$ = conditionnelle(neg(soust(Ent'0', bornesup(ids))),
retirer(Idseq'ID'),stoperreur) ;

$C\llbracket ID[E_1] := E_2 \rrbracket$ = conditionnelle(neg(soust(Ent'0', $V\llbracket E_1 \rrbracket$)) ∧
neg(soust($V\llbracket E_1 \rrbracket$, bornesup(Idseq'ID'))),
changer-ième(Idseq'ID', $V\llbracket E_1 \rrbracket$, $V\llbracket E_2 \rrbracket$), stoperreur).

Cette dernière équation présente l'inconvénient (inexistant du point de vue théorique) de comporter plusieurs occurrences de $V\llbracket E_1 \rrbracket$. Dans le système PERLUETTE cela signifie que le texte du terme correspondant à E_1 sera recopié plusieurs fois dans le premier texte intermédiaire, traduit plusieurs fois dans les phases ultérieures de traduction, calculé plusieurs fois à l'exécution du programme objet généré. D'où l'intérêt de définir une nouvelle modification :

(Idseq, Ent, Ent) → Modif : changer-ième-avec-verif ;

définie par :

changer-ième-avec-vérif(ids, i, j) = conditionnelle(neg(soust(Ent'0',i)) ∧
neg(soust(i,(bornesup(ids))), changer-ième(ids,i,j), stoperreur) ;

et utilisée dans l'équation :

$C\llbracket ID[E_1] := E_2 \rrbracket$ = changer-ième-avec-verif(Idseq'ID', $V\llbracket E_1 \rrbracket$, $V\llbracket E_2 \rrbracket$)

Cela laisse le choix à l'implémenteur de "changer-ième-avec-vérif" de stocker i dans une variable intermédiaire ou un registre ou de recopier le texte de son implementation.

On a utilisé une nouvelle modification : "stoperreur", dont on peut vouloir, selon les cas, différentes définitions. Ici on choisit de rester dans l'état auquel a été appliqué "stoperreur", quelle que soit la suite du programme :

$P \in S \Rightarrow P \in$ appl(stoperreur, S)

erreur = vrai $\in$ appl(stoperreur, S)

$\forall$ m, erreur = vrai $\in S \Rightarrow$ appl(m, S) = S.

Où erreur est une opération modifiable de profil ( ) → Bool. Pour éviter

des contradictions il faut alors revoir les définitions des autres modifications en y ajoutant un test d'erreur. Ce soin est laissé au lecteur.

On a vu quels étaient les changements dans la définition de C causés par l'utilisation de Précondition. $V\llbracket ID[E] \rrbracket$ doit être également redéfini. Cette redéfinition implique des changements beaucoup plus complexes dans la Structure Source de SEQFLEX car dorénavant l'évaluation d'une expression peut provoquer un changement d'état. Ce changement d'état ne peut être dû qu'à une erreur : donc de l'état S, on ne peut passer qu'à l'état appl(stoperreur,S) ou à S.

La définition de $V\llbracket ID[E] \rrbracket$ devient :

$V\llbracket ID[E] \rrbracket$ = ième-avec-vérif(Idseq 'ID', $V\llbracket E \rrbracket$)

où ième-avec-vérif a pour arité (Idseq, Ent) → Ent.

Pour donner les axiomes sur cette nouvelle opération on a besoin d'opérateurs de composition de modification et d'entier ou de booléen :

(Modif, Ent) → Ent : efb_i ;

(Modif, Bool) → Bool : efb_b ;

dont les propriétés sont décrites par les axiomes et les définitions suivantes :

axiomes

add(efb_i(m, i), efb_i(m', i')) = efb_i(concat(m,m'), add(i, i')) ;

...

eg(efb_i(m, i), efb_i(m', i')) = efb_b(concat(m, m'), eg(i, i')) ;

...

N.B : et ainsi de suite pour toutes les opérations à opérandes entiers ou booléens.

définitions

appl(affect(id, efb_i(m, i)), S) = appl(affect(id, i), appl(m, S)) ;

appl(conditionnelle(efb_b(m, b), m', m") = appl(conditionnelle(b,m',m"),
appl(m, S)) ;

...

N.B : et ainsi de suite pour toutes les modifications à opérandes entiers ou booléens.

Le principe de ces définitions est simple : on se met en état d'erreur si un des opérandes d'une modification a provoqué une erreur.

On peut maintenant donner la définition de ième-avec-vérif :

```
ième-avec-vérif(ids,i) = efb_i(conditionnelle(neg(soust(Ent '0',i)) ∧
                                              neg(soust(i,bornesup(ids))),
                                              vide, stoperreur),
                               ième(ids,i)) ;
```

On voit que l'utilisation de Préconditions complique quelque peu la définition du Langage Source : c'est une conséquence du fait qu'on a allégé la spécification de l'implantation.

III.4 - Autre définition sémantique de SEQFLEX

Il a pu paraître étrange aux habitués des types abstraits, de voir définir un langage de manipulation de séquences sans une spécification explicite de ce type abstrait. Pour permettre une comparaison, nous donnons ici une définition de SEQFLEX, basée sur le type séquence, plus proche de celle qui était donnée en [Gau 77].

On se donne donc le type suivant :

```
type Seq ;
opérations
   ( ) → Seq : seq-vide ;
   (Seq, Ent) → Ent : bornesup ;
   (Seq, Ent) → Seq : concaténer ;
   (Seq) → Seq : début ;
   (Seq, Ent, Ent) → Seq : changer ;
   (Seq, Ent) → Ent : ième ;
```

N.B : aucune de ces opérations n'est modifiable.

```
axiomes
   bornesup(seq-vide) = Ent '0' ;
   bornesup(concaténer(s,j)) = add(bornesup(s), Ent '1') ;
   bornesup(début(s)) = soust(bornesup(s), Ent '1') ;
   ième(concaténer(s,j),i) = si eg(i, add(bornesup(s), Ent '1')
                 alors j sinon ième(s, i) ;
   ième(début(s),i) = si dif(i, bornesup(s)) alors ième (s,i) ;
```



```
restrictions
   neg(soust(i, Ent '1')) ⇒ Echec (ième, s, i) ;
   neg(soust(bornesup(s), i)) ⇒ Echec (ième, s, i) ;
   eg(bornesup(s), Ent '0') ⇒ Echec (début, s) ;
   neg(soust(i, Ent '1')) ⇒ Echec (changer, s, i j) ;
   neg(soust(bornesup(s), i)) ⇒ Echec (changer, s, i, j) ;
fin
```

Les types Ent, Bool et Idvar restent inchangés. Par contre, le type Idseq devient :

```
type Idseq ;
opérations
   (Idseq,Idseq) → Bool : eg ;
   (Idseq) → Seq : val-seq mod
axiomes
   eg(Idseq 'C1', Idseq 'C2') = eg(C1, C2) ;
fin
```

Les noms des modifications restent les mêmes mais les définitions changent :

```
type Modif
opérations
   (Idseq) → Modif : init ;
   (Idseq, Ent) → Modif : ajouter ;
   (Idseq) → Modif : retirer ;
   (Idseq, Ent, Ent) → Modif : changer-ième ;
...
définitions
   init(ids) = subst(val-seq(ids), seq-vide) ;
   ajouter(ids, j) = subst(val-seq(ids), concaténer(val-seq(ids),j)) ;
   retirer(ids) = subst(val-seq(ids),début(valseq(ids))) ;
   changer-ième(ids, i, j) = subst(val-seq(ids),changer(ids, i, j)) ;
...
fin
```

Les équations sémantiques sont inchangées par rapport à III.2 sauf :

```
   V⦃ID[E]⦄ = ième(val-seq(Idseq 'ID'), V⦃E⦄) ;
   V⦃longueur(ID)⦄ = bornesup(val-seq(Idseq 'ID')) ;
```

On voit qu'on a augmenté de manière significative le nombre de noms d'opérations intervenant dans la description, d'où une écriture plus lourde de la définition des modifications. Tout cela a des répercussions sur les preuves de représentation, on le verra. Par contre, le terme associé à un programme est très peu modifié et les propriétés des objets traités par le langage sont mieux mises en évidence : il est difficile de conclure en faveur de l'une ou l'autre approche. Si on avait un langage plus compliqué autorisant des affectations ou des passages de séquences en paramètres, la deuxième solution s'imposerait. Elle s'imposerait également dans le cas où on utiliserait pour définir un langage une librairie de types, où il suffirait d'aller chercher des types Tableau, Séquence, Ensemble, prédéfinis ... Un problème ouvert est de déterminer ces types prédéfinis et leur usage : le contexte dans lequel est utilisé un type peut parfois en autoriser des simplifications qu'il serait dommage de négliger (c'est le cas ici pour les séquences) ; d'autre part, le fait de vouloir se ramener à un type prédéfini risque de compliquer la spécification de l'ensemble du langage.

III.5 - Problèmes de consistance et de complétude

Ces problèmes doivent être envisagés et les propriétés nécessaires d'une structure de données associée à un langage doivent être précisées.

Il est évident que tout état résultant de l'application d'une modification de la structure doit être consistant. Cela signifie tout d'abord que les axiomes généraux n'introduisent pas de contradictions et que, par la suite, les modifications ne produisent pas d'état inconsistant. Or, on sait que toute modification est spécifiée comme une composition du subst . subst($f(\lambda),\mu$), de part sa définition, introduit la nouvelle formule $f(\lambda) = \mu$ et supprime les formules qui lui sont contradictoires. Il faut donc vérifier pour chaque modification de la structure

- que le "$f(\lambda) = \mu$" n'est pas en contradiction avec les axiomes généraux,
- que, dans le cas d'une substitution simultanée les formules introduites ne sont pas contradictoires entre elles.

Pour ce qui est de la complétude, il est évident que le type abstrait, associé à un langage, est en général incomplet ; il n'est même pas suffisamment complet puisque les opérations modifiables ("valeur", "ième", ...) ne sont pas définies. Par contre certains types utilisés, dont on supprime les opérations modifiables sont suffisamment complets : Ent et Bool par exemple. Le type Bool est complet, et les opérations externes "neg", "eg", "dif" du type Ent sont toujours définies comme on peut le montrer par récurrence sur les constructeurs.

On peut envisager de poser cette propriété comme une règle, et dans les exemples présentés, cette règle est respectée. Cependant, on peut imaginer des langages de programmation définis incomplètement et donnant une grande liberté à l'implanteur. Dans cette application des types abstraits algébriques à la définition de traducteurs, les problèmes de complétude (suffisante ou non) sont donc beaucoup moins fondamentaux que dans le cas général. Il ne nous paraît pas souhaitable de donner des règles trop restrictives et d'exiger systématiquement la complétude suffisante : après tout, l'incomplétude permet un plus large choix d'implantations, même si, par ailleurs on connait les inconvénients dûs aux langages incomplètement définis : impossibilité de déterminer, indépendamment d'une implémentation particulière, le comportement de certains programmes, diversité des compilateurs ou interprètes, mauvaise portabilité des programmes.

III.6 - Conclusion

Dans ce chapitre nous avons défini une notion d'état, et une notion de modification, ceci afin de rendre compte de la sémantique des instructions.

Le type abstrait associé à un langage a pour "type d'intérêt" le type Modif. On ajoute à ce type abstrait algébrique une partie "définition des modifications" où les opérations à résultat de type Modif sont décrites en terme de appl et de subst. Ce couple <type abstrait algébrique, définition des modifications> est appelé structure d'information par analogie avec les travaux de Finance, Pair et Rémy. La donnée de cette structure d'information et des fonctions qui associent aux phases du langage un terme du type abstrait correspondant constitue la définition d'un langage de programmation, que nous utilisons pour générer et prouver les compilateurs.

Une des originalités de cette approche est la définition d'un état comme l'algèbre initiale correspondant aux axiomes généraux et aux axiomes spécifiques introduits par les modifications. Un état est ainsi caractérisé de manière très abstraite, indépendamment de toute notion de mémoire.

Il est intéressant de faire une comparaison avec la sémantique dite axiomatique : on remarque que la sémantique donnée ici pour les instructions, c'est-à-dire les modifications, est très proche des transformations de prédicats ou des plus-faibles-préconditions. Une des différences est que la sémantique axiomatique détermine les prédicats qui doivent être vérifiés avant l'instruction, en fonction de ceux qu'on veut obtenir après l'exécution de celle-ci. Par contre, ici on détermine l'ensemble des prédicats valides après l'instruction... en fonction de ceux qui étaient valides avant. La différence n'est pas essentielle, et cela apparaît bien dans la similitude de la plupart des définitions. En particulier, pour l'affectation de III.1 on aurait [Dij 76].

wp(affect(id,i),P) = P[i/val(id)]

ce qui est tout à fait semblable à :

<u>appl</u>(affect(id,i),S) = {P|P[i/val(id)]∈S}

La vraie différence est qu'on travaille ici sur un 'énorme prédicat' qui caractérise <u>toutes</u> les propriétés valides avant et après l'instruction : on n'est cependant pas très éloigné de la sémantique axiomatique.

Il est important de remarquer que les notions d'états et de modifications peuvent se décrire d'une manière algébrique. C'est ce qui est fait dans [PG 79]. On a ainsi un cadre théorique plus unifié pour décrire la sémantique des langages de programmation : il est alors nécessaire de faire apparaître la notion d'état de manière explicite dans le type abstrait, comme un nom de type. Les noms d'opérations, et par suite les termes sont interprétés, en ce sens qu'un terme de type Ent comme :

add(val(Id'A'),val(Id'B'))

est considéré comme une application des états dans les entiers ; de même un terme de type Modif est une application des états dans les états.

On peut décrire ces applications comme les opérations d'un type abstrait. On aura :

<u>opérations</u>

(Ent,Etat) → **Z** : eval ;

(Modif,Etat) → Etat : apply ;

La notion de substitution disparaît, et l'affectation s'exprime de la manière suivante :

(Id,Ent,Etat) → Etat : mod-val ;

<u>axiomes</u>

apply(assign(id,i),S) = mod-val(id,i,S) ;

eval(val(id),mod-val(id',i,S)) = <u>si</u> id = id' <u>alors</u> eval(i,S)
<u>sinon</u> eval(val(id),S) ;

On peut traiter de manière analogue toute opération modifiable.

L'intérêt est de faire apparaître plus clairement le rôle des états et de supprimer les modifications primitives. De plus, on verra dans le chapitre suivant que les 'effets de bord' se décrivent plus facilement.

Pour ce qui est de la compilation, on peut faire deux remarques : d'un point de vue pratique, il est inutile de faire apparaître des états dans les textes intermédiaires, donc dans les profils des fonctions (il est important de conserver le type Modif) ; d'un point de vue théorique, il paraît judicieux de ne pas interpréter le système formel associé au langage. Il suffit de remarquer que pour traduire une expression, il n'est pas nécessaire de connaître la valeur qu'elle désignera à l'exécution d'un programme. On évite ainsi - entre autres - les problèmes posés par les définitions récursives de programmes : celles-ci sont traduites en des définitions équivalente si une solution existe ... mais le problème de l'existence d'une solution n'a pas à être posé dans ce contexte (un compilateur ne vérifie pas la terminaison des programmes qu'il traduit). La sémantique définie dans ce chapitre a pour but de spécifier des traductions, non de définir complètement la sémantique des langages. Une idée de base dans cette thèse est que les schémas fonctionnels non interprétés sont l'outil théorique adapté à la spécification de compilateurs, une sémantique au sens classique, des programmes étant une interprétation de ces schémas fonctionnels.

CHAPITRE IV

SEMANTIQUE DES PROCEDURES

On montre ici que le formalisme présenté au chapitre précédent permet de rendre compte d'une manière élégante et rigoureuse des problèmes posés par la sémantique des procédures, les accès à leurs variables locales ou à des variables globales, les passages de paramètres.

La sémantique de ces aspects des langages de programmation n'est pas évidente à décrire, quel que soit le cadre théorique utilisé [Hoa 71, Ern 77, GHL 77, AB 77, Don 76, ...] . Les types abstraits algébriques enrichis de modifications en permettent une définition générale et relativement simple en dépit de la complexité du problème.

IV.1 - Le Langage SIMPROC

C'est un langage classique "de type Pascal", dont nous allons donner une sémantique complète. Il comporte des déclarations de variables entières, de tableaux d'entiers et de procédures qui peuvent être emboitées, récursives, mutuellement récursives. Les paramètres de ces procédures sont passés par valeur ou par référence. On a comme instructions l'affectation, l'appel de procédure et la composition conditionnelle. On trouvera ci-dessous une syntaxe de SIMPROC.


```
<L>   → <LD> ; <LI>
<LD>  → <LD> ; <D> | <D>
<D>   → entier ID | tableau ID [ENT] |ε|
        proc ID (val ID, var ID) debut <LD> ; <LI> fin
<LI>  → <LI> ; <I> | <I>
<I>   → <VAR> := <E> | appeler ID(<E>, <VAR>) |
        si <E> eg <E> alors <LI> sinon <LI> fsi
<E>   → <E> + <T> | <E> - <T> | <E> * <T> | <E> % <T> | <T>
<T>   → <VAR> | (<E>) | ENT
<VAR> →  ID | ID [<E>]
```

Pour simplifier l'exposé, on conviendra que chaque procédure a exactement deux paramètres, le premier passé par valeur, le second passé par référence. De plus, on va considérer que tous les identificateurs sont différents : on les renomme si c'est nécessaire. Enfin, comme au chapitre précédent, on ne s'intéressera qu'aux programmes valides au sens de la compilation : c'est-à-dire ceux qui ne donnent pas lieu à des messages d'erreurs pendant la traduction. Cela permet de simplifier la lecture des fonctions sémantiques en éliminant de nombreux tests.

Une première version de la définition complète de ce langage a été donnée en [GDM 78].

IV.1.1. Les expressions

On va utiliser naturellement les types Ent et Bool spécifiés précédemment.

a) Identificateurs

On a plusieurs types d'identificateurs : des identificateurs de variables simples, de tableaux, de procédures et de paramètres passés par référence (on considère les identificateurs de paramètres passés par valeur comme des identificateurs de variables simples).

D'où les définitions des types suivants :

```
type Var-id ;
operations
   (Var-id, Var-id) → Bool : eg ;
axiomes
   eg(Var-id'C1', Var-id'C2') = eg(C1, C2) ;
fin
type Tab-id ;
operations
   (Tab-id, Tab-id) → Bool : eg ;
   (Tab-id) → Ent : bornesup ;
axiomes
   eg(Var-id'C1', Var-id'C2') = eg(C1,C2) ;
fin
type Proc-id ;
operations
   (Proc-id, Proc-id) → Bool : eg ;
   (Proc-id) → Var-id : par1 ;
   (Proc-id) → Ref-id : par2 ;
axiomes
   eg(Proc-id'C1', Proc-id'C2') = eg(C1,C2) ;
fin
```

L'opération "bornesup" associe à un identificateur de tableau la borne supérieure de celui-ci. Les opérations "par1" et "par2" associent respectivement à un identificateur de procédures l'identificateur du paramètre passé par valeur et celui du paramètre passé par référence.

Chaque identificateur est local à une procédure ou au programme. On conviendra de désigner celui-ci par l'identificateur de procédure PROG. Les règles de localité sont spécifiées par l'opération "portée" qui est définie pour tous les identificateurs, sauf PROG et qui rend l'identificateur de la procédure à laquelle un identificateur est local :

```
type Id = union (Var-id, Tab-id, Proc-id, Ref-id) ;
operations
   (Id) → Proc-id : portée ;
```



```
axiomes
   portée(par1(p)) = p ;
   portée(par2(p)) = p ;
restrictions
   Pré(portée,id) = ¬ eg(id,Proc-id'PROG') ;
fin
```

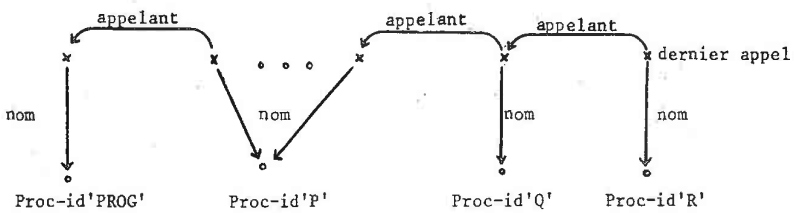
L'opération portée correspond à la notion de portée statique : il s'agit d'une relation entre identificateurs, que l'on peut déduire de l'examen du texte d'un programme et qui reste inchangée pendant toute évaluation de celui-ci.

Les problèmes de portée dynamique sont décrits à l'aide d'un nouveau type abstrait, le type des variables : en effet, une des difficultés dans la description d'un langage tel que SIMPROC est de rendre compte du fait qu'un identificateur local à une procédure désigne des variables différentes pour chaque appel de cette procédure. D'autre part, lorsqu'on rencontre un identificateur non local à la procédure en cours, il faut spécifier quelle variable il désigne c'est-à-dire préciser quel est l'appel de "sa" procédure à considérer.

On est donc amené à introduire un type abstrait "variable" (en Algol 68 on parlerait de "nom" [VW76]) un type "appel de procédure" et une opération qui, étant donné un identificateur et un appel, rend une variable.

b) Le type Appel

Avant de poursuivre la description des variables, on doit donc préciser la notion d'appel de procédure. Intuitivement, à chaque état (au sens du chapitre III) résultant de l'exécution d'une instruction de SIMPROC correspond un certain nombre d'appels en cours, tous différents, chacun "appelant" d'un autre. Chaque appel a un "nom" qui est l'identificateur de la procédure appelée :



Le dessin ci-dessus est un exemple des relations qui existent entre appels dans un état donné : on a une chaîne d'appels. Deux appels peuvent avoir le même nom puisque les procédures sont éventuellement récursives. Le plus ancien appel a pour nom l'identificateur PROG. Le plus récent est désigné par le symbole fonctionnel dernier appel. Tout cela se spécifie par les axiomes généraux suivants :

```
type Appel ;
opérations
    (  ) → Appel : dernier appel mod
    (Appel) → Appel : appelant mod
    (Appel) → Proc-id : nom mod
    (Appel,Appel) → Bool : eg ;
axiomes
    soit a,a',a" ∈ Appel ;
    eg(dernier appel, dernier appel) = vrai ;
    eg(appelant(a), appelant (a')) = eg(a,a') ;
    eg(appelant(a), dernier appel) = faux ;
    eg(a,a') = eg(a',a) ;
    eg(a,a') ∧ eg(a',a") ∧¬eg(a,a") = faux ;
restrictions
    Pre(appelant,a) = ¬ eg(nom(a),Proc-id'PROG') ;
fin
```


Les opérations 'dernier appel', 'appelant' et 'nom' sont modifiables : en effet on verra que les modifications correspondant aux instructions d'appel ou de retour d'une procédure changent le type Appel.

c) Les variables

On a maintenant les éléments pour décrire les relations entre identificateurs, appels et variables. Dans un premier temps, on ne va étudier que le cas des identificateurs de variables simples.

```
    Type Var ;
    Opérations
        (Var) → Ent : val mod
        (Var-id,Appel) → Var : dés ;
        (Var,Var) → Bool : eg ;
    axiomes
    soit v,v',v"∈Var,   id,id'∈Var-id,   a,a'∈Appel ;
      eg(v,v') = eg(v',v) ;
      eg(v,v') ∧ eg(v',v")∧¬eg(v,v") = faux ;
      eg(v,v') ⊃ eg(val(v),val(v')) ;
(1)   eg(id,id') ∧ eg(a,a') ⊃ eg(dés(id,a),dés(id',a')) ;
(2)   ¬(eg(id,id') ∧ eg(a,a')) ∧
          eg(nom(a),portée(id)) ∧ eg(nom(a'),portée(id'))
          ⊃ ¬eg(dés(id,a),dés(id',a')) ;
(3)   ¬eg(nom(a),portée(id)) ⊃ eg(dés(id,a),dés(id,appelant(a))  ;
    restrictions
(4)   Pré(dés,id,a) = eg(portée(id),Proc-id'PROG') ∨ ancêtre(id,nom(a)) ;
    fin
```

Les trois premiers axiomes (ceux qui ne sont pas numérotés) expriment des propriétés ordinaires de l'égalité. L'axiome (1) signifie simplement que deux couples <identificateur, appel> identiques désignent la même variable.

L'axiome (2) est nettement plus complexe : il exprime à la fois :

a) que deux identificateurs différents désignent toujours deux variables différentes ;
b) qu'en cas d'appel récursif les variables d'une procédure sont dupliquées ;
c) que les variables correspondant à des appels de procédures différentes sont toujours différentes.

Ce qui se dit, de manière concise sous la forme : étant donné deux couples non identiques <identificateur, appel> tels que, dans chaque couple l'identificateur soit local à la procédure appelée, ces couples désignent des variables différentes.

L'axiome (3) traite le cas des variables non locales : si un identificateur non local est utilisé au cours d'un appel, la variable qui lui correspond est celle du plus récent appel de la procédure à laquelle il est local.(1)

La restriction (4) exprime les règles de visibilité et de localité des identificateurs. On a besoin, pour l'exprimer, d'enrichir le type Id de l'opération "ancêtre" :

```
Opération
      (Id,Proc-id) → Bool : ancêtre ;
axiome
soit id ∈ Id, p ∈ Proc-id ;
      ancêtre(id,p) = si eg(portée(id),p)
                      alors vrai
                      sinon si eg(p,Proc-id'PROG')
                            alors faux
                            sinon ancêtre(id,portée(p)) ;
fin
```

(1) Ne pas oublier que les identificateurs sont supposés tous différents !

Ce prédicat indique si l'identificateur id peut être utilisé dans le texte de la procédure d'identificateur p. Notons qu'il peut être testé statiquement, par simple examen du texte du programme.

Enfin, si l'opération "val" est modifiable (pour pouvoir rendre compte des affectations), l'opération "dés" ne l'est pas. On verra en traitant les appels de procédures que ce sont les appels qui sont modifiés ; étant donné un identificateur et un appel, la variable désignée est toujours la même.

La valeur sémantique d'un identificateur de variable simple rencontré comme opérande d'une expression est :

val(dés(Var-id'ID'),dernier appel)

et par les axiomes (2) et (3) on sait quel est l'appel en cours à prendre en compte.

On va maintenant traiter le cas des variables désignées par un identificateur de tableau et un entier. Le type Var va être enrichi de l'opération "élément"

```
   Opération
      (Tab-id,Appel,Ent) → Var : elt ;
   axiomes
      soit tid,tid'∈Tab-id,   i,j∈Ent,   a,a'∈Appel ;
(4)   eg(tid,tid') ∧ eg(a,a') ∧ eg(i,j) ⊃ eg(elt(tid,a,i),elt(tid',a',j)) ;
(5)   ¬(eg(tid,tid') ∧ eg(a,a') ∧ eg(i,j)) ∧
        eg(nom(a),portée(tid)) ∧ eg(nom(a'),portée(tid')) ⊃
          ¬ eg(elt(tid,a,i),elt(tid',a',j)) ;
(6)   ¬eg(nom(a),portée(tid)) ⊃
          eg(elt(tid,a,i),elt(tid,appelant(a),i)) ;
(7)   ¬ eg(dés(vid,a),elt(tid,a',i)) ;
   restrictions
(8)   Pré(elt,tid,a,i) = eg(portée(tid),nom(a)) ∨ ancêtre(tid,nom(a)) ;
      neg(soust(i,Ent'1')) ⇒Echec(elt,tid,a,i) ;
      neg(soust(bornesup(tid),i)) ⇒Echec(elt,tid,a,i) ;
   fin
```

Les axiomes (4), (5), (6) expriment respectivement les mêmes propriétés sur les couples <identificateurs de tableau, entier> que les axiomes (1), (2), (3) sur les identificateurs de variable simple. L'axiome (7) spécifie qu'un élément de tableau et une variable simple sont forcément différents. La restriction (8) exprime la même règle de portée statique pour les identificateurs de tableau que la règle donnée par la restriction (4) pour les identificateurs de variable simple. Les deux dernières restrictions spécifient que les débordement d'indices dans les tableaux doivent être testés dans le code généré (autrement dit, l'implémentation du type abstrait associé à SIMPROC).

Il reste à spécifier les opérations permettant d'associer une variable à un identificateur de paramètre passé par référence. Pour cela on a choisi d'introduire un nouveau nom de type :

```
Type Réf-var ;
Opérations
   (Réf-id,Appel) → Réf-var : réf-dés ;
   (Réf-var) → var : réf mod
   (Réf-var,Réf-var) → Bool : eg ;
axiomes
```

NB : ce sont pratiquement les mêmes que pour le type Var et l'opération dés.

```
soit rid,rid' ∈ Réf-id,rv,rv',rv" ∈ Réf-var,a,a' ∈ Appel ;
    eg(rv,rv') = eg(rv',rv) ;
    eg(rv,rv') ∧ eg(rv',rv") ∧ ¬eg(rv,rv") = faux ;
    eg(rv,rv') ⊃ eg(réf(rv),réf(rv')) ;
    eg(rid,rid') ∧ eg(a,a') ⊃ eg(réf-dés(rid,a),réf-dés(rid',a')) ;
    ¬(eg(rid,rid') ∧ eg(a,a')) ∧
       eg(nom(a),portée(rid)) ∧ eg(nom(a'),portée(rid'))
       ⊃ ¬eg(réf-dés(rid,a),réf-dés(rid,a')) ;
    ¬(eg(nom(a),portée(rid)) ⊃ eg(réf-dés(rid,a),réf-dés(rid,appelant(a))) ;
restrictions
    Pré(réf-dés,rid,a) = eg(portée(rid),Proc-id'PROG') ∨ ancètre (rid,nom(a)) ;
fin
```


"réf" est une opération modifiable.

Il serait possible de se passer du type "Réf-var" en définissant la fonction "dés" sur l'union des types Var-id et Réf-var et en en faisant une opération modifiable. Cela compliquerait l'écriture des axiomes (1), (2), (3) car il faudrait tester chaque fois, avant d'établir la non-égalité de deux variables, que les opérandes de dés ne sont pas des Réf-id. C'est pour cette raison que nous avons préféré séparer très clairement les deux cas en utilisant des noms de type et d'opération différents.

On voit que la signification d'un identificateur dans une expression sera différente selon qu'il s'agit d'un "Var-id" ou d'un "Réf-id" :

```
val(dés(Var-id'ID',dernier appel))
val(réf(réf-dés(Réf-id'ID',dernier appel)))
```

On a donné ici la définition d'un type abstrait permettant de définir la sémantique des expressions de SIMPROC, ce qui est fait en IV.1.3. où sont données les équations sémantiques.

Si on supprime les opérations modifiables 'val' et 'ref', et si on considère que l'opération 'portée' est définie pour tous les identificateurs du programme (2), le type Var est suffisamment complet. En particulier l'égalité entre variables est décidable : il suffit pour le montrer de travailler par récurrence sur les constructeurs 'dernier appel' et 'appelant' du type Appel et de considérer que l'égalité entre identificateurs est complètement définie, ce qui est vrai.

IV.1.2 - Les instructions

Comme dans le chapitre précédent, la signification d'une instruction de SIMPROC sera un terme de type modification. A chaque catégorie d'instructions ou de composition d'instructions correspond une opération de ce type abstrait. De plus, comme on a des procédures, la valeur associée à un identificateur de procédure est de type "Modif", ce qui est exprimé par l'opération modifiable "possède".

(2) Ce n'est pas vrai pour tous les états, puisqu'on verra plus loin que 'portée' est définie au fur et à mesure des déclarations. Mais pour tout état où il est licite d'utiliser un identificateur effectivement (c'est-à-dire autrement que dans une déclaration de procédure), 'portée' est suffisamment définie : d'après la définition de SIMPROC, on a rencontré auparavant sa déclaration dans le texte.

On donne ici une définition partielle du type "Modif" associé à SIMPROC : Afin de faciliter la lecture la plupart des opérations sur les modifications seront définies au fur et à mesure de leur apparition dans les équations sémantiques.

```
Type  Modif ;
Opérations
    (Proc-id) → Modif : poss mod
    (Var,Int) → Modif : affecter ;
    (Modif,Modif) → Modif : concat ;
    (Ent,Ent,Modif,Modif) → Modif : cond ;
    ...
définitions
soit i,j  ∈ Int,m,m' ∈ Modif,v ∈ Var ;
    affecter(v,i) = subst(val(v),i) ;
    appl(concat(m,m'),S) = appl(m',appl(m,S)) ;
    eg(i,j) ∈  S ⇒appl(cond(i,j,m,m'),S) = appl(m,S) ;
    ¬eg(i,j) ∈ S ⇒appl(cond(i,j,m,m'),S) = appl(m',S) ;
    ...
fin
```

L'opération 'poss' est modifiable, et une déclaration de procédure introduit dans l'état courant une formule de la forme

$$poss(p) = m$$

où p est l'identificateur déclaré et m la modification correspondant au corps de la procédure.

Pour rendre compte de l'exécution d'une procédure, on se donne la règle :

(1) $poss(p) = m \in S \Rightarrow appl(poss(p),S) = appl(m,S)$

cependant les formules d'égalité entre modifications doivent être limitées si on veut éviter d'introduire des paradoxes. Par exemple considérons la déclaration de procédure suivante :

```
proc P (Val V,var R) début R:=V+X fin
```

où X est un identificateur global.

Les équations sémantiques données au paragraphe suivant spécifient que la modification correspondant au corps de cette procédure est :

```
affecter(réf(ref-dés(Ref-id'R',dernier-appel)),
        add(val(dés(val(dés(Var-id'V',dernier-appel),
            val(dés(Var-id'X',dernier-appel)))
```

que nous abrégerons en

```
affecter(R,add(val(V),val(X)))
```

On a donc introduit dans l'état courant la nouvelle formule :

$$poss(P) = affecter(R,add(val(V),val(X)))$$

Or si dans cet état courant on a une formule du genre :

$$val(X) = Ent'2'$$

on introduit ainsi la formule superflue :

$$poss(P) = affecter(R,add(val(V),Ent'2')))$$

formule qui demeurera dans l'état courant même si on change la valeur de X. En fait, la relation que l'on veut établir entre un identificateur de procédure et une modification doit vérifier une propriété plus forte que la propriété donnée en (1), et qu'on peut écrire :

$$poss(p) = m \in S, \exists m' t.q. S' = appl(m',S) \Rightarrow appl(poss(p),S') = appl(m,S')$$

C'est-à-dire que quels soient les changements d'état possibles, l'équivalence des deux applications reste vraie. D'où la nécessité de limiter cette "égalité" entre modifications à l'identité. Pour cela, on doit changer la définition d'un état qui a été donnée au chapitre précédent : en effet les noms d'opérations à résultat de type modification ne doivent pas vérifier la propriété iv)

$$x = y \Rightarrow f(x) = f(y)$$

La définition 3 du chapitre III devient :

Définition : Soit <T,F,A> la présentation du type abstrait associé à un langage de programmation. Soit O l'ensemble des noms d'opération dont le résultat n'est pas de type Modif. On appelle état un ensemble S de formules de la forme t=t' qui vérifie les propriétés i) ii) et iii) données au chapitre III et la propriété suivante :

iv) pour tout f de O, pour tout $t_1, \ldots t_n$ et $t'_1, \ldots t'_n$ tels que les types des t_i correspondent à l'arité de f on a :
$t_i = t'_i \wedge pré(f(t_1,\ldots t_n)) \wedge pré(f(t'_1,\ldots t'_n)) \in S \Rightarrow$
$f(t_1,\ldots t_n) = f(t'_1,\ldots t'_n) \in S$.

On peut alors conserver la règle (1).

La déclaration considérée précédemment aura pour effet d'introduire la formule

poss(P) = affecter(R,add(Val(V),Val(X)))

et elle seule, par la substitution

subst(poss(P), affecter(R,add(Val(V),Val(X)))

IV.1.3. - Les fonctions sémantiques

Les différentes fonctions sémantiques utilisées pour la définition de SIMPROC sont les suivantes :

$\mathcal{M}$: P → terme de type Modif
$\mathcal{S}$: Proc-id x {DuLDuIuLI} → terme de type Modif
$\mathcal{L}$: Proc-id x VAR → terme de type Var
$\mathcal{V}$: Proc-id x {EuT} → terme de type Ent

Les symboles P,D,I,LD,LI,VAR,E et T sont ceux utilisés comme non-terminaux dans la grammaire de SIMPROC donnée au début du chapitre. $\mathcal{S}$, $\mathcal{L}$ et $\mathcal{V}$ ont un identificateur de procédure comme premier paramètre ; celui-ci indique la procédure à laquelle la partie de programme considérée est locale. (3)

(3) Il est possible d'écrire ces fonctions sémantiques sans utiliser ce paramètre : en effet, cet identificateur peut être noté, à tout endroit d'un programme, comme le résultat de "nom(dernier-appel)". Mais cette solution a pour conséquence des implantation inefficaces : la fonction "nom" doit alors être représentée et calculée à l'exécution alors que son résultat peut toujours être déduit du texte du programme. Nous reviendrons sur les problèmes posés par la distinction des opérations statiques et dynamiques à la fin de ce paragraphe.

a) La fonction $\mathcal{M}$

Un programme est formé d'une liste de déclarations et d'une liste d'instructions qui sont locales ... à ce programme. Sa sémantique est la composition de la sémantique de ces deux listes, précédée d'une initialisation de la chaîne des appels :

$$\mathcal{M}\llbracket LD;LI \rrbracket = concat^3(init, \mathcal{S}_{PROG}\llbracket LD \rrbracket, \mathcal{S}_{PROG}\llbracket LI \rrbracket)$$

Au point de vue des notations, "$concat^n$" indique la concaténation de n modifications, et $\mathcal{S}_P\llbracket U \rrbracket$ est utilisé à la place de $\mathcal{S}$(Proc-id'P',U), ceci afin de faciliter la lecture des équations sémantiques.

La définition de la modification "init" est la suivante :

Opération
( ) → Modif : init ;
Définition
init = subst(nom(dernier-appel),Proc-id'PROG') ;
fin

b) La fonction $\mathcal{S}$

Tout d'abord nous donnons la sémantique des déclarations :

$\mathcal{S}_P\llbracket LD;D \rrbracket$ = concat($\mathcal{S}_P\llbracket LD \rrbracket$, $\mathcal{S}_P\llbracket D \rrbracket$) ;

$\mathcal{S}_P\llbracket$ entier ID $\rrbracket$ = decl-int(Var-id'ID',Proc-id'P') ;

$\mathcal{S}_P\llbracket$ tableau ID[ENT] $\rrbracket$ = decl-tab(Tab-id'ID',Proc-id'P',Ent'ENT') ;

$\mathcal{S}_P\llbracket$ proc ID1(val ID2, var ID3) début LD;LI fin $\rrbracket$ =
decl-proc(Proc-id'ID1',Proc-id'P',Var-id'ID2',
Ref-id'ID3',concat($\mathcal{S}_{ID1}\llbracket LD \rrbracket$, $\mathcal{S}_{ID1}\llbracket LI \rrbracket$)) ;

Les définitions des modifications utilisées sont les suivantes :

opérations
(Var-id,Proc-id) → Modif : decl-int ;
(Tab-id,Proc-id,Ent) → Modif : decl-tab ;
(Proc-id,Proc-id,Var-id,Ref-id,Modif) → Modif : decl-proc ;

<u>définitions</u>
<u>soit</u> vid ∈ Var-id,tid∈Tab-id, pid,pid'∈Proc-id, rid∈Ref-id, m∈Modif,
i ∈ Ent ;
decl-int(vid,pid) = <u>subst</u> (portée(vid),pid) ;
decl-tab(tid,pid,i) = <u>subst</u> (<portée(tid),bornesup(tid)>,<pid,i>) ;
decl-proc(pid,pid',vid,rid,m) = <u>subst</u> (<portée(pid),
par1(pid),par2(pid), poss(pid)>,<pid',vid,rid,m>) ;
<u>fin</u>

Ces définitions spécifient par des substitutions simultanées les caractéristiques des identificateurs déclarés : en effet l'ordre dans lequel sont définies ces caractéristiques n'a aucune importance.

On peut noter que la modification possédée par un identificateur ID1 est "concat(S_{ID1} ⦃LD⦄,S_{ID1} ⦃LI⦄)" c'est-à-dire que le premier argument de la fonction sémantique est modifié.

La plupart des propriétés définies par ces modifications sont statiques et peuvent donc être vérifiées à la traduction : c'est le cas de l'opération "portée" par exemple. On peut alors se demander pourquoi cette opération n'est pas un paramètre de la fonction sémantique. Une réponse est que les propriétés de "portée" sont globales au programme : étant donné un identificateur id utilisé correctement, "portée(id)" correspond toujours le même résultat en tout endroit du programme. Ce n'est pas le cas de "nom (dernier-appel)" que l'on sait calculer statiquement mais dont le résultat varie selon le point du programme considéré : c'est une propriété statique "locale" et de telles propriétés sont intéressantes à calculer au moment de l'évaluation sémantique car elles évitent des calculs lors de la future exécution du programme.

En outre, <u>d'un point de vue formel</u>, il est normal de définir la sémantique d'un langage de programmation sans se préoccuper des aspects statiques et dynamiques de ce langage. La distinction peut d'ailleurs être variable selon l'implantation choisie, voire même inexistante si on interprète le langage.

C'est pourquoi les opérations 'statiques', celles qui sont calculées lors de la traduction ne sont pas en général distinguées des autres dans la définition d'un langage source. Par contre, dans la spécification des choix de représentation cette distinction apparaît tout naturellement : par exemple dans le cas de SIMPROC où les bornes de tableaux sont constantes, l'opération bornesup est représentée par une constante (voir le chapitre V).

Il faut, bien sûr, avoir les éléments pour calculer ces constantes lors de la traduction(4). Ceci est fait à l'aide de tables qui, en toute rigueur, devraient être construites lors de la deuxième phase de traduction, au fur et à mesure que l'on traite les déclarations du terme source : ces déclarations sont représentées par une modification inopérante et par une tabulation des opérations statiques comme 'portée' ou 'bornesup'. Les tables ainsi obtenues sont utilisées pour traduire les termes du genre 'bornesup(t)' par des constantes du type objet.

<u>D'un point de vue pratique</u>, ceci est quelque peu irréaliste : cela retarde en effet considérablement tout diagnostic d'erreur. Dans le système PERLUETTE, ces tables sont construites à la première phase et le premier texte intermédiaire n'est produit que s'il n'y a pas d'erreurs statiquement détectables. Une fois construites, ces tables sont transmises à la deuxième phase de traduction pour éviter d'avoir à les reconstruire.

C'est l'utilisateur de PERLUETTE, c'est-à-dire le concepteur du compilateur qui décide des vérifications à effectuer dès la première phase. L'idéal serait de pouvoir déduire systématiquement les règles statiques à partir de la définition sémantique et de la description des aspects 'dépendants du contexte' de la syntaxe. On n'en est pas là, mais en examinant systématiquement les préconditions, on a un inventaire des vérifications à effectuer : il reste à les identifier comme 'statiques' ou 'dynamiques'.

(4) Rappelons que la spécification des choix de représentation décrit, en pratique, la deuxième phase de traduction du compilateur : Terme Source → Terme Objet

Considérons maintenant la sémantique des instructions, sauf pour les appels de procédure qui seront étudiés au paragraphe suivant :

S_P ⦃ LI;I ⦄ = concat(S_P ⦃ LI ⦄ , S_P ⦃ I ⦄) ;

S_P ⦃ VAR:=E ⦄ = affecter(L_P ⦃ VAR ⦄ , V_P ⦃ E ⦄) ;

S_P ⦃ <u>si</u> E_1 <u>eg</u> E_2 <u>alors</u> LI_1 <u>sinon</u> LI_2 <u>fsi</u> ⦄ = cond(V_P ⦃ E_1 ⦄ , V_P ⦃ E_2 ⦄ , S_P ⦃ LI_1 ⦄ , S_P ⦃ LI_2 ⦄) ;

c) Les fonctions V et L

V est définie de manière très analogue à ce qui était fait au chapitre précédent :

V_P ⦃ ENT ⦄ = Ent'ENT' ;

V_P ⦃ VAR ⦄ = val(L_P ⦃ VAR ⦄) ;

V_P ⦃ E+T ⦄ = add(V_P ⦃ E ⦄ , V_P ⦃ T ⦄) ;

et ainsi de suite pour les trois autres opérations. Pour L , on a :

L_P ⦃ ID ⦄ = <u>si</u> ID ∈ Ref-id
<u>alors</u> ref(ref-des(Ref-id'ID',dernier-appel))
<u>sinon</u> dés(Var-id'ID',dernier-appel) ;

L_P ⦃ ID[E] ⦄ = elt(Tab-id'ID',dernier-appel, V_P ⦃ E ⦄) ;

Les différentes restrictions ((4),(8),...) données pour l'utilisation des identificateurs ne sont pas testées car on a convenu qu'on se limitait à des programmes corrects de ce point de vue.

Le <u>si</u>-<u>alors</u>-<u>sinon</u> utilisé en partie droite de L_P ⦃ ID ⦄ est une méta-conditionnelle. Dans le système PERLUETTE, la construction du terme correspondant à un programme est programmée par des définitions d'attributs sémantiques écrites en LISP : on aura donc un COND de LISP dont la première condition comporte une recherche en table.

IV.1.4. - <u>L'appel de procédure</u>

La sémantique d'un appel de procédure correspond à un changement d'environnement : les identificateurs locaux à la procédure désignent de nouvelles variables. On doit passer les paramètres et les expressions en argument doivent être évaluées dans l'ancien environnement. On doit exécuter le corps de la procédure puis restaurer l'environnement précédent.

On va se donner la modification suivante :

<u>Opération</u>
(Proc-id,Int,Var,Proc-id) → Modif:appel ;

et on a l'équation sémantique :

S_P ⦃ <u>appeler</u> ID(E,VAR) ⦄ = appel(Proc-id'ID', V_P ⦃ E ⦄ , L_P ⦃ VAR ⦄ , Proc-id'P')

avec la précondition suivante :

<u>restriction</u>
<u>Pre</u>(appel,pid1,i,v,pid2) = ancètre(pid2,pid1) ∨ eg(portée(pid1),portée(pid2)) ∨ eg(portée(pid1),pid2) ;

Cette précondition exprime la règle bien connue qui est qu'on ne peut appeler depuis une procédure pid2, qu'une procédure englobante ou une procédure locale à la même procédure que pid2 ou une procédure déclarée dans pid2.

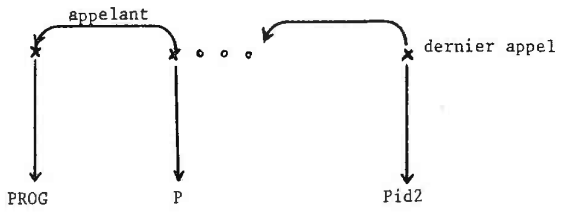
La définition de la modification appel, pour des raisons de lisibilité va se faire en utilisant des modifications auxiliaires :

appel(pid1,i,v,pid2) = $concat^3$(chgt-env(pid1,i,v),poss(pid1), rest-env) ;

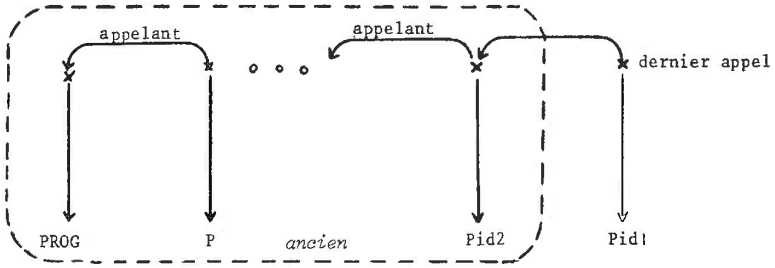
"chgt-env" correspond au changement d'environnement et au passage des paramètres ; "rest-env" correspond à la restauration de l'environnement.

La signification d'un changement d'environnement en terme d'états et de modifications est la suivante :

- le type Appel est enrichi d'une valeur ; en effet si dans l'état d'origine ce type pouvait être représenté graphiquement de la manière suivante :



dans l'état résultant on peut le schématiser par :



- d'autre part, toutes les formules de l'état d'origine où apparaît "dernier-appel" doivent être remplacées dans l'état résultant par une formule obtenue en remplaçant "dernier-appel" par "appelant (dernier-appel)" ;

si on a :

val(dés(id,appelant^{n}(dernier-appel))) = i

dans l'état d'origine, on veut avoir :

val(dés(id,appelant^{n+1}(dernier-appel))) = i

dans l'état résultant. Les axiomes sur les variables permettent de savoir si cette valeur est accessible ou non depuis le dernier-appel de l'état

résultant, et que les variables locales à ce dernier-appel sont différentes de toutes les autres variables.

La définition formelle de ce changement est :

f ∈ S ⟹ f [appelant(dernier-appel)/dernier -appel] ∈ S' ;
nom(dernier-appel) = pid1 ∈ S' ;

C'est une modification différente des affectations traitées jusqu'ici : en effet, on a créé une nouvelle valeur dans le type abstrait "Appel". Cette notion de création d'un nouvel objet est fréquente dans les langages de programmation où elle correspond généralement à une allocation dynamique de mémoire. Il est difficile d'en rendre compte dans les types abstraits ou, plus généralement dans les systèmes formels, autrement que par une modification. En effet, par définition, tout constructeur retourne toujours la même valeur pour les mêmes arguments. Or, ici on veut une valeur de type "Appel" nouvelle pour chaque changement d'environnement : cela peut se faire par une opération à condition d'introduire un compteur.

Il peut sembler choquant d'effectuer une substitution sur une "constante". En fait, ici ainsi que dans le cas général, il ne s'agit pas d'un substitution au sens classique, mais d'un changement de système formel : le type Appel a été sérieusement modifié.

Il reste à exprimer cette modification en terme de <u>subst</u> puisque c'est la modification primitive. Comme dans la définition de <u>subst</u> on va se servir de l'opération auxiliaire dernier-appel', auquel on substituera ensuite dernier-appel. Moyennant cette astuce, le changement d'appel est défini par la substitution simultanée suivante :

<u>subst</u>(<appelant(dernier-appel'),dernier-appel>,
<dernier-appel,dernier-appel'>)

Soit S' l'état résultant de cette substitution appliquée à un état S. On a par définition de <u>subst</u>

S' = {P|P[appelant'/appelant,dernier-appel'/dernier-appel] ∈
S + appelant'(x) = <u>si</u> eg(x,dernier-appel') <u>alors</u>
dernier-appel <u>sinon</u> appelant(x)
+ dernier-appel' = dernier-appel'}

On peut vérifier que pour toute formule

$$f(\text{appelant}^n(\text{dernier-appel})) \in S \Longrightarrow f(\text{appelant}^{n+1}(\text{dernier-appel})) \in S'$$

En effet si $f(\text{appelant}^n(\text{dernier-appel})) \in S$, on a dans l'état intermédiaire :

$$f(\text{appelant}'^{n+1}(\text{dernier-appel}')) = f(\text{appelant}'^n(\text{dernier-appel}))$$

d'après le premier axiome ajouté, donc

$$= f(\text{appelant}^n(\text{dernier-appel}))$$

toujours d'après cet axiome. Or cette formule appartient à S, donc à l'état intermédiaire. On a par conséquent :

$$f(\text{appelant}^{n+1}(\text{dernier-appel})) \in S'$$

On va maintenant donner la définition complète de la modification "chgt-env" en prenant en compte le passage des paramètres :

chgt-env(pid1,i,v) = <u>subst</u>(<appelant(dernier-appel'),
dernier-appel,nom(dernier-appel'),
val(dés(par1(pid1),dernier-appel')),
ref(ref-dés(par2(pid1),dernier-appel'))>,
< dernier-appel,dernier-appel',pid1,i,v>)

i et v ne comportant que des occurences de dernier-appel, l'évaluation des arguments et bien faite dans l'ancien environnement, tandis qu'ils sont affectés à des variables du nouvel environnement.

La restauration de l'environnement est tout à fait simple :

rest-env = <u>subst</u>(dernier-appel,appelant(dernier-appel))

D'après la définition de <u>subst</u>, si

S' = <u>appl</u>(rest-env,S)

S' = {P|P[dernier-appel'/dernier-appel] ∈
S+ dernier-appel' = appelant(dernier-appel)}

Si dans S, on avait une formule de la forme

$$f(\text{appelant}^n(\text{dernier-appel})) \in S \quad n \geq 1$$

$f(\text{appelant}^{n-1}(\text{dernier-appel}) \in S'$ - En effet :

$$f(\text{appelant}^{n-1}(\text{dernier-appel}')) = f(\text{appelant}^n(\text{dernier-appel})$$
$$\in S+ \text{dernier-appel}' = \text{appelant}(\text{dernier-appel})$$

Par contre, toutes les formules relatives au dernier appel de S sont éliminées ; en particulier toutes celles du type (1)

val(des(id,dernier-appel)) = i ∈ S

où id est local au dernier-appel disparaissent. Cela ne serait pas le cas si on utilisait, par exemple, une pile d'appels.

Supposons qu'on ait une opération du genre empiler(dernier-appel, Proc-id'P') qui retourne un appel de nom P dont l'appelant est dernier-appel. Au moment de la restauration, les formules de type (1) disparaissent mais celles de la forme

val(des(id,empiler(dernier-appel,Proc-id'P')) = i

restent - ce qui provoquera un nouvel appel de P avec ses variables locales initialisées à leurs anciennes valeurs ...

Cette spécification de l'appel de procédure à l'avantage de décrire la création de nouvelles variables sans préciser le type d'allocation locale que l'on aura : les problèmes de taille et d'ordre en mémoire n'interviennent pas dans la définition, ce qui est très interessant car cela laisse une grande liberté à l'implanteur du langage. C'est, par rapport aux autres approches mentionnées au début de ce chapitre un point très positif, en plus du fait que cette description traite le problème des variables globales et locales et des paramètres sans aucune restriction.

IV.2 - Sémantique d'autres constructions courantes dans les langages de programmation

IV.2.1 - Remarque sur le profil des fonctions sémantiques

Jusqu'ici on n'a envisagé que des langages simples où l'évaluation des expressions ne provoque pas de changement d'état (pour reprendre la terminologie courante, il n'y a pas 'd'effets de bord'). Cela a pour conséquence que dans les exemples précédents (à l'exception de III.3.3) une instruction a pour signification un terme de type Modif, et une expression un terme ne faisant par intervenir de modifications.

Ceci est insuffisant dès l'instant où on veut rendre compte de langages où il est possible de définir des fonctions et de modifier des variables globales dans le corps de ces fonctions : la sémantique d'une expression où apparaît un appel d'une telle fonction doit rendre compte de ce changement d'état.

Une solution est de modifier le profil des fonctions V et L du paragraphe précédent et de considérer que le résultat de ces fonctions est un couple <changement d'état, résultat>. C'est ce qui est fait dans [FIN 74] pour décrire la sémantique d'Algol 68. Cependant dans d'autres langages une fonction peut effectuer des calculs après avoir évalué son résultat : supposons qu'on enrichisse le langage SEQFLEX décrit en III.3 d'un opérateur dernier qui, étant donné un identificateur de séquence, rend la valeur du dernier élément et le supprime de la séquence ; la suppression doit être décrite comme suivant l'accès à la valeur du dernier élément. Plutôt qu'un couple, il conviendrait donc d'envisager la sémantique d'une expression comme un triplet de termes <m1,v,m2>, où m1 et m2 sont des termes de type Modif décrivant les changements d'états à effectuer avant et après l'évaluation du résultat, qui est décrite par le terme v.

La manipulation de tels triplets quand on compose des expressions, alourdit considérablement l'écriture des fonctions sémantiques. C'est pourquoi il est proposé ici une autre solution, qui n'est pas fondamentalement différente de la précédente mais qui est d'un usage plus facile. On va se donner des opérations de composition des modifications et des valeurs des autres types ;

par exemple :

(Modif,Ent,Modif) → Ent : effet-indirect ;

Cette opération se compose avec les autres opérations, qu'elles retournent des modifications ou non, selon des règles qui doivent être spécifiées. Un exemple de telles règles (qui d'une certaine manière sont des schémas d'axiomes) est donné ci-dessous.

Régle 1 Pour tout f de profil Ent^n → Ent, on a :

```
f(effet-indirect(m1,e1,m1'),...,effet-indirect(mn,en,mn')) =
  effet-indirect(concat^3n(m1,affecter(v1,e1),m1',...,
                           mn,affecter(vn,en),mn'),
                 f(val(v1),...,val(vn)),vide)
```

où v1,...vn sont des variables différentes entre elles et différentes de tous les termes 'dés(id,dernier-appel)' et 'elt(id,i,dernier-appel)' qui vérifient les préconditions dans l'état de départ.

Cette règle correspond à la compréhension intuitive que l'on a de la signification du terme f(x1,...xn) : évaluer séquentiellement les xi en les stockant dans des variables intermédiaires puis calculer f. (Comme on l'a déjà dit, on peut se donner une règle différente).

Régle 2 Pour tout m de profil Ent^n → Modif, on a :

```
m(effet-indirect(m1,e1,m1'),...,effet-indirect(mn,en,mn')) =
  concat^3n+1(m1,affecter(v1,e1),m1',...
              mn,affecter(vn,en),mn',
              m(val(v1),...,val(vn)))
```

Comme exemple d'utilisation de cet opérateur nous donnons la sémantique de l'opérateur dernier évoqué au début de ce paragraphe.

```
V⦃dernier(ID)⦄ = effet-indirect(vide,ième(Idseq'ID',
                                bornesup(Idseq'ID')),
                                retirer(Idseq'ID') )
```

Dans la suite de ce chapitre nous utilisons cet opérateur pour décrire la sémantique des fonctions et nous prenons pour convention les règles 1 et 2. Le choix de cette description des effets indirects permet de ne pas changer les profils des fonctions sémantiques V_P et L_P définies en IV.1.

De plus l'introduction des effets indirects avec les règles 1 et 2 permet de ne pas modifier la définition des compositions de modifications. On doit cependant introduire des opérations d'effet indirect pour tous les types t pour lesquels il existe des opérations ayant des entier en opérande et retournant un résultat de type t qui peut lui-même apparaître en opérande d'une modification. C'est le cas du type Bool dans l'exemple de SEQFLEX.

On va donc avoir parmi les opérations :

(Modif,Bool,Modif) → Bool : effet-indirect.b ;

On aura une règle 1.b de composition des opérations d'effet indirect avec les opérations à résultat de type Bool et à opérandes de type Bool ou Ent. De même, on aura une règle 2.b de composition de 'effet-indirect.b' avec les modifications ayant des opérandes de type Bool. Cette règle 2.b permet de ne pas modifier la définition des conditionnelles ou des itérations (s'il y en a). En effet, dans le cas de SEQFLEX, cette règle indique que, par exemple :

appl(conditionnelle(effet-indirect.b(m,b,vide),m1,m2),S) =
    appl(concat(m,conditionnelle(b,m1,m2)),S) =
    appl(conditionnelle(b,m1,m2),appl(m,S))

ce qui est bien la définition d'une conditionnelle dont l'évaluation de la condition provoque un changement d'état.

La nécessité de variables intermédiaires pour décrire les compositions entre modifications et autres termes est évitée si, comme on le suggère en III.5 on introduit la notion d'état dans le type abstrait algébrique et

une opération eval qui retourne la valeur d'un terme en fonction d'un état donné. On a alors à la place des règles 1 et 2 :

eval(f(effet-indirect(m1,e1,m1'),...effet-indirect(mn,en,mn')),S) =
  f_I(eval(e1,appl(m1,S)),eval(e2,appl(concat^3(m1,m1',m2),S)),...
    eval(en,appl(concat^{2n-1}(m1,m1',...,mn),S)))

appl(m(effet-indirect(m1,e1,m1'),...,effet-indirect(mn,en,mn')),S) =
    m_I(eval(e1,appl(m1,S)),...,eval(en,appl(concat^{2n}(m1,...mn'),S)))

où f_I et m_I sont les fonctions correspondant aux symboles fonctionnels f et m. Il devient d'ailleurs inutile de distinguer les opérations à résultat de type Modif des autres si on définit appl et eval pour tout terme [PG 79].

IV.2.2 - Les fonctions

On enrichit la grammaire du langage SIMPROC des règles suivantes :

<D> → fct ID(val ID,var ID) début <LD> ; <LI> fin
<I> → retour <E>
<T> → ID (<E>,<VAR>)

L'instruction retour doit apparaître dans le corps d'une fonction ; elle peut éventuellement apparaître plusieurs fois dans le corps d'une même fonction. On se limite à des fonctions qui rendent un résultat entier.

Pour rendre compte de cette extension dans le type abstrait associé au langage on ajoute le nom de type suivant :

type Fct-id ;

d'où un changement dans la définition du type Id :

type Id = union(Var-id,Tab-id,Proc-id,Fct-id,Ref-id) ;

Ce nouveau type ayant un rôle très similaire à celui de Proc-id, le profil des fonctions sémantiques va être légèrement modifié, car le premier argument pourra être soit un identificateur de procédure, soit un identificateur de fonction (cf IV.1.3) :

$\mathcal{M}$: P → terme de type Modif
$\mathcal{S}$: {Proc-id∪Fct-id} x {D∪LD∪I∪LI} → terme de type Modif
$\mathcal{L}$: {Proc-id∪Fct-id} x VAR → terme de type Var
$\mathcal{V}$: {Proc-id∪Fct-id} x {E∪T} → terme de type Ent

De même un certain nombre d'opérations qui ont des identificateurs de procédure en argument et qui sont relatifs aux portées vont avoir leur profil modifié :

<u>opérations</u>
(Id) → <u>union</u>(Proc-id,Fct-id) : portée ;
(Appel) → <u>union</u>(Proc-id,Fct-id) : nom ;
(<u>union</u>(Proc-id,Fct-id)) → Var-id : par1 ;
(<u>union</u>(Proc-id,Fct-id)) → Ref-id : par2 ;
(Id,<u>union</u>(Proc-id,Fct-id)) → Bool : ancêtre ;

On modifie de même, dans le type Modif, le type du deuxième argument de 'decl-int','decl-tab' et 'decl-proc', les entités déclarées pouvant être locales soit à une procédure, soit une fonction.

Ces changements dans le type abstrait correspondent au fait que l'unité de portée est maintenant une fonction ou une procédure. On va maintenant enrichir le type abstrait de manière à pouvoir rendre compte de l'appel ou de la déclaration d'une fonction. Une des manières de procéder, qui à notre avis présente l'avantage de changer très peu le type abstrait, est de partir du fait qu'une fonction a un double rôle : effectuer des calculs et retourner un résultat. D'où l'idée de considérer qu'un identificateur de fonction 'possède' une modification et 'désigne' une variable :

<u>opérations</u>
(<u>union</u>(Var-id,Fct-id),Appel) → Var : dés ;
(<u>union</u>(Proc-id,Fct-id)) → Modif : poss ;

La sémantique de <u>retour</u> devient une affectation à la variable désignée :

$\mathcal{S}_P$⦃<u>retour</u> E⦄ = <u>si</u> P ∈ Fct-id
<u>alors</u> affecter(dés(Fct-id'P',dernier-appel),$\mathcal{V}_P$⦃E⦄)
<u>sinon</u> Erreur ;

La déclaration d'une fonction est très semblable à la déclaration d'une procédure : la portée de son identificateur et de ses paramètres est définie et la liaison avec la sémantique du corps de la fonction est établie :

$\mathcal{S}_P$⦃<u>fct</u> ID(<u>val</u> ID1,<u>var</u> ID2)<u>début</u> LD ; LI <u>fin</u>⦄ =
<u>si</u> P ∈ Proc-id <u>alors</u>
decl-fct(Fct-id'ID',Proc-id'P',Var-id'ID1',Ref-id'ID2',
concat($\mathcal{S}_{ID}$⦃LD⦄,$\mathcal{S}_{ID}$⦃LI⦄))
<u>sinon</u>
decl-fct(Fct-id'ID',Fct-id'P',Var-id'ID1',Ref-id'ID2',
concat($\mathcal{S}_{ID}$⦃LD⦄,$\mathcal{S}_{ID}$⦃LI⦄)) ;

avec pour définitions de 'decl-fct' :

(Fct-id,<u>union</u>(Proc-id,Fct-id),Var-id,Ref-id,Modif) → Modif : decl-fct ;

decl-fct(f,p,v,r,m) = <u>subst</u>(<portée(f),par1(f),par2(f),poss(f)>,
<p,v,r,m>) ;

Enfin l'appel d'une fonction utilise l'opérateur d'effet-indirect introduit en IV.2.1 :

$\mathcal{V}_P$⦃ID(E,VAR)⦄ = effet-indirect(appel(Fct-id'ID',$\mathcal{V}_P$⦃E⦄,$\mathcal{L}_P$⦃VAR⦄,Proc-id'P'),
val(dés(Fct-id'ID',dernier-appel)),vide) ;(1)

Le profil de 'appel' est modifié, mais sa définition reste identique(voir IV.1.4) ainsi que les préconditions. Le profil devient :

(<u>union</u>(Proc-id,Fct-id),Ent,Var,<u>union</u>(Proc-id,Fct-id)) → Modif : appel ;

---

(1) Pour être rigoureux il faudrait comme pour la déclaration, tester si P est un identificateur de fonction ou de procédure.

Etant donné la définition de 'appel', les appels de fonction en argument d'appels de fonction ou de procédure ne posent pas de problème : ils sont effectués et évalués dans l'environnement appelant. Du fait de la définition de 'effet-indirect', il n'y a pas de problème non plus si une fonction est appelée plusieurs fois en argument d'un même appel : le résultat de chaque appel sera conservé dans une variable intermédiaire.

En résumé, l'introduction des fonctions nécessite l'adjonction au type abstrait d'un opérateur de composition de modification et d'entier ('effet-indirect'). C'est le changement le plus fondamental les autres étant plus mineurs :

- pour les portées on travaille sur l'union des identificateurs de fonctions et de procédure ;
- on modifie les profils de 'dés' et 'poss' sans en changer les axiomes ;
- on introduit une modification déclaration de fonction ; on pourrait introduire également une opération appel de fonction à résultat entier. Cela n'a pas été fait ici, cette opération se décrivant de manière immédiate à l'aide d''effet-indirect'.

IV.2.3 - Conclusion

La relative facilité avec laquelle on introduit le concept de fonction dans SIMPROC est encourageante : elle semble confirmer la 'bonne qualité' du modèle qu'on s'est donné pour les procédures. Une autres extension de SIMPROC [Gau 80] avec des tableaux à bornes variables se fait très simplement aussi. Ce n'est pas vrai de toutes les extensions (par exemple l'introduction de sous-tableaux à la Algol 68 demande de revoir sérieusement ce qui est fait pour les tableaux) mais jusqu'ici, le type abstrait présenté en IV.1 n'a jamais été remis complètement en cause.

Une autre remarque encourageante, soulignée par M. Sintzoff, est la remarquable convergence de cette présentation avec la définition révisée d'Algol 68 [VW 76], où les notions de 'Nest' et d''environ' sont très semblables aux notions de portée et d'appel utilisée ici. La différence est qu'ici ces deux notions sont complètement formalisées axiomatiquement.

Il n'en reste pas moins qu'un travail important et fort intéressant reste à faire : montrer la consistance de cette définition des procédures avec d'autres définitions comme l'a fait Donahue dans [Don 76] pour les approches axiomatiques et dénotationnelles de la sémantique.

Il faut noter que pour faciliter la tâche du lecteur, les équations sémantiques ont été considérablement simplifiées tout au long de cette présentation : on a considéré en effet que les programmes étaient corrects. La définition complète de SIMPROC comporte de nombreux tests dans la partie droite de ces équations, tests qui sont de deux types : si la condition (généralement la vérification d'une précondition ou l'appartenance d'une constante à un type) peut être vérifiée statiquement, on utilise le si-alors-sinon rencontré précédemment qui correspond à une définition sémantique conditionnelle. Si la condition ne peut être vérifiée statiquement, alors la valeur sémantique est une conditionnelle c'est-à-dire dans l'exemple précédent, un terme commençant par le symbole fonctionnel 'cond'.

CHAPITRE V

SPECIFICATION DES CHOIX D'IMPLANTATION

Dans les chapitres précédents on a montré deux exemples de types abstraits, enrichis de définitions de modifications, permettant de définir deux langages de haut niveau : SEQFLEX et SIMPROC. Autrement dit, on a donné des Structures Sources utilisables pour spécifier des compilateurs de ces langages. Dans ce chapitre, on va donner un exemple de structure d'information définissant un langage de bas niveau. Puis on montrera comment spécifier la traduction des langages SEQFLEX et SIMPROC sous forme d'une représentation de leur Structure Source dans la Structure Objet ainsi définie.

V.1 - Exemple de Structure Objet

Le langage objet considéré est tout à fait classique : il permet de manipuler des valeurs hexadécimales et des adresses, de les charger dans un registre ou de les ranger à une adresse. Il est possible d'indexer une adresse par une valeur. On peut faire des comparaisons de valeurs ou d'adresses qui positionnent un code de condition. Enfin, une instruction peut être étiquetée et on dispose d'instructions de branchement inconditionnelle, conditionnelle et indirecte.

On va avoir parmi les noms de types, les types Valeur et Adresse.

Le type Valeur a une axiomatisation semblable à celle du type Ent, aux noms d'opérations près. Seule la définition de la division diffère puisqu'ici les tests donnent pour résultats des "codes de condition" et non plus des booléens. Dans un premier temps, nous donnons simplement les noms et les profils des opérations. Les axiomes seront donnés après la définition du type "code de condition".


```
type Valeur ;
opérations
   (Valeur, Valeur) → Valeur : plus, moins, mult, div ;
   (Valeur, Valeur) → Bool : eg ;
```

N.B. : il est nécessaire d'introduire cette opération d'égalité afin de pouvoir écrire des axiomes conditionnels et des préconditions. C'est une opération auxiliaire de la Structure Objet et qui ne peut en aucun cas représenter une opération de la Structure Source. Afin de bien mettre en évidence cette distinction, nous noterons les expressions conditionnelles sur les valeurs de la manière suivante : si v = v' alors x sinon y, et "eg" ne sera jamais explicitement utilisée.

```
axiomes
   ...
fin
```

Une autre manière de distinguer l'opétation 'eg' est de scinder la déclaration des opérations en deux parties quand il y a des opérations auxiliaires :

```
opérations
principales
   (Valeur, Valeur) → Valeur : plus, moins, mult, div ;
auxiliaires
   (Valeur, Valeur) → Bool : eg ;
axiomes
...
fin Valeur ;
```

De même on a :

```
type Adresse ;
opérations
principales
   (Adresse, Valeur) → Adresse : index ;
auxiliaires
   (Adresse, Adresse) → Bool : eg ;
```

```
axiomes
Soit a ∈ Adresse,v,v'∈Valeur ;
    index (a, Valeur '0') = a ;
    index (index (a,v), v') = index (a, plus (v, v')) ;
    eg (Adresse 'A', Adresse 'B') = eg (A,B) ;
    eg (a, a) = vrai ; etc ...
restrictions
    Pré (index, a, v) = ¬ (a = Adresse 'nil')
fin Adresse ;
```

La restriction indique qu'il est interdit d'indexer une adresse égale à 'nil'.

On a un type Registre qui comporte seulement des constantes et une opération d'égalité. Sa spécification est donc très simple :

```
type Registre ;
opérations
auxiliaires
   (Registre, Registre) → Bool : eg ;
axiomes
   eg(Registre 'R1', Registre 'R2') = eg (R1, R2) ;
fin Registre ;
```

Les registres et les adresses peuvent contenir indifférement des valeurs ou des adresses, qui peuvent changer d'un état à l'autre, d'où les définitions suivantes :

```
type Contenu = union (Valeur, Adresse) ;
opérations
   (Adresse) → Contenu : ca mod
   (Registre) → Contenu : cr mod
restrictions
   Pré (ca, a) = ¬ (a = Adresse 'nil') ;
fin Contenu ;
```


On peut maintenant donner la définition du type code de condition qui est le résultat de la comparaison de deux 'contenus'. Ce type comporte trois valeurs : lt, eq, gt. Le résultat d'une comparaison est contenu dans un registre interne. On accède au contenu de ce registre par l'opération 'cond'.

```
type Codecond ;
opérations
principales
   (      ) → Codecond : lt, eq, gt ;
   (Contenu, Contenu) → Codecond : test ;
   (      ) → Codecond : cond mod
auxiliaires
   (Codecond, Codecond) → Bool : eg ;
axiomes
Soit c,c'∈Contenu , a,a'∈Adresse, v,v'∈Valeur ;
test(v, plus (v', Valeur '1')) =
        si v = v'
        alors lt
        sinon si v = plus(v', Valeur '1')
              alors eq
              sinon test(v, v') ;
test(v, moins (v', Valeur '1')) =
        si v = v'
        alors gt
        sinon si v = moins (v', Valeur '1')
              alors eq
              sinon test (v, v') ;
test(Adresse 'nil', a) =
        si a = Adresse 'nil'
        alors eq
        sinon gt ;
```

```
test(a,index(a',Valeur '1')) =
        si a = a'
        alors lt
        sinon si a = index(a',Valeur '1')
              alors eq
              sinon test(a,a') ;
test(c',c) = si test(c,c') = eq
             alors eq
             sinon si test(c,c') = lt
                   alors gt
                   sinon lt ;
fin Codecond ;
```

On a maintenant les éléments pour donner les axiomes du type Valeur ;

```
axiomes
Soit v, v' ∈ Valeur ;
    plus(v,Valeur '0') = v ;
    plus(v,plus(v',Valeur '1')) = plus(plus(v,v'),Valeur '1') ;
    ...
    div(v,v') = si test(Valeur '0',v) = gt
                alors si test(Valeur '0',v') = gt
                      alors div(moins(Valeur '0',v),moins(Valeur '0',v'))
                      sinon moins(Valeur '0',div(moins(Valeur '0',v),v'))
                sinon si test(Valeur '0',v') = gt
                      alors moins(Valeur '0',div(v,moins(Valeur '0',v'))
                      sinon si test(Valeur '0',moins(v,v')) = gt
                            alors Valeur '0'
                            sinon plus(div(moins(v,v'),v'),Valeur'1') ;
```

N.B. : l'opération 'eg' sur les valeurs est définie comme l'opération 'eg' sur les entiers.

```
restriction
        test(v',Valeur '0') = eq  ⟹  Echec(div,v,v') ;
fin
```


On a mentionné la possibilité d'étiqueter des instructions dans le langage objet. On va donc se donner un type étiquette et une opération d'étiquetage des modifications.

```
type Eti ;
opérations
auxiliaires
   (Eti,Eti) → Bool : eg ;
principales
   (Eti,Modif) → Modif : étiqueter ;
axiomes
   eg(Eti 'E1',Eti 'E2') = eg(E1,E2)
fin
```

Pour pouvoir définir correctement les modifications étiquetées et les branchements, on a besoin d'enrichir ce type des deux opérations à résultat booléen suivantes :

```
opérations
auxiliaires
   (Modif,Eti) → Bool : sortie, entrée ;
```

Intuitivement, sortie (m,e) est vrai si le texte de la modification m comporte un branchement (conditionnel ou non) à l'étiquette e, et ne comporte pas de modification étiquetée par e · entrée (m,e) est vrai si m comporte une modification étiquetée par e ; 'entrée' sert uniquement à définir 'sortie' dans le cas des compositions de modifications.

La définition de ces prédicats va être donnée pour toute modification ou composition de modifications. Le type Modif de la structure Objet est défini ci-dessous :

```
type Modif ;
opérations
   (Contenu, Registre) → Modif : load ;
   (Contenu, Adresse) → Modif : store ;
   (Contenu, Contenu) → Modif : compare ;
   (Eti) → Modif : brcht ;
   (Eti, Codecond) → Modif : brcht-cond, brcht-neg ;
   (Modif, Modif) → Modif : compseq ;
   (    ) → Modif : suite ;
axiomes
Soit e, e' ∈ Eti,  c, c' ∈ Contenu, r ∈ Registre, a ∈ Adresse,
ccond ∈ Codecond, m, m' ∈ Modif ;
     entrée(load(c,r),e) = faux ;
     entrée(store(c,a),e) = faux ;
     entrée(compare(c,c'),e) = faux ;
     entrée(brcht(e'),e) = faux ;
     entrée(brcht-cond(e',ccond),e) = faux ;
     entrée(brcht-neg(e',ccond),e) = faux ;
     entrée(étiqueter(e',m),e) = si eg(e',e) alors vrai sinon entrée(m,e) ;
     entrée(comseq(m,m'),e) = entrée(m,e) ∨ entrée(m',e) ;
     entrée(suite,e) = faux ;
     sortie(load(c,r),e) = faux ;
     sortie(store(c,a),e) = faux ;
     sortie(compare(c,c'),e) = faux ;
     sortie(brcht(e'),e) = eg(e,e') ;
     sortie(brcht-cond(e',ccond),e) = eg(e,e') ;
     sortie(brcht-neg(e',ccond),e) = eg(e,e') ;
     sortie(étiqueter(e',m),e) = si eg(e',e) alors faux sinon sortie (m,e) ;
     sortie(compseq(m,m'),e) = si entrée(m,e) ∨ entrée(m',e)
                    alors faux sinon sortie(m,e) ∨ sortie(m',e) ;
     sortie(suite,e) = faux ;
```


On va utiliser ces prédicats pour définir les modifications, ou du moins les compositions de modifications susceptibles de représenter des modifications des structures sources : l'effet des branchements ne sera défini que dans quelques cas particuliers, ceux qui sont nécessaires.

définitions (ces définitions comportent des commentaires qui sont entre parenthèses)

```
load(c,r) = subst(cr(r),c) ;
store(c,a) = subst(ca(a),c) ;
compare(c,c') = subst(cond,test(c,c')) ;
```

(on spécifie dans la règle ci-dessous qu'un étiquetage ne change pas la signification d'une modification).

```
¬ sortie(m,e) ⇒ appl(étiqueter(e,m),S) = appl(m,S) ;
```

(on traite maintenant la composition séquentielle).

```
appl(compseq(étiqueter(e,m),m'),S) = appl(étiqueter(e,compseq(m,m')),S) ;
∀ e,sortie(m,e) = faux ⇒ appl(compseq(m,m'),S) = appl(m',appl(m,S)) ;
```

(la règle suivante définit le saut en avant inconditionnel).

```
appl(compseq³(brcht(e),m,étiqueter(e,m')),S) = appl(étiqueter(e,m'),S) ;
```

(on définit maintenant les sauts en avant conditionnels).

```
cc = cond ∈ S ⇒
     appl(compseq³(brcht-cond(e,cc),m,étiqueter(e,m')),S) =
                    appl(étiqueter(e,m'),S) ;
cc ≠ cond ∈ S ⇒
     appl(compseq³(brcht-cond(e,cc),m,étiqueter(e,m')),S) =
                    appl(compseq(m,étiqueter(e,m')),S) ;
```

```
cc = cond ∈ S ⟹
    appl(compseq³(brcht-neg(e,cc),m,étiqueter(e,m')),S) =
                        appl(compseq(m,étiqueter(e,m')),S) ;
cc ≠ cond ∈ S ⟹
    appl(compseq³(brcht-neg(e,cc),m,étiqueter(e,m')),S) =
                        appl(étiqueter(e,m'),S) ;
```

(on ne considère ici les sauts en arrière que dans le cas particulier suivant qui correspond à la représentation d'une itération :

Posons)

```
m = étiqueter(boucle,comseq⁵(m1,brcht-cond(exit,cc),
                             m2,brcht(boucle),étiqueter(exit,m3)))
```

(avec, pour toute étiquette e :)

```
sortie(m1,e) = faux
sortie(m2,e) = faux
```

(on a alors les deux règles suivantes :)

```
cond = cc ∈ appl(m1,S) ⟹
    appl(m,S) = appl(compseq(m1,étiqueter(exit,m3)),S) ;
cond≠cc ∈ appl(m1,S) ⟹
    appl(m,S) = appl(m,appl(compseq(m1,m2),S)) ;
```

(ces deux règles correspondent respectivement à un arrêt et à une itération d'une boucle. On peut se donner d'autres règles mais celles-ci sont suffisantes pour le cas étudié).

(il reste à définir la modification 'suite' qui correspond à une instruction vide :)

```
    appl(suite,S) = S ;
fin Modif ;
```

On a ainsi donné une structure d'information pour un langage objet qui a été choisi volontairement très 'ordinaire'. La spécification incomplète des branchements est sans importance du fait qu'on ne s'intéresse qu'à un

sous-ensemble du langage objet : celui des programmes susceptibles de représenter des programmes source.

De plus, pour décrire ces programmes, on est amené à ajouter dans la Structure Objet des opérateurs 'd'effet de bord' semblables à ceux qui ont été utilisés au Chapitre IV pour traiter les fonctions. En effet, il est extrêmement rare que toutes les opérations d'une Structure Source se représentent uniquement par des composition d'opérations de la Structure Objet, sans que des calculs auxiliaires soient nécessaires pour certaines d'entre elles : par exemple, pour l'implantation de SEQFLEX qui est donnée en V.2, la représentation de l'opération 'ième' de la Structure Source (qui a été donnée en III.3) sera la composition d'une modification (parcours d'une liste chaînée) et d'une opération (accès à un contenu) de la Structure Objet.

On va donc se donner des opérations correspondant à ce type de composition. Quoiqu'elles ne soient pas caractéristiques du langage objet, il est raisonnable de les considérer comme des opérations de la Structure Objet : elles apparaîtront donc dans le deuxième texte intermédiaire (Terme de la Structure Objet) de la traduction. Leur introduction évite en effet, d'avoir à décrire dans la représentation une gestion de résultats intermédiaires qui serait spécifique aux opérations de la Structure Source représentées par de telles compositions de modifications et d'opérations de la Structure Objet : on traitera le problème des résultats intermédiaires au moment de la génération de code, globalement pour toutes les opérations de la Structure Objet. On évite ainsi des inefficacités dans le code engendré qui viendraient d'une distinction, parfaitement injustifiée, entre deux types de résultats intermédiaires : ceux introduits lors de la traduction du Terme Source en Terme Objet et ceux introduits lors de la génération du code.

On ajoute donc à la Structure Objet les opérations suivantes :

```
opérations
    (Modif, Contenu) → Contenu : efb ;
    (Modif, Codecond) → Codecond : efbc, efbnc ;
```

Les deux dernières opérations sont nécessaires pour pouvoir exprimer le résultat de la comparaison de deux contenus dont le calcul entraîne des modifications. Ces trois opérations vérifient des règles analogues à celles qui ont été données au Chapitre IV sur les entiers pour pouvoir décrire les fonctions. En particulier on a

axiomes

ca(efb(m,a)) = efb(m,ca(a))

plus(efb(m,c),efb(m',c')) = efb(compseq^3(m,store(c,int),m'),plus(ca(int),c'));

(Cette propriété est également valable pour toutes les opérations de profil (Contenu, Contenu) → Contenu. 'int' est une adresse de sauvegarde du résultat intermédiaire, dont m' ne modifie pas le contenu et dont le choix n'est pas explicité ici. De même on a :)

test(efb(m,c),efb(m',c')) :
efbc(comseq^3(m,store(c,int),m'), test(ca(int),c')) ;

(par ailleurs :)

efb(m,efb(m',c)) = efb(compseq(m,m'),c) ;

efbc(m,efbc(m',cc)) = efbc(compseq(m,m'),cc) ;

fin

Pour ce qui est de la composition de ces opérations avec des modifications, plutôt que de donner les définitions correspondant à chaque modification de la Structure Objet, on va utiliser le fait qu'un Contenu apparaît toujours en opérande d'une modification définie par un subst et un Codecond apparaît toujours en opérande d'un branchement conditionnel : il suffit donc de se donner les règles de composition avec subst et avec les branchements conditionnels.

Pour le subst sur des termes de type Contenu on se donne :

subst(efb(m,c),efb(m',c')) = compseq^4(m',store(c',int),m,subst(c,c'))

(on évalue d'abord la partie droite de la substitution).

Dans le cas de termes de type Codecond on pose :

subst(cond,efb(m,cc)) = compseq(m,subst(cond,cc)) ;

(de par la définition du type Codecond, c'est le seul genre de substitution qu'on peut avoir).

Pour les branchements conditionnels on a les quatre définitions suivantes

Soit e ∈ Eti, m ∈ Modif, cc ∈ Codecond ;

brcht-cond(e,efbc(m,cc)) = compseq(m,brcht-cond(e,cc)) ;

brcht-cond(e,efbnc(m,cc)) = compseq(m,brcht-neg(e,cc)) ;

brcht-neg(e,efbc(m,cc)) = compseq(m,brcht-neg(e,cc)) ;

brcht-neg(e,efbnc(m,cc)) = compseq(m,brcht-cond(e,cc)) ;

fin

D'autre part, lors des preuves données dans le Chapitre VI on utilisera des versions simplifiées de la propriété donnée pour subst :

subst(c,efb(m',c')) = compseq(m',subst(c,c'))

si c ne contient pas d'occurrence de efb, et de même :

subst(efb(m,c),c') = compseq(m,subst(c,c'))

si c' ne contient pas d'occurrence de efb et s'il n'est pas "modifié" par m : on dit que c n'est pas modifié par m si pour tout état S tel que c = v ∈ S on a c = v ∈ appl(m,S).

Les règles données ici sont, on le verra, trop complètes pour les traductions de langages données en exemple car il s'avère que dans les termes objets produits lors de ces traductions, tout sous-terme de la forme

f(efb(m,c),efb(m',c'))

est tel que c n'est pas modifié par m' et c' n'est pas modifié par m ... et les sauvegardes sont inutiles.

Nous allons maintenant donner un premier exemple de spécification d'implantation, en indiquant dans le même temps quelle méthodologie nous semble appropriée pour l'écriture de telles spécifications.

V.2 - Implantation de SEQFLEX

Dans la Structure Source associée à un langage, on peut distinguer deux sortes de noms d'opération : ceux qui apparaîssent dans les équations sémantiques et ceux qui ne sont utilisés que dans les axiomes.

Du point de vue de la traduction, seule les premiers sont intéressants : en effet, ce sont ces noms d'opérations et eux seuls, qui vont apparaître dans le premier texte intermédiaire. La deuxième phase de traduction (terme du type Source vers terme du type Objet) est donc complètement spécifiée dès lors qu'on a donné une représentation pour chacun d'eux. Par contre, si on veut démontrer la correction d'une implantation, ce qui est l'objet du Chapitre VI, on est amené à spécifier la représentation de certaines opérations auxiliaires afin de vérifier que les axiomes sont conservés.

Avant d'écrire la spécification d'une représentation, il est parfois commode d'enrichir la Structure Objet en introduisant de nouvelles opérations, composées à partir de celles données au départ. Cela permet de concevoir la représentation d'une manière plus structurée, en passant par une étape où on se donne, à partir de la Structure Objet, des opérations adaptées à la représentation que l'on veut spécifier. De plus, cela permet de rendre cette spécification plus lisible. Il ne s'agit pas du tout de changer la Structure Objet, et les noms d'opérations qui apparaîtront dans le deuxième texte intermédiaire sont ceux des opérations de base. En fait ces opérations dérivées peuvent être considérées comme des macro-opérations.

Nous allons maintenant traiter un premier exemple. Il s'agit de l'implantation du langage SEQFLEX, tel qu'il a été défini en III.3.

Les choix d'implantation sont les suivants :

- les séquences sont représentées de manière chaînée ; par conséquent, on va avoir des listes chaînées de blocs de deux mots : un pour la valeur de l'élément de séquence, l'autre pour le pointeur vers un autre élément ;
- chaque identificateur de séquence est représenté par une adresse constante qui désigne un bloc de deux mots : dans le premier se trouve la longueur de la séquence ; dans le deuxième se trouve l'adresse du dernier élément de la séquence, ceci afin d'avoir une représentation plus efficace des instructions allonger et raccourcir. Lorsque la séquence est vide, le premier mot contient la valeur zéro et le deuxième contient l'adresse nil ;
- la mémoire est organisée avec une liste libre chaînée, de blocs de deux mots. L'adresse du premier bloc de cette liste est contenue à l'adresse 'libre' ;
- enfin, chaque identificateur de variable est représenté par une adresse constante, qui contient sa valeur.

La spécification d'une telle représentation est donnée sous la forme d'une liste de rubriques : une pour chaque type de la Structure Source et une pour chaque opération.

A l'intérieur de ces rubriques on trouve, dans le cas d'un type le nom du ou des types représentant et, quand le type comporte des constantes, la représentation de ces constantes est décrite. Dans le cas d'une opération, on trouve la description de sa représentation en terme d'opérations de la structure objet ou de la représentation de ces opérandes (c'est-à-dire que la spécification de la représentation des termes est récursive, tout comme l'est la définition de ρ donnée en II.4).

Pour décrire la traduction des constantes, mais aussi certains calculs qui doivent être faits à la traduction pour obtenir la représentation d'une opération, on a besoin de fonctions de traduction ou de manipulation de textes et de variables permettant d'introduire ces textes dans une représentation. Il s'agit de décrire les traitements effectuées 'à la compilation', en particulier on distinguera ce qui est appelé classiquement les aspects statiques du langage du fait que les opérations correspondantes sont représentées par des constantes dans la Structure Objet (ceci a déjà été discuté en IV.1.3). Encore faut-il spécifier le calcul de cette constante : on a donc besoin d'un métalangage pour spécifier les représentations.

Dans le système PERLUETTE, du fait que le terme Source et le terme Objet sont des formes LISP, et que la traduction de l'un dans l'autre est effectuée en utilisant un ensemble de fonctions LISP, ce métalangage est tout naturellement LISP. Ici, pour des raisons de lisibilité nous utilisons un métalangage à la ALGOL 60, comprenant des appels de fonctions, des

affectations, une instruction conditionnelle et une instruction de répétition.

Dans la présentation qui suit de la spécification de l'implantation de SEQFLEX selon les choix exposés plus haut, l'enrichissement de la Structure Objet par des macro-opérations est précisé au fur et à mesure des besoins. De même le rôle des différentes méta-procédures de traduction est décrit à leur première utilisation. Enfin les variables, noms de procédures et instructions du métalangage sont notés en majuscules.

On donne d'abord la représentation du type Ent et de ses opérations :

```
type Ent :
   Valeur ; (un entier est représenté par une valeur).
   repr Ent 'I' = Valeur 'CONVERTIR(I)' ;
fin Ent ;
```

(La procédure CONVERTIR a pour donnée une chaîne de caractères qui dénote un entier décimal. Elle a pour résultat une chaîne de caractères qui dénote le même nombre en héxadécimal).

```
op(Ent,Ent) → Ent : add ;
   repr add(i,j) = plus(repr i,repr j) ;
fin add ;
op(Ent,Ent) → Ent : soust ;
   repr soust(i,j) = moins(repr i,repr j) ;
fin soust ;
op(Ent,Ent) → Ent : mult ;
   repr mult(i,j) = mult(repr i,repr j) ;
fin mult ;
op(Ent,Ent) → Ent : div ;
   repr div(i,j) = div(repr i,repr j) ;
fin div ;
```

On donne maintenant la représentation du type identificateur de variable et de l'opération qui retourne la valeur d'un tel identificateur.


```
type Idvar ;
   Adresse ;
   repr Idvar 'X' = Adresse 'ALLOC1(X)' ;
fin Idvar ;
```

(La procédure ALLOC1 convertit l'identificateur X en une adresse : la chaîne X est recherchée dans la table des identificateurs déjà alloués. Si elle est présente son adresse est retournée, sinon X est entré dans la table avec l'adresse suivante, qui est rendue comme résultat).

```
op(Idvar) → Ent : valeur ;
   repr valeur (id) = ca(repr id) ;
fin valeur ;
```

On voit que la spécification d'une implantation est très systématique et assez facile à écrire. Nous allons maintenant donner une représentation du type Idseq. Pour cela, on va être amené à utiliser les opérations de composition des modifications et des opérations introduites à la fin de V.1 : en effet le calcul de 'ième' rend un entier : sa représentation est donc une opération qui retourne une valeur. Pour accéder à cette valeur on doit effectuer un parcours de liste chaînée ; très informellement la représentation de ième sera de la forme :

efb(parcours-de-la-liste-chaînée-pour-trouver-l'adresse-a-du ième-élément, ca(a))

La représentation des identificateurs de séquence et des opérations sur les séquences se spécifie de la manière suivante :

```
type Idseq
   Adresse ;
   repr Idseq 'S' = Adresse 'ALLOC2(S)' ;
fin Idseq
```

(La procédure ALLOC2(S) est très semblable à ALLOC1, mais deux mots sont alloués au lieu d'un).

```
op(Idseq) → Ent : bornesup ;
   repr bornesup(ids) = ca(repr ids) ;
fin bornesup ;
```

(La longueur de la séquence est contenue dans le premier mot).

```
op(Idseq,Ent) → Ent : ième ;
```

(La représentation de cette opération est plus complexe : les cas d'échecs doivent être testés ; on a besoin de registres de travail ; on veut générer une boucle, donc des étiquettes).

```
   repr ième(ids,i) = (# (affectations de métavariables)I:=GENREG ;
        A:=GENREG ; BOUCLE:=GENETI ; EXIT:=GENETI#
        efb(compseq[13](test des cas d'échec
                    load(repr i,Registre 'A') ;
                    compare(cr(Registre 'A'),Valeur '1') ;
                    brcht-cond(Eti 'erreur 1',lt) ;
                    load(moins(ca(repr ids),cr(Registre 'A')),Registre 'I'),
                    compare(cr(Registre 'I'),Valeur '0'),
                    brcht-cond(Eti 'erreur 2',lt),
                    fin des cas d'échec et début du parcours
                    load(ca(index(repr ids,Valeur 'I')),Registre 'A'),
                    étiqueter(Eti 'BOUCLE',brcht-cond(Eti 'EXIT',eq)),
                    load(moins(cr(Registre 'I'),Valeur '1'),Registre 'I'),
                    load(ca(index(cr(Registre 'A',Valeur '1')),Registre 'A'),
                    compare(cr(Registre 'I'),Valeur '0'),
                    brcht(Eti 'BOUCLE'),
                    étiqueter(Eti 'EXIT',suite)   ),
                    l'adresse du résultat est dans le registre A
           ca(cr(Registre 'A'))   )) ;
fin ième ;
```

Les métaprocédures GENREG et GENETI retournent sous forme de chaînes de caractères des noms de registres et d'étiquettes. GENREG retourne chaque fois un nouveau nom de registres : à ce niveau, on considère qu'on en a une infinité, l'allocation proprement dite est faite à l'étape suivante. Les métavariables I, A, BOUCLE et EXIT sont locales à la représentation de ième.

Or cette représentation peut être récursive car i peut très bien contenir une occurence de ième. Mais dans ce cas repr i utilisera d'autres versions de ces variables, c'est-à-dire des registres et des étiquettes différentes.

Il reste maintenant à donner la représentation des opérations de type booléen qui interviennent dans les équations sémantiques : 'eg' et 'dif'. On va représenter un booléen par un code de condition et pour exprimer la comparaison de deux valeurs on utilisera les opérateurs d'effet de bord efbc et efbnc mentionnés précédemment. Rappelons que ceux-ci sont caractérisés par les propriétés suivantes :

```
opérations
   (Modif,Codecond) → Codecond : efbc, efbnc ;
axiomes
Soit eti ∈ Eti, m ∈ Modif,ccond ∈ Codecond ;
   brcht-cond(eti,efbc(m,ccond)) = compseq(m,brcht-cond(eti,ccond)) ;
   brcht-cond(eti,efbnc(m,ccond)) = compseq(m,brcht-neg(eti,ccond)) ;
   brcht-neg(eti,efbc(m,ccond)) = compseq(m,brcht-neg(eti,ccond)) ;
   brcht-neg(eti,efbnc(m,ccond)) = compseq(m,brcht)cond(eti,ccond)) ;
fin
```

Comme on n'a pas de résultats intermédiaires, rien ne s'oppose à supprimer les occurences de 'efbc' et 'efbnc' dans l'étape de représentation : on pourrait donc faire de ces opérations des macro-opérations, et des quatres axiomes ci-dessus des règles de réécritures données en même temps que la représentation et appliquées à la PHASE2 de traduction. D'un point de vue méthodologique cela ne change que peu de choses à la spécification et rien à la preuve. On va donc les considérer comme des opérations.

La spécification de la représentation du type Bool s'écrit :

```
type Bool ;
   Codecond ;
fin
```

(Les constantes 'vrai' et 'faux' n'interviennent pas dans les équations sémantiques, leur représentation n'est pas donnée ici).

```
op(Ent,Ent) → Bool : eg ;
   repr eg(i,j) = efbc(compare(repr i,repr j),eq) ;
fin eg ;
```

(Composé avec un branchement conditionnel, ce terme se réécrira comme une composition : comparaison puis branchement si égalité).

```
op(Ent,Ent) → Bool : dif ;
   repr dif(i,j) = efbnc(compare(repr i,repr j),eq) ;
fin dif ;
```

Il reste à donner la représentation des modifications. On va auparavant se donner deux macro : libérer (a) supprime le premier bloc de la liste d'adresse a (c'est-à-dire que l'adresse de ce premier bloc est contenu dans index (a,1)) et le rajoute en tête de la liste libre ; occuper (a) supprime le premier bloc de la liste libre et l'ajoute en tête de la liste d'adresse a. On a les définitions suivantes :

```
macro-opérations
   (Adresse) → Modif : occuper, libérer ;
définitions
Soit a ∈ Adresse ;
   occuper (a) = compseq^6(
            compare(ca(libre),Adresse'nil'),
            brcht-cond(Eti 'débordement',eq),
            store(ca(libre),aux),
            store(ca(index(ca(aux),Valeur '1'),libre),
            store(ca(index(a,Valeur '1')),index(ca(aux),Valeur '1')),
            store(ca(aux),index(a,Valeur '1'))   ) ;
```



```
   libérer(a) = compseq^4(
            store(ca(index(a,Valeur '1')),aux),
            store(ca(index(ca(aux),Valeur '1')),index(a,Valeur '1')),
            store(ca(libre),index(ca(aux),Valeur '1')),
            store(ca(aux),ca(libre))) ;
fin
```

Les adresse 'libre' et 'aux' sont exclusivement réservées à la gestion de la liste libre.
Avant de donner la représentation des différentes modifications, rappelons brièvement les cas d'échec qui doivent être testés par la représentation : on doit vérifier qu'on ne retire pas le dernier élément d'une séquence vide et que lorsqu'on change le ième élément d'une séquence l'indice est bien compris entre les bornes.

```
op(Idseq) → Modif : init ;
   repr init(ids) = compseq(store(Valeur '0',repr ids),
                            store(Adresse 'nil',index(repr ids,Valeur '1'))) ;
fin init ;
```

(La longueur est initialisée à zéro. La liste ne contient aucun bloc).

```
op(Idseq,Ent) → Modif : ajouter ;
   repr ajouter(ids,i) = compseq^3(occuper(repr ids),
                  store(plus(ca(repr ids),Valeur '1'),repr ids),
                  store(repr i,ca(index(repr ids,Valeur '1')))   ) ;
fin ajouter ;
```

(Un bloc est ajouté en tête de la liste, la longueur est augmentée de 1, la valeur à ajouter est rangée dans le nouveau bloc).

```
op(Idseq) → Modif : retirer ;
   repr retirer(ids) = compseq^4(compare(ca(repr ids), Valeur '0'),
                  brcht-cond(Eti 'erreur3', eq),
                  libérer(repr ids),
                  store(moins(ca(repr ids), Valeur '1'), repr ids) ) ;
fin retirer ;
```

(Le cas d'échec : longueur de la liste égale à zéro, est testé ; le premier bloc est libéré et la longueur est diminuée de 1).

```
op(Idseq, Ent, Ent) → Modif : changer-ième ;
   repr changer-ième(ids, i, j) = ( # (initialisation des metavariables)
      I := GENREG ; A := GENREG ; BOUCLE := GENETI ;
      EXIT := GENETI #
      compseq^14 ( test des cas d'échec
               load(repr i, Registre 'A'),
               compare(cr(Registre 'A'), Valeur '1'),
               brcht-cond(Eti 'erreur1', lt),
               load(moins(ca(repr ids), cr(Registre 'A')), Registre 'I'),
               compare(cr(Registre 'I', Valeur '0'),
               brcht-cond(Eti 'erreur2', lt),
               fin des cas d'échec et début de la recherche du ième élément
               load(ca(index(repr ids, Valeur '1')), Registre 'A'),
               étiqueter(Eti 'BOUCLE', brcht-cond(Eti EXIT', eq)),
               load(moins(cr(Registre 'I'), Valeur '1'), Registre 'I'),
               load(ca(index(cr(Registre 'A'), Valeur '1')), Registre 'A'),
               compare(cr(Registre 'I'), Valeur '0'),
               brcht(Eti 'BOUCLE'),
               rangement de la nouvelle valeur à la ième place
               étiqueter(Eti 'EXIT', store(repr j, cr(Registre 'A')))   )) ;
fin changer-ième ;

op(Idvar, Ent) → Modif : affecter ;
   repr affecter(id,i) = store(repr i, repr id) ;
fin affecter ;
L'affectation est représentée par un rangement

op ( ) → Modif : rien ;
   repr rien = suite ;
fin rien ;
```

On a donné la représentation des modifications élémentaires. Il reste à spécifier l'implantation des compositions de ces modifications.


```
op(Modif, Modif) → Modif : concat ;
   repr concat(m, m') = compseq(repr m, repr m') ;
fin concat ;

op(Bool, Modif, Modif) → Modif : conditionnelle ;
   repr conditionnelle(b, m, m') = (#  NEG := GENETI ; EXIT := GENETI #
      compseq^5(brcht-neg(Eti 'NEG', repr b),
               repr m, brcht(Eti 'EXIT'),
               etiqueter(Eti 'NEG', repr m'),
               etiqueter(Eti 'EXIT', suite) )) ;
fin conditionnelle ;

op(Bool, Modif) → Modif : itération ;
   repr itération(b, m) = (# BOUCLE := GENETI ; EXIT := GENETI #
      compseq^4(etiqueter(Eti 'BOUCLE',
                        brcht-neg(Eti 'EXIT', repr b )),
               repr m, brcht(Eti 'BOUCLE'),
               etiqueter(Eti 'EXIT', suite))) ;
fin itération ;
```

On a ainsi complètement spécifié la deuxième phase de traduction de SEQFLEX : tous les symboles fonctionnels apparaissant en partie droite d'une équation sémantique ont été définis en terme d'opérations de la structure Objet. Il reste, pour finir de spécifier la traduction, à décrire la génération du code proprement dit. Un travail important est nécessaire : prouver que l'on a donné une implantation correcte de SEQFLEX. La preuve complète de la représentation qui vient d'être présentée est donnée dans le chapitre VI, comme exemple de la méthode générale de preuve.

L'idéal serait, bien sûr, de proposer une méthode permettant, à partir des axiomes de la Structure Source et de la Structure Objet et de la spécification des choix fondamentaux, de construire une spécification complète, correcte. Une première approche de ce problème de la construction systématique de représentations correctes pour les types abstraits algébriques a été étudiée dans [Gau 78, GT 78]. C'est un problème difficile et ouvert : quelques résultats ont été obtenus sur des exemples d'écoles. Dans l'état actuel des choses, la seule méthodologie utilisable pour obtenir une spécification de représentation correcte d'un type abstrait est d'écrire complètement la représentation et d'en faire une preuve a posteriori en utilisant les axiomes. C'est cette méthode qui est présentée dans cette thèse.

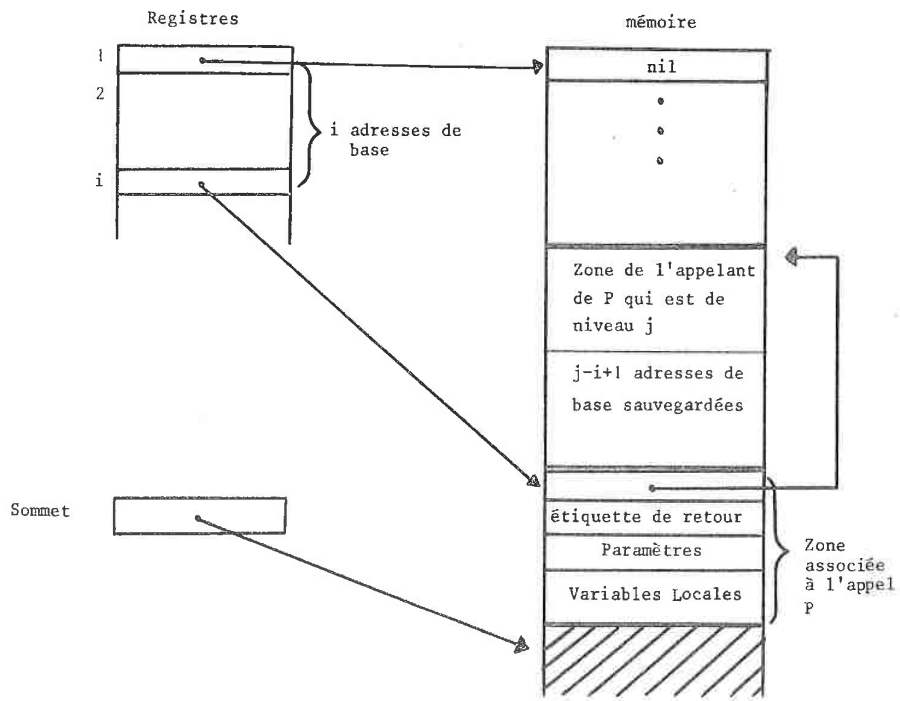
V.3 - Implantation de SIMPROC

Cet exemple de spécification reprend la Structure Source donnée en IV.1 pour le langage SIMPROC et la Structure Objet donnée au début de ce chapitre. L'intérêt de cet exemple est qu'il comporte la spécification de traitements non triviaux aussi bien à la compilation qu'à l'exécution.

Les choix d'implantation sont très classiques :

- on gère la mémoire, à l'exécution, comme une pile de zones, chacune correspondant à un appel de procédure en cours ;
- on considère qu'on a une infinité de registres. Pendant l'exécution d'un appel d'une procédure de niveau d'imbrication i, les i premiers registres contiennent les adresses de bases des i zones de mémoires associées à cet appel et aux appels en cours des procédures englobantes ;
- chaque zone de mémoire associée à un appel de procédure contient :
  . dans le premier mot l'adresse de la zone mémoire de l'appelant,
  . dans le deuxième mot l'étiquette de retour,
  . dans les deux mots suivants, les paramètres,
  . on a ensuite les variables et tableaux locaux à la procédure appelée ;
- La première adresse libre après la pile des zones de mémoire est contenue dans un registre spécial : 'sommet'.

Tout cela peut se schématiser par le dessin ci-après, qui décrit l'état de la mémoire et des registres pendant l'exécution d'une procédure P de niveau i (le programme est considéré comme une procédure de niveau 1) :



Selon ce schéma, un appel est représenté par une adresse, celle de la zone qui lui est associée. Un identificateur de variable ou de tableau est représenté par un déplacement dans cette zone, c'est-à-dire une valeur. La variable désignée par un identificateur, pour un certain appel est obtenue par indexation.

```
type Appel ;
   Adresse
fin Appel ;
type Var-id ;
   Valeur ;
   repr Var-id 'V' = Valeur 'ALLOC1(V)' ;
   (ALLOC1(V) recherche dans la table des allocations le déplacement associé
   à V, et si V n'y est pas lui associe un déplacement en fonction de sa
   portée).
fin Var-id ;
```

```
op(Var-id, Appel) → Var : dés ;
   repr dés(id, a) = index(cr(Registre 'NIV(PORTEE(TEXT(id)))', repr id)
   (TEXT est une meta procédure prédéfinie qui retourne le texte d'une
    constante. PORTEE recherche dans une table l'identificateur de la
    procédure où id a été déclaré. Cette table représente l'opération portée ;
    on a :
          repr portée (id) = repr Proc-id 'PORTEE(TEXT(id))' ;
    enfin la spécification de NIV est :
    NIV(X) = SI X = PROG  ALORS  1 SINON NIV(PORTEE(X)) + 1 ;)
fin dés ;
```

Dans la suite on notera

BASE(id) = NIV(PORTEE(TEXT(id)))

le numéro du registre de base correspondant à la portée de id.

On voit apparaître ici la classique différence entre statique et dynamique : l'opération 'portée' de la structure source qui est évaluée à la compilation, a son résultat représenté par une constante. Cette constante est obtenue à partir d'une table. Cette table pourrait être construite par la deuxième phase du traducteur puisque les déclarations apparaissent dans le Terme Source. Pour des raisons d'efficacité (voir IV.1.3), dans la version actuelle du système elle est construite par PHASE1 et transmise à PHASE2, en même temps que d'autres tables donnant, par exemple, les bornes des tableaux et tous les renseignements statiques sur les identificateurs du programme.

On va donner maintenant la représentation des procédures, c'est-à-dire la représentation du type Proc-id et des modifications decl-proc et appel.

```
type Proc-id ;
   Eti ;
   repr Proc-id 'P' = Eti 'P' ;
fin Proc-id ;

op(Proc-id, Var-id, Ref-id, Proc-id, Modif) → Modif : declproc ;
   repr decl-proc(p, id1, id2, q, m) = (# SAUT := GENETI
        (le corps de la procédure n'est pas exécuté, on se donne donc une
         étiquette pour se brancher à la fin) #
```



```
        compseq⁴(brcht(Eti 'SAUT'),
                 étiqueter(Eti 'TEXT(p)', repr m),
                 brcht(ca(index(cr(Registre 'en cours'), Valeur '1'))),
                 étiqueter(Eti 'SAUT', suite)) ;
   (Le corps de la procédure est étiqueté du nom de la procédure et complété
    à la fin par un branchement à l'étiquette de retour - qui sera contenue
    dans le deuxième mot de la zone mémoire -)
fin decl-proc ;
```

Pour définir l'appel de procédure plus facilement, on se donne deux macro-opérations de sauvegarde multiple et de chargement multiple : en effet, à l'appel d'une procédure de niveau i depuis une procédure de niveau j on a à sauvegarder, puis restaurer j-i+1 registres de base. Par exemple on utilise :

```
macro-opérations
   (Registre, Adresse, Valeur) → Modif : sauvmult ;
   (Adresse, Registre, Valeur) → Modif : loadmult ;
définitions
   sauvmult(Registre 'I', a, Valeur 'K') =
      SI K = 1 ALORS  store(cr(Registre 'I'), a)
      SINON  compseq(store(cr(Registre 'I'), a), sauvmult(Registre 'I +1',
                     index(a, Valeur '1'), Valeur 'K-1'))
      FSI ;
   loadmult(a, Registre 'I', Valeur 'K') =
      SI K = 1 ALORS  load(ca(a), Registre 'I')
      SINON  compseq(load(ca(a), Registre 'I'), loadmult(index(a, Valeur '1'),
                     Registre 'I +1', Valeur 'K-1'))
      FSI ;
fin
```

D'où la définition suivante de la représentation de l'appel de procédure :

```
op(Proc-id, Ent, Var, Proc-id) → Modif : appel ;
   repr appel(p, i, v, q) = (# P := TEXT(p) ; Q := TEXT(q) ; I := NIV(P) ;
      J := NIV(Q), K := J-I+1 ; RET := GENETI #
      compseq⁹(SI  K = 0  ALORS  suite  SINON
               compseq(sauvmult(Registre 'I', cr(Registre 'sommet'),Valeur 'K'),
                     load(plus(cr(Registre 'sommet'), Valeur 'K'),
                          Registre 'sommet'))
```

```
        On a sauvegardé les registres de base.
        FSI,
        On sauvegarde l'adresse de l'appelant dans le premier mot de
        la future zone.
        store(cr(Registre 'J'), cr(Registre 'sommet')),
        On passe les paramètres : les registres de base sont inchangés,
        ils sont donc évalués dans l'environnement précédent.
        store(repr i, index(cr(Registre 'sommet'), Valeur '2')),
        store(repr v, index(cr(Registre 'sommet'), Valeur '3')),
        On change d'environnement.
        load(cr(Registre 'sommet'), Registre 'I'),
        load(plus(cr(Registre 'sommet'), Valeur '4'), Registre 'sommet'),
        On sauvegarde l'étiquette de retour et on se branche à la
        procédure.
        store(Eti 'RET', index(cr(Registre 'I'), Valeur '1')),
        brcht(Eti 'P'),
        étiqueter(Eti 'RET',
        compseq(load(moins(cr(Registre 'I'), Valeur 'K'), Registre
                    'sommet'),
              SI  K = 0  ALORS  suite  SINON
              loadmult(cr(Registre 'sommet'), Registre 'I', Valeur 'K')
              FSI))
        On restaure les registres de base et le sommet après le retour ;
fin appel ;
```

Le reste de la spécification ne présente pas de difficultés. Pour les tableaux, on a :

```
type Tab-id ;
   Valeur ;
   repr Tab-id 'T' = Valeur 'ALLOC2(T)' ;
   (ALLOC2(T) recherche dans la table des allocations le déplacement associé
   à T. Si T n'est pas dans cette table, un déplacement lui est associé en
   fonction de sa portée. Le résultat de ALLOC2(T) est ce déplacement diminué
   de 1 car la borne inférieure des tableaux est égale à 1. Cela permet une
   représentation plus efficace de 'elt').
fin Tab-id ;
```



```
op(Tab-id, Appel, Ent) → Var : elt ;
   repr elt(t,a,i) = (# I := GENREG #
        efb(Vérification des cas d'échec.
            compseq5(load(repr i, Registre 'I'),
                        compare(cr(Registre 'I'), Valeur '1'),
                        brcht-cond(Eti 'erreur 1', lt),
                        compare(cr(Registre 'I'), Valeur 'BSUP(TEXT(t))'),
                        brcht-cond(Eti 'erreur', gt)),
            Adresse résultat.
            index(cr(Registre'BASE(t)', plus(repr t, cr(Registre 'I')))) ;
fin elt ;
```

Pour les paramètres passés par référence, on a une représentation presque identique à celle des identificateurs de variable, sauf que le déplacement est toujours le même :

```
type Ref-id ;
   Valeur
   repr Ref-id 'R' = Valeur '3' ;
fin Ref-id ;

op(Ref-id, Appel) → Ref-var : ref-dés ;
   repr ref-dés(id, a) = index(cr(Registre 'BASE(id)'), repr id) ;
fin ref-dés ;
```

Il reste à représenter les variables et les entiers : la spécification n'est pas donnée ici 'in extenso'. Les types Var et Ref-Var sont représentés par des Adresses, le type Ent par des Valeurs ; les opérations 'réf' et 'val' sont représentées par l'opération 'ca'.

Enfin, la représentation des modifications autres que l'appel et la déclaration de procédure est donnée ici pour mémoire : elle est très semblable à celle des modifications de SEQFLEX.

```
op(Var, Ent) → Modif : affecter ;
   repr affecter(v,i) = store(repr i, repr v) ;
fin affecter ;
```

```
op(Modif, Modif) → Modif : concat ;
   repr concat(m, m') = compseq(repr m, repr m') ;
fin concat ;

op(Ent, Ent, Modif, Modif) → Modif : cond ;
   repr cond(i,j,m,m') = (# ETI1 := GENETI ; ETI2 := GENETI #
            compseq^6(  compare(repr i, repr j),
                        brcht-neg(Eti 'ETI1', eq),
                        repr m,
                        brcht(Eti 'ETI2'),
                        étiqueter(Eti 'ETI1', repr m'),
                        etiqueter(Eti 'ETI2', suite)) ;
fin cond ;
```

Pour les déclarations d'entiers et de tableaux, il existe plusieurs variantes. La plus 'naïve' et la moins efficace quant au code généré est la suivante :

```
op(Var-id, Proc-id) → Modif : decl-int ;
   repr decl-int(id,p) = (# A := ALLOC1(TEXT(id)) #
        load(plus(cr(Registre 'sommet'), Valeur '1'), Registre 'sommet')) ;
  (à la compilation, on effectue l'allocation si elle n'était pas déjà faite
   car il peut y avoir des références en avant. A l'exécution on augmente de un
   le pointeur 'sommet')
fin decl-int ;

op(Tab-id, Ent, Proc-id) → Modif : decl-tab ;
   repr decl-tab(id, i,p) = (# A := ALLOC2(TEXT(t)) #
        load(plus(cr(Registre 'sommet'), repr i), Registre 'sommet') ;
fin decl-tab ;
```

Il est plus efficace de n'augmenter qu'une fois le pointeur 'sommet' au moment d'un appel de procédure. Cela signifie que dans l'appel de procédure, au moment du changement d'environnement, au lieu d'augmenter ce pointeur de quatre (pointeur vers l'appelant, étiquette de retour, paramètres) on l'augmente d'une certaine constante TAILLE(P) qui est calculable à partir de la table des portées, dès l'instant où l'on sait sur combien de mots sont représentés les entiers et comment sont représentés les tableaux. Dans ce cas, les déclarations sont représentées par une modification vide.

CHAPITRE VI

PREUVES D'IMPLANTATION

On a vu dans le chapitre II comment prouver correcte la représentation d'un type abstrait algébrique par un autre. Nous allons évidemment utiliser ces résultats pour prouver que les axiomes généraux du langage source sont conservés. Cependant, l'introduction de modifications complique un peu le problème des preuves. Dans ce chapitre, nous donnons tout d'abord une méthode de preuve de représentation des modifications. Puis, à titre d'exemple, nous donnons la preuve complète de l'implantation du langage SEQFLEX donnée en V.2.

VI.1 - Preuve de Représentation des Structures d'Information.

VI.1.1 - Théorème fondamental

Etant donné un type abstrait T et un type abstrait O, rappelons qu'une représentation de T par O est une fonction ρ qui, à tout symbole fonctionnel f du type T associe une composition $\rho(f)$ de symboles fonctionnels du type O, autrement dit, une opération dérivée du type O.

Par définition, ρ s'étend aux termes du type T de la manière suivante :

$$\rho(f(t_1,\ldots, t_n)) = = \rho(f)\ (\rho(t_1),\ldots, \rho(t_n))$$

où = = indique l'identité entre termes.

ρ est correcte si les axiomes du type T sont conservés. Soit un axiome de la forme :

$$f(x_1,\ldots x_n) = g(x_1,\ldots x_n)$$

où $x_1,\ldots, x_n$ peuvent comporter des variables libres, il faut que :

$$\rho(f)\ (\rho(x_1,\ldots, \rho(x_n)) = \rho(g)\ (\rho(x_1,\ldots \rho(x_n))$$

soit un théorème dans le sous-ensemble du type O défini par les invariants de représentation.

Les invariants de représentations sont les fonctions caractéristiques du domaine d'arrivée de ρ, $\simeq$ est l'interprétation de l'égalité, et si x est une variable libre de type t, $\rho(x)$ est par définition une variable libre du type qui représente t.

Toutes les notions ont été présentées très précisément en II.4.

Avec l'introduction de la notion d'état, et des modifications, on est amené à se donner un certain nombre de définitions sur la représentation d'un état et sur les critères de correction de la représentation des modifications.

Définition 1 : Représentant d'un état

Soit ρ une représentation correcte d'un type T. Soit $S_{A \cup B}$ un état d'une structure d'information obtenue en ajoutant à T des modifications (A est l'ensemble des axiomes généraux, ceux de T ; B est l'ensemble des axiomes spécifiques à S). On appelle représentant de $S_{A \cup B}$ l'état $S_{\rho(A) \cup \rho(B)}$ où $\rho(A)$ désigne les représentations des axiomes généraux A, $\rho(B)$ celles des axiomes spécifiques. On le note $\rho(S_{A \cup B})$.

Théorème 1

Si $f \in S_{A \cup B}$, alors $\rho(f) \in \rho(S_{A \cup B})$.

La démonstration est immédiate : si $f \in S_{A \cup B}$, il existe une démonstration de la forme

$$a_1 \wedge a_2 \wedge \ldots \wedge a_n \Rightarrow f_1 \Rightarrow \ldots \Rightarrow f$$

où $a_1, \ldots a_n$ sont des axiomes de $A \cup B$. Comme les axiomes sont conservés par ρ et que les règles d'inférence sont les mêmes, on a une démonstration de $\rho(f)$ à partir de $\rho(A) \cup \rho(B)$ par simple réécriture

$$\rho(a_1) \wedge \rho(a_2) \wedge \ldots \wedge \rho(a_n) \Rightarrow \rho(f_1) \Rightarrow \ldots \Rightarrow \rho(f)$$

d'où

$$\rho(f) \in \rho(S_{A \cup B}) \quad \square$$

Définition 2 :

Soit un état S et une fonction de représentation . On dit qu'un état S_1 est une représentation de S si :

a) S_1 est consistant

b) $f \in \rho(S) \Rightarrow f \in S_1$

Cette définition exprime l'idée intuitive suivante : un état S peut avoir une infinité de représentations. En effet des formules "accessoires", propres à la Structure Objet, peuvent y être contenues. Mais il est indispensable que la représentation de toute formule de S appartienne à chacune des représentations de S. Autrement dit :

$$\rho(S) \subseteq S_1$$

Définition 3 :

Soit ρ une représentation d'une structure d'information T par une structure d'information O. Une modification m de O est dite sans effet par rapport à ρ si, pour tout état S de T, si S_1 est une représentation de S :

appl(m, S_1) est une représentation de S.

Définition 4 :

Soit T et O deux structures d'information et ρ une représentation de T par O.

a) si f est une opération modifiable de T, si $\rho(f)$==g, g doit être une opération modifiable de O.

b) Soit D_f le domaine de f, c'est-à-dire le type des opérandes de f. L'opération 'eg' de D_f doit être représentée par l'opération 'eg' de $\rho(D_f)$. (Dans le cas où f a plusieurs opérandes, si D_{fi} est le type du ième opérande, on doit avoir, pour tout i, l'opération 'eg' de D_{fi} représentée par l'opération 'eg' de $\rho(D_{fi})$.).

Cette condition est nécessaire pour que les modifications de f soient représentées correctement par des modifications de g : si on modifie g en $\rho(\lambda)$, on modifie g pour tout les $\rho(x)$ tels que $eg(\rho(\lambda), \rho(x))$ = vrai, et eux seuls.

c) Soit S un état de T. Un état S_1 de O est une représentation acceptable de S par ρ si :

- S_1 est une représentation de S.
- pour tout couple d'opérations f, h où f est modifiable et où f et h sont représentées par la même opération ($\rho(f) == \rho(h)$) on a :

$t1 \in D_f$ et pré$(f(t_1)) \in S$, $t2 \in D_h$ et pré$(h(t2)) \in S$ [1] $\Longrightarrow$

$$eg(\rho(t1), \rho(t2)) = \text{faux} \in S_1.$$

Cette dernière propriété est nécessaire pour éviter qu'une modification de la représentation de f n'ait pour résultat un état où la représentation de h a été modifiée : par exemple, si on a plusieurs opérations représentées par l'opération 'ca' du chapitre V, les ensembles d'adresses représentant les domaines de ces opérations doivent être disjoints (et le rester) dans tout état représentant.

On a maintenant les éléments pour énoncer le théorème sur lequel est basée la méthode de preuve de représentation des modifications : il traite en effet de la représentation de la modification primitive <u>subst</u>$(f(\lambda),\mu)$. On s'est limité au cas où f est représentée par une opération de la Structure Objet, g.

<u>Théorème 2</u> :

Soit T et O deux structures d'information et ρ une représentation correcte [2] de T par O.

Soit f une opération modifiable de T. Si

a) S' = <u>appl</u>(<u>subst</u>$(f(\lambda), \mu)$, S),

b) S_1 est une représentation acceptable de S,

Alors S'_1 = <u>appl</u>(subst$(\rho(f(\lambda)),\rho(\mu)),S_1)$ est une représentation acceptable de S'.

---

(1) pré(t)$\in$S indique que les préconditions sur t sont vérifiées dans S. Par exemple si t==g(u), si on a une restriction <u>Pré</u>(g,x), pré(t) = <u>Pré</u>(g,u) $\wedge$ pré(u)$\in$S. S'il n'y a pas de préconditions sur g on pose, par définition, pré(t)=pré(u) dans tout état S.

(2) C'est-à-dire que pour les types abstraits correspondant à T et O, ρ est une représentation correcte au sens du chapitre II. De plus, ρ vérifie les propriétés a) et b) de la définition 4.

Tout au long de la démonstration, on utilisera les notations suivantes :

- g est l'opération de O qui représente f : $\rho(f)$==g ;
- on se donne deux opérations f' et g', de même profil respectivement que f et g, soumises aux même préconditions que f et g (s'il y en a) et vérifiant :

  $\rho(f')$ == g'

  $(a_{f'})$ f'(x) = <u>si</u> eg(x,λ) <u>alors</u> μ <u>sinon</u> f(x)

  $(a_{g'})$ g'(x) = <u>si</u> eg(y,$\rho(\lambda)$) <u>alors</u> $\rho(\mu)$ <u>sinon</u> g(y)
- on appelle T' (respectivement O') la structure d'information obtenue en enrichissant T (respectivement O) de l'opération f' et de l'axiome $a_{f'}$ (respectivement g' et $a_{g'}$).

La démonstration utilise les deux lemmes suivants :

<u>Lemme 1</u> : Soit S un état de T et S_1 une représentation acceptable de S par ρ. Alors $S_1 + a_{g'}$ est une représentation acceptable de $S + a_{f'}$ par ρ.

<u>Démonstration du lemme 1</u> :

Elle se fait en deux temps : on montre d'abord que $S_1 + a_{g'}$ est une représentation de $S + a_{f'}$. On montre ensuite que c'est une représentation acceptable.

a) Considérons la représentation de l'axiome $a_{f'}$, $\rho(a_{f'})$. Elle s'écrit :

$\rho(f')(\rho(x)) \simeq$ <u>si</u> $\rho(eg)(\rho(x), \rho(\lambda))$ <u>alors</u> $\rho(\mu)$ <u>sinon</u> $\rho(f)(\rho(x))$

soit, par définition de ρ:

g'(y) $\simeq$ <u>si</u> eg(y,$\rho(\lambda)$) <u>alors</u> $\rho(\mu)$ <u>sinon</u> g(y)

où '$\simeq$' est l'interprétation de l'égalité sur le co-domaine C_f de f. Par définition, si ρ est correcte, c'est une relation de congruence sur le co-domaine C_g de g, compatible avec les axiomes sur C_g. Par conséquent, l'égalité induite par les axiomes sur C_g est contenue dans cette relation et on a $t = t' \Longrightarrow t \simeq t'$ dans C_g

donc

$$a_{g'} \Longrightarrow \rho(a_{f'})$$

Comme S_1 est une représentation de S

$$\rho(S + a_{f'}) \subseteq S_1 + \rho(a_{f'}) \subseteq S_1 + a_{g'}$$

$S_1 + a_{g'}$ est donc une représentation de $S + a_{f'}$.

b) On va montrer maintenant que $S_1 + a_{g'}$ est une représentation acceptable par ρ.

Soit t un terme de T' tel que $\text{pré}(t) \in S+a_{f'}$. En utilisant l'axiome $a_{f'}$ un nombre fini de fois, on peut trouver un terme R(t) de T (c'est à dire sans f') tel que

$$t = R(t) \in S+a_{f'}$$

et

$$\text{pré}(R(t)) \in S$$

R est défini par :

$R(\wedge) = \wedge$ où $\wedge$ est le mot vide

$R(h(u)) == h(R(u))$ si h n'est pas f'

$R(f'(u)) == \mu$ si $eg(u, \lambda)$=vrai $\in S+a_{f'}$

$R(f'(u)) == f(R(u))$ si $eg(u, \lambda)$=faux $\in S+a_{f'}$

de même, dans O', on se donne pour tout terme t un terme $R_1(t)$, sans occurrence de g', tel que

$$t=R_1(t) \in S_1+a_{g'}$$

D'après les définitions de ρ, R et R_1, on a

$$\rho(R(t)) = R_1(\rho(t)) \in S_1+a_{g'}$$

(La démonstration de cette propriété est donnée dans le lemme 1bis).

Soit maintenant h1 et h2 deux opérations de T' telles que $\rho(h1)==\rho(h2)$. f' ne peut intervenir dans un tel couple car f' est représentée par g' qui n'est utilisée que pour cette représentation.

Soit $t1 \in D'_{h1}$ et $t2 \in D'_{h2}$

où D'_{h1} et D'_{h2} sont les termes de T' qui peuvent apparaître comme opérandes de h1 et h2. De plus on considère que :

$$\text{pré}(h1(t1)) \in S+a_{f'}$$

$$\text{pré}(h2(t2)) \in S+a_{f'}$$

Du fait que S_1 est une représentation acceptable de S par ρ, et par définition de R on a :

$$R(t1) \in D_{h1} \text{ et } \text{pré}(h1(R(t1))) \in S$$

$$R(t2) \in D_{h2} \text{ et } \text{pré}(h2(R(t2))) \in S$$

$$\Longrightarrow eg(\rho(R(t1)), \rho(R(t2))) = \text{faux} \in S_1$$

donc

$$eg(R_1(\rho(t1)), R_1(\rho(t2)) = \text{faux} \in S_1+a_{g'}$$

d'où

$$eg(\rho(t1, \rho(t2)) = \text{faux} \in S_1+a_{g'}.$$

$S_1+a_{g'}$ est donc une représentation acceptable par ρ de $S+a_{f'}$ □

<u>Lemme 1bis</u>

$$\rho(R(t)) = R_1(\rho(t)) \in S_1+a_{g'}$$

<u>Démonstration</u> : Elle se fait par récurrence sur la longueur de t.

Posons $t == h(u)$ avec

$$\rho(R(u)) = R_1(\rho(u)) \in S_1+a_{g'}$$

<u>1er cas</u> h n'est pas f'. Dans ce cas on a

$$R(h(u)) == h(R(u))$$

$$\rho(R(h(u)) == \rho(h) \ \rho(R(u))$$

d'autre part

$$\rho(h(u)) == \rho(h) \ \rho(u)$$

$$R_1(\rho(h(u)) == \rho(h) \ R_1(\rho(u))$$

en effet $\rho(h)$ n'est pas g' puisque g' représente uniquement f'.

comme $\rho(R(u)) = R_1(\rho(u)) \in S_1+a_{g'}$

on a (par définition d'un état)

$$\rho(h)(\rho(R(u))) = \rho(h)(R_1(\rho(u))) \in S_1+a_{g'}$$

<u>2ème cas</u> h est f' : $t == f'(u)$.

On a deux cas à distinguer :

. $eg(u, \lambda)$ = vrai $\in S+a_{f'}$

Dans ce cas $eg(\rho(u), \rho(\lambda))$ = vrai $\in S_1+a_{g'}$

$$R(f'(u)) == \mu$$

$$\rho(R(f'(u))) == \rho(\mu)$$

et

$\rho(f'(u)) == g'(\rho(u))$

d'où

$R_1(\rho(f'(u)) == \rho(\mu)$

par conséquent la propriété est vérifiée.

. $eg(u, \lambda)$ = faux $\in S_1+a_{g'}$

on a alors $eg(\rho(u), \rho(\lambda))$ = faux $\in S_1+a_{g'}$

$R(f'(u)) == f(R(u))$

$\rho(R(f'(u)) == g(R(u))$

d'autre part

$\rho(f'(u)) == g'(\rho(u))$

d'où

$R_1(\rho(f'(u)) == g(R_1(\rho(u)))$

par hypothèse de récurrence

$\rho(R(u)) = R_1(\rho(u)) \in S_1+a_{g'}$

et par définition d'un état

$g(\rho(R(u)) = g(R_1(\rho(u)) \in S_1+a_{g'}$ □

On donne maintenant le deuxième lemme sur lequel est basée la démonstration du théorème.

<u>Lemme 2</u> : Soit t un terme de T. On a (si pré(t) $\in$ S) :

$\rho(t[f'/f]) = \rho(t)[g'/g] \in S_1+a_{g'}$

<u>Démonstration</u> : Soit t == h(u), où u est un terme (ou un n-uplet de termes, ou le mot vide) qui vérifie par hypothèse de récurrence :

$\rho(u[f'/f]) = \rho(u)[g'/g] \in S_1+a_{g'}$

a) On va distinguer le cas où h est f, c'est-à-dire :

$t == f(u)$

On a alors :

$\rho(t[f'/f]) == \rho(f(u)[f'/f])$

$== \rho(f'(u[f'/f]))$

$== g'(\rho(u[f'/f]))$

et par hypothèse de récurrence sur u :

$\rho(t[f'/f]) = g'(\rho(u)[g'/g]) \in S_1+a_{g'}$.

D'autre part

$\rho(t)[g'/g] == \rho(f(u))[g'/g]$

$== g'(\rho(u)[g'/g])$

La propriété est donc vérifiée pour les termes de la forme f(u).

b) Soit t == h(u) où h n'est pas f. On a :

$\rho(t[f'/f]) == \rho(h(u)[f'/f])$

$== \rho(h)(\rho(u[f'/f]))$

et par récurrence :

$\rho(t[f'/f]) = \rho(h)(\rho(u)[g'/g]) \in S_1+a_{g'}$

Si h n'est pas représentée par g, on a

$\rho(t)[g/g'] == \rho(h(u))\ [g/g']$

$== \rho(h)(\rho(u)[g'/g])$

et la propriété est vérifiée.

Si h est représentée par g, la démonstration est moins immédiate. En effet, on a vu que :

$\rho(t[f'/f]) = g(\rho(u)[g'/g]) \in S_1+a_{f'}$

et par définition de ρ :

$\rho(t)[g'/g] == g'(\rho(u)[g'/g])$

Il faut donc montrer que

$g(\rho(u)[g'/g]) = g'(\rho(u)[g'/g]) \in S_1+a_{g'}$

où encore

(1) $g(\rho(u[f'/f])) = g'(\rho(u[f'/f])) \in S_1+a_{g'}$

Pour cela, on va utiliser l'axiome $a_{g'}$ et le fait que $S_1+a_{g'}$ est une représentation acceptable de $S+a_{g'}$ (ce qui a été montré dans le lemme 1).

Rappelons l'axiome $a_{g'}$:

$g'(y)$ = <u>si</u> $eg(y,\rho(\lambda))$ <u>alors</u> $\rho(\mu)$ <u>sinon</u> $g(y)$

La propriété (1) est vraie si :

(2) $eg(\rho(u[f'/f]), \rho(\lambda))$ = faux $\in S_1+a_{g'}$

Or h et f sont deux opérations de T représentées par la même opération de O.

On a $t == h(u)$ et $\text{pré}(t) \in S$

Par définition de f' (même profil et même préconditions que f) on a

$$u \in D_h \Longrightarrow u[f'/f] \in D'_h$$

où D'_h est le domaine de définition de h dans T'

et $\text{pré}(t) \in S \Longrightarrow \text{pré}(t[f'/f]) \in S+a_{f'}$

d'autre part :

$$\lambda \in D_f \Longrightarrow \lambda \in D'_f$$

Les représentations de u [f'/f] et de λ vérifient la propriété (2) puisque $S_1+a_{g'}$ est une représentation acceptable par ρ de $S+a_{f'}$.
Ceci termine la démonstration du lemme 2 □

<u>Démonstration du théorème</u> :

Par définition de <u>subst</u> on a

$$S' = \{P \mid P[f'/f] \in S+a_{f'}\}$$

où P est de la forme t = t'.

De même S'_1 est défini par

$$S'_1 = \{Q \mid Q[g'/g] \in S_1+a_{g'}\}$$

S'_1 est une représentation de S' si pour tout P de S', ρ(P) appartient à S'_1. Si P appartient à S', alors P[f'/f] appartient à $S+a_{g'}$.

D'après le lemme 1, $S_1+a_{g'}$ est une représentation de $S+a_{f'}$. Donc :

$$\rho(P[f'/f]) \in S_1+a_{g'}$$

P est de la forme t = t'. P[f'/f] est de la forme

$$t[f'/f] = t'[f'/f]$$

et ρ(P[f'/f]) est de la forme $\rho(t[f'/f]) \simeq \rho(t'[f'/f])$

Or d'après le lemme 2 :

$$\rho(t[f'/f]) = \rho(t)[g'/g] \in S_1+a_{g'}$$

$$\rho(t'[f'/f]) = \rho(t')\,[g'/g] \in S_1+a_{g'}$$

et on a vu plus haut que

$$\rho(t[f'/f]) \simeq \rho(t'[f'/f]) \in S_1+a_{g'}$$

d'où

$$\rho(t)[g'/g] \simeq \rho(t')[g'/g]) \in S_1+a_{g'}$$

c'est-à-dire

$$\rho(P)[g'/g] \in S_1+a_{g'}$$

et par définition de S'_1

$$\rho(P) \in S'_1$$

S'_1 est donc une représentation de S'.

C'est une représentation acceptable de S' par ρ si pour tout couple d'opérations h1, h2 représentées par la même opération on a

$$t1 \in D_{h1} \quad \text{et} \quad \text{pré}(h1(t1)) \in S'$$

$$t2 \in D_{h2} \quad \text{et} \quad \text{pré}(h2(t2)) \in S'$$

$$\Longrightarrow eg(\rho(t1),\ \rho(t2)) = \text{faux} \in S'_1$$

autrement dit

$$eg(\rho(t1),\ \rho(t2))[g'/g] = \text{faux} \in S_1+a_{g'}$$

soit, d'après le lemme 2

(3) $$eg(\rho(t1[f'/f]),\ \rho(t2[f'/f])) = \text{faux} \in S_1+a_{g'}$$

Or, par définition de f' (même profil et mêmes préconditions que f) on a

$$t1[f'/f] \in D'_{h1} \qquad t2[f'/f] \in D'_{h2}$$

et par définition de S'

$$\text{pré}(h1(t1))[f'/f] \in S+a_{f'}$$

$$\text{pré}(h2(t2))[f'/f] \in S+a_{f'}$$

Si h1 et h2 ne sont pas f, on a

$$\text{pré}(h1(t1[f'/f])) \in S+a_{f'}$$

$$\text{pré}(h2(t2[f'/f])) \in S+a_{f'}$$

et du fait que $S_1+a_{g'}$ est une représentation acceptable de $S+a_{f'}$ par ρ, la propriété (3) est vérifiée.

Si h1 est identique à f on a

$$\text{pré}(f'(t1[f'/f])) \in S+a_{f'}$$

et comme les préconditions sur f et f' sont identiques

$$\text{pré}(f(t1[f'/f])) \in S+a_{f'}$$

La propriété (3) est donc vérifiée dans tous les cas et S'_1 est une représentation acceptable de S' par ρ. □

Le théorème qu'on vient de démontrer correspond au cas très particulier où une opération modifiable de la Structure Source est représentée par une opération modifiable de la Structure Objet. (dans l'implantation de SEQFLEX donnée en V.2, c'est le cas des opérations 'valeur' et 'bornesup')

On peut étendre ce théorème au cas où $\rho(f)$ est de la forme g o h où g est modifiable et où h est une composition d'opérations <u>non modifiables</u>. Dans ce cas, l'opération auxiliaire g' sur laquelle on travaille pour démontrer le théorème est définie par :

$$g'(h(y)) = \underline{\text{si}}\ eg(h(y),h(\rho(\lambda)))\ \underline{\text{alors}}\ \rho(\mu)\ \underline{\text{sinon}}\ g(h(y))$$

L'hypothèse sur la représentation de l'égalité dans D_f est que cette représentation vérifie :

$$\rho(eg(x,\lambda)) == eg(h(\rho(x)),\ h(\rho(\lambda)))$$

De plus, la notion de représentation acceptable S_1, de S par ρ, devient : si f1 est une autre opération de la Structure Source, représentée par g o h1, si t est un opérande correct de f dans S, si t1 est un opérande correct de f1 dans S on a :

$$eg(h(\rho(t)),\ h1(\rho(t1)) = \text{faux} \in S_1.$$

Enfin, on a une autre extension du théorème au cas où la représentation de f est de la forme :

$$efb(m,\ g(x))$$

où m est sans effet par rapport à ρ (voir la Définition 3) et où g est une opération modifiable. (C'est le cas de la représentation de l'opération 'ième' dans SEQFLEX).

Soit

$$S' = \underline{appl}(\underline{subst}(f(\lambda),\mu),\ S)$$

et

$$S'_1 = \underline{appl}(\underline{subst}(\rho(f(\lambda)),\ \rho(\mu)),\ S_1)$$

où S_1 est une représentation acceptable de S. Par définition de 'efb' (voir V.1) on a

$$S'_1 = \underline{appl}(\underline{subst}(efb(m,\ g(x)),\ \rho(\mu)),\ S_1)$$
$$= \underline{appl}(\underline{subst}(g(x),\ \rho(\mu)),\ \underline{appl}(m,\ S_1))$$

m étant sans effet sur ρ, $\underline{appl}(m,\ S_1)$ est une représentation acceptable de S. Dans la mesure où les opérandes x de g vérifient les hypothèses du théorème (représentation de l'égalité sur D_f par l'égalité sur D_g et non recouvrement avec les représentations des domaines des autres opérations représentées par g), on a un état résultant S'_1 qui est une représentation acceptable de S' par ρ.

VI.1.2 - <u>La méthode de preuve</u>

En utilisant les méthodes données en II.4, plus particulièrement en II.4.2 puisqu'on utilise une fonction de représentation, et les résultats qui ont été donnés au début de ce chapitre, on peut prouver correcte la représentation d'une Structure d'Information par une autre.

Nous suggérons la méthode suivante :

a) dans un premier temps on considère le type abstrait obtenu en supprimant les modifications et les opérations modifiables. On démontre de manière classique (II.4.2) que la représentation de ce type abstrait est correcte, c'est-à-dire que les axiomes généraux sont conservés. Cette démonstration amène à :

- donner la représentation d'opérations auxiliaires qui n'apparaissent pas dans les équations sémantiques mais sont utilisées dans les axiomes ;
- établir une interprétation de l'égalité pour chaque type à représenter ;
- dégager des invariants de représentation.

b) on étudie alors la représentation des opérations modifiables. On vérifie d'abord que l'égalité sur leur domaine de définition est représentée par l'égalité. Il faut montrer ensuite qu'on a une représentation acceptable par ρ de l'état initial. Cet état initial S_o, est l'état engendré par les axiomes généraux (Définition 5 de III.1) et il faut prouver que ρ est telle que si

$$\rho(f) == \rho(h)$$

pour tout opérande t de f tel que pré(t) soit vraie dans S_o, pour tout opérande t' de h tel que pré(t') soit vraie dans S_o on a

$$eg(\rho(t),\rho(t')) = \text{faux} \in S_o.$$

De ce fait, on peut être amener à introduire de nouveaux invariants de représentation.

c) On prouve ensuite, pour chaque modification que sa représentation est correcte. Pour cela, on a trois propriétés à vérifier pour une modification m :

- la représentation de m conserve les invariants de représentation qui ont été établis en a) et b). C'est en effet indispensable puisque ces invariants font partie des propriétés qui assurent la conservation des axiomes généraux et le fait que la représentation des états par ρ est acceptable.
- si m comporte dans sa définition un subst($f(\lambda)$,μ) alors la représentation de m comporte une (ou des) modification(s) qui peut être montrée équivalente à subst($\rho(f(\lambda))$,$\rho(\mu)$)
- les autres modifications intervenant dans la représentation de m sont sans effet sur ρ.

On va voir sur un exemple que si la preuve d'une implantation est longue, elle n'est pas très difficile et peut être développée méthodiquement.

VI.2 - Preuve de l'implantation de SEQFLEX

VI.2.1 - Types et opérations ne faisant pas intervenir de modifications

Les types Ent, muni des opérations add, soust, mult, eg, et Valeur, muni des opérations plus, moins, mult, eg (eg(v,v') est noté v = v') sont identiques, aux noms des opérations près. Donc le deuxième représente correctement le premier et l'interprétation de l'égalité sur les entiers est simplement l'égalité sur les valeurs. Ceci à condition que la procédure de conversion des constantes : CONVERTIR soit correcte, ce qu'on va supposer être démontré par ailleurs.

Les deux divisions sont définies par des axiomes différents, le premier fait intervenir les booléens et le second les codes de condition. On va donc d'abord prouver correcte la représentation du type Bool par le type Codecond. On reviendra ensuite à la preuve de la représentation de la division.

Nous rappelons ici la représentation du type Bool :

```
type Bool ;
   Codecond ;
fin Bool ;
op (Ent,Ent) → Bool : eg ;
   repr eg(i,j) = efbc(compare(repr i, repr j),eq) ;
fin eg ;
op (Ent,Ent) → Bool : dif ;
   repr dif(i,j) = efbnc(compare(repr i,repr j),eq) ;
fin dif ;
```

Les axiomes qui définissent eg sont les suivants

```
eg(Ent'C1',Ent'C2') = eg(C1,C2) ;
eg(i,add(i,Ent'1')) = faux ;
eg(i,soust(i,Ent'1')) = faux ;
eg(i,i') = eg(i',i) ;
eg(add(i,Ent'1'),add(i',Ent'1') = eg(i,i') ;
eg(soust(i,Ent'1'),soust(i',Ent'1')) = eg(i,i') ;
eg(soust(i,Ent'1'),add(i',Ent'1')) = eg(i,add(add(i',Ent'1'),Ent'1'))) ;
```

Pour prouver que ces axiomes sont conservés, on a besoin de repr vrai, de repr faux et de l'interprétation de l'égalité sur les représentations de Booléens . Posons :

repr vrai = efbc(compare(v,v'),test(v,v')) ;
repr faux = efbnc(compare(v,v'),test(v,v')) ;

Ces deux définitions ne sont pas arbitraires :

en opérande d'un branchement conditionnel, la représentation de vrai provoque un branchement et la représentation de faux ne le provoque pas. Pour l'interprétation de l'égalité, selon le même critère, on va avoir

efbc(compare(v,v'),ccond) ≃ si test(v,v') =
ccond alors repr vrai sinon repr faux ;
efbnc(compare(v,v'),ccond) ≃ si test(v,v') = ccond
alors repr faux sinon repr vrai ;

Intuitivement, tout terme commençant par efbc ou efbnc, et qui mis en opérande d'un branchement conditionnel provoque un branchement, est équivalent à vrai, sinon il est équivalent à faux.

En effet, sont équivalents à repr vrai tous les termes :

efbc(compare(v,v'),ccond) tels que ccond = test(v,v')

et

efbnc(compare(v,v'),ccond) tels que ccond ≠ test(v,v')

De même les termes suivants sont équivalents à repr faux :

efbc(compare(v,v'),ccond) tels que ccond ≠ test(v,v')
efbnc(compare(v,v'),ccond) tels que ccond = test(v,v')

Ces deux classes d'équivalence sont disjointes. Il est donc impossible de montrer que repr vrai ≃ repr faux. Par contre, on a bien reflexité, symétrie et transivité de la relation ≃ : c'est une conséquence de ces mêmes propriétés de l'égalité sur les codes de condition.

Revenons aux axiomes à vérifier.

Pour vérifier le premier, on va utiliser l'axiome sur les valeurs suivant :

eg(Valeur'C1',Valeur'C2') = eg(C1,C2)

et pour éviter d'avoir à spécifier une représentation de eg(C1,C2), on va reformuler l'axiome et travailler en faisant une analyse de cas :

repr eg(Ent'C1',Ent'C2') ≃ si eg(C1,C2) alors repr vrai sinon repr faux

La partie gauche se réécrit :

efbc(compare(Valeur'CONVERTIR(C1)',Valeur'CONVERTIR(C2)'),eq)

La procédure CONVERTIR conserve, si elle est correcte, l'égalité et l'inégalité.

Si eg(C1,C2) est vrai, alors on a , d'après l'axiome mentionné ci-dessus

Valeur'CONVERTIR(C1)' = Valeur'CONVERTIR(C2)'

Donc, d'après les axiomes sur les codes de condition, un 'test' sur ces deux valeurs a pour résultat 'eq' et on a bien un terme équivalent à 'repr vrai'.

Dans le cas contraire, le résultat sera différent de eq et on a bien un terme équivalent à 'repr faux'.

Le deuxième axiome est :

eg(i,add(i,Ent'1')) = faux

Il se réécrit

efbc(compare(repr i,plus(repr i,Valeur'1')),eq) ≃ repr faux

Or on sait que par les axiomes que :

test(v,plus(v',Val'1')) = si v = v' alors lt ...

donc la partie gauche est équivalente à repr faux.

Le troisième axiome se vérifie de manière identique au précédent.

Le quatrième :

eg(i,i') = eg(i',i)

devient :

efbc(compare(repr i,repr i'),eq) = efbc(compare(repr i',repr i),eq)

or on a l'axiome

```
test(c',c) = si test(c,c') = eq alors eq
             sinon si test(c,c') = lt alors gt
                   sinon lt ;
```

Si la partie gauche est équivalente à 'repr vrai' la partie droite l'est aussi. Si elle est équivalente à 'repr faux' la partie droite l'est également. L'axiome est donc bien conservé.

Pour montrer que l'axiome ci-dessous est conservé :

eg(add(i,Ent'1'),add(i',Ent'1')) = eg(i,i') on a besoin de montrer que les codes de condition vérifient la propriété suivante :

test(plus(v,Valeur'1'),plus(v',Valeur'1')) = test(v,v')

ce qui se montre facilement un utilisant le premier axiome sur les codes de condition et le dernier (pseudo-commutativité). Il est alors immédiat de montrer que l'axiome en question est conservé.

Pour les deux derniers axiomes, la démarche est très similaire.

Pour vérifier la correction de la représentation de l'opération 'dif', on considère l'axiome qui la définit :

dif(i,j) = si eg(i,j) alors faux sinon vrai ;

il se réécrit

```
efbnc(compare(repr i,repr j),eq) = si repr i = repr j
                alors repr vrai sinon repr faux ;
```

A propos de cette réécriture, rappelons que si-alors-sinon doit avoir un booléen comme premier opérande. La représentation de eg, dans cette position ne peut pas être un code de condition : c'est l'égalité sur les valeur comme cela a été dit au début de ce paragraphe.

La distinction entre Code de condition, qui est un type de la Structure Objet et Booléen qui est un type auxiliaire nécessaire à l'écriture des axiomes conditionnels est fondamentale.

Pour revenir à la vérification de la représentation de 'dif', test(repr i,repr j) est égal à 'eq' si repr i = repr j.
Dans ce cas, la représentation de dif(i,j) est bien équivalente à repr faux. De même si on n'a pas repr i = repr j, cette représentation est équivalente à repr vrai.

On a donc prouvé que le type Bool était représenté correctement.

On va maintenant donner une représentation du prédicat sur les entiers 'neg', car ce prédicat intervient dans la définition de la division :

repr neg(i) = efbc(compare(Valeur'0',repr i),gt)

Cette représentation est correcte car les trois axiomes

```
neg(Ent'0') = faux ;
neg(soust(i,Ent'1')) = si eg(i,Ent'0') alors
                       vrai sinon neg(i) ;
neg(add(i,Ent'1')) = si eg(i,soust(Ent'0',Ent'1'))
                     alors faux sinon neg(i) ;
```

sont conservés.

On donne ici la démonstration pour les deux premiers axiomes ; le premier devient :

efbc(compare(Valeur'0',Valeur'0'),gt) = repr faux
or test(Valeur'0',Valeur'0') = eq

Le deuxième se réécrit :

```
efbc(compare(Valeur'0',moins(repr i,Valeur'1')),gt) = si repr i =
            Valeur'0' alors repr vrai sinon repr neg(i)
```

D'après le deuxième axiome sur 'test' on a

```
test(Valeur'0',moins(repr i,Valeur'1')) = si Valeur'0' = repr i
            alors gt
            sinon si Valeur'0' = moins(repr i,Valeur'1')
                  alors eq
                  sinon test(Valeur'0',repr i) ;
```

On fait une analyse par cas de la partie droite :

- repr i = Valeur'0' : d'après la propriété ci-dessus la partie gauche est équivalente à repr vrai puisque le résutat de 'test' est 'gt'

- dans l'autre cas, il y a deux possibilités à examiner :

a) moins(repr i, Valeur'1') = Valeur'0'

Dans ce cas, le résultat du test est eq, donc la partie gauche de l'axiome est équivalente à repr faux également car

test(Valeur'0',repr i) = test(Valeur'0',Valeur'1') = lt ≠ gt

b) dans tous les autres cas, on sait que

repr i ≠ Valeur'0' repr i ≠ Valeur'1'

D'après le deuxième axiome sur test on a

test(Valeur'0',moins(repr i,Valeur'1')) = test(Valeur'0',repr i)

donc la partie gauche est équivalente à

efbc(compare(Valeur'0',repr i),gt)

c'est-à-dire à repr neg(i) □

On peut maintenant vérifier la représentation de la division : l'axiome qui définit la division dans le type Valeur est la traduction, cas par cas, de celui définissant la division sur les entiers. Il en est de même pour la restriction sur le diviseur.

On a donc montré que la représentation des type Ent et Bool de la Structure Source était correcte.

VI.2.2 - Représentation des opérations modifiables

La première de ces opérations est 'valeur' qui a des opérandes de type Idvar. Le type Idvar comporte deux axiomes :

eg(Idvar'C1',Idvar'C2') = eg(C1,C2)
eg(id,id') ⊃ eg(valeur(id),valeur(id'))

Sa représentation est la suivante :

```
type Idvar
    Adresse ;
    repr Idvar'X' = Adresse'ALLOC1(X)' ;
fin Idvar
op(Idvar) → Ent : valeur ;
    repr valeur(id) = ca(repr id) ;
fin valeur ;
```

Pour que la représentation soit correcte, il faut qu'elle conserve l'égalité : deux identificateurs identiques doivent être représentés par la même adresse, deux identificateurs différents par deux adresses différentes. Cette propriété dépend de la procédure ALLOC1. On supposera que ALLOC1 a été prouvée correcte, c'est-à-dire qu'elle vérifie cette spécification.

L'opération 'valeur' qui est modifiable, est représentée par l'opération 'ca' qui l'est également. On notera ρ(Idvar) le domaine de définition de ρ(valeur), qui, rappelons-le, doit être disjoint des domaines de définition des autres opérations représentées par 'ca'.

Pour le type Idseq, le problème est très similaire sauf qu'on a deux opérations modifiables, bornesup et ième, et deux cas d'échec qui doivent être vérifiés par la représentation.

```
type Idseq ;
opérations
    (Idseq,Idseq) → Bool : eg ;
    (Idseq) → Ent : bornesup mod
    (Idseq,Ent) → Ent : ième mod
axiomes
    eg(Idseq'C1',Idseq'C2') = eg(C1,C2) ;
    ...
restrictions
    neg(soust(i,Ent'1')) ⟹ Echec(ième ids,i) ;
    neg(soust(bornesup(ids),i)) ⟹ Echec(ième,ids,i) ;
fin
```

Pour ce qui est de l'égalité, ALLOC2 doit avoir les mêmes propriétés que ALLOC1, c'est-à-dire que deux identificateurs de séquence différents doivent être représentés par des adresses différentes.

On a deux opérations modifiables : 'bornesup' et 'ième' dont les représentations sont de la forme

repr bornesup(ids) = ca(repr ids)
repr ième(ids,i) = efb(m,ca(ad))

Les domaines de définition de ces représentations doivent être disjoints entre eux et disjoints du domaine de définition de valeur. Cela signifie que pour tout idv et ids :

repr idv ≠ repr ids

Notons adr ième(ids,i)(3) l'adresse qui est en opérande de 'ca' dans la représentation de ième. De même, on doit avoir pour tout idv, ids, i dans tout état représentant :

repr idv ≠ adr ième(ids,i)
repr ids ≠ adr ième(ids,i)

Nous reviendrons sur ces propriétés dans le paragraphe sur les invariants de représentation.

Pour ce qui est des restrictions, on va prendre pour convention qu'un branchement conditionnel à une étiquette 'erreur i' est une représentation correcte d'un cas d'échec si la condition du branchement est une représentation de la condition d'échec.

On a vu que

repr ième(ids,i) = efb(m,ca(a))

plus précisément

```
repr ième(ids,i) = efb(compseq4(load(repr i,Registre'A'),
                                compare(cr(Registre'A'),Valeur'1'),
                                brchtcond(Eti'erreur1',lt) ; m1) ;
                       ca(cr(Registre'A'))) ;
```

(3) Il s'agit d'une fonction de représentation "auxiliaire", d'où la notation utilisée.

On va montrer que si

neg(soust(i,Ent'1')) ≈ vrai

appartient à l'état courant S, alors le calcul de ième provoque un branchement à l'étiquette erreur1.

Rappelons que :

load(v,r) = subst(cr(r),v)

Soit

S_1 = appl(load(repr i,Registre'A'),S)

Par définition de subst on a

cr(Registre'A') = repr i ∈ S_1

donc :

```
appl(compseq(compare(cr(Registre'A'),Valeur'1'),
             brcht-cond(Eti'erreur1',lt)),S1) =
appl(compseq(compare(repr i,Valeur'1') ;
             brcht-cond(Eti'erreur1', lt), S1) =
appl(brcht-cond(Eti'erreur1',
                efbc(compare(repr i,Valeur'1'),lt)),S1)
```

Or

```
repr neg(soust(i,Ent'1')) ≈ efbc(compare(Valeur'0',moins(repr i,Valeur'1')),gt)
   ≈ efbc(compare(moins(repr i,Valeur'1'),Valeur'0'),lt)
```

d'après les règles de commutativité de test

≈ efbc(compare(repr i,Valeur'1'),lt)

d'après les propriétés des entiers et le fait que

test(plus(v,Valeur'1'),plus(v',Valeur'1')) = test(v,v')

Donc la représentation de ième est telle que si la représentation de la première condition d'échec est vérifiée on se branche à erreur1, ce qui est bien la propriété voulue.

La vérification du deuxième cas d'échec est parfaitement similaire.

On a maintenant démontré la correction de la représentation des expressions de SEQFLEX. Un certain nombre de propriétés des métaprocédures de conversion ont été mises en évidence comme nécessaires. Une dernière étape de cette démonstration serait de les prouver, par des méthodes, maintenant classiques, de preuve de programme à partir du texte de ces procédures (qui dans PERLUETTE, est du LISP).

VI.2.3 - Invariants de représentation

Dans le paragraphe précédent un certain nombre de propriétés de la représentation ont été mentionnées comme indispensables pour avoir des représentations acceptables des états (par exemple la représentation différente d'un identificateur de variable et d'un identificateur de séquence). Ces propriétés vont être complétées ici par les caractéristiques de la gestion de la mémoire : on va spécifier les états de la Structure Objet susceptibles de représenter des états de la Structure Source, et ces représentations doivent être évidemment acceptables.

Rappelons les invariants de représentations qui sont apparus en VI.2.2 :

1/ non recouvrement des représentations des différents identificateurs

2/ non recouvrement des adresses de la représentation des ièmes éléments d'une séquence entre elles :

eg(adr elt(ids1,i),adr elt(ids2,j)) = eg(ids1,ids2) ∧ eg(i,j)

et avec les représentations des identificateurs.

On va introduire ici d'autres invariants. Certaines règles paraîtront peut-être évidentes : il est cependant essentiel de les énoncer précisément pour les utiliser dans la démonstration. Intuitivement on veut avoir :

3/ pas de partage de blocs entre la liste libre et les représentations des séquences et des identificateurs

4/ non circularité de la liste libre (et des représentations des séquences, c'est une conséquence de 2/).

5/ non recouvrement des représentations des identificateurs et des éléments de séquence avec les adresses 'libre' et 'aux' qui sont réservées à la gestion de la liste libre.

On voit que ces derniers invariants sont une spécification informelle du rôle de la liste libre : il est normal qu'on ait à les introduire explicitement, et il est normal qu'ils ne soient pas apparus en VI.2.2 car il s'agit d'optimisation de la gestion de la mémoire ; on peut très bien avoir une représentation correcte qui ne gère pas la mémoire au mieux.

Formellement, l'invariant 1/ s'écrit :

eg(idv,idv') ⇔ repr idv = repr idv'(4)
eg(ids,ids') ⇔ repr ids = repr ids'
repr idv ≠ repr ids
repr idv ≠ index(repr ids,Valeur'1')
repr ids ≠ index(repr ids',Valeur'1')

L'invariant 5/, pour ce qui est des identificateurs s'écrit :

repr idv ≠ libre repr idv ≠ aux
repr ids ≠ libre repr ids ≠ aux
index(repr ids,Valeur'1') ≠ libre
index(repr ids,Valeur'1') ≠ aux

D'autre part :

aux ≠ libre

---

(4) Comme dans le chapitre V on note a = a' l'opération eg(a,a') sur les adresses.

On notera (INV1) cet ensemble de propriétés. Les identificateurs étant représentés par des adresses constantes il peut être vérifié une fois pour toute et ne peut pas être changé par les modifications.

Pour spécifier les autres invariants, qui sont relatifs à des listes chaînées, on va se donner une opérations sur les adresses, à résultat booléen dont la signification est "a' est chaîné à a".

```
chaîné(a',a) = si a  = Adresse 'nil' alors faux
               sinon si a' = ca(index(a,Valeur'1'))
                     alors vrai
                     sinon chaîné(a',index(a,Valeur'1'))
```

L'invariant 2/ s'écrit :

¬eg(ids,ids') ∧ chaîné(a,repr ids) ⊃ ¬chaîné(a,repr ids')
¬eg(ids,ids') ⊃ ¬ chaîné(repr ids,repr ids')

avec en plus, la non circularité des listes en général qui est spécifiée en 4/ et la propriété suivante :

¬chaîné(repr idv,repr ids)

L'invariant 3/ s'écrit :

chaîné(a,repr ids) ∧ chaîné(a,ca(libre)) = faux
¬chaîné(repr ids,ca(libre))
¬chaîné(repr idv,ca(libre))

On notera (INV2) l'ensemble de ces propriétés qui doivent être vérifiées pour tout état représentant un état source.

La non circularité des listes 4/ devient :

(INV3) ¬chaîné(a,a)

En plus de ces trois ensembles de propriétés on doit préciser que le deuxième mot d'un bloc ne peut être utilisé à d'autres usages. Posons

chaîné(a',a) = appartient(index(a',Valeur'1'),a)

On a alors le quatrième invariant suivant :

```
(INV4)   appartient(a,ca(libre)) ⊃ ¬ chaîné(a,ca(libre))
                                 ∧ ¬ chaîné(a,repr ids)
                                 ∧ (a ≠ repr ids) ∧ (a ≠ repr idv)
         appartient(a,repr ids)  ∧ ¬ eg(ids,ids') ⊃ ¬ chaîné(a,ca(libre))
                                      ∧ ¬ chaîné(a,repr ids')
                                      ∧ (a ≠ repr ids') ∧ (a ≠ repr idv)
```

Il est important, avant de continuer la démonstration, de savoir quelles modifications élémentaires conservent ces trois derniers invariants. C'est l'objet des deux lemmes suivants :

Lemme 1 Soit une adresse a telle que

a) ∀ids, a ≠ index(repr ids,Valeur'1') et
   ¬ appartient(a,repr ids)

b) a ≠ libre et a ≠ index(ca(libre),Valeur'1')
   et ¬ appartient(a,ca(libre))

alors toute modification de la forme store(v,a) conserve les invariants (INV2), (INV3) et (INV4)

Démonstration

Par définition : store(v,a) = subst(ca(a),v). Cette substitution laisse le prédicat "chaîné" inchangé puisque, par définition de 'chaîne (a',a")', son résultat ne dépend que du contenu d'adresses b telles que 'appartient (b,a")'. D'autre part, d'après (INV2) tout bloc chaîné appartient soit à la liste libre soit à une séquence. Sinon le résultat de chaîné est faux. D'après les hypothèses a) et b), cette modification laisse donc le prédicat chaîné inchangé, donc conserve les invariants.

Lemme 2 Toute modification load, compare, brcht, brcht-cond, brcht-neg conserve les invariants de représentation.

C'est évident puisque les liens de chaînage ne sont pas changés par ces modifications.

VI.2.4 - Modifications sans effets sur la représentation

De même qu'il est important, avant de commencer la preuve de la représentation des modifications, de connaître les modifications qui conservent les invariants, il est essentiel de savoir quelles sont les modifications "sans effet de bord". La définition de ces modifications a été donnée en VI.1 (Définition 3). Nous donnons ici une définition différente qui à l'avantage de donner une méthode pour prouver qu'une modification a cette propriété.

Définition 3bis : Une modification m de la Structure Objet est dite "sans effet sur la représentation" si elle ne modifie pas la représentation d'une opération modifiable de la Structure Source.

Soit op de profil $(t_1,\ldots,t_n) \to t_o$ une telle opération vérifiant :

repr $op(x_1,\ldots,x_n) \simeq$ repr $x_o \in S$

où xo est une constante du type to et $\simeq$ la représentation de l'égalité pour ce type. Si, quelque soit op, modifiable, pour tout $x_1,\ldots,x_n$ et pour tout S on a

repr $op(x_1,\ldots,x_n) \simeq$ repr $x_o \in$ appl(m,S)

m est sans effet sur la représentation. On dira 'sans effet'.

La définition 3bis est équivalente à la définition 3 : en effet, si S est la représentation d'un état de la Structure Source, appl(m,S), d'après la remarque précédente contient la représentation des mêmes axiomes spécifiques que S. Les deux états sont donc deux représentations d'un même état "Source".

En utilisant cette définition, on peut montrer que dans la représentation qui nous intéresse, certaines modifications sont sans effet.

Lemme 3 : Toute modification de la forme load(v,r) ou compare (v,v') est sans effet.

Démonstration :

La Structure Source comporte trois opérations modifiables : Valeur, bornesup, ième.

Elles sont représentées par des termes de la forme ca(a). Or ca n'est pas modifié par load et compare.

Lemme 4 : Si a est de type Adresse et vérifie les hypothèses du Lemme 1 et que, en plus

$\forall$idv , a $\neq$ repr idv

$\forall$ids , a $\neq$ repr ids $\wedge$ $\neg$ chaîné(a,repr ids)

alors toute modification de la forme

store(v,a)

est sans effet.

Intuitivement, c'est évident : cette modification laisse les invariants inchangés et l'adresse dont le contenu est modifié ne correspond pas à la représentation d'un objet de SEQFLEX.

Démonstration :

On considère d'abord l'opération valeur

repr valeur(idv) = ca(repr idv)

Par hypothèse a est différent de repr idv donc les formules de la forme

repr valeur(idv) = repr Ent'i'

sont conservées.

Pour l'opération bornesup on a :

repr bornesup(ids) = ca(repr ids)

et comme a est différent de repr ids on a bien la propriété voulue.

Pour ième(ids,i), le principe de la démontration est de montrer que

repr ième(ids,i) = ca(a') $\Longrightarrow$ chaîné(a',repr ids)

comme pour tout ids on a $\neg$ chaîné(a,repr ids), store(v,a) ne provoque aucun effet de bord sur repr ième.

Pour faire cette démonstration (et d'autres dans la suite) on a besoin du lemme suivant :

Lemme 5 : Posons repr ième (ids,i) = ca(adr ième(ids,i)) où adr est la fonction de représentation de profil Ent → Adresse qui a déjà été utilisée pour prouver l'implantation des restrictions sur ième.

Si eg(bornesup(ids),Ent'0') on a :

(1) adr ième(ids,bornesup(ids)) = ca(index(repr ids,Valeur'1'))

et si

¬neg(soust(i,Ent'1'))
¬neg(soust(bornesup(ids),i))
¬eg(i,bornesup(ids))

on a :

(2) adr ième(ids,i) = ca(index(adr ième(ids,add(i,Ent'1')),Valeur'1'))

Démonstration :

On sait que repr ième(ids,i) = efb(m,ca(cr(Reg'A'))
Donc adr ième(ids,i) = efb(m,cr(Reg'A'))

Afin de faciliter la lecture de la démonstration nous rappelons ici la définition de la modification m :

```
compseq13(load(repr i,Registre'A'),
          compare(cr(Registre'A'),Valeur'1') ;
          brcht-cond(Eti'erreur1',lt) ;
          load(moins(ca(repr ids),cr(Registre'A')),Registre'I'),
          compare(cr(Registre'I'),Valeur'0'),
          brcht-cond(Eti'erreur2',lt) ;
          load(ca(index(repr ids,Valeur'1')),Registre'A'),
étiqueter(Eti'BOUCLE',brcht-cond(Eti'EXIT',eq)),
          load(moins(cr(Registre'I'),Valeur'1'),Registre'I'),
          load(ca(index(cr(Registre'A'),Valeur'1')),Registre'A'),
          compare(cr(Registre'I'),Valeur'0'),
          brcht(Eti'BOUCLE'),
étiqueter(Eti'EXIT',suite) )
```


D'après les hypothèses, qu'il s'agisse des formules (1) ou (2), on n'est pas dans des cas d'échec. La vérification des restrictions de ième, permet d'affirmer que l'exécution des six premières modifications ne provoquera pas de branchement aux étiquettes d'erreur. Soit S_1 l'état obtenu en appliquant un état initial S ces modifications.

On a cr(Reg'A') = repr i $\in S_1$
cr(Reg'I') = moins(ca(repr ids),cr(Reg'A')) $\in S_1$

donc cr(Reg'I') = moins(ca(repr ids),repr i) $\in S_1$

de plus cond ≠ lt $\in S_1$
cond = test(cr(Reg'I'),Val'0') $\in S_1$

La démonstration des propriétés (1) et (2) se fait par induction sur les calculs effectués dans la boucle.

Soit S_2 = appl(load(ca(index(repr ids,Val'1')),Reg'A'),S_1)

on a ca(Reg'A') = ca(index(repr ids,Val'1')) $\in S_2$

et toutes les formules ci-dessus, sauf la première appartiennent à S_2.

Considérons maintenant les instructions étiquetées par BOUCLE. Par définition on sait que si

m1 = $compseq^4$(étiqueter(boucle,brcht-cond(exit,eq)),
m2,brcht(boucle),étiqueter(exit,suite))

alors si cond = eq $\in S_2$
on a appl(m1,S_2) = S_2

sinon appl(m1,S_2) = appl(m1,appl(m2,S2))

Or cond = eq $\in S_2$ si test(cr(Reg'I'),Valeur'0') = eq $\in S_2$

c'est-à-dire que d'après les autres propriétés de S_2 :
moins(ca(repr ids),repr i) = Valeur'0'

comme repr bornesup(ids) = ca(repr ids)
dans le cas où i = bornesup(ids) l'état final est S_2 et on a

adr ième(ids,i) = cr(Registre'A')
= ca(index(repr ids,Valeur'1'))

On a donc montré la propriété (1).

Considérons maintenant les états

S_{p+1} = appl(m2,S_p)

où m2 est le corps de la boucle : incrémentation du registre I, progression suivant le lieu de chaînage du contenu du registre A, comparaison du registre I avec zéro.

Si v et a sont respectivement des constantes de type Valeur et Adresse, on a :

cr(Registre'I') = v $\in S_p$ $\Longrightarrow$ cr(Registre'I') = moins(v,Val'1')$\in S_{p+1}$
cr(Registre'A') = a $\in S_p$ $\Longrightarrow$ cr(Registre'A') = ca(index(a,Val'1'))$\in S_{p+1}$

L'état final S_f correspond au cas où

cr(Reg'I') = Val'0'

Or, en S_2 on avait :

cr(Reg'I') = moins(ca(repr ids),repr ids)

On voit que si on calcule ième (ids,add(i,Ent'1')) on appliquera une fois de moins la modification m. On aura donc

adr ième(ids,i) = ca(index(adr ième(ids,add(i,Ent'1')),Valeur'1'))

c'est-à-dire la propriété (2) □.

VI.2.4 - Propriétés des macro-modifications occuper et libérer

Théorème 1 :

∀ ids, si ca(libre) ≠ Adresse'nil', occuper(repr ids) conserve les invariants de représentation.

Démonstration :

Soit S tel que les invariants de représentation soient vrais et

ca(libre) ≠ Adresse'nil' $\in$ S

On veut montrer que ∀ ids, les invariants sont vrais dans

S' = appl(occuper(repr ids),S)

Posons repr ids = t et reprenons la définition de occuper

occuper(t) = $compseq^3$(compare(ca(libre),Adresse'nil'),
brcht-cond(Eti'débordement',eq),m)

avec

m = $compseq^4$(store(ca(libre),aux),
store(ca(index(ca(aux), Valeur'1'),libre),
store(ca(index(t,Valeur'1'),index(ca(aux), Valeur'1')),
store(ca(aux), index(t,Valeur'1')))

D'après les définitions de compseq, compare et brcht-cond et la propriété donnée pour S en hypothèse on a :

S' = appl(occuper(t),S) = appl(m,S_1)

avec

S_1 = appl(subst(cond,test(ca(libre),Adresse'nil')),S)

D'après le lemme 2 les invariants sont vérifiés dans S_1.

On a maintenant la propriété cond ≠ eq $\in S_1$
en plus de ca(libre) ≠ Adresse'nil' $\in S_1$

Il reste à montrer que m conserve les invariants : on va le faire état intermédiaire par état intermédiaire.

Soit S_2 = appl(store(ca(libre),aux),S_1)

"aux" est une adresse spécifique destinée à servir de mémoire de travail pour occuper et libérer. Pour que S_2 soit valide, il faut que l'adresse "aux" vérifie les hypothèses du lemme 1 (ce qui est dit dans les invariants).

On a par définition de "store" :

ca(aux) = ca(libre) $\in S_2$

et les propriétés de S_1 sont conservées.

Soit S_3 = appl(store(ca(index(ca(aux),Valeur'1'),libre),S_2)

Le lemme 1 ne s'applique pas mais les invariants sont, malgré tout, conservés :

ca(aux) = ca(libre) $\in S_2 \Longrightarrow$

S_3 = appl(subst(ca(libre),ca(index(ca(libre),Valeur'1')),S_2)

= {P|P[ca'/ca] $\in S_2$+ca'(a) = si a = libre
alors ca(index(ca(libre),Valeur'1'))
sinon ca(a)}

Supposons qu'on ait chaîné(b,ca(libre)) $\in S_2$

Cela peut s'écrire (par définition de S_2)

(b = ca(index(ca(libre),Valeur'1')) $\vee$ chaîné(b,
ca(index(ca(libre),Valeur'1')) $\in S_2$

ce qui entraîne

b = ca(libre)$\vee$chaîné(b,ca(libre)) $\in S_3$

donc

chaîné(b,ca(libre)) $\in S_3$ seulement si chaîné(b,ca(libre)) $\in S_2$

c'est-à-dire qu'on n'a pas introduit de contradiction avec les invariants relatifs à ca(libre).

Pour ce qui est des prédicats chaîné(b,repr ids) ils ont été laissés inchangés. S_3 vérifie donc les invariants.

Par contre ca(aux) $\neq$ ca(libre) $\in S_3$ et on a introduit, entre autres :

chaîné(ca(aux),ca(libre)) $\in S_3$
appartient(index(ca(aux),Valeur'1'),ca(libre)) $\in S_3$

(à cause de l'invariant de non circularité) et

chaîné(ca(aux),repr ids) $\in S_3$
appartient(index(ca(aux),Valeur'1'),repr ids) $\in S_3$
ca(aux) $\neq$ repr ids $\in S_3$

(à cause de l'incompatibilité avec ca(libre) dans S_2)

Soit maintenant

S_4 = appl(store(ca(index(t,Valeur'1'),index(ca(aux),Valeur'1')),S_3)

On est dans un des cas d'application du Lemme 1 grâce aux propriétés de l'état S_3 mentionnées ci-dessus. Les invariants sont conservés mais on a ajouté les propriétés

ca(index(ca(aux),Valeur'1') = ca(index(t,Valeur'1')) $\in S_4$

d'où

chaîné(b,t) $\in S_4 \Longrightarrow$ chaîné(b,ca(aux)) $\in S_4$

La dernière modification donne l'état final S' :

S' = appl(store(ca(aux),index(t,Valeur'1')),S_4)

on a

ca(index(t,Valeur'1')) = ca(aux) $\in$ S'

c'est-à-dire

chaîné(ca(aux),t) $\in$ S'

or on sait (depuis S_3) que

$\forall$ ids $\neg$ chaîné(ca(aux), repr ids) $\in$ S'
$\neg$ chaîné(ca(aux),ca(libre) $\in$ S'

Cette nouvelle formule n'est donc pas en contradiction avec les invariants □

Corollaire : occuper(repr ids) a pour effet sur la représentation que

repr ième(ids,i) = v $\in$ S $\Longrightarrow$
repr ième(ids,soust(i,Ent'1')) = v $\in$ S'

ou v est une constante de type Valeur et ou

S' = appl(occuper(repr ids),S).

C'est le seul effet de occuper (repr ids) sur la représentation.

Démonstration :

On voit, d'après le Lemme 4 que, comme l'adresse 'aux' vérifie les hypothèses de ce Lemme, les trois premières modifications de occuper sont sans effets sur la représentation.

L'application de ces trois modifications a pour résultat l'état S_4, avec comme propriétés :

¬ chaîné(ca(aux),repr ids) $\in S_4$
ca(aux) $\neq$ repr ids $\in S_4$
ca(index(ca(aux),Valeur'1')) = ca(index(repr ids,Valeur'1'))

ca(repr ids) et ca(index(repr ids,Valeur'1')) n'ayant pas été modifiés, on a

repr ième(ids,i) = v $\in$ S $\Longrightarrow$ repr ième(ids,i) = v $\in S_4$

Or on a

S' = appl(store(ca(aux),index(repr ids,Valeur'1')),S_4)

Considérons le cas où : i = bornesup(ids). D'après le lemme 5 :

adr ième(ids,bornesup(ids)) = ca(index(repr ids,Valeur'1')) $\in S_4$

Soit a la constante de type Adresse telle que

ca(index(repr ids,Valeur'1')) = a $\in S_4$

et supposons que

repr ième(ids,bornesup(ids)) = ca(a) = v $\in S_4$

D'après la définition de store on a :

ca(index(repr ids,Valeur'1')) = ca(aux) $\in$ S'

Or d'après les propriétés de S_4 on sait que

ca(index(ca(aux),Valeur'1')) = a $\in S_4$

et cette propriété est conservée dans S' car ca(aux) n'est pas chaîné à repr ids, donc on a :

index(repr ids,Valeur'1') $\neq$ ca(aux) et par ailleurs : repr ids $\neq$ ca(aux)

Donc on a :

ca(index(ca(index(repr ids,Valeur'1')),Valeur'1')) = v $\in$ S'

soit, d'après le lemme 5 :

repr ième(ids,soust(bornesup(ids),Ent'1')) = v $\in$ S'

En utilisant la relation de récurrence du lemme 5, on voit que le corollaire est vérifié pour i = bornesup(ids) et pour tout i < bornesup et supérieur à 1.

Si i = 1 on a

repr ième(ids,Ent'1') = v $\in$ S

ca(ca(index(adr ième(ids,Ent'1'),Valeur'1'))) = v $\in$ S'

De même on montre le théorème suivant et son corollaire :

Théorème 2 :

Pour tout identificateur de séquence ids, si :

ca(index(repr ids,Valeur'1')) $\neq$ Adresse'nil'

alors libérer (repr ids) conserve les invariants de représentation.

Corollaire :

Soit S' = appl(libérer(repr ids),S) $\Longrightarrow$
libérer (repr ids) a pour seuls effets sur la représentation :

repr ième(ids,i) = v $\in$ S
repr ième(ids,add(i,Ent'1')) $\in$ S'

pour tout i tel que i $\geq$ 1 et i < bornesup(ids)

De plus

adr ième(ids,Ent'1') = Adresse'nil'

VI.2.5 - Preuve de la représentation des modifications élémentaires

Les différents lemmes et théorèmes qui viennent d'être présentés vont être maintenant utilisés pour vérifier la représentation des modifications. Cette vérification se fera en deux parties : on va tout d'abord traiter les modifications élémentaires, c'est-à-dire celles qui n'ont pas d'opérandes de type Modif. On traitera ensuite les opérations de composition de modifications en considérant que leurs opérandes sont représentés correctement.

Rappelons que pour chaque modification élémentaire, on doit démontrer que sa représentation

- effectue bien les substitutions spécifiées dans la définition : si subst(f(x),y) apparaît dans la définition de m, subst(repr f(x),repr y) doit apparaître dans la définition de repr m ou alors, on peut avoir subst(g(w),z) avec g(w) $\simeq$ repr f(x) et z $\simeq$ repr y.
- conserve les invariants de représentation
- est sans effets autres que ceux spécifiés par la définition.

Vérification de init

La définition de init est la suivante :

init(ids) = subst (bornesup(ids),Ent'0')

La représentation donnée pour init est :

repr init(ids) = compseq(store(Valeur'0',repr ids),
store(Adresse'nil',index(repr ids,Valeur'1')))

Rappelons que, d'autre part :

repr bornesup(ids) = ca(repr ids)
repr Ent'0' = Valeur'0'

Posons

S' = appl(repr init(ids),S)

Le premier point à vérifier est que la substitution de la définition est bien effectuée.

Or

subst(repr bornesup(ids),repr Ent'0') = subst(ca(repr ids),Valeur'0')

qui est la définition du premier 'store'. Le store qui suit n'affecte pas repr bornesup(ids).

Le deuxième point est que les invariants de représentation sont conservés : par définition de chaîné on a

$\forall$ a $\in$ Adresse, $\neg$chaîné(a,repr ids) $\in$ S'

car ca(index(repr ids),Valeur'1') = Adresse'nil' $\in$ S' d'après le deuxième store.

D'autre part, le premier store vérifie les hypothèses du lemme 1, donc laisse les invariants inchangés, et chaîné(a,repr ids') ainsi que chaîné(a,ca(libre)) sont laissés inchangés par le deuxième store.

Le troisième point se vérifie facilement car les deux modifications composant repr init vérifient les hypothèses du lemme 4 pour tout repr idv et repr ids' tel que $\neg$eg(ids,ids'). On n'a donc pas d'effets autres que ceux spécifiés.

Vérification de ajouter :

La définition de ajouter est :

ajouter(ids,j) = concat(subst(bornesup(ids),add(bornesup(ids),Ent'1')),
subst(ième(ids,bornesup(ids)),j))

Sa représentation est :

repr ajouter(ids,j) = compseq^3(occuper(repr ids),
store(plus(ca(repr ids),Valeur'1'),repr ids),
store(repr j,ca(index(repr ids,Valeur'1')))

Soit S' = appl(repr ajouter(ids,j),S)

Le premier point se vérifie facilement car la deuxième modification correspond à :

subst(repr bornesup(ids),repr add(bornesup(ids),Ent'1')) La troisième correspond à :

subst(repr ième(ids,bornesup(ids)),repr j). On a donc les substitutions voulues, appliquées dans l'ordre spécifié (d'après les définitions de concat et compseq).

Pour le deuxième point, on sait par le théorème 1 que occuper(repr ids) conserve les invariants. Les deux 'store' vérifient les hypothèses du Lemme 1, donc les conservent également.

Le troisième point est plus délicat à vérifier. On sait, par le corollaire du théorème 1 que occuper(repr ids) a des effets sur la représentation. Mais

store(plus(ca(repr ids),Valeur'1'),repr ids) =
subst(repr bornesup(ids),repr add(bornesup(ids),Ent'1'))

en a également. Soit m cette modification.

On sait d'après le Lemme 5, que dans l'état S, pour tout i tel que :

$i \geq 1$ $\quad$ $i <$ bornesup(ids)

adr ième(ids,i) = ca(index(adr ième(ids,add(i,Ent'1')),Valeur'1')
adr ième(ids,bornesup(ids)) = ca(index(repr ids,Valeur'1')

Posons

repr bornesup(ids) = repr b $\in S_1$
repr bornesup(ids) = repr add(b,Ent'1') $\in S_1'$

où b est une constante entière et ou $S_1' =$ appl(m,S_1)

Alors

repr ième(ids,b) = v $\in S_1 \Longrightarrow$
repr ième(ids,add(b,Ent'1')) = v $\in S_1'$

car adr ième(ids,bornesup(ids)) n'a pas été modifié. Il en est de même pour tout i compris dans les bornes de la séquence et strictement supérieur à 1.

On voit que les effets de la modification m annulent ceux de occuper(repr ids) :

repr ième(ids,i) = v $\in$ S $\Longrightarrow$
repr ième(ids,soust(i,Ent'1')) = v $\in$ appl(occuper(repr ids),S)

soit S_1 cet état :

repr ième(ids,i) = v $\in$ appl(m,S_1)

On doit traiter à part le cas i = 1. D'après le corollaire du théorème 1

repr ième(ids,Ent'1') = v $\in$ S $\Longrightarrow$
ca(ca(index(adr ième(ids,Ent'1'),Valeur'1'))) = v $\in S_1 \Longrightarrow$
ca(ca(index(adr ième(ids,Ent'2'),Valeur'1'))) = v $\in$ appl(m,S_1) $\Longrightarrow$
repr ième(ids,Ent'1') $\in$ appl(m,S_1).

Donc les deux premières modifications de repr ajouter sont sans effets autres que la première substitution spécifiée pour ajouter. La dernière modification est sans effets autres que la deuxième substitution spécifiée pour ajouter.

Vérification de retirer

La définition de retirer est :

retirer(ids) = subst(bornesup(ids),soust(bornesup(ids),Ent'1'))

on a la restriction

eg(bornesup(ids),Ent'0') $\Longrightarrow$ Echec(retirer,ids)

la représentation de retirer est :

repr retirer(ids) = compseq^4(compare(ca(repr ids),Valeur'0'),
brcht-cond(Eti'erreur3',eq)
libérer(repr ids),
store(moins(ca(repr ids), valeur'1'),repr ids))

Pour ce qui est du premier point, la représentation de retirer provoque bien un branchemnet à une erreur si on est dans le cas d'échec ; si on n'est pas dans ce cas, la substitution demandée est bien représentée.

```
compseq(compare(ca(repr ids),Valeur'0'),
        brcht-cond(Eti'erreur3,eq)) =
        brcht-cond(Eti'erreur3',efbc(compare(ca(repr ids),
                                     Valeur'0'),eq)) =
        brcht-cond(Eti'erreur3',efbc(compare(repr bornesup(ids),
                                     repr Ent'0'),eq)) =
        brcht-cond(Eti'erreur3',repr eg(bornesup(ids),Ent'0'))
```

On a donc bien un branchement conditionnel à une erreur si la condition d'échec est vraie.

Si celle-ci n'est pas vrai, alors on a

```
store(moins(ca(repr ids),Valeur'1'),repr ids) =
subst(repr bornesup(ids),repr sousl(bornesup(ids),Ent'1'))
```

ce qui est bien la substitution demandée.

. deuxième point :

D'après le Lemme 2, les deux premières modifications conservent les invariants. D'après le théorème 2 libérer(repr ids) également. D'après le Lemme 1, le dernier store aussi. repr retirer(ids) conserve donc les invariants.

. troisième point :

Les deux premières modifications n'ont pas d'effets (Lemme 3) ; libérer(repr ids) a les effets de bord suivants (d'après le corollaire du théorème 2) :

Soit S_1 = appl(libérer(repr ids),S)

alors

repr ième(ids,i) = v $\in$ S $\Longrightarrow$
repr ième(ids,add(i,Ent'1')) = v $\in$ S_1

et adr ième(ids,Ent'1') = Adresse'nil' $\notin$ S_1

La modification

m = store(moins(ca(repr ids),Valeur'1'),repr ids)

annule cet effet de bord car elle est équivalente à

subst(repr bornesup(ids),repr sousl(bornesup(ids),Ent'1'))

et on a

repr ième(ids,add(i,Ent'1')) = v $\in$ S_1 $\Longrightarrow$
repr ième(ids,i) = v $\in$ appl(m,S_1)

de plus

repr ième(ids,Ent'1') est bien défini.

La représentation de retirer est donc correcte.

Vérification de changer-ième

La définition de changer-ième est :

changer-ième(ids,i,j)= subst(ième(ids,i),j)

Les restrictions sont les mêmes que pour ième(ids,i).
La représentation est de la forme

repr changer-ième(ids,i,j) = compseq(m,store(repr j,cr(Registre'A'))

Le premier point se vérifie facilement car on sait que

repr ième(ids,i) = efb(m,ca(cr(Registre'A'))

donc

subst(repr ième(ids,i),j) =
subst(efb(m,ca(cr(Registre'A'))),repr j) =
compseq(m,subst(ca(cr(Registre'A')),repr j)) =
compseq(m,store(repr j,cr(Registre'A')) =
repr changer-ième(ids,i,j).

Les restrictions sont correctement représentées puisqu'on a montré que celles sur 'ième' l'étaient.

Pour les deuxième et troisième points, m est une composition séquentielle de modifications de la forme load, compare, brcht-cond, qui, d'après les lemmes 2 et 3 conservent les invariants et sont 'sans effet'.

D'autre part, cr(Registre'A'), d'après le Lemme 5 est égal à adr ième(ids,i) ; en utilisant ce lemme et en raisonnant par recurrence sur i, on montre qu'un store à cette adresse conserve les invariants.

La représentation de changer-ième est donc correcte.

Vérification de affecter

La définition de affecter est :

affecter(idv,i) = subst(valeur(repr idv),i)

Sa représentation est :

repr affecter(idv,i) = store(repr i,repr idv)

De plus

repr valeur(idv) = ca(repr idv)

Le premier point se vérifie donc immédiatement. Pour ce qui est du deuxième point, repr idv, par définition de (INV1) vérifie les hypothèses du Lemme 1. Donc la représentation de affecter conserve les invariants. Quand au troisième point, il découle du fait que les représentations des opérations modifiables sont correctes et que la représentation des identificateurs de variable préserve égalité et inégalité.

On a ainsi démontré correcte la représentation des modifications élémentaires. On va maintenant prouver la représentation des compositions de modifications.

VI.2.5 - Composition de modifications

Avant de commencer la démonstration, on va rappeler brièvement quelles sont les compositions de modifications dans la Structure Source et la Structure Objet, et établir quelques propriétés des modifications de la Structure Objet qui représentent des modifications de la Structure Source.

Dans la Structure Source, on a les opérations suivantes sur les modifications :

(Modif,Modif) → Modif : concat ;
(Bool,Modif,Modif) → Modif : conditionnelle ;
(Bool,Modif) → Modif : itération ;

Dans la Structure Objet on a :

(Modif,Modif) → Modif : compseq ;
(Eti,Modif) → Modif : étiqueter ;
(Eti) → Modif : brcht ;
(Eti,Codecond) → Modif : brcht-cond,brcht-neg ;

De plus, dans la Structure Objet on a deux prédicats :

(Modif,Eti) → Bool : sortie, entrée ;

sortie(m,e) indique qu'il existe un branchement, conditionnel ou non, à l'étiquette e dans m, et que e est une étiquette extérieure à m.

entrée(m,e) indique que e est une étiquette intérieure à m, c'est-à-dire qu'il existe une modification étiqueter(e,m1) dans m.

Pour prouver la correction des représentations de concat, conditionnelle et itération qui ont été données au chapitre précédent, on a besoin du Lemme suivant :

Lemme 6 : Soit m une modification de la Structure Source et e une étiquette de la Structure Objet ne correspondant pas à un cas d'erreur. Alors on a

sortie(repr m,e) = faux

Du fait qu'on a introduit les opérateurs, efb et efbc, il faut étendre les prédicats sortie et entrée aux types Contenu et Codecond. On a également pour tout booléen

sortie(repr b,e) = faux

et pour tout entier et tout identificateur

sortie(repr i,e) = faux

Démonstration : Elle se fait par induction sur le type Source. Considérons par exemple, l'opération 'conditionnelle'. Supposons la propriété vraie pour les opérandes b,m,m' de cette opération. Alors elle est vraie pour

conditionnelle(b,m,m')

En effet, on a :

repr conditionnelle(b,m,m') = (#NEG:=GENETI;EXIT:=GENETI#

compseq^5(brcht-neg(Eti'NEG',repr b),
repr m,
brcht(Eti'EXIT'),
étiqueter(Eti'NEG',repr m'),
étiqueter(Eti'EXIT',suite)))

Donc

entrée(repr conditionnelle(b,m,m'),Eti'NEG') = vrai
entrée(repr conditionnelle(b,m,m'),Eti'EXIT') = vrai

D'après les axiomes sur 'sortie' et 'compseq' donnés en V.1, on en déduit :

sortie(repr conditionnelle(b,m,m'),Eti'NEG') = faux
sortie(repr conditionnelle(b,m,m'),Eti'EXIT) = faux

De plus, pour tout étiquette e ne correspondant pas à un cas d'erreur, on a sortie(m",e) = faux pour tous les opérandes de compseq^5, d'où

sortie(repr conditionnel(b,m,m'),e) = faux.

La démonstration est similaire pour toute les opérations de la Structure Source donnant pour résultat une modification. Etant donné sa longueur et sa simplicité elle n'est pas donnée ici.

Vérification de la concaténation

Par définition, on a :

appl(concat(m,m'),S) = appl(m',appl(m,S))

On s'est donné pour représentation :

repr concat(m,m') = compseq(repr m,repr m')

Or la définition de compseq est :

$\forall$ e, sortie(m,e) = faux $\Longrightarrow$ appl(compseq(m,m'),S) = appl(m',appl(m,S))

Il est immédiat que si S ne comporte pas une condition d'échec pour m on a :

appl(repr concat(m,m'),S) = appl(repr m',appl(repr m,S))

d'après le lemme précédent.

D'autre part, si repr m et repr m' conservent les invariants et n'ont pas d'autres effets que ce qui est spécifié dans les définitions de m et m', repr concat(m,m') à la même propriété.

Vérification de la conditionnelle

Par définition, on a :

b = vrai $\in$ S $\Longrightarrow$ appl(conditionnelle(b,m,m'),S) = appl(m,S)
b = faux $\in$ S $\Longrightarrow$ appl(conditionnelle(b,m,m'),S) = appl(m',S)

La représentation a été donnée dans la démonstration partielle du Lemme 6.

Les règles sur les modifications brcht et brcht-neg de la Structure Objet sont les suivantes :

appl(compseq^3(brcht(eti),m,étiqueter(eti,m')),S) =
appl(étiqueter(eti,m'),S)

cc $\neq$ cond $\in$ S $\Longrightarrow$ appl(compseq^3(brcht-neg(eti),m,étiqueter(eti,m')),S) =
appl(compseq(m,étiqueter(eti,m')),S)

Démonstration :

Rappelons que

repr b ≃ repr vrai si

repr b ≃ efbc(compare(v,v'),test(v,v'))

On a alors :

appl(compseq(brcht-neg(Eti'NEG',repr b),m),S) =

appl(compseq^3(compare(v,v') ; brcht-neg(Eti'NEG',test(v,v')),m),S) =

appl(compseq(brcht-neg(Eti'NEG',test(v,v')),m),S_1)

avec cond = test(v,v') ∈ S_1

et S_1 est une représentation du même état source que S.

D'où

appl(repr conditionnelle(b,m,m'),S) =

appl(compseq^5(brcht-neg(Eti'NEG',test(v,v')),

repr m,brcht(Eti'EXIT'),...),S_1)

Donc si repr b ≃ repr vrai on a

= appl(compseq^4(repr m,brcht(Eti'EXIT'),...,étiqueter

(Eti'EXIT',suite)),S_1)

= appl(compseq(repr m,étiqueter(Eti'EXIT',suite),S_1)

= appl(étiqueter(Eti'EXIT',suite),appl(repr m,S_1))

= appl(repr m,S_1) = S'

S_1 étant une représentation de l'état source d'origine, S' est, d'après le théorème fondamental, une représentation de l'application de m à cet état :

repr b ≃ repr vrai ∈ S ⟹

appl(repr conditionnelle(b,m,m'),S) =

appl(compare(v,v'),appl(repr m,S)

qui représente le même état que appl(repr m,S)

La première propriété de la conditionnelle est donc vérifiée.

La deuxième est donnée ici très rapidement. On note ≃ la relation entre états de la Structure Objet qui représentent le même état de la Structure Source.

repr b ≃ repr faux si

repr b ≃ efbnc(compare(v,v'),test(v,v'))

appl(compseq^5(brcht-neg(Eti'NEG',repr b),...),S) =

appl(compseq^6(compare(v,v'),brcht-cond(Eti'NEG',test(v,v')),

...,étiqueter(Eti'NEG,repr m'),

étiqueter(Eti'EXIT',suite)),S) =

appl(compseq(compare(v,v'),étiqueter(Eti'NEG',repr m'),

étiqueter(Eti'EXIT',suite)),S) ≃

appl(repr m',S) □

Vérification de l'itération

Par définition on a :

b = vrai ∈ S ⟹ appl(itération(b,m),S) =

appl(itération(b,m),appl(m,S))

b = faux ∈ S ⟹ appl(itération(b,m),S) = S

La représentation donnée est :

repr itération(b,m) =

compseq^4(étiqueter(Eti'BOUCLE',brcht-neg(Eti'EXIT',repr b)) ;

repr m,brcht(Eti'BOUCLE),

étiqueter(Eti'EXIT',suite))

D'autre part, pour les branchements en arrière (comme brcht(Eti'BOUCLE') on s'est donné la règle :

Soit

m = étiqueter(boucle,compseq^5(m1,brcht-cond(exit,cc),

m2,brcht(boucle),étiqueter(exit,m3))

avec

sortie(m1,e) = faux ∧ sortie(m2,e) = faux ∀ e.

On a :

cond = cc ∈ appl(m1,S') ⟹
appl(m,S) = appl(compseq(m1,étiqueter(exit,m3))),S)

et

cond ≠ cc ∈ appl(m1,S) ⟹
appl(m,S) = appl(m,(compseq(m1,m2),S))

Considérons d'abord le cas simple où b = faux

```
repr(itération(b,m)) =
     compseq^4(étiqueter(Eti'BOUCLE',brcht-neg(Eti'EXIT',
               repr faux)),repr m,brcht(Eti'BOUCLE'),
               étiqueter(Eti'EXIT',suite))
appl(repr itération(b,m),S) =
     appl(étiqueter(Eti'BOUCLE',
          compseq^5(compare(v,v'),brcht-cond(Eti'EXIT',
                    test(v,v')),repr m,brcht(Eti'BOUCLE'),
                    étiqueter(Eti'EXIT',suite)),S) =
     appl(compseq(compare(v,v'),étiqueter(Eti'EXIT',suite)),S) = S'
```

Par définition de appl(compare(v,v'),S)

Or S' représente le même état que S : 'compare' et 'suite' sont sans effets sur la représentation.

Dans le cas où b = vrai, on a

repr b ≃ efbnc(compare(v,v'),cc) avec cc ≠ test(v,v')

d'où

```
appl(repr itération(b,m),S) ≃
     appl(étiqueter(Eti'BOUCLE',
          compseq^5(compare(v,v'),
                    brcht-cond(Eti'EXIT',cc) ;
                    repr m,brcht(Eti'BOUCLE'),
                    étiqueter(Eti'EXIT',suite)),S)
```


or cc ≠ test(v,v') donc on doit appliquer la deuxième règle :

```
= appl(repr itération(b,m),
       appl(compseq(compare(v,v'),repr m),S))
≃ appl(repr itération(b,m),appl(repr m,S)) □
```

Avec cette vérification, se termine la preuve de l'implantation de SEQFLEX. On voit que cette preuve est longue (certaines démonstrations ont due être données incomplétement) mais très systématique. Elle a l'énorme avantage d'obliger à préciser très clairement les propriétés de l'implantation : la rédaction de cette preuve a fait apparaître plusieurs oublis dans une première spécification de la représentation : test pour savoir si la liste libre est vide, augmentation de 1 de la borne supérieure dans la représentation de ajouter ...

Il faut noter qu'il s'agit d'une implantation compliquée, quoique le langage soit à première vue simple et que tous les problèmes susceptibles d'apparaître dans une preuve ont été abordés. L'établissement des invariants de représentation est loin d'être triviale dans ce cas. Il en est de même pour la notion d'effets sur la représentation.

Il paraît indispensable, pour mener à bien de manière agréable de telles preuves, de disposer d'un système de démonstration. Le travail de l'implanteur serait d'établir les théorèmes, lemmes et vérifications à faire. De tels systèmes existent déjà pour des problèmes très voisins : AFFIRM pour les types abstraits (sans modifications) et L.C.F pour des problèmes plus généraux. La preuve de l'implantation des expressions d'un petit langage exemple a été déjà obtenue en utilisant une première version d'AFFIRM. Parmi les suites à donner à ce travail, l'expérimentation de ces deux systèmes pour établir les preuves d'implantation est, à notre avis, un point essentiel.

La preuve de la même implantation de SEQFLEX en utilisant la Structure Source donnée en III.4 est d'une longueur tout à fait similaire : les invariants sont moins nombreux, mais on a à spécifier au plus grand nombre de représentations d'opérations auxiliaires. Cette variante de la Structure Source a donc à la fois peu d'incidence sur la définition sémantique (par conséquent sur la spécification de l'implantation) et sur la preuve de cette implantation. On peut conjecturer que c'est une propriété générale : si on

spécifie une Structure Source avec peu d'opérations, les invariants de représentation sont plus longs à établir, mais la preuve des opérations est plus courte. Si on utilise beaucoup d'opérations auxiliaires, la vérification des axiomes sur ces opérations est plus longue, mais introduit de fait un certain nombre d'invariants. Les variantes dans le choix de la Structure Source ou donc probablement peu d'influence sur la complexité des preuves de représentation. Le seul vrai problème dans le choix d'une Structure Source est d'éviter de faire de la surspécification en dotant le langage Source de propriétés non indispensables et en limitant ainsi ses implantations possibles.

CONCLUSION

Nous proposons dans cette thèse une sémantique des langages de programmation originale, bien adaptée à la spécification et à la preuve de compilateurs. Ce travail se situe à la frontière de deux disciplines de l'informatique : l'étude théorique des langages de programmation et la compilation de ceux-ci. Les intégrer dans une même étude de manière à obtenir des retombées pratiques fondées sur des bases théoriques rigoureuses n'a pas toujours été facile, mais le bilan est positif puisqu'on a abouti à un générateur de compilateurs, basé sur une méthode formelle de spécification et de preuve de compilateurs, qui a été expérimenté avec succès sur des exemples réalistes.

Cependant, ce serait une erreur de limiter les applications de la théorie présentée ici à un générateur de compilateurs particulier : nous avons, en fait, dégagé une méthode générale de conception et de réalisation des compilateurs qui est simple, raisonnable et facilement applicable dans la pratique. Selon cette méthode, les langages source et objet sont considérés indépendamment l'un de l'autre. La première étape de traduction met en cause seulement le langage source et produit ce que nous avons appelé un terme ; mais les habitués de la compilation appellent ce premier texte intermédiaire un arbre abstrait. Cet arbre est très proche de la structure de données interne manipulée par les éditeurs orientés-vers-un-langage qui se généralisent aujourd'hui [Cou 78, HKM 77]. Le type de compilateur que nous proposons a la qualité d'être facilement compatible avec un tel éditeur.

La troisième phase de traduction est uniquement dépendante du langage objet et peut être réalisée une fois pour toutes pour une machine donnée. La traduction d'un langage se ramène alors à une reécriture de termes, ce qui est facilement implémentable de manière performante.

La méthode de preuve est probablement moins directement exportable vers un contexte industriel : en effet, l'établissement des axiomes pour les deux langages est une tâche difficile et sans outil d'aide à la démonstration la preuve est très fastidieuse. Ce dernier point devrait pouvoir se résoudre

à court terme, il n'en est pas de même du premier.

C'est pourquoi parmi les problèmes qui restent à étudier nous mettons en première place la méthodologie : trouver quel type abstrait est associé à un langage, comment le construire, quels choix faire s'il y en a, que se passe-t-il quand on enrichit le langage ? De plus, dans la mesure ou on n'a pas de méthode pour les établir, comment valider les axiomes ? Cela revient à se poser le problème de la cohérence avec d'autres sémantiques, problème déjà évoqué à la fin du chapitre IV.

Toujours dans le cadre méthodologique, une autre question se pose qui dépasse largement le problème de la compilation : c'est l'aide à la spécification de la représentation des gros types abstraits. En particulier, est-il possible de construire des représentations complètes à partir des choix fondamentaux ? D'autre part, comment peuvent se composer des représentations standard de sous-types en fonction de l'ensemble du type?

On voit que les problèmes ouverts ne manquent pas, dont le moindre n'est pas de situer plus précisément cette sémantique par rapport aux théories déjà existantes.

BIBLIOGRAPHIE

[AB77] Apt K.R, De Bakker J.W.,
Semantics and Proof Theory for Pascal Procedures - Proceeding of 4th Coll. on Automata Languages and Programming - Springer Verlag (1977).

[ABB76] Anderson E., Belz F., Blum E.,
SEMANOL (73) A metalanguage for programming the semantics of programming languages. Acta Informatica, n° 6. 1976, p. 109-131.

[ADJ75] Goguen J.A., Thatcher J.W., Wagner E.G., Wright J.B.,
Initial algebra semantics and continuous algebras. JACM n° 24, 1977, p. 68-95.

[AU77] Aho A.V., Ullmann J.D.,
Principles of compiler design, Addison-Wesley, 1977.

[de B67] de Bakker J.W.,
Formal definition of programming languages. Mathematical Centrum Tracts 16, Amsterdam, 1967.

[BG77] Burstall R.M., Goguen J.A.,
Putting theories together to make specifications. Proceedings of the 5th international joint conference on artificial intelligence, 1977, p. 1045-1058.

[BL69] Burstall R.M., Landin P.J.,
Programs and their proofs : an algebraic approach. Machine Intelligence 4, Edinburgh University Press, 1969, p. 17-43.

[Blu69] Blum E.K.,
Towards a theory of semantics and compilers for programming languages. Journal of Computer and System Sciences, Vol 3, 1969, p. 248-275.

[Bou77] Boullier P.,
Le système SYNTAX, Manuel d'utilisation, Groupe Langages et Traducteurs, IRIA, 1977.

[Bou80] Boullier P.,
Génération automatique d'analyseurs syntaxiques avec rattrapage d'erreurs, Journées Francophones sur la production assistée de logiciel, Genève 1980.

[Bur70] Burstall R.M.,
Formal description of program structure and semantics in first order logic. Machine Intelligence 4, Edinburgh University Press, 1970, p. 79-98.

[Cou78] Couhault A.M.,
Editeur syntaxique LIS et Constructeur. Rapport SESORI, IRIA 1978.

[Coo78] Cook S.A.,
Soundness and Completeness of an Axiom System for Program Verification , SIAM J. COMPUT. Vol 7, n° 1, (Feb. 78) pp. 70-80.

[Den 76] Dendien-Gaudel M.C.,
Application des structures de données à la description et à la preuve de traducteurs. Congrès AFCET-Informatique, Gif-sur-Yvette, 1976.

[Der79] Deransart P.,
Synthèse automatique de traducteurs définis par attributs sémantiques. Actes du 2ème congrès AFCET-IRIA en reconnaissance des formes et intelligence artificielle, Toulouse, 1979, pp. 111-120.

[Des80] Deschamp Ph.,
Thèse de docteur-ingénieur - à paraître.

[Dij76] Dijkstra E.W.,
A discipline of programming, Prentice-Hall 1976.

[DM78] Deschamp Ph., Mazaud M.,
Phase 1 : Manuel d'utilisation, Groupe Langages et Traducteurs, IRIA, 1978.

[Don76] Donahue J.E.,
Complementary definitions of programming language semantics - Lectures Notes in Computer Science n° 42 - Springer Verlag 1976.

[Ern77] Ernst G.N.,
Rules of Inference for Procedure Calls. Acta Informatica 8, 1977.

[Eva62] Evans A.,
An Algol 60 Compiler. ACM National Conference, Denver, 1962.

[Fel66] Feldman J.A.,
A formal semantics for computer languages and its application in a compiler-compiler, CACM 9, n° 1, 1966, p. 3-9.

[FG68] Feldman J.A., Gries D.
Translator writing systems. CACM 11, n° 2, 1968, p. 77-103.

[Fin74] Finance J.P.,
Contribution à la formalisation d'un langage de programmation, Application à Algol 68. Thèse de Spécialité, Université de Nancy I, 1974.

[Fin76] Finance J.P.,
Une méthode de formalisation de la sémantique d'un langage de programmation. RAIRO rouge 10, 1976, n° 8, p. 5-32 et n° 12, p. 5-21.

[Flo67] Floyd R.W.,
Assigning meanings to programs. Proc. Symposium in applied Mathematics, Mathematical Aspects of Computer Science 19, 1967, p. 19-32.

[Gau77] Gaudel M.C.,
A formal approach to translator specification. Information Processing 1977, North-Holland, 1977.

[Gau78] Gaudel M.C.,
Spécifications incomplètes mais suffisantes de la représentation des types abstraits. Rapport Laboria n° 320, IRIA 1978.

[Gau80] Gaudel M.C.,
Specification of compilers as abstract data type representation. International Workshop on semantics-directed compilers. Aarhus. Jan. 1980. (actes à paraître chez Springer - Verlag).

[GDM78] Gaudel M.C., Deschamp Ph., Mazaud M.,
Semantics of Procedures as an Algebraic Abstract Data Type , Rapport Laboria n° 334, IRIA 1978.

[GH78] Guttag J.V., Horning J.J.,
The algebraic specification of abstract data type. Acta Informatica 10, n° 1, 1978, p. 27-52.

[GH79] Guttag J.V., Horning J.J.,
Axioms for a display interface. May 1979. à paraître dans POPL 1980.

[GHL77] Guttag J.V., Horning J.H., London R.L.,
A Proof Rule for Euclid Procedures - Proceedings of the IFIP TC2 Working Conference on Formal Description of Programming Concepts, North Holland 1977.

[GHM76] Guttag J.V., Horowitz E., Musser D.R.,
Abstract data types and software validation. CACM 21, n° 12, 1978, p. 1048-1064.

[GMW77] Gordon M., Milner R., Warsworth C.,
Edinburgh L.C.F. Internal report CSR-11-77, Edinburgh University, 1977.

[GO77] Goguen J.A.,
Abstract errors for abstract data types. Formal description of programming concepts (Neuhold Ed.) North-Holland, 1978, p. 491-526.

[GP79] Gaudel M.C., Pair C.,
Construction de compilateurs basée sur une sémantique formelle. Acte des Journées Francophones sur la certification du logiciel. Genève 1979.

[Gu78,79] Guttag J.V.,
Note on type abstraction (version 2). to appear in IEEE transactions on Software Engineering.

[GT78] Gaudel M.C., Terrine G.,
Synthèse de la représentation d'un type abstrait par des types concrets. Actes du congrès AFCET-TTI, 1978, p. 434-445.

[GTW78] Goguen J.A., Thatcher J.W., Wagner E.G.,
Abstract data types as initial algebras and the correctness of data representations. Current Trends in Programming Methodology 4, (Yeh R. Ed), Prentice-Hall, 1978, p. 80-149.

[HKM77] Huet G., Kahn G., Maurice P.,
Environnement de programmation PASCAL. Rapport SESORI, IRIA, 1977.

[Hoa69] Hoare C.A.R.,
An axiomatic basis of computer programming. CACM 12, n° 10, 1969, p. 576-583.

[Hoa71] Hoare C.A.R.,
Procedures and Parameters, An Axiomatic Approach . Symposium on Semantics of Algorithmic Languages (E. Engler Ed.) Lecture Notes in Math. Vol 188, Springer Verlag 1971.

[Hoa72] Hoare C.A.R.,
Proofs of correctness of data representation. Acta Informatica 1, n° 1, 1972, pp. 271-281.

[Iro61] Irons E.
A syntax-directed compiler for Algol 60. CACM 4, n° 1, 1961, p. 51-55.

[KB70] Knuth D.E., Bendix P.B.,
Simple word problems in universal algebras. Computational problems in abstract algebra, (Leech J. ed), Pergamon Press, 1970, p. 263-297.

[Knu68] Knuth D.,
Semantics of context-free languages. Math. Systems Theory 2, n° 2, 1968, p. 127-145.

[Kos71] Koster C.H.A.,
Affix Grammars. Algol 68 Implementation, North Holland 1971.

[Liv78] Livercy C.,
Théories des programmes : schémas, preuves, sémantique. Dunod informatique, Paris, 1978.

[LLS68] Lauer P., Lucas P., Stigleitner H.,
Method and notation for the formal definition of programming languages. Technical Report TR 25.087, IBM Laboratory, Vienna 1968.

[Lon71] London R.L.,
Correctness of two compilers of a LISP subset. A.I. memo 151, Standford University, 1971.

[Lon72] London R.L.,
Correctness of a compiler of a LISP subset. Conference on proving assertions about programs, New Mexico state university, 1972.

[Lor74] Lorho B.,
De la définition à la traduction des langages de programmations. Thèse d'état, Toulouse 1974.

[Lor76] Lorho B.,
DELTA : manuel d'utilisation, Groupe Langages et Traducteurs, IRIA, 1977.

[McC61] Mac Carthy J.,
Towards a mathematical science of computation. Proceedings IFIP Congress 1962, p. 21-28.

[McC63] Mac Carthy J.,
A basis for a mathematical theory of computation. Computer programming and formal systems, Amsterdam 1963, p. 33-70.

[Mil77] Milne R., Verifying the correctness of implementations. Advanced course on semantics of programming languages, Antibes, 1977.

[MLB76] Marcotty M., Ledgard H.F., Bochmann G.V.,
A sampler of formal definitions. Computing Surveys 8, n° 2, 1976, p. 191-276.

[Mor73] Morris F.L.,
Advice on structuring compilers and proving them correct . P.O.P.L. 1973, Boston, p. 144-152.

[Mos75] Mosses P.D.,
Mathematical Semantics and Compiler Generation , Ph.D Thesis, Oxford Univ., 1975.

[Mos78] Mosses P.D.,
S.I.S. a compiler generator system using denotational semantics, Reference Manual, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1978.

[MP67] Mac Carthy J., Painter J.A.,
Correctness of a compiler for arithmetic expressions. Proceedings of a symposium in applied mathematics, Mathematical aspects of computer science n° 19, 1967, p. 33-41 .

[Mus77] Musser D.R.,
A data type verification system based on rewrite rules. 6th Texas conference on computing systems, Austin 1977, p. 22-31.

[MW72] Milner R., Weyrauch R.,
Proving Compiler Correctness in a Mechanized logic. Machine Intelligence 7, Edinburgh University Press, 1972, p. 51-70.

[NN66] Nivat M., Nolin N.,
Contribution to the definition of Algol semantics. IFIP working conference on formal language description languages, (T.B. Steel Jr Ed.), Baden 1966, p. 148-159.

[Pai75] Pair C.,
Formalization of the notions of data, information and information structure. Data base management systems (Klimbie-Koffman ed.), North-Holland, 1975, p. 149-167.

[PG79] Pair C., Gaudel M.C.,
Formal description of compilers, abstract data types and algebraic semantics of programming languages. WG2.2 meeting, Varsovie 1979.

[Rem74] Remy J.L.,
Structures d'information, formalisation des notions d'accès et de modification d'une donnée. Thèse de spécialité, Université de Nancy I, 1974.

[Sch67] Schoenfield J.R.,
Mathematical Logic. Addison-Welsey, 1967.

[SS71] Scott D., Strachey C.,
Towards a mathematical semantics for computer languages. Programming Research Group Monograph, PRG-6, Oxford 1971.

[SW75] Spitzen J., Wegbreit B.,
The verification and synthesis of data structures, Acta Informatica 4, 1975, p. 127-144.

[Ten76] Tennent R.D.,
The denotational semantics of programming languages. CACM 19, n° 8, 1976, pp. 437-453.

[TWW78] Thatcher J.W., Wagner E.G., Wright J.B.,
Data type specification : parametization and the power of specification techniques. 10th annual symposium on Theory of Computing, San Diego, 1978, p. 119-132.

[TWW79] Thatcher J.W., Wagner E.G., Wright J.B.,
More on advice on structuring compilers and proving them correct. R.C. 7588, IBM T.J. Watson Research Center, Yorktown Heights, Feb. 1979.

[WS76] Wegbreit B., Spitzen J.M.,
Proving properties of complex data stru-tures. J.A.C.M. 23, n° 2, 1976, p. 389-396.

[VW69] Van Wijngaarden A., Mailloux B.J., Peck J.E., Koster C.H.A.,
Report on the Algorithmic Language Algol 68. MR101, Mathematische Centrum, Amsterdam, 1969.

[VW76] Van Wijngaarden and al.
Revised Report on the Algorithmic Language Algol 68. Springer Verlag 1976.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique