Algèbre de programmes dans un univers type

Saddek Bensalem

THESE

présentée à

Institut National Polytechnique de Grenoble

pour obtenir le grade de

Docteur DE 3ème CYCLE
INFORMATIQUE

par

Saddek BENSALEM

ALGEBRE de PROGRAMMES dans un UNIVERS TYPE

Thèse soutenue le 20 décembre 1985 devant la commission d'Examen :

Monsieur	J. MOSSIERE	:	Président
Messieurs	D. BERT P. JACQUET Ph. JORRAND M. SINTZOFF		Examinateurs

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : Monsieur M. MERMET
Directeur des Etudes et de la formation : Monsieur J. LEVASSEUR
Directeur des recherches : Monsieur J. LEVY
Secrétaire Général : Mademoiselle M. CLERGUE

Professeurs de 1ère Catégorie

COINDE	Alexandre	Gestion
GOUX	Claude	Métallurgie
LEVY	Jacques	Métallurgie
LOWYS	Jean-Pierre	Physique
MATHON	Albert	Gestion
RIEU	Jean	Mécanique - Résistance des matériaux
SOUSTELLE	Michel	Chimie
FORMERY	Philippe	Mathématiques Appliquées

Professeurs de 2ème catégorie

HABIB	Michel	Informatique
PERRIN	Michel	Géologie
VERCHERY	Georges	Matériaux
TOUCHARD	Bernard	Physique Industrielle

Directeur de recherche

LESBATS	Pierre	Métallurgie

Maîtres de recherche

BISCONDI	Michel	Métallurgie
DAVOINE	Philippe	Géologie
FOURDEUX	Angeline	Métallurgie
KOBYLANSKI	André	Métallurgie
LALAUZE	René	Chimie
LANCELOT	Francis	Chimie
LE COZE	Jean	Métallurgie
THEVENOT	François	Chimie
TRAN MINH	Canh	Chimie

Personnalités habilitées à diriger des travaux de recherche

DRIVER	Julian	Métallurgie
GUILHOT	Bernard	Chimie
THOMAS	Gérard	Chimie

Professeur à l'UER de Sciences de Saint-Etienne

VERGNAUD	Jean-Maurice	Chimie des Matériaux & chimie industrielle

****

Au moment où je mets la dernière touche à cette thèse, je tiens à remercier :

Monsieur J. Mossière, Professeur à l'Institut Polytechnique de Grenoble, de m'avoir fait l'honneur de présider le jury de cette thèse,

Monsieur D. Bert, chargé de recherche au C.N.R.S, qui m'a éclairé de ses idées synthétiques tant pour l'approfondissement que pour la présentation de ce travail ; je lui dois de m'avoir donné goût à l'informatique fondamentale et je n'oublierai pas qu'il m'a consacré beaucoup de son temps,

Monsieur P. Jacquet, Maître assistant à l'ENSIMAG, qui n'a cessé de me prodiguer critiques, conseils, suggestions, encouragements et répliques dans des discussions passionnées et fort instructives,

Monsieur Ph. Jorrand, Directeur de recherche au C.N.R.S, de m'avoir accordé sa confiance et d'avoir accepté de juger ce travail,

Monsieur M. Sintzoff, Professeur à l'université catholique de Louvain, qui s'est montré intéressé par mes résultats ; je suis sensible à sa participation au jury.

je tiens à remercier particulièrement les deux groupes LPG et FP2 parmi lesquels : R. Echahed, P. Drabik, J.C. Reynaud, J. M. Hufflen, Ph. Schnoebelen, S. Rogé, Annick & J. C. Marty, M. B. Ibañez

TABLE des MATIERES

1. INTRODUCTION

On sait depuis longtemps que la notation fonctionnelle et spécialement les langages "applicatifs" (par opposition à "impératifs" ou algorithmiques), ont donné lieu à des résultats remarquables dans le domaine des **preuves de programmes** [Mc Carthy 63] [Burstall 69a] [Burstall 69b], comme dans celui de la **transformation de programmes** [Burstall 77] [Williams 82]. En effet, dans ces langages, une expression est équivalente à une formule mathématique, dans laquelle l'égalité syntaxique entraîne l'égalité sémantique puisqu'une même variable est associée à une même valeur dans toute l'expression ; cela n'est pas le cas dans les langages algorithmiques habituels, à cause de la présence implicite d'un "état" et des phénomènes de modifications cachées de cet état (le problème des "effets de bord"). En l'absence d'effets de bord, le langage peut être considéré comme un système formel dans lequel les termes sont des programmes, et les théorèmes sont des identités de termes [Turner 81].

Cependant, un certain nombre de limitations apparaissent, du fait que l'on ne dispose en général que d'un nombre fixe de constructions, et que les objets sont faiblement structurés. Backus lui-même souligne ce point [Backus 85, p. 81]. Une manière d'introduire une plus forte structuration est de typer les objets et les fonctions.

On sent la nécessité d'une **programmation fonctionnelle typée**, où les formes fonctionnelles ne sont pas déterminées a priori.

Certaines de ces formes fonctionnelles (opérateurs du 2e ordre) sont liées aux types et constituent des stratégies de "calcul" de ces types [Klaeren 84] [Norström 81]. Par exemple, dans les structures "libres", une méthode de calcul fondamentale est **l'homomorphisme** [Burstall 85]. Dans cette thèse, nous suivons l'approche du typage par **types abstraits** [Guttag 78] [Guttag 78] [Goguen 81]. Les valeurs d'un type (carrier) sont construites inductivement à l'aide de "constructeurs". Outre que la définition algébrique des types complète naturellement l'axiomatisation algébrique des programmes, il est donc possible de s'appuyer sur la structure des termes pour définir une algèbre de programmes plus riche que celle développée jusqu'à maintenant.

Les idées que nous proposons dans cette thèse sont issues d'un projet de langage de **spécification** et de **programmation générique** (LPG) [Bert 83b]. Nous nous sommes plus particulièrement intéressé à l'aspect "programmation au moyen de LPG". En LPG, on a la possibilité de créer des exemplaires (instances) des types génériques et des opérateurs génériques là où on en a besoin, sans passer par une déclaration intermédiaire. Sachant que le type "seq" est générique (type des séquences linéaires), on peut écrire dans un profil d'opérateur les types seq[entier], seq[bool], seq[seq[entier]], etc..., sans avoir à donner un nom à chacune des instances. Evidemment, deux occurrences de seq[entier] représentent le même type. De même **un opérateur générique** peut être instancié dans une expression, soit d'une manière implicite, soit d'une manière explicite. Un opérateur générique avec instance explicite est tout à fait l'équivalent d'un opérateur du second ordre, ou **forme fonctionnelle**. On verra comment il est facile de définir un opérateur "alpha" paramétré par un opérateur f : t1 -> t2, tel que alpha[f] : seq[t1] -> seq[t2] applique l'opérateur "f" à tous les éléments de la séquence d'entrée pour fournir la séquence résultat. Comme l'utilisateur peut définir de nouveaux types ou opérateurs, il peut également déclarer de nouvelles formes fonctionnelles sans aucune difficulté.

Dans la première partie de cette thèse, nous présentons le langage LPG, et les traits qui seront utiles pour la suite, en particulier le mécanisme de la **généricité** et de **l'instanciation**. Ensuite nous proposons une méthode de

définition et d'axiomatisation d'opérateurs génériques liés aux **structures libres** : séquences et arbres, puis aux **structures non libres** : vecteurs, ensembles, enfin des exemples de programmes avec ces opérateurs génériques.

Dans la seconde partie, nous définissons une extension de la syntaxe de LPG de façon à pouvoir manipuler des "**fonctions**" et non plus simplement des "**termes**" ; nous donnerons un jeu d'opérateurs et de combinateurs de bases avec leur axiomatisation, ainsi que des règles d'équivalence pour chaque structure. L'ensemble de ces règles constitue une **algèbre de programmes dans un univers typé**.

La troisième partie, est une application à la transformation et la preuve de programmes de l'approche algébrique typée qui a été développée dans la première partie. Nous donnerons quelques exemples classiques d'optimisation et de preuve de programmes fonctionnels avec notre formalisme, ainsi qu'une comparaison de notre approche avec les travaux réalisés dans le domaine des transformations et des preuves de programmes.

2. Méthodes de définition des opérateurs génériques.

2.1 La notion d'opérateurs.

L'une des plus importantes notions qui distingue les langages fonctionnels des langages algorithmiques habituels, est la notion d'opérateur. Par opérateur on entend un "objet" qui admet comme paramètre une ou plusieurs fonctions pour rendre une autre fonction. Cette notion a été introduite par Kenneth Iverson [Iverson 62] [Iverson 79] dans le langage APL. A la suite de cette introduction, les opérateurs sont devenus un concept très utile dans ce style de programmation. Depuis, plusieurs propositions d'opérateurs intéressants ont été faites, dans [Backus 78], [Burstall 80], [Chiarini 80], [Morris 80] et [Wadler 81]. Malheureusement, tous ces opérateurs sont pauvres, du point de vue de la structure algébrique, et des propriétés mathématiques qu'on peut leur associer, alors qu'on sait que les connaissances mathématiques jouent un rôle important dans le développement et le raisonnement sur les programmes. Les opérateurs constituent la base de la programmation fonctionnelle. Ces limitations sont dues en réalité aux langages supports.

2.2 Les limitations.

Dans ce paragraphe, notre but est d'illustrer les limitations

mentionnés ci-dessus, à travers deux langages qui nous semblent les plus importants : APL et FFP. Le premier parce qu'il a joué un rôle très important dans le développement de la programmation à l'aide d'opérateurs. Le second à cause des idées nouvelles qu'il a proposées.

1- Le langage APL contient un ensemble très riche d'opérateurs. Seulement leur utilisation est limitée, à cause des raisons suivantes :

- il n'est pas possible d'appliquer un opérateur à une fonction définie par l'utilisateur. Si on considère l'opérateur "reduction" dans APL, Il prend comme argument une fonction, par exemple la fonction "plus", pour donner la fonction "somme". Cet opérateur, en réalité doit connaître une propriété sur la fonction, qui est l'existence d'un élément neutre. il n'y a encore aucun moyen de spécifier un tel élément pour les fonctions définies par l'utilisateur dans APL.

- on ne peut pas définir ses propres opérateurs car dans APL, les fonctions ne peuvent pas prendre comme argument des fonctions.

- on n'a aucun moyen de typer le résultat des fonctions définies par l'utlisateur. La notion de type n'existe pas dans APL.

2- Dans FFP, J. Backus a tenté de résoudre certains des points soulignés ci-dessus, comme suit :

- toute fonction peut être appliquée à n'importe quel élément du domaine des objets.

- on peut appliquer les formes fonctionnelles (ou opérateurs au sens de Iverson) prédéfinies à n'importe quelle fonction, en représentant les fonctions comme objets du même domaine, et les formes fonctionnelles comme des fonctions qui opèrent sur la représentation d'autres. Avec cette solution, on laisse un aspect important non résolu : le fait qu'on ne peut pas associer des lois algébriques entre les formes fonctionnelles définies par l'utilisateur.

- il en est de même pour spécifier des propriétés sur les fonctions définies par l'utilisateur. Si l'on considère la forme fonctionnelle "insert" qui est équivalente à l'opérateur de réduction dans APL, elle utilise un élément neutre à droite (pour l'insertion à droite), mais il n'y a aucune facilité dans le langage, qui permet de définir un tel élément. En fait ce n'est pas formellement spécifié.

2.3 Une solution.

Une approche qui nous semble intéressante, et qui permet de résoudre les problèmes évoqués ci-dessus, est d'envisager une fusion possible entre les deux notions suivantes :

- les types abstraits,
- les opérateurs ou formes fonctionnelles,

de façon à ce que les opérateurs, puissent être définis dans le contexte de la structure algébrique à laquelle ils appartiennent. On peut ainsi leur associer des propriétés mathématiques. Un tel developpement n'est possible, que si le langage support admet les "types génériques", et la "généricité structurale" [Jacquet 78](par opposition à la généricité incrémentale qui est celle des langages traditionnels, où par exemple l'opérateur "+" peut s'appliquer en même temps aux entiers, réels, etc...), autrement dit un langage de programmation générique [Bert 79]. Un tel langage existe, c'est LPG [Bert 83b].

2.4 Présentation de LPG

2.4.1 Présentations. algèbres.

LPG [Bert 83b] est un langage applicatif fondé sur le formalisme des spécifications algébriques des types abstraits. La notion syntaxique de base est la présentation.

Une présentation est un triplet (S,Ω,E), où S est un ensemble de "sorte s" (ou nom de types), Ω un ensemble de noms de fonctions caractérisés par un profil, c'est-à-dire leur domaine et leur codomaine, et E est un ensemble d'équations ou axiomes sur les opérateurs. On note $\Sigma = (S,\Omega)$ la signature de

la présentation. Avec les constructions habituelles [Goguen 78] [Ehrich 82], on peut définir $T(\Sigma,X)_s$ qui est l'ensemble des termes de sorte s ($s \in S$), engendrés par un ensemble de variables typées, noté X. L'algèbre des termes est appelée T_Σ, et l'algèbre des termes qui "satisfait" les équations E est $T_{\Sigma,E} = T_\Sigma/\equiv_E$, où $\equiv_E$ est la plus petite relation de congruence qui contient E. D'une manière générale, les modèles d'une présentation (S,Ω,E) sont des algèbres hétérogènes (A,F) telles que :

- $A = \{ A_s / s \in S \}$ est une famille d'ensembles.
- F est un ensemble d'applications telles que :

$$\forall\ \omega \in \Omega, \omega : (s_1, \ldots, s_n) \rightarrow s$$

on a $f_\omega \in F$ tel que :

$$f_\omega : (A_{s1}, \ldots, A_{sn}) \rightarrow A_s$$

Note : $(A_1, \ldots, A_n)$ est mis pour le produit cartésien $A_1 \times \ldots \times A_n$

- $\forall\ e \in E$ on a (A,F) satisfait e, noté $(A,F) \models e$

Autrement dit : les équations sont interprétées par des propositions vraies dans tous les modèles.

2.4.2 Unités de définition en LPG.

En LPG, on peut définir des <u>présentations de type</u> : $\tau = (\Sigma,E)$ dont la sémantique est donnée par l'algèbre $T_{\Sigma,E}$. Un exemple de présentation de type est celle des entiers naturels.

Exemple1:

```
type Entier sortes entier          --entier est la sorte du type Entier.
constructeurs                      -- définition des constructeurs.
    succ : entier -> entier
    zero : -> entier
operateurs
    + : (entier,entier) -> entier
variables
```

-- ensemble de variables quantifiées
-- universellement.

n,m : entier

axiomes

1: zero + n ==> n

2 : succ(n) + m ==> succ(n+m)

fin

Dans cette notation "constructeurs" est un mot clé qui indique quels sont les opérateurs constructeurs des valeurs du type [Guttag 78], [Huet 80b], [Goguen 80].

Notons au passage que les équations notées par "==>" sont les équations "exécutables", c'est-à-dire que l'interpréteur pourra les utiliser pour l'évaluation symbolique des expressions; celles notées par "==" sont des équations non-exécutables. Toutes les équations sont vues néanmoins de la même façon du point de vue de l'axiomatisation et des preuves.

Exemple2 :

type Bool sortes bool

constructeurs

vrai, faux : -> bool

operateurs

non : bool -> bool

et, ou : (bool,bool) -> bool

variables

b,c : bool

axiomes

1: non (vrai) ==> faux

2 : non (faux) ==> vrai

3 : b et vrai ==> b

4 : b et faux ==> faux

5 : b ou c ==> non (non (b) et non (c))

fin

On dispose de présentations d'enrichissement qui permettent de définir de nouveaux opérateurs sans introduire de nouvelles sortes. Comme exemple, nous déclarons un enrichissement de bool et de entier, avec l'opérateur = : (entier,entier) -> bool.
En LPG 1.8, on écrit :

Exemple3 :

```
enrich Egalite_des_entiers      -- nom de l'enrichissement.
operateurs
    = : (entier,entier) -> bool
variables
    n,m : entier
axiomes
    1 : zero = zero ==> vrai
    2 : succ(n) = zero ==> faux
    3 : zero = succ(m) ==> faux
    4 : succ(n) = succ(m) ==> n = m
fin
```

Le troisième concept introduit en LPG est celui des présentations de propriétés. Une propriété p = (Σ,E) a comme sémantique l'ensemble de ses modèles, la différence avec la sémantique d'un type c'est qu'on s'intéresse pas simplement à l'algèbre initiale. Intuitivement, une propriété caractérise des types et des opérateurs (les algèbres effectives) qui satisfont les équations de la propriété. Voici, comme exemple les propriétés de la structure de monoïde et celle de l'égalité (qui utilise la sorte bool) :

Exemple4 :

```
prop Monoide sortes t     -- t est une sorte."formelle".
operateurs
    e : -> t              -- e et plus sont des opérateurs
    plus : (t,t) -> t
variables
    x,y,z : t
```

axiomes

1 : plus(e,x) == x

2 : plus(x,e) == x

3 : plus(x,plus(y,z)) == plus(plus(x,y),z)

fin

prop Egalite **sortes** t

operateurs

= : (t,t) -> bool

variables

x,y,z : t

axiomes

1 : x = x == vrai

2 : x = y == y = x

3 : x = y et y = z et non(x = z) == faux

fin

Dans les types et les enrichissements, on peut déclarer des modèles de propriétés par la clause "modeles". La clause :

modeles

nom _du _modele : p_1[T_1, ...,T_n operateurs F_1, ..., F_k]

déclare un modèle de p_1, et signifie que les sortes et les opérateurs entre crochets vérifient les équations de la propriété p_1 (cf. exemple du type "seq" dans § 2.4.3).

De plus, on peut établir des morphismes [Burstall 84] (cf. note p.17) entre les propriétés par les clauses "satisfait", "herite", et "combine", de telle sorte que des modèles peuvent être déduits automatiquement par le système [Bert 83a] [Bert 82b]. La clause "satisfait p_1 [T_1, ..., T_n operateurs F_1, ..., F_k]", dans une propriété p_2, indique que l'on peut construire un modèle de p_1 à partir de tout modèle de p_2. Pour cela il faut (et il suffit) que les équations de p_1 (après, substitution par les sortes et les opérateurs effectifs soient vraies dans la théorie présentée par p_2. De même dans p_2, la clause :

"herite p_1 [T_1, ..., T_n operateurs F_1, ..., F_k]"

déclare un morphisme entre p_1 et p_2 et signifie que les équations de p_1 doivent être ajoutés (après substitution par les sortes et opérateurs effectifs), à la définition de p_2.

Exemple5 :

```
-- Tyfo est une propriété avec une sorte, dont les modèles
-- sont des ensembles sans aucune fonction.
prop Tyfo sortes t1
fin

-- "Egalite" décrit les ensembles avec un opérateur d'égalité
prop Egalite sortes t2
operateurs
    = : (t2,t2) -> bool
axiomes
...                          -- équations de l'égalité
satisfait tyfo[t2]           -- inclusion de propriétés
fin

-- Ordre_total décrit les ensembles ordonnés.
-- l'inclusion des propriétés indique qu'un modèle de l'ordre total contient un -- modèle de l'égalité.
prop Ordre_total sortes t3
operateurs
    <=,= : (t3,t3) -> bool
variables
    x,y,z : t
axiomes
    1 : x <= x == vrai
    2 : x <= y et y <= x == x = y
    3 : x <= y et y <= z et non(x <= z) == faux
    4 : x <= y ou y <= x == vrai
satisfait Egalite [t3 operateurs =]
fin
```

De ces trois déclarations, on déduit les morphismes :

Tyfo Egalite Ordre_total

L'information associée aux flèches doit indiquer évidemment la correspondance des sortes et des opérateurs.

Note : D'une façon intuitive et sans rappeler la théorie que l'on pourra trouver dans [Burstall 84] [Burstall 79], un morphisme entre deux propriétés p = (Σ,E) et p' = (Σ',E') est :

(1) un morphisme de signature f : Σ -> Σ' (qui fait correspondre les sortes et les opérateurs entre eux).

(2) les équations p sont des théorèmes dans p', autrement dit f(e) est valide dans E', $\forall e \in E$.

2.4.3 Généricité, instanciation :

Une unité est générique, lorsqu'elle est paramétrée par une propriété appellée propriété éxigée. Les sortes et opérateurs de cette propriété sont les sortes et opérateurs formels de l'unité. La paramétrisation la plus répandue est le paramétrisation par un type. La propriété qui décrit tous les types est "Tyfo".

Le type générique des séquences linéaires exige Tyfo pour le type de ses éléments :

```
type Seq exige Tyfo[t]
sortes seq
constructeurs
    nil : -> seq[t]
    <+ : (t,seq[t]) -> seq[t]
operateurs
    vide? : seq[t] -> bool
    tete : seq[t] -> t
    reste : seq[t] -> seq[t]
axiomes                    -- axiomatisation des opérateurs
```

modeles
    -: tyfo[seq[t]] -- "-" indique un modèle anonyme.
fin

Après cette déclaration, on peut utiliser directement des exemplaires de la sorte "seq" comme seq[bool], seq[entier]. La rubrique modeles indique que les exemplaires de "seq" appartiennent à la classe caractérisée par "Tyfo" et donc peuvent être des paramètres de seq. Cela permet de construire des expressions de sorte comme "seq[seq[entier]]".

Les opérateurs d'une unité générique sont eux aussi génériques. Prenons un exemple simple : soit la propriété d'ordre total définie ci-dessus . A l'aide de cette propriété, il est possible de définir un ou plusieurs opérateurs de tri sur les séquences.
L'entête de l'unité générique doit être :

enrich Op_de_tri **exige** Ordre_total [t **operateurs** <=,=]
operateurs
    trier : seq[t] -> seq[t]
...
fin

Un exemplaire de l'opérateur "trier" est paramétré non seulement par la sorte (qui est la sorte des éléments de la séquence donnée en paramètre), mais également par les opérateurs "<=,=" de la propriété exigée. Lors de chaque occurrence de trier, la sorte et les opérateurs formels doivent être complétement determinés. Cela peut être fait de deux manières.

Comme autre exemple de type générique, il est intéressant de donner les définitions des types prédéfinis : ensemble et vecteur, d'où on a les définitions suivantes :

type Vecteur **exige** tyfo[t]
sortes vecteur
constructeurs

```
    vect : entier -> vecteur[t]
    rg : (vecteur[t],entier,t) -> vecteur[t]
operateurs
    val : (vecteur[t],entier) -> t
    # : vecteur[t] -> entier -- taille du vecteur en paramètre
variables
    v : vecteur[t], i,j : entier, x,y : t
axiomes                          -- équation sur les constructeurs
    1 : rg(rg(v,i,x),j,y) ==> si i = j alors rg(v,j,y)
                              sinon rg(rg(v,j,y),i,x) fsi
                              -- axiomes de val et # omis
modeles
    formel_vect : Tyfo[vecteur[t]]
fin


type Ensemble exige Egalite [t operateurs =]
sortes ens
constructeurs
    e_vide : -> ens[t]
    <+ : (t,ens[t]) -> ens[t]
operateurs
    # : ens[t] -> entier          -- cardinalité
    @ : (t,ens[t]) -> bool        -- appartenance
    +, &,- : (ens[t],ens[t]) -> ens[t]
                              -- union, intersection et différence d'ensembles
variables
    x,y : t,
    s : ens[t]
axiomes                       -- équations sur les constructeurs
    1 : x <+ (x <+ s) == x <+ s
    2 : x<+(y<+s) == y <+( x <+ s)
                              -- axiomatisation de #, @, +, &, - omises
modeles
    f_ens : Tyfo[ens[t]]
```

fin

2.4.3.1 Instanciation explicite.

Les opérateurs sont données directement à l'emplacement de l'appel :

trier[<=,=](s)

où s est seq[entier] et <= et = les opérateurs d'ordre et d'égalité de type (entier,entier) -> bool. Une autre occurrence pourrait être :

trier[>=,=](s)

2.4.3.2 Instanciation implicite.

Les opérateurs ne sont pas donnés à l'appel mais dans la clause "modeles", qui déclare un ordre total pour le type des entiers par exemple . On a dans une unité :

modeles

croissant : ordre_total [entier **operateurs** <=,=]

Un appel:

trier(s)

où 's est seq[entier], est automatiquement considéré comme paramétré par les opérateurs "<=,=". S'il y a zero ou plus d'une liaisons possibles lors d'une instanciation implicite, l'appel n'est pas determiné, et un message apparaît à la compilation.

Remarques :

1- Si le type des paramètres d'un opérateur ne suffit pas à déterminer les types effectifs d'un modèle de propriété exigée, on doit qualifier l'instance de cet opérateur par le

type du résultat. C'est le cas en particulier de toutes les constantes génériques.

Exemple :

nil : seq[entier]

2- Les types des paramètres et du résultat d'un opérateur doivent contenir tous les types formels de la propriété exigée.

Exemple :

enrich Exemple **exige** toto [t1,t2 **operateurs** f1,f2]
operateurs
g : seq[t1] -> t1
...
fin

L'opérateur g ne pourra jamais être appelé correctement à l'extérieur de l'unité "Exemple", parce que le type t2 ne figure pas dans son profil, et ne pourra pas être déduit des paramètres de l'appel.

2.4.4 Définition des expressions.

Une expression LPG peut être :

- Une constante ou littéral prédéfini
  3, "abc", ..., [1,2,3], {'a','b','c'}
- Une variable;
- Un opérateur, éventuellement instancié avec des paramètres
  3+x, trier[<=,=](s)

- Un produit cartésien
  $(expr_1, ...,expr_n)$
- Une expression conditionnelle
  si $expr_1$ **alors** $expr_2$ **sinon** $expr_3$ **fsi**
- Une expression de liaison
  soit v1 = expr1, v2 = expr2, ...

dans expr
fsoit

Parmi les opérateurs prédéfinis, on peut citer en particulier l'opérateur "projection" de produit cartésien, noté

!1(x) : première composante,
!2(x) : deuxième composante,
...

et l'opérateur générique de "curryfication", $ défini par :

$[f,x](y) == f(x,y)

Il faut noter enfin que les paramètres formels d'un opérateur générique décrits dans la propriété exigée de cet opérateur, sont des opérateurs ou des constantes (opérateurs sans paramètres ou zero-aires). Les paramètres effectifs sont des opérateurs qui doivent avoir un profil compatible avec celui des paramètres formels. Les paramètres constants peuvent avoir comme correspondants soit des opérateurs constants soit des littéraux prédéfinis, soit des variables. Cette dernière possibilité, jointe à l'opérateur "$", est à la base de l'utilisation intensive des opérateurs génériques.

La sémantique d'un opérateur, sous forme d'équations exécutables, peut être donnée par cas sur les constructeurs [Guttag 78], comme dans l'enrichissement Egalite_des_entiers, ou bien directement par une formule, comme la définition de "ou" dans le type Bool.

2.5 Théories, méthodes de preuves.

2.5.1 Théories associées aux présentations.

Etant donné une famille d'algèbre {(A,F)}, on note th({A,F}) la théorie associée, c'est-à-dire l'ensemble des équations qui sont vraies dans toutes les algèbres. Pour une présentation de propriété p = (Σ,E), la théorie présentée par p est th(mod(p)) où mod est l'ensemble des modèles de p. Cette théorie est habituellement appelée la théorie équationnelle de p. Le système logique qui permet

d'engendrer th(mod(p)) à partir de E est défini par les règles suivantes (|- est le symbole d'infèrence $t, t_1, t_2, \ldots, t_i, t'_i \in T(\Sigma,X)$ et t[t'/x] pour parler du terme t dans lequel la variable x est remplacée par le sous terme t') : sous la liste d'hypothèses h notée comme suit ⌊ h ⌋. h peut être vide et dans ce cas, on ne le notera pas.

(1) |- e ($\forall e \in E$)

(2) |- $t == t$

(3) $t_1 == t_2$ |- $t_2 == t_1$

(4) $t_1 == t_2, t_2 == t_3$ |- $t_1 == t_3$

(5) $t_1 == t_2$ |- $t_1[t_3/x] == t_2[t_3/x]$

($x \in X$ et x est une variable libre dans t_1 et t_2)

(6) $t_1 == t'_1, \ldots, t_n == t'_n$ et $\omega\ (s_1, \ldots, s_n) \rightarrow s \in \Omega$

|- $\omega\ (t_1, \ldots, t_n) == \omega\ (t'_1, \ldots, t'_n)$

Il a été démontré [Birkhoff 35], que toute formule valide dans tous les modèles est un théorème de la théorie équationnelle, autrement dit :

$\forall\ t_1 == t_2,\ p$ |- $t_1 == t_2 <=>\ t_1 == t_2 \in$ th(mod(p))

Les règles conditionnelles sont des cas particulier de l'implication logique. Pour manier les implications, nous utiliserons la notation :

⌊$e_1, e_2, \ldots, e_n$⌋ e

qui signifie $e_1 \wedge e_2 \wedge \ldots \wedge e_n \Rightarrow e$

où e, e_i sont des équations.

Un modèle (A,F) satisfait une équation conditionnelle :

(A,F) |= ⌊$e_1, e_2, \ldots, e_n$⌋ e

ssi (A,F) |= e dès que ($\forall$ i) $1 \leq i \leq n$ (A,F) |= e_i

Voici un certain nombre de règles applicables aux axiomes conditionnels :

(7) e_1, ⌊ e_1 ⌋ e_2 |- e_2 (modus ponens)

(8) t_1== si t alors t_2 sinon t_3 fsi

|- ⌊ t == vrai ⌋ $t_1 == t_2$, ⌊ t == faux ⌋ $t_1 == t_3$ (règle du si)

(9) (e_1 |- e_2) |- ⌊ e_1 ⌋ e_2 (déduction sous hypothèse)

La théorie associée à une présentation de type $\tau = (\Sigma,E)$ est donnée par $th(T_{\Sigma,E})$. On l'appellera la théorie inductive de τ, on a $th(mod(\Sigma,E)) \subseteq th(T_{\Sigma,E})$, mais $th(T_{\Sigma,E})$ contient également beaucoup d'autres théorèmes. Il n'existe pas de système logique complet qui permette d'atteindre $th(T_{\Sigma,E})$ à partir de E. Néanmoins, un certain nombre de théorèmes peuvent être obtenus par des règles d'induction liées à la construction des termes du type. Ce sont les règles d'induction sur les constructeurs [Burstall 69b] [Goguen 80] [Huet 80b]. Pour le type entier par exemple, les constructeurs étant {zero,succ}, on peut formuler une règle d'induction par rapport à une variable n du type entier qui apparaît dans une équation EQUA de la forme $t_1 == t_2$

EQUA [zero/n]

$$\vdash (\forall\ n)\ EQUA$$

⌊EQUA[n_0/n]⌋ EQUA[succ(n_0)/n]

Une autre façon d'obtenir des théorèmes, au lieu d'utiliser la règle d'induction, c'est de faire une preuve par consistence. A ce sujet, les premières propositions ont été faites par [Musser 80], redéfini par [Kapur 80], [Goguen 80], [Huet 80a], [Huet 80b], [Lankford 80], [Dershowitz 83], et puis par [Kapur 84] où ce dernier propose un théorème analogue à celui de Birkhoff qui concerne la complétude de la preuve inductive, c'est ce qu'ils appellent le théorème de complétude inductive.

2.5.2 Méthodes de preuves.

En LPG, lorsqu'on déclare qu'une algèbre est un modèle d'une propriété, ou qu'une propriété contient une autre propriété à l'aide de la clause "satisfait", on doit vérifier une condition sémantique qui revient à montrer que certaines équations appartiennent à une certaine théorie.
Par exemple on a déclaré que la présentation Ordre_total satisfait l'Egalité, il faut donc montrer que les équations :

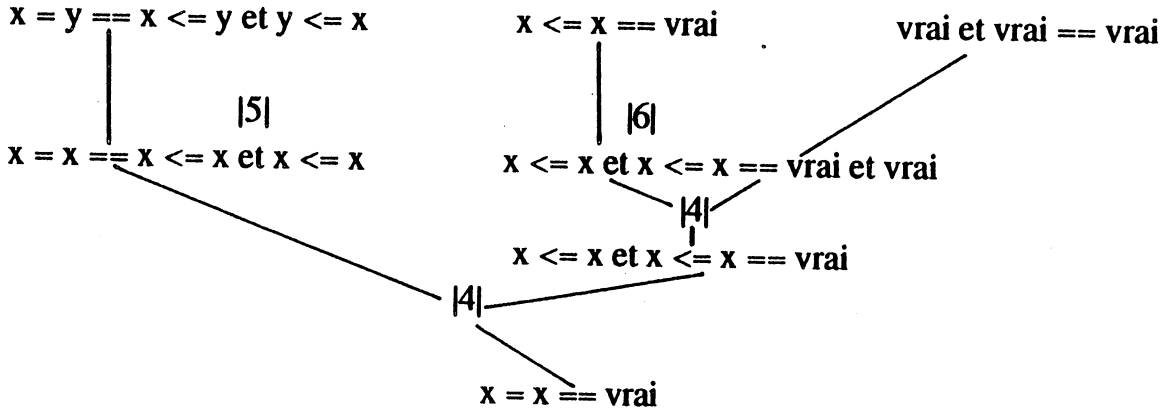
(1) x = x == vrai

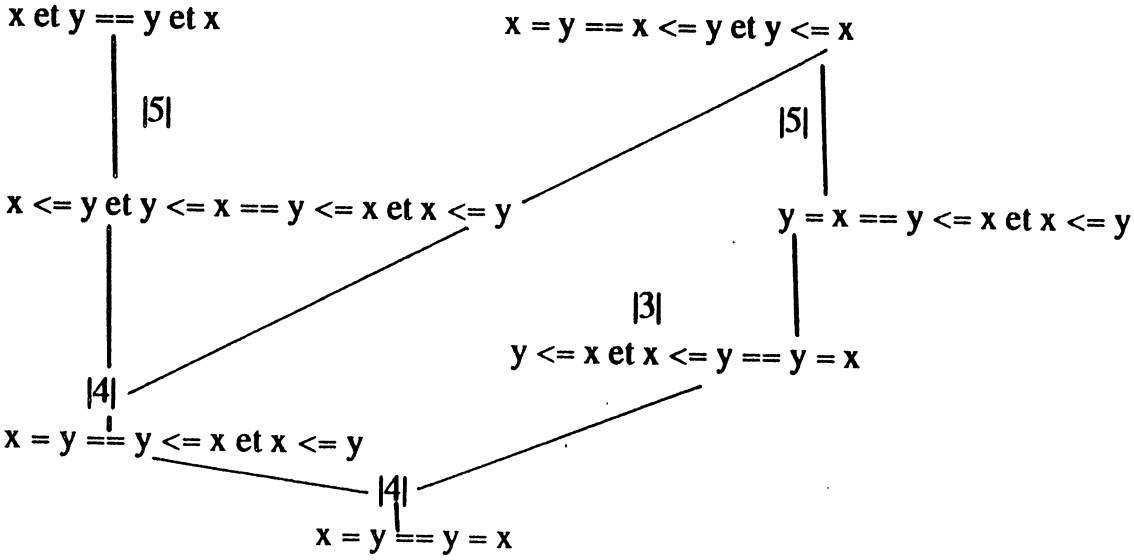
(2) x = y == y =x
(3) x = y et y = z et non(x = z) == faux

appartiennent bien à la théorie de l'Ordre_total. On suppose que l'axiomatisation des booléens comprend les équations :

x et y == y et x
vrai et vrai == vrai

La démonstration de (1) et (2) par exemple, revient à construire des arbres d'inférence (on note |j|, l'utilisation de l'inférence numéro j).



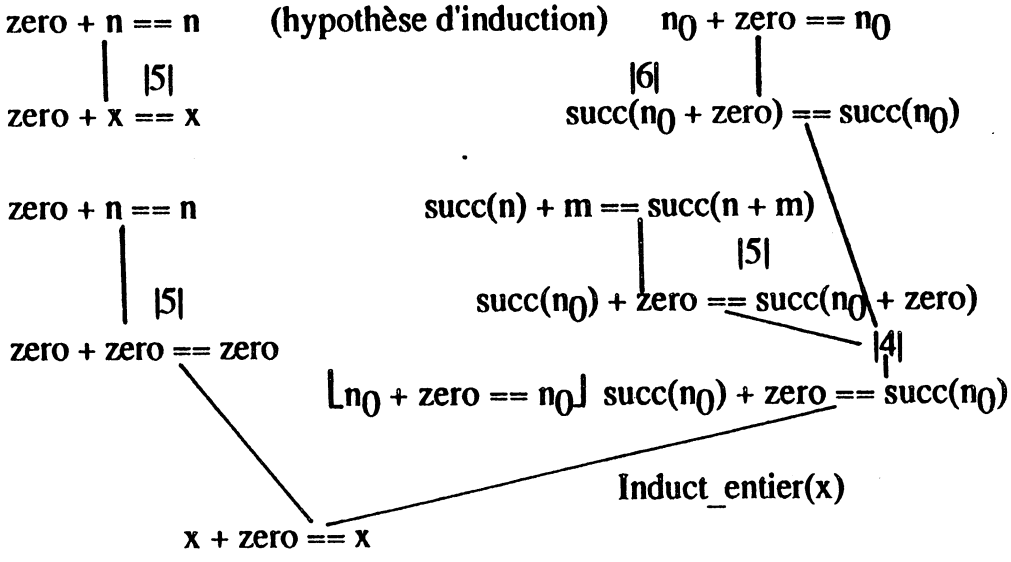


De même, la déclaration "entier est un modèle de monoïde" demande la preuve que les équations :

(4) zero + x == x
(5) x + zero == x
(6) x + (y + z) == (x + y) + z

appartiennent à la théorie de Entier.

Les équations (5) et (6) ne peuvent être démontrées que par induction.
Donnons par exemple les arbres d'inférence de (4) et (5)



Plutôt que de faire une preuve en utilisant un arbre d'inférence, il est plus facile de partir du but à atteindre, et utilisant les axiomes de la présentation, et montrer que ce but est valide en le décomposant en sous-buts.
Une méthode de décomposition d'un but dans une théorie est appelée une tactique ; un enchaînement de tactiques est une stratégie [Gordon 79] [Schmidt 83] [Gordon 82].

Dans le cas des théorèmes équationnels, une tactique pour

démontrer $t_1 == t_2$ est de réécrire t_1 et t_2 suivant une certaine relation de réécriture "->" de telle sorte que :

$(\exists\, t_3)$ $t_1 \overset{*}{\rightarrow} t_3$ et $t_2 \overset{*}{\rightarrow} t_3$

avec $\overset{*}{\rightarrow}$ la fermeture réflexive transitive de -> .

La relation de réécriture que l'on considère est :

$t_1 \rightarrow t_2 \iff (\exists\, t_3, t_4, t_5)\ t_1 = t_3\,[t_4/x]\ t_2 = t_3\,[t_5/x]$

où x est libre dans t_3

et $\exists\ a_1 = a_2 \in E$

$\exists\ \sigma$: substitution aux variables, telle que

$(\sigma(a_1) = t_4$ et $\sigma(a_2) = t_5)$

ou $(\sigma(a_2) = t_4$ et $\sigma(a_1) = t_5)$

Pour les axiomes conditionnels, une tactique possible est celle de la décomposition par cas :

t_1 == si t alors t_2 sinon t_3 fsi

est un théorème, si

$\lfloor t == \text{vrai} \rfloor\ t_1 == t_2$

et $\lfloor t == \text{faux} \rfloor\ t_1 == t_3$

sont des théorèmes.

La troisième tactique utilisée sera celle de l'induction. Pour les entiers naturels, la tactique associée à la règle donnée précédement est :

$(\forall\, n : \text{entier})$ $t_1 == t_2$

est un théorème, si

$t_1[\text{zero}/n] == t_2[\text{zero}/n]$

et $\lfloor t_1[n_0/n] == t_2[n_0/n] \rfloor\ t_1[\text{succ}(n_0)/n] == t_2[\text{succ}(n_0)/n]$

sont des théorèmes.

Enfin, on doit maintenant généraliser les tactiques aux preuves de théorèmes de la forme $\lfloor e_1, \ldots, e_n \rfloor$ e. C'est la tactique de la preuve sous conditions :

$\lfloor e_1, \ldots, e_n \rfloor$ e est un théorème

si e est un théorème de la théorie considérée à laquelle on ajoute $e_1, \ldots, e_n$ comme

axiomes supplémentaires, dans lesquels les variables sont liées.

Voici un exemple de l'utilisation de ces tactiques pour la preuve de l'associativité du "+" (6) :

but : (6) $x + (y + z) == (x + y) + z$
Induction sur x
sous-buts : (6-1) $zero + (y + z) == (zero + y) + z$
(6-2) $\lfloor x_0+(y+z) == (x_0+y)+z \rfloor\ succ(x_0) + (y+ z) == (succ(x_0)+y) +z$
réécriture de 6-1 :
$zero + (y + z)$ (membre gauche)

(entier,1,->) ==> $y + z$ (-> indique le sens de l'utilisation de l'axiome entier 1)
$(zero + y) + z$ (membre droit)
(entier,1,->) ==> $y + z$
6-1 **démontré** (car les deux membres se réécrivent dans le même terme.)

but:6-2 $\lfloor x_0+(y+z)==(x_0+y)+z \rfloor succ(x_0)+(y+z) == (succ(x_0)+y)+z$
réécriture du conséquent
$succ(x_0) + (y + z)$
(entier,2,->) ==> $succ(x_0 + (y + z))$
(hyp, 6-2,->) ==> $succ(x_0 + y) + z$
(entier,2,<-) ==> $(succ(x_0) + y) + z$
6-2 **démontré** (le membre gauche se réécrit dans le membre droit)

Il est facile, à partir de la trace d'une démonstration par sous-buts de reconstruire l'arbre d'inférence de la preuve.

2.6 Définition des opérateurs génériques.

Dans cette partie, nous allons définir un ensemble d'opérateurs génériques puissants pour les types prédéfinis tels que séquence, vecteur et ensemble. De cette manière chaque structure de donnée aura son propre ensemble d'opérateurs. La plupart de ces opérateurs peuvent être vus comme des itérateurs, ou

des "formes fonctionnelles" selon la terminologie de [Backus 78]. La partie générique ou formelle de ces opérateurs est décrite par une propriété. La méthode consiste à définir la propriété qui caractérise les parties formelles, puis à décrire l'opérateur dans un enrichissement qui exige cette propriété.

2.6.1 Opérateurs génériques sur les séquences

- La fonction "iota" [Iverson 79], permet de construire une séquence (cf. la définition des séquences §2.4.3) d'entiers de 1 à n, où n est le paramètre de cette fonction. L'enrichissement "operations" déclare "iota" telle que, par exemple :

iota(3) ==> [1,2,3]

```
enrich operations
operateurs
    iota : entier -> seq[entier]
    prive it_iota : (entier,entier) -> seq[entier]
variables
    x, n : entier
axiomes
    1 : iota(x) ==> it_iota(x,0)
    2 : it_iota(x,n) ==> si x = n alors [:entier]
                         sinon (n+1) <+ it_iota(x,n+1)
                         fsi
fin
```

- "alpha" applique l'opérateur passé en paramètre générique à tous les éléments de la séquence en paramètre, pour générer une nouvelle séquence.

exemple :

alpha[succ]([1,2,3]) : seq[entier] ==> [2,3,4]

alpha[iota] ([1,2,3]) : seq[seq[entier] ==> [[1], [1,2], [1,2,3]]

L'opérateur générique "alpha" est similaire à la fonction "mapcar" dans LISP, et "forall" dans FP. Il est implicite dans certains opérateurs APL.

```
prop op_formel sortes t1 t2
operateurs
    f : t1 -> t2
satisfait Tyfo[t1], Tyfo[t2]
fin

enrich transformation_seq
exige op_formel [t1,t2 operateurs f]
operateurs
    alpha : seq[t1] -> seq[t2]
variables
    a : t1
    s : seq[t1]
axiomes
    1 : alpha(nil : seq[t1]) ==> nil : seq[t2]
    2 : alpha(a<+s) ==> f(a)<+alpha(s)
fin
```

-"init" est l'opérateur qui construit une séquence dont les éléments sont calculés d'après la fonction passée en paramètre générique :

```
    init[f](n) ==> [f(1), f(2), ..., f(n)]
```

Exemple :

```
    init[id](3) ==> [1,2,3]
```

```
prop init_formel sortes t
operateurs
    f : entier ->t
satisfait tyfo[t]
fin

enrich constuire_seq exige init_formel [t operateur f]
operateurs
```

```
    init : entier -> seq[t]
    prive init_iter : (entier,entier) -> seq[t]
variables
    n,i,j : entier
axiomes
    1 : init(n) ==> init_iter(0,n)
    2 : init_iter(i,n) ==> si i=n alors nil : seq[t]
                           sinon soit j=succ(i)
                                 dans f(j)<+init_iter(j,n)
                                 fsoit
              fsi
fin
```

- "hom" réalise un homomorphisme [Burstall 69b] [Von henke 75] [Burstall 85], qui va des séquences dans un type quelconque.

Exemples :

```
    hom[nil,<+]([1,2,3]) :seq[entier]==> [1,2,3]      (identité)
    si f : (t,entier) -> entier     telle que :
    f(a,n) ==> succ(n), on a
    hom[0,f]([1,2,3]) :entier == 3                    (longueur)
```

-"/" est un cas particulier d'homomorphisme où le type du résultat est le même que celui des éléments de la séquence. C'est l'opérateur "reduce" dans APL, et le "insert" dans FP.

Exemple :

```
    /[0,+]([1,2,3]) ==> 6
```

Note : "hom" et "/" associe les éléments à droite, c'est à dire l'expression ci-dessus est évaluée comme suit :

```
    (1+(2+(3+0)))
```

```
prop Image sortes t1 dom
operateurs
    n : -> dom
    f : (t1,dom) -> dom
satisfait Tyfo[t1], Tyfo[dom]
fin

enrich hom_sur_seq
exige Image [t2, dom2 operateurs nc, cop]
operateurs
    hom : seq[t2] -> dom2
variables
    a : t2
    s : seq[t2]
axiomes
    1 : hom([ : t2]) ==> nc
    2 : hom(a.s) ==> cop(a,hom(s))
fin

enrich reduction_seq
exige Monoide [t operateurs zero, plus]
operateurs
    / : seq[t] -> t
variables
    s : seq[t]
axiomes
    1 : /(s) == hom[zero,plus](s) : t
fin
```

-"hom_inv" est un opérateur générique qui utilise deux fonctions. La première est une fonction booléenne, et la seconde est une fonction qui génére une séquence à partir d'une valeur donnée en paramètre. La fonction booléenne retourne la valeur "vrai" pour indiquer la fin de la séquence. Dans le cas où elle retourne "faux", la seconde fonction fournit un couple de valeurs : la première valeur est la tête de la séquence, la seconde est utilisée pour générer le reste de la séquence.

Exemple :

hom_inv[zero?,f](5) ==> [5,4,3,2,1]

où zero?(x) ==> x = 0

et f(x) ==> (x,pred(x))

```
prop Image_inv sortes t dom
operateurs
    p : dom -> bool
    f : dom -> (t,dom)
satisfait Tyfo[t], Tyfo[dom]
fin

enrich homomorphisme_inverse  exige Image_inv [t, dom operateurs p, f]
operateurs
    hom_inv : dom -> seq[t]
variables
    x : dom
    y : (t,dom)
axiomes
1 : hom_inv(x) ==> si p(x) alors [:t]
                   sinon  soit y = f(x)
                          dans !1(y)<+hom_inv(!2(y))
                          fsoit
                    fsi
fin
```

On remarque que la propriété "image" décrit la catégorie des "structures" séquences, où les opérateurs sont des images des constructeurs du type Seq, alors que dans la catégorie "image_inv", les opérateurs sont des images des sélecteurs.

2.6.2 Opérateurs génériques sur les vecteurs.

Le type des vecteurs (tableau à une dimension, cf. la

définition des vecteurs § 2.4.3) n'est pas défini comme une structure libre à partir des constructeurs "vect" et "rg" [Bert 83b]. On simule dans la spécification algébrique le fait qu'un vecteur est une fonction d'un sous ensemble des entiers dans le type des éléments. De ce fait, il est possible de proposer plusieurs jeux d'opérateurs génériques "intéressants" sur les vecteurs, chaque opérateur représentant une certaine stratégie de calcul. Les opérateurs ayant un vecteur en paramètre décrivent des itérations sur les éléments du vecteur. Il est souvent utile, lorsqu'on itère sur un vecteur, de choisir le sens de l'itération et de pouvoir tenir compte de l'indice courant. C'est ce qui est proposé par les opérateurs "iter_haut" et "iter_bas".

Par exemple pour calculer le maximum d'un vecteur et l'indice de ce maximum, on peut écrire :

```
    v0 : -> (entier,entier)
    f : (entier,entier),entier,entier) -> (entier,entier)
avec
    v0 == (0,0)
    f(y,i,j,) == si j > !1(y) alors (j,i)
                 sinon y
                 fsi
et on a
    iter_haut[v0,f](/<1,3,2,3>/) == (3,2)
    iter_bas[v0,f](/<1,3,2,3>/) == (3,4)

prop Pour_iteration sortes t1 t2
operateurs
    init : -> t2
    pas : ( t2,entier,t1,) -> t2
satisfait  Tyfo[t1], Tyfo[t2]
fin

enrich iteration_vect   exige Pour_iteration [t1,t2 operateurs init,pas]
operateurs
    iter_haut : vecteur[t1] -> t2
    prive iter_haut_rep : (vecteur[t1],entier) -> t2
```

```
    iter_bas : vecteur[t1] -> t2
    prive iter_bas_rep : (vecteur[t1],entier) -> t2
variables
    i,j : entier
    v : vecteur[t1]
axiomes
    1 : iter_haut(v) ==> iter_haut_rep(v,#(v))
    2 : iter_haut_rep(v,i) ==> si i = 0 alors init
                               sinon pas( iter_haut(v,pred(i)),i,val(v,i))
                               fsi
    3 : iter_bas(v) ==> iter_bas_rep(v,0)
    4 : iter_bas_rep(v,i) ==> si i = #(v) alors init
                              sinon soit j = succ(i)
                                  dans pas(iter_bas_rep(v,j),j,val(v,j))
                                  fsoit
                              fsi
fin
```

L'effet de iter_haut est :
iter_haut[n,f](/<x_1, ..., x_k>/) == f(... f(f(n,1,x_1),2,x_2), ..., k,x_k)
pour iter_bas :
iter_bas[n,f](/ <x_1, ..., x_k>/) == f(... f(f(n,k,x_k),k-1,x_{k-1}), ..., 1,x_1)

L'opérateur qui construit un vecteur est plus facile à décrire car il s'appuie sur les sélecteurs. La propriété des parties formelles est donnée par :

```
prop Pour_construire_vect
sortes t dom
operateurs
    n : dom -> entier
    g : (dom,entier) -> t
satisfait Tyfo[t], Tyfo[dom]
fin

enrich constuire_vect
```

```
exige Pour_consruire_vect [t,dom operateurs n,g]
operateurs
    gen : dom -> vecteur[t]
    prive genrec : (dom,entier) -> vecteur[t]
variables
    x : dom
    i,j : entier
axiomes
    1 : gen(x) ==> genrec(x,0)
    2 : genrec(x,i) ==> si i=n(x) alors vect(n(x)):vecteur[t]
                        sinon soit j=succ(i)
                              dans rg(genrec(x,j),j,g(x,j))
                              fsoit
                        fsi
fin
```

L'effet de gen[n,g](x) est de construire un vecteur de taille n(x), et tel que l'élément i ait la valeur g(x,i). Autrement dit, le vecteur représente la fonction li.g(x,i) : nat -> t, restreinte au domaine [1,n(x)].

En utilisant les facilités de la surcharge en LPG, on retrouve l'opérateur "alpha"; paramétrés par la même propriété que pour les séquences.

```
enrich transformation_vect exige op_formel [t1,t2 operateurs f]
operateurs
    alpha : vecteur[t1] -> vecteur[t2]
    prive alpha_iter : (vecteur[t1],entier) -> vecteur[t2]
variables
    v : vecteur[t1]
    i, j : entier
axiomes
    1 : alpha(v) ==> alpha_iter(v,0)
    2 : alpha_iter(v,i) ==> si i = #(v) alors vect(#(v)) : vecteur[t2]
```

```
                    sinon soit j = succ(i)
                        dans rg(alpha_iter(v,j),j,f(val(v,j)))
                        fsoit
                fsi
fin
```

2.6.3 Opérateurs génériques sur les ensembles

On note ens[t] un ensemble (cf. la définition des ensembles §2.4.3) dont les éléments sont de type t et [t -> bool] sa fonction caractéristique. La fonction qui génère cet ensemble à partir d'un domaine D peut être vu comme un opérateur générique : gen[g] : D -> ens[t] avec g : (D,t) -> bool tel que gen[g](x) = { a | g(x,a) == vrai }. Il est très facile de remarquer que gen[g](x) est exactement la définition mathématique des ensembles infiniment engendrés. Pour définir un tel opérateur de façon à générer des ensembles finis, le type des éléments doit être fini et donc énumerable. En LPG on obtient aisément la définition suivante :

```
prop Enumere sorte t
operateurs
    ord : t -> entier
    ordinv : entier -> t
    card : -> entier
    <=,= : (t,t) -> bool
variables
    x,y : t
    n : entier
preconditions
    1 : ordinv(n) => valeur_indefinie si n<1 ou n>card fsi
axiomes
    1 : ordinv(ord(x)) == x
    2 : ord(ordinv(n)) == n
    3 : x=y == ord(x) = ord(y)
    4 : x<=y == ord(x)<=ord(y)
satisfait Ordre_total [t operateurs <=,=]
```

```
fin

prop Pour_gen_ens
sortes t dom
operateurs
    g : (dom,t) -> bool
satisfait Tyfo[t], Tyfo[dom]
fin

prop Pour_gen_enu sortes t dom
operateurs
    g : (dom,t) -> bool
    ord : t -> entier
    ordinv : entier -> t
    card : -> entier
    <=,= : (t,t) -> bool
combine Pour_gen_ens[t,dom operateurs g],
        Enumere[t operateurs ord,ordinv,card,<=,=]


enrich gen_ens
    exige Pour_gen_enu[t,dom operateurs g,ord,ordinv,card,<=,=]
operateurs
    gen : dom -> ens[t]
    prive gen_rec : (dom,entier) -> bool
variables
    x,y : dom
    i,j : t
axiomes
    1 : gen(x) ==> gen_rec(x,0)
    2 : gen_rec(x,i) ==>  si i=card alors e_vide:ens[t]
                          sinon soit j=succ(i), y=ordinv(j)
                                dans si g(x,y) alors y<+gen_rec(x,j)
                                     sinon gen_rec(x,j)
                                     fsi
```

fsoit

fsi

fin

Une autre façon de définir un opérateur générique qui génère un ensemble , et qui est constructive, c'est de définir l'opérateur "hom_inv" comme pour les séquences. D'ailleurs on défini aussi les opérateurs "alpha" et "init". Les propriétés exigées doivent avoir les opérateurs images des constructeurs de l'ensemble mais aussi un opérateur d'égalité exigé pour le type des éléments de l'ensemble. Pour cela, il suffit de prendre les mêmes propriétés que pour les séquences, auxquelles on ajoute un opérateur formel "eq" (car le type ensemble exige l'égalité sur ses éléments) et ensuite il faut déclarer que cette propriété "herite egalite [t operateurs eq].

-"init" : cet opérateur a la même fonction que pour les séquences. Il construit un ensemble dont les éléments sont calculés d'après la fonction passée en paramètre générique.

Exemple :

init[id,=](3) : ens[entier] ==> {1,2,3}

prop Init_formel_eq **sortes** t
operateurs
f : entier -> t
eq : (t,t) -> bool
herite Egalite [t **operateurs** eq]
fin

enrich construire_ens **exige** Init_formel_eq [t **operateurs** f,eq]
operateurs
init : entier -> ens[t]
prive init_iter : (entier,entier) -> ens[t]
variables

```
    n, i, j : entier
axiomes
    1 : init(n) ==> init_iter(n,0)
    2 : init_iter(n,i) ==> si i = n alors { : t }
                           sinon soit j = succ(i)
                                 dans f(j) <+ init_iter(n,j)
                                 fsoit
                           fsi
fin
```

-"alpha" applique l'opérateur passé en paramètre générique à tous les éléments de l'ensemble en paramètre, pour générer un nouvel ensemble.

Exemple :

alpha[alpha[carre,=],=]({{1},{2},{3}}) : ens[ens[entier]] ==> {{1},{4},{9}}

```
enrich transformation_ens
    exige op_formel_eq [t1,t2 operateurs f,eq]
operateurs
    alpha : ens[t1] -> ens[t2]
variables
    a : t1
    e : ens[t1]
axiomes
    1 : alpha(e_vide : ens[t1]) ==> e_vide : ens[t2]
    2 : alpha(a<+e) ==> f(a)<+alpha(e)
fin
```

-"hom_inv", cet opérateur générique d'homomorphisme inverse utilise, comme son équivalent pour les séquences, une fonction booléenne et une fonction pour générer un ensemble à partir d'une valeur donnée en paramètre.

Exemple :

hom_inv[vide?, f,=]([1,2,2,4,5]):ens[entier] == {1,2,4,5}
où f(s) ==> (tete(s),reste(s))

```
prop Pour_constr_ens sortes t,dom
operateurs
    p : dom -> bool
    f : dom -> (t,dom)
herite Egalite [t operateur eq]
fin

enrich construire_ens  exige Pour_constr_ens [t,dom operateurs  p,f,eq]
operateurs
    hom_inv : dom -> ens[t]
variables
    x : dom
axiomes
    1 : hom_inv(x) ==> si p(x) alors e_vide:ens[t]
                       sinon soit y=f(x)
                            dans !1(y)<+hom_inv(!2(y))
                             fsoit
                       fsi
fin
```

- Pour l'opérateur d'homomorphisme sur les ensembles que l'on note aussi "hom", la propriété exigée doit avoir les opérateurs images des constructeurs, ainsi que l'égalité pour les éléments du type ensemble.

```
prop Image_et_eq  sortes t1 dom
operateurs
    n : -> dom
    f : (t1,dom) -> dom
    eq : (t1,t1) -> bool
variables
```

x, y : t1, z : dom

axiomes

1 : f(x,f(x,z)) == f(x,z)

2 : f(x,f(y,z)) == f(y,f(x,z))

herite Egalite [t1 **operateurs** eq]

fin

enrich hom_sur_ens **exige** Image_et_eq [t1, dom **operateurs** n,f,eq]

operateurs

hom : ens[t1] -> dom

variables

a : t1

e : ens[t1]

axiomes

1 : hom(e_vide : ens[t1]) ==> n

2 : hom(a<+e) ==> f(a,hom(e))

fin

Exemple :

hom[e_vide,+,=]({{1},{2},{3}}) : ens[entier] ==> {1,2,3}

L'opérateur "+" étant ici l'union des ensembles.

Dans le cas où les sortes t1, dom représentent le même type, la condition sur la fonction passée en paramètre à l'opérateur "hom", dans la propriété "image_et_eq" se simplifie par la proposition suivante:

Proposition 1:

Si on a : n : -> dom et f : (dom,dom) -> dom où n est un élément neutre de f alors :

$$(1)\left\{\begin{array}{l} f(x,f(x,y)) == f(x,y) \\ \\ f(x,f(y,z)) == f(y,f(x,z)) \end{array}\right\} <=> (2)\left\{\begin{array}{ll} f(x,x) == x & \text{(idempotence)} \\ f(x,y) == f(y,x) & \text{(commutativité)} \\ f(x,f(y,z)) == f(f(x,y),z) & \text{(associativité)} \end{array}\right.$$

Preuve :

(1) => (2)

sous-buts : ⌊ f(x,f(x,y) == f(x,y), f(x,f(y,z)) == f(y,f(x,z)) ⌋

f(x,x) == x, f(x,y) == f(y,x), f(x,f(y,z)) == f(f(x,y),z)

réécriture de : f(x,x)

(n élément neutre) ==> f(x,f(x,n))

(antécédent1,->) ==> f(x,n)

(n élément neutre) ==> x

réécriture de : f(x,y)

(n élément neutre) ==> f(x,f(y,n))

(antécédent2,->) ==> f(y,f(x,n))

(n élément neutre) ==> f(y,x)

réécrtiture de : f(x,f(y,z))

(f. commutative) ==> f(x,f(z,y))

(antécédent2,->) ==> f(z,f(x,y))

(f. commutative) ==> f(f(x,y),z)

(1) => (2) est démontrée

(2) => (1)

sous-buts : ⌊ f(x,x) == x, f(x,y) == f(y,x), f(x,f(y,z)) == f(f(x,y),z) ⌋

f(x,f(x,y)) == f(x,y), f(x,f(y,z)) == f(y,f(x,z))

réécriture de : f(x,f(x,y))

(antécédent3,->) ==> f(f(x,x),y)

(antécédent1,->) ==> f(x,y)

réécriture de : f(x,f(y,z))

(antécédent2,->) ==> f(,f(y,z),x))

(antécédent3,->) ==> f(f(z,y),x)

(antécédent2,->) ==>f(y,f(x,z))

(2) => (1) est démontrée et par conséquent la proposition l'est aussi.

Remarque : Il faut noter que cette proposition est très utile dans la preuve, puisque généralement on a les propriétés de commutativité et d'associativité sur les opérateurs manipulés.

2.6.4 Autres sortes d'opérateurs génériques.

On peut aussi définir des opérateurs génériques non liés à des structures de données. C'est le cas par exemple de l'opérateur "tantque".

```
prop Pour_tantque sortes t
operateurs
    c : t -> bool
    f : t -> t
satisfait Tyfo[t]
fin

enrich boucle exige Pour_tantque [t operateurs c,f]
operateurs
    TANTQUE : t -> t
variables
    x : t
axiomes
1 : TANTQUE (x) ==> si c(x)  alors TANTQUE (f(x))
                    sinon x  fsi
fin
```

2.7 Possibilité de définir ses propres types et opérateurs génériques.

Si dans le paragraphe de la définition des opérateurs génériques, on a défini un ensemble minimal d'opérateurs pour chaque structure prédéfinie, c'est dans un but pratique pour l'algèbre de programmes (cf. chap. 3). Mais

il est toujours possible d'avoir ses propres opérateurs génériques aussi bien pour les structures de données déjà définies que pour les types que l'utilisateur veut définir lui-même; comme le souligne l'exemple suivant :

Exemple :

1- Définition du type générique arbre binaire (étiqueté) qu'on note arb_b[t]

type Arb_binaire **exige** Tyfo[t]
sortes arb_b
constructeurs
    vide : -> arb_b[t]
    noeud : (arb_b[t],t,arb_b[t]) -> arb_b[t]
fin

2- Définition d'un opérateur générique d'homomorphisme "hom" sur "arb_b[t].

prop Image_arb **sur** t dom
operateurs
    nc : -> dom
    f : (dom,t,dom) -> dom
satisfait Tyfo[t], Tyfo[dom]
fin

enrich hom_sur_arbre **exige** Image_arb [t,dom **operateurs** nc,f]
operateurs
    hom : arb_b[t] -> dom
variables
    a1, a2 : arb_b[t]
    x : t
axiomes
    1 : hom (vide : arb_b[t]) == nc
    2 : hom (noeud(a1,x,a2)) == f (hom(a1),x,hom(a2))

fin

3- Définition de l'opérateur générique "alpha" sur "arb_b[t]". La propriété exigée pour la définition de cet opérateur est la même que pour "alpha" sur les séquences.

enrich transformation_arbre **exige** Op_formel[t1,t2 **operateurs** f]
operateurs
    alpha : arb_b[t1] -> arb_b[t2]
variables
    x : t1
    a1, a2 : arb_b[t1]
axiomes
    1 : alpha(vide : arb_b[t1]) == vide : arb_b[t2]
    2 : alpha (noeud(a1,x,a2)) == noeud (alpha(a1),f(x),alpha(a2))
fin

4- Définition de l'opérateur générique "hom_inv" sur "arb_b[t].

prop Image_inv **sur** t, dom
operateurs
    p : dom -> bool
    g : dom -> (dom,t,dom)
satisfait tyfo[t], tyfo[dom]
fin

enrich hom_inv_arb **exige** image_inv [t,dom **operateurs** p,g]
operateurs
    hom_inv : dom -> arb_b[t]
variables
    x : dom
    u : (dom,t,dom)
axiomes

```
1 : hom_inv(x) ==>  si p(x) alors vide
                    sinon soit u = g(x)
                          dans noeud(hom_inv(!1(u)),!2,hom_inv(!3(u)))
                          fsoit
                    fsi
fin
```

Remarque : on aurait pu définir le type "abin" arbre binaire non étiqueté, ainsi que les opérateurs "hom" et "hom_inv" de la même façon que le type "arb_b[t]". Les constructeurs du type, les fonctions paramètres du "hom" et les fonctions paramètres du "hom_inv auront pour profil respectivement vide:-> abin, noeud:(abin,abin) -> abin, nc : -> dom , f:(dom,dom) -> dom, p:dom->bool, g: dom -> (dom,dom)

2.8 Exemples de programmes.

Exemple1 :

Une première utilisation à laquelle on peut faire référence, et qui illustre parfaitement l'utilisation des opérateurs génériques se trouve dans [Bert 84a] [Bert 84b]. Ce programme décrit un analyseur qui vérifie les conditions d'une grammaire LL(1).

Exemple2 :

On veut utiliser les propriétés de l'arbre binaire ordonné pour faire un tri sur les séquences, d'où on obtient les programmes suivants :

```
enrich arbre_binaire_ord exige Ordre_total [t operateurs <=,=]
operateurs
    inserer : (t,arb_b[t]) -> arb_b[t]
variables
    a1, a2 : arb_b[t]
```

```
    m,m1 : t
axiomes
    1 : inserer(m,vide:arb_b[t]) ==> noeud(vide:arb_b[t],m,vide:arb_b[t])
    2 : inserer(m,noeud(a1,m1,a2) ==> si m <= m1
                                          alors noeud(inserer(m,a1),m1,a2)
                                          sinon noeud(a1,m1,inserer(m,a2))
                                      fsi
fin

enrich tri_seq exige Ordre_total[t operateurs <=,=]
operateurs
    tri : seq[t] -> seq[t]
    f : (seq[t],t,seq[t]) -> seq[t]
variables
    s1,s2 : seq[t]
    x : t
axiomes
    1 : tri(s) ==> hom[nil,f] (hom[vide,inserer](s1): arb_b[t]) : seq[t]
    2 : f(s1,x,s2) ==> s1+(x<+s2)
fin
```

où, "+" est la concaténation de deux séquences.
- hom[nil,f] construit la séquence ordonné à partir de l'arbre, cet opérateur fait un parcours infixé de l'arbre.

Exemple3 :

Pour calculer la somme des carrés des n premiers entiers naturels, on obtient le programme suivant :

```
enrich entier
operateurs
    sommecarre : seq[entier] ->entier
    carre : (entier,entier) -> entier
```

variables
n : entier
axiomes
1 : sommecarre(n) ==> /[0,+] (alpha[carre](iota(n)): seq[entier])
2 : carre(n) ==> n * n
-- "+" et "*" sont l'addition et le produit sur les entiers naturels.
fin

Exemple4 :

On veut calculer la fonction de fibonacci, en utilisant les opérateurs génériques "hom" et "hom_inv" sur le type "abin" qu'on suppose défini. On obtient le programme suivant :

enrich fibonacci
operateurs
fib : entier -> entier
f : entier ->(entier,entier)
zero_ou_un? : entier -> bool
variables
x : entier
axiomes
1 : fib(x) ==> hom[1,+](hom_inv[zero_ou_un?,f](x):abin):entier
2 : f(x) ==> soit u = pred(x) dans (pred(u),u) fsoit
3 : zero_ou_un?(x) == x = 0 ou x = 1
-- zero_ou_un? teste si l'argument vaut 0 ou 1, et pred est la fonction predecesseur.
fin

3. Algèbre des opérateurs génériques

Notre but fondamental dans ce chapitre, est de doter les programmes construits à partir des opérateurs génériques (définis au chapitre 2 §2.6) d'une structure mathématique plutôt que syntaxique, de façon à pouvoir définir une algèbre de programmes. Une telle algèbre nous permet de raisonner sur les programmes et de les transformer. Suivant cette idée, nous allons étendre LPG, à un langage de manipulation de "fonctions" et non plus simplement de "termes". En effet les langages fonctionnels au niveau "objet" présentent certains "inconvénients" [Backus 85].

3.1 Problèmes des programmes fonctionnels au "niveau objet"

Dans la programmation fonctionnelle au niveau objet, la construction d'un nouveau programme à partir d'un autre implique l'application du programme donné aux objets ou bien à des variables objets jusqu'à la construction d'un résultat objet. Le nouveau programme est alors obtenu par abstraction des variables objets. Dans [Backus 85] une telle approche presente des problèmes, parmi ceux-ci on

peut citer les deux points suivants :

1- Contrairement aux programmes algorithmiques, qui utilisent des constructions standards (composition, boucle, etc ...), ces programmes utilisent la lambda abstraction comme construction principale pour construire des programmes. La lambda abstraction n'est pas un combinateur. Elle utilise une variable et une expression pour former soit un programme (une fonction) ou une autre expression. Ses propriétés sont plus syntaxiques qu' algébriques. Une telle approche ne permet pas de produire une algèbre de programmes sur laquelle baser des résultats au sujet des programmes. Puisqu'elle construit un programme à partir de deux entités qui ne sont pas des programmes.

2- Le raisonnement sur les programmes concerne les opérations sur les objets, puisque les programmes ne sont pas construits à partir de combinateurs. Ceci conduit à des théorèmes sur les objets, et ne permet pas d'avoir facilement des théorèmes généraux sur les programmes.

Pour avoir des théorèmes généraux , qui permettent de manipuler et transformer les programmes, les programmes doivent être construits par application d'opérations (combinateurs, composition, formes fonctionnelles, etc...) sur d'autres programmes déjà existants. Il faut noter que les opérations doivent avoir des propriétés algébriques [Backus 85].

3.2 L'avantage des programmes au "niveau fonction".

Au niveau fonction, les programmes sont construits à partir d'autres programmes en utilisant des opérations, qui possèdent les propriétés suivantes :

1- Quand on applique ces opérations, on obtient des résultats compréhensibles. Autrement dit si C(f,g) est un programme construit à partir de deux programmes f et g par l'opération C, et si on connait ce que font f et g alors on peut facilement comprendre ce que fait C(f,g).

2- Posibilité de developper une algèbre de programme [Backus 78] [Backus 81] [Williams 82]. Cette algèbre est constituée d'un ensemble puissant de règles et de théorèmes généraux, qui expriment des équivalences intéressantes entre les programmes.

Les deux points soulignés ci-dessus, nous permettent de considérer les opérations sur les programmes et leur structure algébrique, plutôt que la structure et la représentation des programmes eux-mêmes. Un programme est le résultat d'une opération, et non une entité syntaxique dont la structure est un ensemble de mot-clé, et dont la sémantique est une suite de transitions d'états.

3.3 L'utilité des types.

Il y a deux sortes de typage, le premier est le typage au sens "logique" c'est à dire créer une hiérarchie au niveau des fonctions (fonctions et fonctions d'ordre supérieur). Le second, qui nous intéresse dans ce paragraphe, est le typage des langages de programmation.

La présence des types au niveau des langages fonctionnels, apporte des avantages importants, même si on perd une certaine simplicité dans l'utilisation. Ceci permet d'éviter la vérification systématique du type à l'exécution, et autorise un contrôle statique. Un autre avantage est de pouvoir utiliser les types abstraits. Ainsi on obtient, des programmes modulaires, une cohérence des types utilisés et une forte structuration. Une proposition dans ce sens est celle de [Guttag 81] qui introduit les types abstraits dans FP.

3.4 Extension de la syntaxe de LPG

Nous allons définir un sur-ensemble de LPG qui est un langage de manipulation de "fonctions" et non plus simplement des termes. Les primitives des langages fonctionnels ont été décrites par [Backus 78], avec de nombreuses règles d'algèbre de programmes. Nous allons reprendre certaines de ces primitives. Le langage fonctionnel qui contient LPG est evidemment un langage typé.

Une valeur v dont le profil est :

v : -> T

qui est une valeur simple dans l'ensemble des "objets", peut être vue comme un opérateur d'arité 0, donc de l'ensemble des "fonctions". La règle d'application doit être bien typée, en ce sens que si :

f : T_1 -> T_2
x : -> T_1
alors f(x) : -> T_2

Par la suite t, t_1, t_2, ... seront mis pour des types quelconques.

- Les types prédéfinis [Bert 83b] sont :
bool, entier, car, chaine, seq[t], ens[t], vecteur[t]

- Le domaine des fonctions contient les valeurs définies et les exceptions qui sont plus définies que "⊥" (la valeur indéfinie).

Les fonctions sont déterminées de la manière suivante :
- les constantes et littéraux prédéfinis, sont des fonctions zéro-aires :
3 : -> entier

- et les opérateurs avec paramètres sont des fonctions n-aires, n>0 :

+ : (entier,entier) -> entier
trier[<=, =] : seq[t] -> seq[t]

Une fonction est définie par :
f == E
où f est un symbole de fonction, et E une expression de fonction qui peut contenir f. Chaque fonction est stricte. Une fonction f n'est pas définie en x si f(x) == ⊥

Les opérateurs prédéfinis seront des fonctions typées par :

$!1 : (t_1, \ldots, t_n) \rightarrow t_1$
$!2 : (t_1, \ldots, t_n) \rightarrow t_2$
...
si $f : (t_1, \ldots, t_n) \rightarrow t$
$x : (t_1, \ldots, t_i)$
alors $\$[f,x] : (t_{i+1}, \ldots, t_n) \rightarrow t$

On dispose de l'identité
$id : t \rightarrow t$

Les constructeurs de base sont :

- La composition, notée "o"

$f : t_2 \rightarrow t_3$
$g : t_1 \rightarrow t_2$
$f \circ g : t_1 \rightarrow t_3$

Cas particulier de composition; l'application :

$f : t_2 \rightarrow t_3$
$g :\rightarrow t_2$
$f \circ g :\rightarrow t_3$

- Le produit cartésien :
$f_1 : t \rightarrow t_1$
...
$f_n : t \rightarrow t_n$
alors $psi[f_1, \ldots, f_n] : t \rightarrow (t_1, \ldots, t_n)$

- La condition :

$f_1 : t_1 \rightarrow bool$
$f_2 : t_1 \rightarrow t_2$ $\qquad$ SI $[f_1, f_2, f_3] : t_1 \rightarrow t_2$
$f_3 : t_1 \rightarrow t_2$

La condition est considérée comme forme fonctionnelle ([Backus 78] [Williams 82]). L'avantage d'une telle approche permet de maintenir strictes toutes les fonctions.

- La "fonctionnalisation" ou passage d'une constante à une fonction avec argument

$y : \rightarrow t_2$

alors $\qquad C[y] : t_1 \rightarrow t_2$

Les équivalences sémantiques des constructeurs de base sont données par :

$id(x) == x$

$f \circ g\ (x) == f(g(x))$

$f \circ g == f(g)$ $\qquad$ (cas particulier)

$psi\ [f_1, \ldots, f_n]\ (x) == (f_1(x), \ldots, f_n(x))$

$C[y]\ (x) == y$

3.4.1 Les combinateurs.

Les combinateurs que nous définissons sont utiles dans le developpement de l'algèbre de programmes. Ils sont liés au produit cartésien dans le cas général. D'ailleurs on reprend l'opérateur "psi" déjà défini.
Les opérateurs combinateurs sont :

(1) le produit cartésien

$f_1 : t \rightarrow t_1$
$f_2 : t \rightarrow t_2$ $\qquad \Rightarrow$ $psi[f_1, \ldots, f_n] : t \rightarrow (t_1, \ldots, t_n)$
...

$f_n : t \rightarrow t_n$
tel que : $psi[f_1, \ldots, f_n]\ (x) == (f_1, \ldots, f_n)$

(2) la distribution :

$f_1 : t_1 \rightarrow t'_1$
$f_2 : t_2 \rightarrow t'_2 \qquad \Rightarrow phi[f_1, \ldots, f_n] : (t_1, \ldots, t_n) \rightarrow (t'_1, \ldots, t'_n)$
...
$f_n : t_n \rightarrow t'_n$
tel que : $phi[f_1, \ldots, f_n]\ (x_1, \ldots, x_n) == (f_1(x_1), \ldots, f_n(x_n))$

(3) l'application composée

$f_1 : (t_2 , t_3) \rightarrow t_3$
$\Rightarrow mu[f_1, f_2] : (t_1 , t_3) \rightarrow t_3$
$f_2 : t_1 \rightarrow t_2$
tel que : $mu\ [f_1, f_2]\ (x,y) == f_1(f_2(x), y)$

(4) pour T qui est le type seq ou le type vect, on a :

$<> : (T[t_1], \ldots, T[t_n]) \rightarrow T[(t_1, \ldots, t_n)]$
$>< : T[(t_1, \ldots, t_n)] \rightarrow (T[t_1], \ldots, T[t_n])$

Sémantique pour les séquences par exemple :

$<> ([x_{11}, x_{12}, \ldots, x_{1n}], \ldots, [x_{i1}, \ldots, x_{in}], \ldots, [x_{m1}, \ldots, x_{mn}]) ==$
$[(x_{11}, \ldots, x_{i1}, \ldots, x_{n1}), \ldots, (x_{1n}, \ldots, x_{in}, \ldots, x_{nn})]$
et inversement

$>< ([(x_{11}, \ldots, x_{m1}), \ldots, (x_{1i}, \ldots, x_{mi}), \ldots, (x_{1n}, \ldots, x_{mn})]) ==$
$([x_{11}, x_{12}, \ldots, x_{1n}], \ldots, [x_{i1}, \ldots, x_{in}], \ldots, [x_{m1}, \ldots, x_{mn}])$

3.4.2 Exemples de programmes dans cette notation.

Grâce à cette notation, toute les équations qui définissent un

opérateur peuvent être vus comme une identité entre fonctions ou expressions de fonctions. Si on considère les exemples donnés au § 2.7 (partie 1), on aurait (pour simplifier la notation, on remplace "==>" par "=="):

Exemple 1 :

enrich tri_seq **exige** Ordre_total[t **operateurs** <=,=]
operateurs
tri : seq[t] -> seq[t]
f : (seq[t],t,seq[t]) -> seq[t]

axiomes
1 : tri == hom[nil,f]:seq[t] o hom[vide,inserer]:arb_b[t]
2 : f == + o psi[!1,<+o psi[!2,!3]
fin

où, "+" est la concaténation de deux séquences.

- hom[nil,+ o psi[!1,<+ o psi[!2,!3]]] construit la séquence ordonné à partir de l'arbre, cet opérateur fait un parcours infixé de l'arbre.
Notons que hom[nil,<+ o psi[!2, + o psi[!1,!3]]] est un parcours préfixé et hom[nil,+ o psi[+ o psi[!1,!3],<+ o psi[!2,C[nil]]] est un parcours postfixé.

Exemple2 :

Pour calculer la somme des carrés des n premiers entiers naturels, on obtient:

enrich Entier
operateurs
somme_carre : seq[entier] -> entier

axiomes
1 : somme_carre == /[0,+]:entier o alpha[* o psi[id,id]]:seq[entier] o iota
fin
"+" et "*" sont l'addition et le produit sur les entiers naturels.

Exemple 3 :

Pour fibonacci on obtient :

enrich fibonacci
operateurs
fib : entier -> entier
zero_ou_un? : entier -> bool
axiomes
1 : fib == hom[1,+]:entier o hom[zero_ou_un?, psi[pred o pred, pred]]:abin
2 : zero_ou_un? == ou o psi[= o psi[id,C[0]], = o psi[id,C[1]]]
fin

3.5 Axiomatisation des opérateurs et combinateurs.

Nous donnons ci-après trois jeux de règles d'équivalence. Toutes ces règles se déduisent de la définition même des opérateurs. le typage n'est pas indiqué, mais il est supposé correct par rapport aux règles de constructions des opérateurs.

3.5.1 Préliminaire

Une équation entre fonctions implique une égalité entre les domaines de définition. Si dom(f) est le domaine pour lequel la fonction f est définie, on a :

f == g |- dom(f) = dom(g)

lorsque, par construction, les domaines des deux fonctions membre de l'égalité peuvent être différents (exemple la règle a.6 §3.5.2), on indique dans la précondition les propiétés sur les domaines qui assurent cette égalité forte. Si l'on note f | d la fonction f restreinte au domaine d, c'est à dire

$f \mid d\ (x) == f(x)$ si $x \in d$
$== \perp$ sinon

avec la propriété évidente :

$f \mid dom(f) == f$

alors la règle a.6 peut se lire :

$C[h] \circ g == C[h]$ si $dom(C[h]) \subseteq dom(g)$
ou encore
$C[h] \circ g == C[h] \mid dom(g)$

3.5.2 Les règles de base.

$\vdash g \circ id == g$ (a.1)
$\vdash id \circ g == g$ (a.2)
$\vdash (f \circ g) \circ h == f \circ (g \circ h)$ (a.3)
$\vdash SI\ [b,r,q] \circ h == SI\ [b \circ h, r \circ h, q \circ h]$ (a.4)
$\vdash h \circ SI\ [b,r,q] == SI\ [b, h \circ r, h \circ q]$ (a.5)
$dom(C[h]) \subseteq dom(g) \vdash C[h] \circ g == C[h]$ (a.6)
$\vdash f \circ \$[g,h] == \$[f \circ g, h]$ (a.7)
$\vdash \$[SI\ [b,f,g],h] == SI\ [\$[b,h],\$[f,h],\$[g,h]]$ (a.8)
$\vdash \$[f,h] \circ g == f \circ psi\ [h,g]$ (a.9)

3.5.3 Les règles du SI et des booléens

On note q % p pour indiquer que l'expression p apparaît dans q et q % (r/p) pour parler de l'expression q dans laquelle la sous-expression p est remplacée de manière uniforme par l'expression r. Ces règles sont reprises de [Guttag 78], à la suite de [Mc Carthy 63] [Boyer 75].

$\vdash$ SI [C[vrai],q,r] == q (b.1)
$\vdash$ SI [C[faux],q,r] == r (b.2)
$\vdash$ et == SI [!1,!2, C[faux]] (b.3)
$\vdash$ ou == SI [!1,C[vrai],!2] (b.4)
$\vdash$ non == SI [id,C[faux],C[vrai]] (b.5)
$\vdash$ SI [p,q,q] == q (b.6)
$\vdash$ SI [p,C[vrai],C[faux]] == p (b.7)
$\vdash$ SI [SI [p,q,r],a,b] == SI [p,SI [q,a,b],SI[r,a,b]] (b.8)
$\vdash$ SI [p,q % p,r] == SI [p,q % (C[vrai]/p),r] (b.9)
$\vdash$ SI [p,q,r % p] == SI [p,q,r % (C[faux]/p)] (b.10)

Ces règles, auxquelles on pourrait ajouter la définition des opérateurs d'implication et d'équivalence forment un système complet pour le calcul propositionnel.

3.5.4 Les règles d'équivalence des combinateurs.

Dans ces règles, on trouve une équivalence conditionnelle ; cela signifie que l'égalité après le symbole " $\vdash$" n'est vraie que si le ou les antécédents sont vrais également. "#" est l'opérateur "longueur" d'une séquence ou d'un vecteur, S[i] : i^{ieme} élément d'une séquence.

$\forall$ i $1 \le i \le n$ $dom(f_i) \subseteq \cap\ dom(f_j) \vdash$
$!i \circ psi[f_1, \ldots, f_i, \ldots, f_n] == f_i$ (c.1)

$\forall i\, 1 \le i \le n$ $dom(f_i \circ !i) \subseteq \Pi\ dom(f_j) \vdash$
$!i \circ phi[f_1, \ldots, f_i, \ldots, f_n] == f_i \circ !i$ (c.2)

$\Pi\ dom(f_i) = \cap\ dom(f_i \circ !i) \vdash$
$phi[f_1, \ldots, f_i, \ldots, f_n] = psi[f_1 \circ !1, \ldots, f_i \circ !i, \ldots, f_n \circ !n]$ (c.3)

$\vdash phi[f_1, \ldots, f_n] \circ psi[h_1, \ldots, h_n] == psi[f_1 \circ h_1, \ldots, f_n \circ h_n]$ (c.4)
$\vdash phi[f_1, \ldots, f_n] \circ phi[h_1, \ldots, h_n] == phi[f_1 \circ h_1, \ldots, f_n \circ h_n]$ (c.5)
$\vdash mu[h_1, h_2] \circ psi[f_1, f_2] == h_1 \circ psi[h_2 \circ f_1, f_2]$ (c.6)

$\vdash mu[h_1,h_2] \circ phi[f_1, f_2] == h_1 \circ phi[h_2 \circ f_1, f_2]$ (c.7)

$\vdash psi[f_1, \ldots,f_n] \circ h == psi[f_1 \circ h, \ldots,f_n \circ h]$ (c.8)

$\vdash h \circ mu[f_1,f_2] == mu[h \circ f_1,f_2]$ (c.9)

$\vdash mu[mu[f_1,f_2],f_3] == mu[f_1,f_2 \circ f_3]$ (c.10)

$\vdash mu[f,id] == f$ (c.11)

$\vdash <> \circ phi[psi[f_{11}, \ldots,f_{1n}], \ldots, psi[f_{n1}, \ldots,f_{nn}]] ==$
$psi[phi[f_{11}, f_{21}, \ldots,f_{n1}], \ldots,phi[f_{1n}, \ldots,f_{nn}]]$ (c.12)

$\vdash psi[!1, \ldots, !i, \ldots,!n] == id$ (c.13)

$\vdash phi[id, \ldots,id] == id$ (c.14)

$\vdash <> \circ >< == id$ (c.15)

$\# \circ !1 == \# \circ !2 \vdash >< \circ <> == id$ (c.16)

3.6 Algèbre des opérateurs génériques.

Dans notre formalisme, un programme peut être vu comme une construction obtenue à partir d'opérateurs simples et/ou opérateurs génériques. Cette construction utilise les combinateurs (cf. § 3.3) et la composition.

Comme nous l'avons fait pour les combinateurs, et opérateurs de base , il existe des équivalences intéressantes au niveau des compositions d'opérateurs. Nous donnerons certaines de ces règles sous forme d'axiomes et d'autres sous forme de théorème et nous les démontrerons formellement à partir des définitions. Parmi les règles données ci-après certaines sont reprises de [Backus 78]. Le typage n'est pas indiqué, mais il est supposé correct par rapport aux règles de constructions des opérateurs.

Avant de décrire des ensembles de règles qui sont propres pour chaque structure, donnons les règles "génériques" des types vecteurs, séquences et ensembles.

3.6.1 Règles "génériques" des séquences, vecteurs et ensembles.

$\vdash$ alpha[f_1] o alpha[f_2] o ...o alpha[f_n] == alpha[f_1 o f_2 o...o f_2] (g.1)
$\vdash$ alpha[id] == id (g.2)

3.6.2 Règles "génériques" aux séquences et vecteurs.

$\vdash$ psi[alpha[f_1], ..., alpha[f_n]] == >< o alpha[psi[f_1, ..., f_n]] (g.3)
$\vdash$ <> o phi[alpha[f_1], ..., alpha[f_n]] == alpha[phi[f_1, ...,f_2]] o <> (g.4)

$\forall i\ 1 \leq i \leq n$ $dom(f_i) \subseteq \cap\, dom(f_j) \vdash$
alpha[!i] o <> o psi[f_1, ..., f_i, ..., f_n] == f_i (g.5)

$\vdash$ alpha[f] o !i o >< == alpha[f o !i] (g.6)

3.6.3 Règles et théorèmes généraux sur les séquences.

$\vdash$ /[n,f] == hom[n,f] (s.1)
$\vdash$ hom[n,f_1] o alpha[f_2] == hom[n,mu[f_1,f_2]] (s.2)
$\vdash$ hom[nil,<+] == id (s.3)
$\vdash$ hom_inv[vide?,psi[tete,reste]] == id (s.4)
$\vdash$ alpha[f] o hom_inv[p,psi[f_1,f_2] == hom_inv[p,psi[f o f_1,f_2] (s.5)

$\vdash$ phi[hom[n_1,c_1],...,hom[n_m,c_m]] o >< ==
hom[psi[n_1,...n_m],psi[c_1 o phi[!1,!1],...,c_m o phi[!m,!m]]] (s.6)

$\vdash$ alpha[f] == hom[nil,mu[<+,f]] (s.7)
$\vdash$ init[f] == alpha[f] o init[id] (s.8)
$\vdash$ iota == init[id] (s8')

Théorème 1 (génération de la forme récursive) :

h == hom[n,f] o hom_inv[p,psi[f1,f2]] |-

h == SI[p,C[n],f o psi[f1,h o f2]] (s9)

Preuve :

réécriture de : h

(antécédent,->) ==> hom[n,f] o hom_inv[p,psi[f1,f2]](x)

(def.composition,->) ==> hom[n,f](hom_inv[p,psi[f1,f2]](x))

(def.hom_inv,->) ==> hom[n,f](si p(x) alors nil

sinon f1(x)<+hom_inv[p,psi[f1,f2]](f2(x)))

(régles du SI,->) ==> si p(x) alors hom[n,f](nil)

sinon hom[n,f](f1(x)<+hom_inv[p,psi[f1,f2]](f2(x)))

(def.hom1,->) ==> si p(x) alors n

sinon hom[n,f](f1(x)<+hom_inv[p,psi[f1,f2]](f2(x)))

(def.hom2,->) ==>si p(x) alors n

sinon f(f1(x),hom[n,f](hom_inv[p,psi[f1,f2]](f2(x))))

réécriture de : SI[p,C[n],f o psi[f1,h o f2]](x)

(def. du SI,->) ==> si p(x) alors C[n](x) sinon f o psi[f1,h o f2](x)

(def. du C[],->) ==> si p(x) alors n sinon f o psi[f1,h o f2](x)

(def. psi,->) ==> si p(x) alors n sinon f o (f1(x), h o f2(x))

(composition,->) ==> si p(x) alors n sinon f(f1(x),h(f2(x)))

(antécédent,->) ==> si p(x)alors n

sinon f(f1(x),hom[n,f](hom_inv[p,psi[f1,f2]](f2(x))))

==> vrai

le théorème est démontré

Ce schéma de fonction récursive linéaire a été souvent donné dans la litérature sans trop de justifications [Kieburtz 81] [Backus 85].
Nous remarquons ici qu'il correspond très exactement au fait que la fonction h : dom -> dom est égale à une composition h1 o h2 où h2 : dom -> seq[t] est un homomorphisme

inverse, et h1 : seq[t] -> dom est homomorphisme.

En exploitant les résultats présentés dans [Kieburtz 81] [Backus 85], on peut donner la forme itérative de la fonction h par le théorème suivant :

Théorème2 (suppression de récursivité) :

SI[p,C[n],f o psi[f1,h o f2]] == h, monoide [t operateurs n,f] |-
h == !2 o TANTQUE[non o p o !1,psi[f2 o !1,f o phi[f1,id]]] o psi[id,C[n]] (s.10)

Ce passage à la forme itérative de la fonction h a été prouvé de deux manières différentes :

1- Dans [Kieburtz 81] , une preuve est donné par induction sur le point fixe.
2- Dans [Backus 85], une preuve est donné en utilisant le théorème d'expansion linéaire.

Ce résultat de génération récursive est tout à fait général dans le sens suivant : soit une structure libre X définie comme solution de l'équation sur les domaines [Lehmann 77] X = F(X), où F est composée des constructeurs d'union disjointe et de produit cartésien, et dont les objets sont finis (finiment engendrés); soient les formes fonctionnelles hom : X -> D et hom_inv : D -> X construits comme pour les séquences alors la composition de ces deux opérateurs telle que :

h == hom o hom_inv

est équivalente à une fonction (récursive si le domaine est récursif) h : D -> D dont la "structure" est isomorphe à F.

Par exemple, avec $F_1(D) == 1 + D$, la fonction récursive image est :

h == SI[p,C[n],f o h o g]

où p, n, f, g sont des paramètres de l'homorphisme et de l'homomorphisme inverse. De même, le foncteur correspondant aux arbres binaires purs, qui est $F_2(D) == 1 + D \times D$ a une fonction image de la forme :

$$h == SI[p,C[n], f \circ psi[h \circ f1, h \circ f_2]$$

Théorème3 :

monoide [t operateurs n, f], |- hom[n,f] ==
!2 o TANTQUE[non o vide? o !1,psi[reste o !1,f o phi[tete,id]]] o psi[id,C[n]]
(s.11)

Preuve :

réécriture de : hom[n,f]
(id,->) ==> hom[n,f] o id
(s.4,<-) ==> hom[n,f] o hom_inv[vide?,psi[tete,reste]
(s.8,->) ==> SI[vide?,C[n],f o psi[tete,h o reste]
(s.9,->) ==>
!2 o TANTQUE[non o vide? o !1, psi[reste o !1,f o phi[tete,id]]] o psi[id,C[n]]

Les structures inductives se prêtent bien à des équivalences de haut niveau. Ce qui permet de donner les théorèmes suivants :

Théorème4 (de composition) :
$n_2 == g \circ n_1$, c_2 o phi[id,g] == $g \circ c_1$ |-
hom[n_2,c_2] == g o hom[n_1,c_1] (s.12)

Preuve :

Pour la preuve, on est obligé de descendre au niveau des termes, à cause de l'utilisation des définitions même des opérateurs.

Induction :

sous-buts : 1.1 $hom[n_2,c_2](nil) == g \circ hom[n_1,c_1](nil)$
1.2 $(hom[n_2,c_2](s_0) == g \circ hom[n_1,c_1](s_0))$
hom[n2,c2](a<+s0) == g o hom[n1,c1](a<+s0)
réécriture de : $hom[n_2,c_2](nil)$
(def.hom1,->) ==> n_2
réécriture de : $g \circ hom[n_1,c_2](nil)$
(def.hom1,->) ==> $g \circ n_1$
(antécédent s.11,->) ==> n_2

(1.1) est démontrée.
réécriture de : $hom[n_2,c_2](a<+s_0)$
(def. hom2,->) ==> $c_2(a, hom[n_2,c_2](s_0))$
réécriture de : $g \circ hom[n_1,c_1](a<+s_0)$
(def. hom2,->) ==> $g \circ c_1(a,hom[n_1,c_1](s_0)$
(antécédent s.11,->) ==> $c_2 \circ phi[id,g](a,hom[n_1,c_1](s_0))$
(def. phi,->) ==> $c_2(id \circ a,g \circ hom[n_1,c_1](s_0))$
(antécédent 1.2,->) ==> $c_2(a,hom[n_2,c_2](s_0))$

1.2 est démontré et par conséquent (s.11) est démontré.

L'opérateur "hom_inv" n'est pas forcément la fonction réciproque pour l'opérateur "hom". Les conditions pour que la composition entre ces deux opérateurs soit égale à l'identité, sont données par le théorème suivant :

Théorème 5 (d'identité) :

avec g == hom_inv[p,f] on a :
$p \circ n == vrai, p \circ c \circ psi[!1,!2] == faux, f \circ c == id_{(t,dom)}$ |-
$hom\_inv[p,f] \circ hom[n,c] == id_{seq[t]}$ (s.13)

$id_{(t,dom)}$ signifie que la fonction d'identité a comme profil : (t,dom) -> (t,dom), et $id_{seq[t]}$ a pour profil : seq[t] -> seq[t].

Preuve :

Induction

sous-buts : 1.1 hom_inv[p,f] o hom[n,c](nil) == nil

1.2 ⌊ hom_inv[p,f] o hom[n,c](s_0) == s_0⌋

hom_inv[p,f] o hom[n,c](a<+s0) == a<+s0

réécriture de : hom_inv[p,f] o hom[n,c](nil)

(def. hom1,->) ==> hom_inv[p,f](n)

(def. hom_inv,->) ==> si p(n) alors nil sinon

!1(f(n))<+hom_inv[p,f](!2(f(n))

(antécédent s.12,->) ==> si vrai alors nil sinon

!1(f(n))<+hom_inv[p,f](!2(f(n))

(règle du SI) ==> nil

1.1 est démontré.

réécriture de : hom_inv[p,f] o hom[n,c](a.s_0)

(def. hom2,->) ==> hom_inv[p,f](c(a,hom[n,c](s_0)))

(def.hom_inv,->) ==> si p(c(a,hom[n,c](s_0)) alors nil sinon

!1(f(c(a,hom[n,c](s_0))))<+hom_inv[p,f](!2(f(c(a,hom[n,c](s_0)))))

(antécésents.8,->) ==> si faux alors nil sinon

!1(f(c(a,hom[n,c](s_0))))<+hom_inv[p,f](!2(f(c(a,hom[n,c](s_0)))))

(si,->) ==> !1(f(c(a,hom[n,c](s_0))))<+hom_inv[p,f](!2(f(c(a,hom[n,c](s_0)))))

(antécédents.8,->) ==> !1(a,hom[n,c](s_0))<+hom_inv[p,f](!2(a,hom[n,c](s_0)))

(def. !,->) ==> a<+hom_inv[p,f](!2(a,hom[n,c](s_0)))

(def.!,->) ==> a<+hom_inv[p,f](hom[n,c](s_0))

(antécédent 1.2,->) ==> a<+s_0

1.2 est démontré et par conséquent le théorème aussi.

Théorème 6 (de fusion) :

n == psi[n_1,n_2], c == psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] |-

psi[hom[n_1,c_1],hom[n_2,c_2]] == hom[n,c] (s.14)

Preuve :

Induction

sous-buts : 1.1 psi[hom[n_1,c_1],hom[n_2,c_2]](nil) == hom[n,c](nil)

1.2 ⌊psi[hom[n_1,c_1],hom[n_2,c_2]](s_0) == hom[n,c](s_0)⌋

psi[hom[n_1,c_1],hom[n_2,c_2]](a<+s_0) == hom[n,c](a<+s_0)

réécriture de : psi[hom[n_1,c_1],hom[n_2,c_2]](nil)

(def. psi,->) ==> (hom[n_1,c_1](nil),hom[n_2,c_2](nil))

(def.hom1,->) ==>(n_1, hom[n_2,c_2](nil))

(def.hom.1,->) ==>(n_1,n_2)

(def.psi,<-) ==> psi[n_1,n_2]

réécriture de : hom[n,c](nil)

(def.hom1,->) ==> n

(antécédent1,->) ==> psi[n_1,n_2]

1.1 est démontrée

réécriture de : psi[hom[n_1,c_1],hom[n_2,c_2]](a<+s_0)

(def.psi,->) ==> (hom[n_1,c_1] (a<+s_0), hom[n_2,c_2](a<+s_0))

(def.hom2,->) ==>(c_1(a,hom[n_1,c_1](s_0)), hom[n_2,c_2](a<+s_0))

(def.hom2,->) ==>(c_1(a,hom[n_1,c_1](s_0)), c_2(a,hom[n_2,c_2](s_0)))

réécriture de : hom[n,c](a<+s_0)

(def.hom2,->) ==> c(a,hom[n,c]($\tilde{s}_0$))

(antécédent1.2,<-) ==> c(a, psi[hom[n_1,c_1],hom[n_2,c_2]](s_0))

(def.psi,->) ==> c(a, (hom[n_1,c_1](s_0),hom[n_2,c_2](s_0)))

(antécédent2,->)==> psi[c_1o phi[id,!1],c_2 o phi[id,!2]]
(a,(hom[n_1,c_1](s_0),hom[n_2,c_2](s_0)))

(def.phi,->) ==>(c_1(a,hom[n_1,c_1](s_0)),c_2(a,hom[n_2,c_2](s_0)))

1.2 est démontrée, et par conséquent le théorème aussi.

3.6.4 Règles sur les vecteurs.

|-iter_haut[n,f_1] o alpha[f_2] == iter_haut[n,f_1o phi[id,id,f_2]] (v.1)

|-iter_bas[n,f_1] o alpha[f_2] == iter_bas[n,f_1o phi[id,id,f_2]] (v.2)

|- val o phi[alpha[f],id] == f o val (v.3)
o v == C[n] |- iter_bas[n,rg] o v == v (v.4)
o v == C[n] |- iter_haut[n,rg] o v == v (v.5)
|- alpha[f_1] o gen[n,f_2] == gen[n,f_1 o f_2] (v.6)
|- gen[#, val] == id (v.7)
C[1]≤i, i≤n |- val o psi[gen[n,f],i] == f o psi[id,i] (v.8)

Pour les structures non libres , on n'a évidement pas de résultats généraux, néanmoins dans le cas particulier que l'on a ici, les vecteurs, on a une équivalence itérative :

|- iter_haut[n,f] o gen[k,g] ==
!1 o TANTQUE[<= o psi[!2,C[k]],psi[f o psi[!1,!2,g o psi[!3,!2]],succ o !2,!3]]
o psi[C[n],C[1],id] (v.9)

|- iter_bas[n,f] o gen[k,g] ==
!1 o TANTQUE[>= o psi[!2,C[1]],psi[f o psi[!1,!2,g o psi[!3,!2]], pred o !2,!3]]
o psi[C[n],C[k],id] (v.10)

3.6.5 Règles et théorèmes sur les ensembles.

Pour le type ensemble, on utilisera les règles et théorèmes dérivés des séquences.

|- init[f,=] == alpha[f] o init[id,=] (e.1)
|- hom[e_vide,<+,=] == id (e.2)
|- hom[n,f_1,=] o alpha[f_2] == hom[n,mu[f_1,f_2],=] (e.3)
|- alpha[f_1] o hom_inv[p,f_2,=] == hom_inv[p,phi[f_1,id] o f_2,=] (e.4)

Théorème 7(génération de la forme récursive) :

$h == hom[n,f,=] \circ hom\_inv[p,psi[f_1,f_2],=]$ |-
$h == SI[p,C[n],f \circ psi[f_1,h \circ f_2]]$ (e.5)

A cause de la proposition1 (cf. chap 2), la condition associée au théorème de suppression de la récursivité se simplifie pour les ensembles à une condition sur le type . Ce qui permet de donner le théorème suivant :

Théorème 8 (passage à la forme itérative) :

$h == hom[n,f,=] \circ hom\_inv[p,psi[f_1,f_2],=]$, f : (t,t) -> t |-
$h == !2 \circ TANTQUE[non \circ p \circ !1,psi[f_2 \circ !1,f \circ phi[f_1,id]]] \circ psi[id,C[n]]$ (e.6)

Preuve :

Réécriture de h :
(antécédent1,->) ==> $hom[n,f,=] \circ hom\_inv[p,psi[f_1,f_2,=]$
(théorème 7,->) ==> $SI[p,C[n],f \circ psi[f_1,h \circ f_2]$
d'après la proposition1 on a :
monoide [t operateurs n,f] et $h == SI[p,C[n],f \circ psi[f_1,h \circ f_2]$
C'est exactement la même forme récursive pour les séquences. D'où :
(théorème2,->) ==>
$!2 \circ TANTQUE[non \circ p \circ!1,psi[f_2 \circ!1,f \circ phi[f_1,id]]] \circ psi[id,C[n]]$

Théorème 9 (de composition) :

$n_2 == g \circ n_1$, $c_2 \circ phi[id,g] == g \circ c_1$ |-
$hom[n_2,c_2,=] == g \circ hom[n_1,c_1,=]$ (e.7)

Thèorème 10 (cas particulier) :

p o n == vrai, p o c == C[faux], f o c == $id_{(t,dom)}$ |-
hom_inv[p,f,=] o hom[n,c,=]==$id_{ens[t]}$ (e.8)

Théorème 11 (de fusion sur les ensembles) :

|- psi[hom[n_1,c_1,=],hom[n_2,c_2,=]] ==
hom[psi[n_1,n_2],psi[c_1 o phi[id , !1],c_2 o phi[id, !2],=] (e.9)

Note : la preuve des théorèmes ci-dessus est analogue aux preuves des théorèmes sur les séquences.

Proposition 2 :

Si c_1 et c_2 satisfont les conditions exigées pour l'opérateurs générique "hom" sur les ensembles, alors " psi[c_1 o phi[id,!1],c_2 o phi[id,!2]]" les satisfait aussi.

Preuve :

Les conditions exigées pour "hom" sur les ensembles sont :
c o psi[!1,c o psi[!2,!3]] == c o psi[!2,c o psi[!1,!3]] (1)
c o psi[!1,c] == c (2)
c_1 et c_2 satisfont les conditions ci-dessus, d'où on a les hypothèses suivantes :
c_1 o psi[!1,c_1 o psi[!2,!3]] == c_1 o psi[!2,c_1 o psi[!1,!3]] (1.a)
c_2 o psi[!1,c_2 o psi[!2,!3]] == c_2 o psi[!2,c_2 o psi[!1,!3]] (1.b)
c_1 o psi[!1,c_1] == c_1 (2.a)
c_2 o psi[!1,c_2] == c_2 (2.b)
dire que " psi[c_1 o phi[id,!1],c_2 o phi[id,!2]]" satisfait les conditions exigées pour "hom" revient à vérifier les égalités suivantes :
-Pour la condition (1) :

psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o
psi[!1,psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o psi[!2,!3] ==
psi[c_1 o phi[id,!1],c_2 o psi[id,!3]] o psi[!2,psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o psi[!1,!3]] (*)
-Pour la condition (2) :

psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o psi[!1,psi[c_1 o phi[id,!1],c_2 o phi[id,!2]]]
== psi[c_1 o phi[id,!1],c_2 o phi[id,!2]]

Afin de faciliter la lisibilité de la preuve posons d'abord que l'égalité notée (*) est de la forme suivante :

$A \circ B == A_1 \circ B_1$

tel que :

A == psi[c_1 o phi[id,!1],c_2 o phi[id,!2]]
B == psi[!1,psi[c_1 o phi[id,!1],c_2 o psi[id,!2]] o D]
où D == psi[!2,!3]
et
A_1== psi[c_1 o phi[id,!1],c_2 o phi[id,!2]]
B_1== psi[!2,psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o D_1]
D_1== psi[!1,!3]

Preuve de l'égalité (*) :

- réécriture B :

psi[!1,psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o D]
(c.10) ==> psi[!1,psi[c_1 o phi[id,!1],c_2 o phi[id,!2] o D]]
(c.4) ==> psi[!1,psi[c_1 o psi[id o !2,!1 o !3],c_2 o psi[id o !2,!2 o !3]]]
(a.2) ==> psi[!1,psi[c_1 o psi[!2,!1 o !3],c_2 o psi[!2,!2 o !3]]]
Le membre gauche de l'égalité (*) (à démontrer) se simplifie à :

psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o
psi[!1,psi[c_1 o psi[!2,!1 o3],c_2 o psi[!2,!2o!3]]]

Posons que :

psi[!1,psi[c_1 o psi[!2,!1 o !3],c_2 o psi[!2,!2o!3]]] == B'
-réécriture du membre gauche de l'égalité (*) :

psi[c_1 phi[id,!1],c_2 o phi[id,!2]] o B'

(c.10) ==> psi[c_1 o phi[id,!1] o B',c_2 o phi[id,!2] o B']
(c.4)(a.2) ==> psi[c_1o psi[!1,c_1o psi[!2,!1 o !3]],c_2 o psi[!1,c_2 o psi[!2,!2 o !3]]]

-réécriture de B_1 :

psi[!2,psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o D_1]
(c.10) ==> psi[!2,psi[c_1 o phi[id,!1] o D_1,c_2 o phi[id,!2] o D_1]]
==> psi[c_1 o psi[!1,!1o!2],c_2 o psi[!1,!2o!2]] o psi[!2,psi[!1,!1o 3],
c_2 o psi[!1,!2 o !3]]]
posons que :
psi[!2,psi[!1,!1 o !3],c_2 o psi[!1,!2 o !3]]] == B'_1

-réécriture du membre droit de l'égalité (*) :

psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o B'_1
(c.10) ==> psi[c_1 o psi[!1,!1 o!2] o B'_1,c_2 o psi[!1,!2 o !2] o B'_1]
(c4.)(a.2) ==> psi[c_1o psi[!2,c_1o psi[!1,!1o !3]],c_2o psi[!2,c_2o psi[!1,!2o !3]]]

Ainsi l'égalité (*) devient :

psi[c_1 o psi[!1,c_1 o psi[!2,!1 o !3]],c_2 o psi[!1,c_2 o psi[!2,!2 o !3]]]
== psi[c_1 o psi[!2,c_1 o psi[!1,!1 o !3]],c_2 o psi[!2,c_2 o psi[!1,!2 o !3]]]

Prouver l'équation ci-dessus, qui est l'égalité entre deux produits cartésiens à deux éléments, revient à prouver les deux égalités suivantes :

c_1 o psi[!1,c_1 o psi[!2,!1 o !3]] == c_1 o psi[!2,c_1 o psi[!1,!1 o !3]] (*1)

c_2 o psi[!1,c_2 o psi[!2,!2 o !3]] == c_2o psi[!2,c_2 o psi[!1,!2 o !3]] (*.2)

- réécriture du membre gauche de (*.1) :

(c.10) ==> c_1 o psi[!1,c_1 o psi[!2,!3]] o psi[!1,!2,!1 o !3]
(hyp.1a) ==> c_1 o psi[!2,c_1 o psi[!1,!3]] o psi[!1,!2,!1 o !3]
(c.10) ==> c_1 o psi[!2,c_1 o psi[!1,!1 o !3]]
==> vrai
l'égalité (*.1) est prouvée.

- réécriture du membre gauche de (*.2) :

(c.10) ==> c_2 o psi[!1,c_2 o psi[!2,!3]] o psi[!1,!2,!2 o !3]
(hyp. 1.b) ==> c_2 o psi[!2,c_2 o psi[!1,!3]] o psi[!1,!2,!2 o !3]
(c.10) ==> c_2 o psi[!2,c_2 o psi[!1,!2 o !3]]
==> vrai

les égalités (*.1) et (*.2) sont prouvées, et par conséquent l'égalité (*) l'est aussi.

Deuxième condition :

Toujours pour la lisibilité de la preuve, posons que le membre gauche de l'équation (**) est de la forme :

psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o E

où :

E == psi[!1,psi[c_1 o phi[id,!1],c_2 o phi[id,!2]]]

- réécriture du membre gauche de l'équation (**)

psi[c_1 o phi[id,!1],c_2 o phi[id,!2]] o E
(c.10) ==> psi[c_1 o phi[id,!1] o E,c_2 o phi[id, !2] o E]
(c.4)(a.2)==> psi[c_1o psi[!1,c_1o psi[!1,!1o !2]],c_2 o psi[!1,c_2o psi[!1,!2o !2]]]

d'où l'équation (**) devient :

psi[c_1 o psi[!1,c_1 o phi[id,!1]],c_2 o psi[!1,c_2 o phi[id, !2]]] ==
psi[c_1 o phi[id,!1],c_2 o phi[id,!2]]

Prouver une telle équation, qui est l'égalité entre produit cartésien à deux éléments, revient à prouver, les deux égalités suivantes (par la règle (c.3) :

c_1 o psi[!1,c_1 o psi[!1,!1 o !2]] == c_1 o psi[!1,!1 o !2] (**.1)
c_2 o psi[!1,c_2 o psi[!1,!2 o !2]] == c_2 o psi[!1,!2 o !2] (**.2)

-réécriture du membre gauche de (**.1)

(c.10) ==> c_1 o psi[!1,c_1] o psi[!1,!1 o !2]
(hyp. 2a) ==> c_1 o psi[!1,!1 o !2]
==> vrai

-réécriture du membre gauche de (**.2)

(c.10) ==> c_2 o psi[!1,c_2] o psi[!1,!2 o !2]
(hyp.2.b) ==> c_2 o psi[!1,!2 o !2]
==> vrai

Ainsi les équations (**1) et (**2) sont prouvées et par conséquent l'équation (**) l'est aussi.

4. Applications à la transformation et à la preuve de programmes

L'approche algébrique a été développée par Backus [Backus 78], dans le but de raisonner sur les programmes fonctionnels par manipulation (transformation du source au source). Suivant la même idée, nous avons développé une algèbre d'opérateurs génériques (cf. chapitre 3) dans un univers typé, en utilisant un formalisme qui permet d'associer des propriétés (associativité, commutativité, etc. ...) aux fonctions.

Dans ce chapitre, nous allons donner deux applications évidentes de cette approche algébrique typée, la première étant l'application à la tranformation, et la seconde l'application à l'équivalence de programmes.

4.1 Transformation de programmes.

Avant de donner notre approche de transformation, il

est intéressant de comparer l'approche de différents auteurs dans ce domaine.

4.1.1 L'approche de [Wadler 81].

Dans son article, il présente un ensemble d'opérateurs sur les listes, avec un ensemble de quatre règles de transformation dans le but d'éliminer le parcours et la création des listes intermediaires inutiles. Comme exemple on a :

```
(red[f,a]) (map g) xs -> (red[h,a]) xs
                            where h[a1, x] = f[a1,g x]
```

est équivalente à la règle :

$$\vdash hom[n,f1] \circ alpha[f2] == hom[n,mu[f1,f2] \qquad (s.2)$$

La différence est que l'équation au niveau fonction nous permet immédiatement de faire la substitution d'un membre d'une équation par un autre dans toute expression de fonction, par contre au niveau objet l'utilisation d' une règle équivalente est moins directe et exige une certaine indirection inutile, c'est le cas de la clause "where" qui est utilisé par manque de combinateur.

D'après Wadler, cet ensemble minimal de règles est "complet" dans le sens où toute composition de deux opérateurs adjacent peut être combinée en un seul opérateur. Si on considère un simple programme qui prend une liste de liste d'entiers pour rendre une liste d'entiers (c'est à dire appliquer la concaténation), et ensuite on fait le produit des éléments de cette liste. On obtient dans son formalisme le programme suivant :

```
(red[*,1])(red[append,nil] xss)
```

où red[f,a] est un opérateur équivalent à l'opérateur d'insertion (/[a,f]), "append" la concaténation, et "nil" la liste vide.
Aucune règle dans l'ensemble qu'il propose, ne peut combiner une telle composition de ces deux opérateurs en un seul. ce qui est contradictoire avec son résultat.

4.1.2 L'approche de [Kieburtz, Shultis 81]

Ils utilisent le formalisme de FP pour dériver des théorèmes pour l'élimination de la récursivité de la forme :

f = p -> q ; h o [i,f o j] (proposition 2)

et aussi pour les fonctions non linéaires de la forme :

f = p -> q ; h o [f o r, f o s] (proposition 5)

f = p ->q ; h o [r, h o [f o s, f o t]] (proposition 6)

Ces formes récursives sont des cas particulier de notre résultat sur la génération de fonctions récursives (cf. chapitre 2). L'ensemble des théorèmes qu'ils proposent est très utile pour l'élimination de la récursivité. D'ailleurs, on peut exploiter ces résultats, surtout que notre formalisme est bien adapté à ce type de transformation. En effet, l'étape clé de ce type de transformation est due en réalité aux propriétés des fonctions (associativité, existence d'un élément neutre, ...), malheureusement comme on l'avait souligné déjà au début du chapitre 2, FP n'a aucun moyen qui permet d'associer des telles propriétés aux fonctions définies par l'utilisateur.

4.1.3 L'approche de [Burstall, Darlington 77]

C'est une méthode de transformation de définitions récursives découverte indépendament par Burstall et Darlington [Darlington 74] [Burstall 77] d'une part, et Manna et Waldinger [Manna 73][Manna 75] d'autre part. Cette méthode est définie de la manière suivante :
les équations manipulées sont des équations récursives, les règles de transformation sont les suivantes :

- la définition : introduire une équation dont le membre gauche n'est pas un exemplaire du membre gauche d'une définition existante.

- l'instanciation : elle consiste à réaliser des substitutions sur une équation.

- le dépliage : si un terme contient un exemplaire de membre gauche d'équation on peut remplacer celui-ci par l'exemplaire du membre droit correspondant.

- le pliage : c'est l'opération inverse du dépliage.

- l'abstraction : c'est l'introduction d'une clause "where" pour transformer une équation en utilisant des identificateurs auxiliaires.

- l'application de propriétés sur les fonctions de base, par exemple l'application de l'associativité, etc. ...

Exemples ([Burstall 77]) :

1- Le but est de produire une forme "itérative" de factorielle.

(1)	fact(0) <= 1	définition
(2)	fact(n+1) <= (n+1)*fact(n)	définition
(3)	f(n,u) <= u*fact(n)	définition
(4)	f(0,u) <= u*fact(0)	instanciation
(5)	f(0,u) <= u*1	dépliage
(6)	f(0,u) <= u	propriété de *
(7)	f(n+1,u) <= u*fact(n+1)	instanciation
(8)	f(n+1,u) <= u*((n+1)*fact(n))	dépliage
(9)	f(n+1,u) <= (u*(n+1))*fact(n)	associtivité de *
(10)	f(n+1,u) <= f(n,u*(n+1))	pliage
(11)	fact(n+1) <= f(n,n+1)	

On obtient donc la définition de fact à partir de la fonction f qui est trivialement convertible en itération.

```
f(0,u) <= u
f(n+1,u) <= f(n,u*(n+1))
```

2- A partir de la définition habituelle de la fonction de Fibonacci qui se déroule en un temps exponentiel, on en dérive une solution qui la calcule en un temps linéaire.

(1)	f(0) <= 1	définition
(2)	f(1) <= 1	définition
(3)	f(x+2) <= f(x+1)+f(x)	définition
(4)	g(x) <= <f(x+1),f(x)>	définition
(5)	g(0) <= <f(0+1),f(O)>	instanciation
(6)	g(0) <= <f(1),f(1)>	prop. de +
(7)	g(0) <= <1,1>	dépliage de 1 et 2
(8)	g(x+1) <= <f((x+1)+1),f(x+1)>	instanciation
(9)	g(x+1) <= <f(x+2),f(x+1)>	pro. de +
(10)	g(x+1) <= <f(x+1) +f(x),f(x+1)>	dépliage
(11)	g(x+1) <= <u+v,u> where <u,v> = <f(x+1),f(x)>	abstraction
(12)	g(x+1) <= <u+v,u> where <u,v> = g(x)	pliage
(13)	f(x+2) <= u+v where <u,v> = <f(x+1),f(x)>	abstraction
(14)	f(x+2) <= u+v where <u,v> = g(x)	pliage

d'où la définition de f devient

f(0) <= 1
f(1) <= 1
f(x+2) <= u+v where <u,v> = g(x)

avec

g(0) <= <1,1>
g(x+1) <= <u+v,u> where <u,v> <= g(x)

A propos de cette méthode, on peut remarquer les deux points suivants :

1- comme on a pu le remarquer dans les deux exemples ci-dessus, certaines étapes clefs de la transformation, qui sont d'ailleurs capitales pour la suite des transformations,

nécessitent des trouvailles judicieuses. La généralisation de définitions existantes (étape 3 de l'exemple1), et l'introduction de nouvelles définitions (étape 4 de l'exemple2). On peut dire que la puissance de cette méthode dépend pour une large part de l'utilisateur.

2- c'est une méthode qui a l'inconvénient de n'être que partiellement correcte : il faut prouver la correction d'une transformation ou bien trouver des conditions l'assurant [Kott 80] (nombre de dépliages doit être supérieur ou égal au nombre de pliages).

4.1.4 Notre approche .

Les règles que nous avons données sont basées sur l'identité. Elles constituent un système formel pour la transformation de programmes. Ce système formel a pour vocabulaire les combinateurs ; les termes sont des fonctions et les règles d'inférence sont les règles de transformation. La méthode consiste, une fois qu'un programme est écrit comme une composition de fonctions, à chercher à le transformer en un programme plus "efficace", dans le sens suivant :

- élimination des parcours inutiles des structures .
- élimination des calculs de structures intermédiaires.
- produire des formes itératives,

tout ceci en utilisant les règles d'équivalences sur les opérateurs génériques.

4.1.5 Exemples de transformation.

Exemple1 :

Si on considère l'exemple 2 du § 3.4.2 qui calcule la somme des carrés des n premiers entiers naturels, on doit générer la séquence [n, ...,1], puis la séquence intermédiaire inutile [n^2, ...,4,1], et ensuite parcourir cette même séquence pour faire la somme. Soit à optimiser cette fonction, on obtient :

La fonction booléenne "zero?" teste si l'argument est égal à zéro. "pred" est la fonction

prędecesseur.

som_car == hom[0,+] o alpha[carre] o hom_inv[zero?, psi[id,pred]]

D'après la règle (s.5) on a :

som_car == hom[0,+] o hom_inv[zero? , psi[carre o id,pred]]

D'après la règle (a.1) on a :

== hom[0,+] o hom_inv[zero? , psi[carre,pred]

D'après la règle (s.9) on obtient :

som_car == SI[zero?,C[0],+ o psi[carre,pred]]

La propriété monoide [entier operateur 0,+] étant vérifiée, d'après la règle (s.10) on a :

som_car == !2 o TANTQUE[non o zero? o !1,
psi[pred o !1, + o phi[carre,id]]] o psi[id,C[0]]

Exemple 2 : la fonction factorielle sous toutes ces formes.

Le programme consiste à générer une séquence [n,...,1], ensuite à calculer le produit des éléments.

fact == hom[1,*] o hom_inv[zero?, psi[id,pred]

D'après la règle (s.9), on obtient :

fact == SI[zero?, C[1], * o psi[id,fact o pred]]

La propriété monoide[entier operateurs 1,*] est vérifiée, on déduit d'après la règle (s.10) :

fact == !2 o TANTQUE[non o zero?, psi[pred o !1, *o phi[id,id]]] o psi[id,C[1]]

D'après la règle (c.17) :

fact == !2 o TANTQUE[non o zero?, psi[pred o !1, *]] o psi[id,C[1]]

Exemple 3 : Un calcul sur les vecteurs.

On veut programmer la fonction qui teste si un vecteur donné est ordonné (la fonction d'ordre peut être formelle). La spécification est :

$\forall i,\ 1 \leq i < taille(v),\ val(v,i) <= val(v,succ(i))$
("<=" est l'opérateur inférieur ou égal)

si l'on remplace la quantification par un ET généralisé on a :

$ET_{pour\ i=1,pred(taille(v))}\ (val(v,i) <= val(v,succ(i)))$

Le programme se compose de trois fonctions.

f1 : vecteur[t] -> vecteur[(t,t)]
qui construit les couples d'éléments à tester;

f2 : vecteur[(t,t)] -> vecteur[bool]
qui compare les éléments;

f3 : vecteur[bool] -> bool
qui calcule le ET de toutes les valeurs, d'où, avec

couple : (vecteur[t],i) -> (t,t)
défini par :
couple (v,i) == (val(v,i),val(v,succ(i)))

la formule du programme :
ordonne == iter_haut[vrai, et o psi[!1,!3]] o alpha[<=] o gen[pred o taille,couple]

d'après la règle (v.6), on obtient :
ordonne == iter_haut[vrai, et o psi[!1,!3]] o gen[pred o taille, <= o couple]

d'après la règle (v.5) :
ordonné == !1o TANTQUE[<= o psi[!2,pred o taille o !3],psi[et o psi[!1,!3] o
psi[!1,<= o couple o psi[!3,!2]], succ o !2,!3]] o psi[C[vrai],C[1],id]

d'après les règles (c.1) et (c.8), on obtient :
ordonné == !1o TANTQUE[<= o psi[!2,pred o taille o !3],
psi[et o psi[!1,<= o couple o psi[!3,!2]], succ o !2,!3]] o psi[C[vrai],C[1],id]

Notons, qu'en appliquant la règle (v.1) au lieu de (v.6), on aurait obtenu le même résultat.

Exemple 4 :

Soit la définition de la concaténation sur les séquences définie par :
s1 + s2 == hom[s2,<+](s1)

ou, en notation fonctionnelle, et en "curryfiant" le second argument
$[f,u] o v == f o psi[v,u]
$[+,u] == hom[u,<+]

et la longueur : long : seq[t] -> entier définie par
long == hom[0,+] o alpha[C[1]]

On étudie la fonction long2 :(seq[t],seq[t]) -> entier dont la définition naïve est :
long2(s1,s2) == long(s1 + s2) [Darlington 82]

d'où :
$[long2,!2] == hom[0,+] o alpha[C[1]] o hom[!2,<+]

d'après la règle (s.2) on a :
$[long2,!2] == hom[0,mu[+,C[1]]] o hom[!2,<+]

d'après (s.12), on a :
$[long2,!2] == hom[n2,c2]

Il suffit de calculer n2 et c2 qui satisfont les équations :
(a) n2 == hom[0, mu[+,C[1]]] o !2
(b) c2 o phi[id, hom[0,mu[+,C[1]]]] == hom[0, mu[+,C[1]]] o <+

La définition de hom donne immédiatement la condition suffisante :
c2 == mu[+,C[1]]

Le résultat est donc :

$[long2,!2] == hom[long o !2, mu[+,C[1]]

ou, en réintroduisant les variables

long2(s1,s2) == hom[long(s2), mu[+,C[1]]] (s1)

Exemple 5 :

Soit à simplifier la fonction :

sl : seq[entier] -> entier

telle que :

sl(s) == somme(s) + long(s)

avec :

somme(s) == hom[0,+](s) et long(s) == hom[0, mu[+,C[1]]](s)

d'où la formule :

sl == + o psi[hom[0,+],hom[0,mu[+,C[1]]

on pose plusun == mu[+,C[1]]

D'après la règle de fusion (s.14), on a :

sl == + o hom[+ o psi[0,0],h] où h est tel que :

h == psi[+ o phi[id,!1] , plusun o phi[id, !2]]

D'après la règle (s.12), on a :

sl == hom[c,f]

où :

c == + o psi[0,0] == 0

f o phi[id,+] == + o psi[+ o phi[id,!1] , plusun o phi[id,!2]]

En introduisant pour la lisibilité les variables (x,(y,z)) , on a :

f(x,y+z) == (x+y) + (plusun(x,z))

or plusun(x,z) == succ(z) d'où : f(x,y+z) == (x+y) + succ(z)

A l'aide de l'associativité et des équations définissant le "+" des entiers, on en déduit

f(x,y+z) == succ((x)+(y+z))

Pour que cette identité soit toujours vérifiée, il suffit que f : (entier,entier) -> entier soit défini par :

f(n,m) ==succ(n+m)

D'où la nouvelle définition de sl :

sl == hom[0,f]

qui fait un seul parcours de la séquence paramètre au lieu de deux.

Exemple 6:

Soit à optimiser la fonction de tri (exemple1 §3.4.2) qui est :

tri(s) == hom[nil,parcours] o hom[vide,inserer](s)

où parcours == + o psi[!1,<+ o psi[!2,!3]]

En appliquant la règle (s12), on obtient :

hom[n2,c2] == hom[nil,parcours] o hom[vide,inserer]

Pour que cette identité soit vraie, il faut vérifier les conditions suivantes :

(a) n2 == hom[nil,parcours] o vide

(b) c2 o phi[id,hom[nil,parcours]] == hom[nil,parcours] o inserer

Pour (a), d'après la définition de "hom" on a :

n2 == nil

Pour (b), on introduit des variables pour des raisons de lisibilité :
c2(m2,hom[nil,parcours](a)) == hom[nil,parcours](inserer(m2,a))

En calculant par cas, on a :
c2(m2,hom[nil,parcours](vide)) ==
hom[nil,parcours](inserer(m2,vide))
=> c2(m2,nil) == [m2]

c2(m2,hom[nil,parcours](noeud(a1,m1,a2))) ==
hom[nil,parcours](inserer(m2,noeud(a1,m1,a2)))

=> c2(m2,hom[nil,parcours](a1)+m1<+hom[nil,parcours](a2)) ==
si m2 < m1 alors
hom[nil,parcours](inserer(m2,a1))+m1<+hom[nil,parcours](a2)
sinon hom[nil,parcours](a1)+m1<+hom[nil,parcours](inserer(m2,a2))
fsi

ou c2 : (t,seq[t]) -> seq[t].

Sachant que hom[nil,parcours] est une fonction surjective sur les séquences, on généralise en prenant :
hom[nil,parcours](a1) == s1
hom[nil,parcours](a2) == s2
d'où (c) : c2(m2,s1+m1<+s2) == si m2 < m1
alors c2(m2,s1)+m1<+s2
sinon s1+m1<+c2(m2,s2) fsi

La définition de c2 n'est pas exécutable en LPG puisque "+" n'est pas constructeur sur les séquences. D'autres langages accepteraient ce genre de définitions [Lindstrom 85]. Cette définition signifie que, quelle que soit la décomposition de la séquence en paramètre c2(m2,s) en c2(m2,s1+m1<+s2), alors on a l'égalité (c). Pour obtenir un programme exécutable, il suffirait de définir une fonction de dichotomie des séquences :

d(s) == (s1,m1,s2) $(s \neq nil)$

telle que s = s1+m1<+s2. Le programme exécutable est alors :

```
c2(m2,nil) == [m2]
c2(m2,a<+s) == soit (s1,m1,s2) ≐ d(a<+s)
                dans si m2<m1
                        alors c2(m2,s1)+m1<+s2
                         sinon s1+m1<+c2(m2,s2)
                         fsi
                fsoit
```

tri(s) == hom[nil,c2](s)

4.1.6 Remarques.

On ne sait pas trouver de système de réécriture équivalent au système formel d'équivalence donné pour deux raisons :

- la première raison est technique, on ne peut pas déterminer l'ordre d'orientation des règles de transformation, donc on ne connaît pas la "forme normale" des programmes.

- la seconde raison est d'ordre théorique : certaines règles sont conditionnnelles. Ces conditions sont de trois sortes :

1) le cas où la condition est une propriété (cf. la règle s.11) ; on peut la vérifier soit par simple consultation des tables associées aux propriétés, soit en faisant une preuve à l'aide du démonstrateur automatique OASIS [Barberye 83].

2) le cas où la condition porte sur le type d'une fonction (cf. la règle (e.6)) ; ce cas aussi peut être vérifié en utilisant le mécanisme du contrôle des types.

3) le cas où la condition est une équation, exemple :

(1) n2 == g o n1, (2) c2 o phi[id,g] == g o c1 |-
g o hom[n1,c1] == hom[n2,c2]

l'équation (1) est une définition de la fonction n2, par contre (2) est une équation non simple qui représente une égalité d'ordre sémantique, dont l'inconnue est une fonction. La résolution d'une telle équation (du second ordre) par un système automatique n'est pas évidente.

Si on veut implémenter notre méthode de transformation de programmes, le processus de transformation ne peut être qu'heuristique, ou bien guidé par l'utilisateur.

4.2 Preuve de propriétés sur les programmes

4.2.1 Les méthodes inductives.

Ces méthodes ont le même prinicipe que la récurrence en mathématiques. Si on veut prouver une propriété d'un programme sur un certain domaine, on peut le faire par induction sur le programme (induction dite de point fixe), sur le domaine des données (induction structurelle), ou sur la propriété ([Manna 77]). Cette dernière possibilité est peu utilisée car il faut d'abord que la propriété s'y prête.

4.2.1.1 Induction du point fixe.

Le prinicipe est le suivant ([Park 69][Manna 73]) : pour prouver une propriété P sur une fonction définie par une équation récursive, f = F(f) il suffit de montrer :

P(⊥) et f (P(f) => P(F(f))) (⊥ est la valeur "indéfinie")

On qualifie cette induction de point fixe parce qu'elle est basée sur la méthode de calcul du point fixe qui est la limite de F(⊥). Cette méthode est bien adapté aux langages

fonctionnels puisqu'elle manipule des équations récursives. En réalité elle n'est pas valide pour toute propriété P. On sait cependant que cette méthode est correcte pour les propriétés admissibles, c'est à dire pour les propriétés qui s'écrivent $P(f) = \alpha(f) \subseteq ß(f)$ où α et ß sont des fonctionnelles continues et l'inégalité $\subseteq$ symbolise la relation "est moins définie que". Notons qu'une propriété fausse pour " $\perp$" ne peut évidement pas être montrée par cette méthode même si elle est vraie pour la fonction définie par le point fixe.

Exemple ([Manna 73]) :

$f(x)$ = si $p(x)$ alors x sinon $f(f(h(x)))$

On veut montrer que le point fixe (minimal) de cette fonctionnelle F vérifie fix o fix = fix à l'aide de la propriété suivante :

$p(f) :: fix \circ f = f$

(1) $fix(\perp)(x) = fix(\perp)$

$= $ si $p(\perp)$ alors $\perp$ sinon $fix(fix(h(\perp)))$ (déf. du point fixe)

$= $ si $\perp$ alors $\perp$ sinon $fix(fix(h(\perp)))$ ($p(\perp) = \perp$)

$= \perp$ (déf. de la condition)

$= \perp(x)$ (déf. du point fixe)

donc $fix \circ f = f$

(2) $fix \circ f = f$

$fix(F(f)(x)) = fix($si $p(x)$ alors x sinon $f(f(h(x)))$

$= $ si $p(x)$ alors $fix(x)$ sinon $fix(f(f(h(x))))$

$= $ si $p(x)$ alors x sinon $fix(f(f(h(x))))$ (déf. de fix)

$= $ si $p(x)$ alors x sinon $(f(f(h(x))))$ (hyp. d'induction)

$= F(f(x))$ (déf. de F)

donc $P(f) \Rightarrow P(F(f))$

puisque cette propriété peut se mettre sous la forme :

$$fix \circ f \subseteq f \quad \text{et} \quad f \subseteq fix \circ f$$

Cette méthode permet également de montrer des propriétés de systèmes de fonctions

récursives et en plus de fonctions définies séparement en utilisant l'induction parallèlement sur chacune d'elle. En effet cette méthode est puissante mais des problèmes se posent pour les preuves de terminaison, il est impossible de montrer p(f) :: h $\subseteq$ f, h étant une fonction quelconque, puisque h $\subseteq \perp$ n'est jamais vraie.

4.1.1.2 Induction structurelle [Burstall 69].

C'est une méthode applicable sur un domaine D bien fondé, c'est à dire muni d'une relation d'ordre partiel ne possédant pas de chaîne décroissante infinie.

Le principe de cette méthode est le suivant :
pour montrer une propriété P sur D il suffit de prouver :

$$\forall a \in D\ (\forall b \in D, b<a \Rightarrow P(b)) \Rightarrow P(a)$$

Dans la pratique, on montre P d'une manière inconditionelle pour le plus petit élément et on suppose l'existence d'un élément inférieur pour le cas général.

Exemple ([Manna 73]) :

Le programme suivant calcule la "fonction 91"
f(x) = si x>100 alors x-10 sinon f(f(x+11))

montrons f(x) = 91 pour tout x < 100, on utilise pour cela la relation << suivante :

$y<<x \Leftrightarrow x<y<100$

(1) f(100) = f(f(111)) = f(101) = 91

(2) l'hypothèse est $\forall$ y tel que y<<x, f(y) = 91
autrement dit : $\forall$ y tel que y<<x, f(y) = 91

or f(x) = f(f(x+11))
si x+11>100 alors f(x+11) = x+1
et f(f(x+11)) = f(x+1)

or x+1 << x puisque x<x+1<100 donc f(x+11) = 91 par hyp. et f(x) = 91

Contrairement à la méthode d'induction du point fixe, l'induction structurelle peut être utilisée pour prouver des théorèmes mathématiques généraux, plutôt que juste des propriétés de programmes et permet la preuve de terminaison. Cependant, si on se limite à prouver des propriétés de programmes (plus précisement quand l'ordre bien fondé peut être défini récursivement), Milner [Milner 72] a montré que l'induction structurelle peut être remplacée par l'induction du point fixe.

4.2.2 Méthode d'induction récursive [Mc Carthy 63]

La méthode repose sur le principe suivant : pour prouver l'équivalence de deux fonctions f1 et f2 sur un certain sous-domaine S de D, c'est à dire que f1(x) = f2(x) pour tout x ∈ S, il suffit de trouver une fonctionnelle F telle que :

(1) f1 est un point fixe de F, c'est à dire f1 = F[f1]
(2) f2 est un point fixe de F, c'est à dire f2 = F[f2]
et (3) fix(F)(x) est définie pour tout x ∈ S.

Autrement dit : deux fonctions qui vérifient la même équation récursive sont égales quand elles convergent; ceci permet en fait de montrer l'équivalence de deux fonctions en les réécrivant sous une même forme récursive.

Exemple ([Leroy 74]) :

Soient les fonctions "pred" et "succ" sur les entiers et la propriété P :: succ(pred(x)) = x ; la fonction "pred" étant définie pour x ≠ 0. On définie l'addition sur les entiers par :

plus(x,y) = si y = 0 alors x sinon succ(plus(x,pred(y))

on peut vérifier que plus(x,0) = x ; montrons plus(0,x) = x en considérant deux fonctions f(x) = plus(0,x) et g(x) = x

- f(x) = plus(0,x) = si x = 0 alors 0 sinon succ(plus(0,pred(x)))
= si x = 0 alors 0 sinon succ(f(pred(x))) (par déf. de f)

- g(x) = si x = 0 alors 0 sinon x
= si x = 0 alors 0 sinon succ(pred(x)) (par la prop. P)
= si x = 0 alors 0 sinon succ(g(pred(x)) (par déf. de g)

f et g se mettent donc sous la forme commune :

F(x) = si x = 0 alors 0 sinon succ(F(pred(x))

On a bien l'équivalence entre plus(0,x) = x dans le domaine de convergence de cette définition, qui est l'ensemble des entiers (ce qui reste toutefois à démontrer). C'est donc une méthode naturelle et élégante qui n'utilise pas explicitement l'induction, les seuls problèmes venant du choix de la définition récursive commune, qui n'est pas toujours facile à faire, et de son domaine de convergence qu'il faut calculer d'une autre manière (sinon on peut réaliser des preuves valides sur un domaine vide).

4.2.3.Notre approche : Règles d'inférence associées aux opérateurs génériques.

Par construction, il est possible d'associer des règles d'inférence aux opérateurs génériques. La possibilité de définir des nouvelles règles d'inférence doit aller de pair avec les définitions de types et d'opérateurs. Les langages ou outils de démonstration possèdent en général des moyens pour définir de nouvelles règles et les tactiques associées [Gordon 79] [Barberye 83] [Bergstra 83]; la définition d'opérateurs génériques est un bon moyen d'introduire des tactiques puissantes. Les formules de ces règles sont des formules de logique du premier ordre et du second ordre, et qui tiennent compte de l'égalité. Ces règles ont pour but d'exprimer les relations entre les opérateurs génériques et leurs paramètres fonctions ; les opérateurs formels sont caractérisés le cas échéant, par des propriétés. La structuration de ces règles est guidée par la décomposition de l'univers en différents types.

- **Règles générales.**

(Egalité générique)

$$\frac{f1 == h1, \ldots, fn == hn}{g[f1, \ldots, fn] == g[h1, \ldots, hn]}$$

(Simplification à droite)

$$\frac{f \circ g == h \circ g,\ \text{surjective[t1,t2 operateurs g]}}{f == h}$$

(Simplification à gauche)

$$\frac{g \circ f == g \circ h,\ \text{injective[t1,t2 operateurs g]}}{f == h}$$

- **Règles d'inférence sur les séquences.**

(is1)

$$\frac{P(f(x))\ (\forall x)}{P(S[i] \circ \text{alpha}[f](s))\ \forall i,\ 1 \leq i \leq \#(s)}$$

(is2)

$$\frac{P(nil,n)\ \ P(s,m) \Rightarrow P(a<+s,f(a,m))}{(\forall s)\quad P(s,\text{hom}[n,f](s))}$$

(is3)

$$p(x) == \text{vrai} \Rightarrow P(x,nil),$$
$$\frac{p(x) == \text{faux} \Rightarrow [P(!2(g(x)),s) \Rightarrow P(x,!1(g(x)) <+ s)]}{P(x,\text{hom\_inv}[p,g](x))}$$

(is4)

$$\frac{P(f1(x))\ \forall x}{P(S[i] \circ \text{hom\_inv}[p,\text{psi}[f1,f2]](y))\ \forall i, 1 \leq i \leq \#(\text{hom\_inv}[\ldots](y))}$$

(is5 : égalité1)

$$\frac{\text{vide?(s1)} == \text{vrai, vide?(s2)} == \text{vrai}}{s1 == s2}$$

(is6 : égalité2)

$$\frac{s1 == a1 <+ u1,\ s2 == a2 <+ u2,\ a1 == a2,\ u1 == u2}{s1 == s2}$$

- Règle d'inférence sur les arbres.

On donne seulement la règle associée à l'opérateur d'homomorphisme sur les arbres.

(ia1)

$$\frac{P(\text{vide},n),\quad P(a1,x) \text{ et } P(a2,y) \Rightarrow P(\text{noeud}(a1,r,a2),f(x,r,y))}{(\forall a) \qquad P(a,\text{hom}[n,f](a))}$$

- Règles d'inférence sur les opérateurs de calcul.

(SI)

$$\frac{c(x) == \text{vrai} \Rightarrow P(f1(x)),\ c(x) == \text{faux} \Rightarrow P(f2(x))}{P(SI[c,f1,f2](x))}$$

(TQ)

$$\frac{P(x),\ [P(y) \text{ et } c(y) == \text{vrai}] \Rightarrow P(f(y))}{P(\text{TANTQUE}[c,f](x))}$$

- Règles d'inférence sur les vecteurs.

(iv1)

$$\frac{[1 \leq i \leq \#(v1),\ \#(v1) = \#(v2)] \Rightarrow \text{val}(v1,i) == \text{val}(v2,i)}{v1 == v2}$$

(iv2)

$$\frac{P(x,0)\quad 1\leq i\leq \#(v) \Rightarrow [P(y,pred(i)) \Rightarrow P(g(i,val(v,i),y),i)]}{P(iter\_haut[x,g](v),\#(v))}$$

(iv3)

$$\frac{P(x,\#(v))\quad 0\leq i\leq \#(v) \Rightarrow [P(y,succ(i)) \Rightarrow P(g(succ(i),val(v,succ(i)),y),i)]}{P(iter\_bas[x,g](v),0)}$$

- Règles d'inférence sur les ensembles.

(ie1)

$$\frac{P(n,e\_vide)\quad P(m,e) \Rightarrow P(c(a,m),a<+e)}{(\forall e)\quad P(hom[n,c,=](e),e)}$$

(ie2)

$p(x) == vrai \Rightarrow P(e\_vide,x)$,

$$\frac{p(x) == faux \Rightarrow [P(!2(g(x)),e) \Rightarrow P(x,!1(g(x))<+e)]}{P(x,hom\_inv[p,g,=](x))}$$

4.2.3.1 Application : l'induction structurelle, par la méthode d'homomorphisme.

Nous allons donner une application concrète de la notion d'homomorphisme unique entre l'algèbre initiale (qu'on note $T_{\Sigma,E}$), et toute Σ-algèbre A (soit dans notre cas les algèbres de la catégorie décrite par la propriété exigée par l'opérateur d'homomorphisme). En effet, la présence de l'opérateur d'homomorphisme "hom" permet de démontrer par simple réécriture des théorèmes qui sont en général démontrés par induction sur les termes. D'une façon générale la méthode consiste à :

Etant donné deux fonctions f1,f2 prouver que :

f1(x) == f2(x) dans A $\forall x \in T_{\Sigma,E}$

revient à trouver "hom" tel que :

f1(x) == hom[n1,g1](x)
f2(x) == hom[n2,g2](x)

et (1) n1 == n2, g1 == g2

(2) si on a des équations sur les constructeurs, il faut vérifier ces équations avec <n1,g1>, <n2,g2>.

Remarques :

1- L'opérateur "hom" est surjectif par construction, car on ne s'intéresse qu'à l'algèbre des termes (A) qui a pour éléments les images des constructeurs de l'algèbre initiale ($T\Sigma$,E), par cet opérateur d'homomorphisme "hom".

2- Il n'est pas nécessaire de formuler de prime abord les fonctions sous forme d'homomorphisme. Une meilleure méthode consiste à trouver des compositions d'opérateurs plus simples et appliquer ensuite les règles d'équivalence (cf. chapitre 3) pour synthétiser les opérateurs d'homomorphisme adéquats.

- Justification de cette méthode :

La méthode que nous venons d'énoncer est équivalente à l'induction structurelle sur les termes. En effet :

Soient c1: -> T et f: T -> T les constructeurs du type T ; si on veut démontrer que :

f1(x) == f2(x) $\forall x \in T_{\Sigma,E}$ ceci revient à :

(1) f1(c) == f2(c)

(2) $\lfloor$f1(x0) == f2(x0)$\rfloor$ f1(f(x0)) == f2(f(x0))

on a :

f1(c) == f2(c) <=> n1 == n2, et

$\lfloor$f1(x0) == f2(x0)$\rfloor$ f1(f(x0)) == f2(f(x0)) <=> g1 == g2

preuve :

1- f1(c) == f2(c) => n1 == n2

f1(c) == hom[n1,g1](c)
f2(c) == hom[n2,g2](c)

or f1(c) == f2(c) donc :
hom[n1,g1](c) == hom[n2,g2](c)
en appliquant la définition de "hom" (axiome1) :
n1 == n2

2- n1 == n2 => f1(c) == f2(c)

f1(c) == hom[n1,g1](c)
== n1 (def. de "hom" axiome1)
f2(c) == hom[n2,g2](c)
== n2 (def. de "hom" axiome1)
or n1 == n2 donc :
f1(c) == f2(c) .

3- ⌊f1(x0) == f2(x0)⌋ f1(f(x0)) == f2(f(X0)) => g1 == g2

f1(f(x0)) == hom[n1,g1](f(x0))
== g1(hom[n1,g1](x0) (def. de "hom" axiome2)

f2(f(x0)) == hom[n2,g2](f(x0))
== g2(hom[n2,g2](x0) (def. de "hom" axiome2)

or f1(f(x0)) == f2(f(X0)) donc :
g1(hom[n1,g1](x0)) == g2(hom[n2,g2](x0))

d'après l'hypothèse d'induction f1(x0) == f2(x0) :
d'où : hom[n1,g1](x0) == hom[n2,g2](x0)

g1(hom[n1,g1](x0)) == g2(hom[n1,g1](x0))

or l'opérateur "hom" est surjectif donc:

g1 == g2

4- g1 == g2 => ⌊f1(x0) == f2(x0)⌋ f1(f(x0)) == f2(f(x0))
g1(x) == g2(x)

L'hypothèse dans le membre droit se réécrit :

hom[n1,g1](x0) == hom[n2,g2](x0)

en composant à gauche avec g1 et g2, on obtient :

g1 o hom[n1,g1](x0) == g2 o hom[n2,g2](x0)

D'après la définition du hom (axiome 2) on a:

hom[n1,g1](f(x0)) == hom[n2,g2](f(x0))

donc on a bien :

⌊f1(x0) == f2(x0)⌋ f1(f(x0)) == f2(f(x0))

Ainsi on a bien démontrer l'équivalence entre la méthode d'homomorphisme et l'induction structurelle.

4.2.3.2 Exemples :

Exemple1 :

On veut démontrer le théorème d'associativité de la concaténation soit :

(s1 + s2) + s3 == s1 + (s2 + s3)

par définition :

s1 + s2 == hom[s2,<+](s1)

d'où

(1) (s1 + s2) + s3 == hom[s3,<+] o hom[s2,<+](s1)

(2) s1 + (s2 + s3) == hom[hom[s3,<+](s2),<+](s1)

le membre droit de (1) se réécrit d'après la règle (s.12) en :

hom[n2,c2] == hom[s3,<+] o hom[s2,<+]

n2 == hom[s3,<+](s2) (a)

c2 o phi[id,hom[s3,<+]] == hom[s3,<+] o <+ (b)

(a) est une définition de n2.

(b) est une occurrence de la définition de l'opérateur "hom" (axiome2), donc :

c2 == <+

ainsi :

(s1 + s2) + s3 == hom[hom[s3,<+](s2),<+](s1)

s1 + (s2 + s3) == hom[hom[s3,<+](s2),<+](s1)

on a bien l'égalité entre (1) et (2) puisque :

hom[s3,<+](s2) == hom[s3,<+](s2) (s3 == s3, <+==<+)

et <+ == <+

Exemple2 :

On veut démontrer que :

s + nil == s

où "nil" est la séquence vide, et "+" est la concaténation.

par définition :

(1) s + nil == hom[nil,<+](s)

(2) s == id o s

== hom[nil,<+](s)

on a bien s+nil == s, puisque nil == nil, <+ == <+.

Exemple3 : on veut démontrer que :

rev(s1 + s2) == rev(s2) + rev(s1) (1)

où "rev" est la fonction qui inverse une séquence, par définition on a :

rev : seq[t] -> seq[t]
rev(s) == hom[nil,+>](s)

avec "+>" la fonction qui ajoute un élément à la fin d'une séquence tel que :

+> : (t,seq[t]) -> seq[t]
+>(a,s) == s+[a]

et "+" est la concaténation donc :

(2) rev(s1 + s2) == hom[nil,+>] o hom[s2,<+](s1)

pour le membre droit de (1) on doit avoir une fonction d'homomorphisme en fonction de s1 (de façon à pouvoir la comparer avec (2)), en faisant une définition par cas sur s1 on obtient :

(3) rev(s2)+rev(s1) == hom[hom[nil,+>](s2),+>](s1)

le membre droit de (2) se réécrit d'après la règle (s.12) en :

hom[n2,c2] == hom[nil,+>] o hom[s2,<+]
n2 == hom[nil,+>](s2)(a)
c2 o phi[id,hom[nil,+>]] == hom[nil,+>] o <+ (b)

(a) est une définition de n2.
(b) est une occurrence de la définition de l'opérateur "hom" (axiome2), donc :

c2 == +>

ainsi :

rev(s1 + s2) == hom[hom[nil,+>](s2),+>] (s1)
rev(s2)+rev(s1) == hom[hom[nil,+>](s2),+>](s1)

on a bien l'égalité entre (1) et (2) puisque :

hom[nil,+>](s2) == hom[nil,+>](s2) (nil == nil, +>==+>)
+> == +>

Exemple 4 :

Soit le programme qui calcule la somme des n premiers entiers naturels :

somme(n) == hom[0,+] o hom_inv[zero?,psi[id,pred]](n)

On veut démontrer que : somme(n) == n*(n+1)/2 où "/" est la division entière, ou encore :

hom[0,+] o hom_inv[zero?,psi[id,pred]](n) == n*(n+1)/2

La propriété P est définie par :

P(n,s) :: somme(s) == (n*succ(n))/2
D'après la règle d'inférence (is3) :

(1) zero?(n) == vrai => hom[0,+](nil) == n <= n = 0

(2) zero?(n) == faux
=> [hom[0,+](s) == (pred(n)*succ(pred(n)))/2)
=> (hom[0,+](n<+s) == (n*succ(n))/2)]

hom[0,+](n<+s) ==> n + hom[0,+](s)
==> n + (pred(n) * n)/2
==> (2 * n + (pred(n) * n))/2
==> ((2 + pred(n)) * n)/2
==> (succ(n) * n)/2
==> (n * succ(n))/2

Exemple 5 :

- "iter_haut" est un opérateur d'itération sur les tableaux. Cet opérateur permet de tenir compte de l'indice courant. Par exemple pour calculer le maximum d'un tableau

et le premier indice de ce maximum, on peut écrire (§ 2.6.2) :

v0 : -> (entier,entier)
f : (entier, entier),entier,entier) -> (entier,entier)

avec : v0 ==> (0,0)
f(y,i,j) ==> si j>!1(y) alors (j,i) sinon y fsi

On veut prouver que l'élément obtenu est bien le maximum en utilisant cet opérateur d'itération.

Soit le prédicat P défini de la façon suivante :

$P(x,j) :: (\forall\ i)\ 1 \le i \le j \Rightarrow\ !1(x) \ge val(v,i)$

L'application de la règle d'inférence (iv2) donne :

$(\forall\ i)\ 1 \le i \le 0 \Rightarrow\ !1(v0) \ge val(v,i)$
==> vrai

Pour le cas général on a :

$1 \le i \le \#(v) \Rightarrow [(\forall\ k)\ 1 \le k \le pred(i) \Rightarrow$
$!1(y) \ge val(v,k) \Rightarrow ((\forall\ k) 1 \le k \le i \Rightarrow\ !1(f(i,val(v,i)),y) \ge val(v,k)]$

Par la définition de f ; la dernière inégalité devient :
==> !1(si val(v,i) > !1(y) alors (val(v,i),i) sinon y fsi) ≥ val(v,k)
==> (si val(v,i) > !1(y) alors val(v,i) sinon !1(y) fsi) ≥ val(v,k)

cas1 : val(v,i) > !1(y) == vrai
$\Rightarrow ((\forall\ k)\ 1 \le k \le pred(i) \Rightarrow val(v,k) \le\ !1(y)$ et $!1(y) < val(v,i)) \Rightarrow$
$((\forall\ k)\ 1 \le k \le i \Rightarrow val(v,k) \le val(v,i)$
==> vrai (en décomposant [1,i] en [1;pred(i)],[i,i])

cas2 : val(v,i) > !1(y) == faux

==> ((∀ k) 1 ≤ k ≤ pred(i) => val(v,k) ≤ !1(y) et !1(y) ≤ val(v,i)) =>
((∀ k)1≤ k ≤i => val(v,k) ≤ !1(y))

==> vrai

d'où ∀ i 1 ≤ i ≤ #(v) !1(iter_haut[v0,f](v)) ≥ val(v,i)

Exemple 6 :

Opérateurs définis par homomorphisme :

+ : (nat,nat) -> nat	-- addition
+ : (seq[t],seq[t]) -> seq[t]	-- concaténation
lg : seq[t] -> nat	-- longueur.

Définitions :

(6) lg(s) == hom[0,plusun](s)

(7) s1 + s2 == hom[s2,<+](s1)

(8) n + m == hom[m,succ](n)

Théorème : lg(s1 + s2) == l(s1) + l(s2)

Application des définitions (6) (7) (8) :

hom[0,plusun](hom[s2,<+](s1)) == hom[lg(s2),succ](hom[0,plusun](s1)

On a la forme générale : f1(f2(s1)) == f3(f4(s1))

avec : f1 == hom[0,plusun]
f2 == hom[s2,<+]
f3 == hom[lg(s2),succ]
f4 == hom[0,plusun]

Démonstration :

On pose P(s,m) :: f1(m) == f3(f4(s))

Si l'on prouve les sous-buts :

(1) P(nil,s2) :: f1(s2) == f3(f4(nil))

(2) P(s,m) => P(a<+s,a<+m) :: f1(m) == f3(f4(s)) => f1(a<+m) == f3(f4(a<+s))

alors on aura montré le but :

f1(hom[s2,<+](s)) == f3(f4(s))

qui est bien le théorème à prouver.

Preuve des sous-buts par réécriture :

(1) f1(s2) ==> hom[0,plusun](s2)

f3(f4(nil)) ==> hom[lg(s2),succ](hom[0,plusun](nil))
=(1)=> hom[lg(s2),succ] (0)
=(3)=> lg(s2)
=(6)=> hom[0,plusun](s2)

(2) conséquent : f1(a<+m) ==> hom[0,plusun](a<+m)
=(2)=> plusun(a, hom[0,plusun](m))
=(3)=> succ(hom[0,plusun](m))

f3(f4(a<+s)) ==> hom[lg(s2),succ](hom[0,plusun](a<+s))
=(2)=> hom[lg(s2),succ](plusun(a, hom[0,plusun](s)))
=(5)=> hom[lg(s2),succ](succ(hom[0,plusun](s)))
=(4)=> succ(hom[lg(s2),succ](hom[0,plusun](s)))

et d'après l'antécédent (hypothèse d'induction) : ok.

5. CONCLUSION

Nous avons développé une algèbre de programmes dans un langage typé. Pour cela nous avons utilisé les types abstraits et la généricité avec instanciation explicite. Une telle approche nous a permis d'acquérir les points suivants :

- Il apparaît clairement que le typage apporte une plus grande richesse d'expression à l'algèbre de programmes, grâce en particulier aux opérateurs génériques d'homomorphisme et d'homomorphisme inverse. Ce sont d'ailleurs des opérateurs utiles dans la programmation [Burstall 85].

- Les langages de spécification algébrique, et surtout LPG, sont bien adaptés à la description de l'algèbre de programmes, car ils permettent d'exprimer des conditions sur les opérateurs (par exemple associativité, monoïde, ...) qui sont nécessaires à la validité de certaines règles.

- La possibilité de déclarer de nouveaux types abstraits doit aller de pair avec la possibilité de définir de nouveaux opérateurs génériques. Ces derniers permettent de structurer l'univers de définition ; la programmation est dès lors plus concise, et

davantage guidée par les types : on peut parler de "programmation orientée types".

- L'emploi des règles d'inférence associées aux opérateurs génériques nous donne des méthodes générales, quoique la validité de certaines de ces règles n'ait pas été prouvée.

Une des conséquences de ce travail est la réalisation d'un système de transformation de programmes adapté à l'environnement LPG [Bensalem 85] ; il comprendrait en particulier une fonction de définition des règles de transformation, un mécanisme d'unification des expressions pour déterminer le motif qui peut être transformé, et un interface interactif pour gérer l'enchaînement des transformations réalisées.

Un autre objectif plus ambitieux : étudier l'emploi de techniques basées sur les combinateurs [Turner 82] [Cousineau 85] pour l'évaluation des opérateurs génériques ; par exemple : étant donné des constructeurs et un opérateur d'homomorphisme, remplacer au moyen de combinateurs ces constructeurs par leurs images. A l'instar des combinateurs du langage KRC, on peut prévoir une évaluation paresseuse de l'opérateur homomorphisme inverse, ce qui permettrait de générer des structures infinies.

On sait que l'homomorphisme permet d'engendrer la classe des fonctions récursives primitives [Burstall 85] [Burstall 69b] [Paulson 84]. Une autre étude théorique consisterait à se donner un jeu d'opérateurs (par exemple hom, hom_inv, ...), et à caractériser la classe des fonctions calculables à l'aide de ces opérateurs.
Rappelons que dans une structure libre, on sait générer des fonctions récursives à l'aide de la composition de l'homomorphisme et de l'homorphisme inverse. On peut se poser le problème inverse : étant donné une fonction récursive, peut-on l'exprimer sous forme d'une composition semblable ? L'avantage d'une telle formulation : on ne manipule que des expressions fonctionnelles non récursives.

6. BIBLIOGRAPHIE

[Backus 78] J. Backus. "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs." CACM, vol. 21, 8, pp. 613-641 (1978).

[Backus 81] J. Backus. "The Algebra of Functional Programs : Function Level Reasoning, Linear Equations, and Extended Definitions." LNCS, 107, Springer-Verlag, Heideberg.

[Backus 85] J. Backus. "From Function Level Semantics to Program Transformation and optimization." Mathematical Foundations of Software Development, TAPSOFT Berlin, Vol. 1 CAAP'85, LNCS 185 (1985).

[Barberye 83] G. Barberye, T. Joubert, M. Moufron, E. Paul. "Manuel OASIS". Note technique CNET, NT/PAA/CLC/LSC/959, mai (1983).

[Bensalem 85] S. Bensalem, D. Bert. "Une expérience d'utilisation des types abstraits génériques et des opérateurs de second ordre dans la spécification, la construction et la transformation des programmes". RR. LIFIA-24-IMAG-524, BIGRE N° 45 (1985).

[Bergstra 83] J. A. Bergstra, J. Terlouw. "Standard Model Semantics for DSL, A Data Type Specification Language". Acta Informatica 19, pp. 97-113 (1983).

[Bert 79] D. Bert. " La PROGRAMMATION GENERIQUE : construction de logiciel, spécification algébrique et vérification." Thèse d'Etat, USM Grenoble, (1979).

[Bert 82] D. Bert. "Construction de spécifications en LPG." Compte-Rendu des journées Greco-Groplan, Greco Programmation (1982).

[Bert 83a] D. Bert. "Refinements of Generic Specifications With Algebraic Tools." IFIP'83, ed. REA. MASON, North-Holland, pp. 815-820 (1983) .

[Bert 83b] D. Bert. "Manuel de référence de LPG, version 1.2." RR. 408, IMAG, Grenoble (1983).

[Bert 84a] D. Bert, S. Bensalem. "Spécification et programmation en LPG." RR. 12, LIFIA, Grenoble (1984).

[Bert 84b] D. Bert, S. Bensalem. "Algèbre des opérateurs génériques et transformation de programmes en LPG." RR. 14, LIFIA, Grenoble (1984).

[Birkhoff 35] G. Birkhoff. "On the Structure of Abstract Algebras." Proc. Cambrige Phil. Soc., Vol. 29, pp. 433-454 (1935).

[Boyer 75] R. Boyer, J. Moore. "Proving Theorems About LISP Function." J. ACM, Vol 22, 1, pp. 129-144 (1975).

[Burstall 69a] R. M. Burstall. "Proving Properties of Programs by structural Induction, Computer J., Vol. 12, 1, pp. 41-48 (1969).

[Burstall 69b] R. M. Burstall. "Programs and Their Proofs : an Algebraic Approach". Machine Intelligence 4, pp. 17-43 (1969).

[Burstall 77] R. M. Burstall, J. Darlington. "A Transformation System for developing Recursive Programs." Journal of the ACM, Vol. 24, 1, pp. 44-67 (1977).

[Burstall 79] R. M. Burstall, J. A. Goguen. "The Semantics of Clear, a Specification Language". In Proceeding of the 1979 Copenhague Winter School on Abstract Software Specification, LNCS 86, Springer-Verlag, pp. 292-332 (1980).

[Burstall 80] R. M. Burstall, D. B. Mac Queen, D. T. Sannella. " HOPE : an experimental Applicative Language." Internal Report, CSR-62-80, Univ. of Edinburgh (1980).

[Burstall 84] R. M. Burstall. J. A. Goguen. "Introducing Instiutions". In E. Clarke and D. Kozen (editor), Proceedings, Logics of Programming Workshop, LNCS Vol. 164, Springer-Verlag, pp. 221-256 (1984).

[Burstall 85] R. M. Burstall. "Inductively defined functions." Proc. MFSD 85, Lecture Notes in Computer Science 185, pp. 92-96 (1985).

[Chiarini 80] A. Chiarini. " On FP Languages Combining Forms." SIGPLAN Notices, Vol. 15, 9, pp. 25-27 (1980).

[Darlington 75] J. Darlington. "Applications of program transformation to program synthesis". In Proceedings of International Symposium on Proving and Improving Programs (Arc-et-Senans, France, juillet 1-3). IRIA, Le Chesnay, France, pp. 133-144 (1975).

[Darlington 82] J. Darlington. "Program Tranformation." Functional Programming and its Applications, An Advanced Course, Cambridge University Press, pp. 193-215 (1982).

[Dershowitz 83] N. Dershowitz. "Applications of the Knuth-Bendix Completion Procedure." Laboratory Operation, Aerospace Corporation, Aerospace Report No. ATR-83 (8478) -2, (1983).

[Ehrich 82] H. D. Ehrich. "On the Theory of Specification, Implementation and Prameterization of Abstract Data Types." J. ACM, 29, 1, 1982.

[Goguen 78] J. A. Goguen, J. W. Thatcher, E. G. Wagner. "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types." In R. Yeh (ed.), Current Trends in Programming Methodology, Prentice-Hall, (1978).

[Goguen 80] J. A. Goguen. "How to Prove Inductive Hypothesis Without Induction." Proc. 5th Conf. on Automated Deduction, LNCS, 87, Springer-Verlag, 1980, pp. 356-372.

[Gordon 79] M. J. Gordon, R. Milner, C. P. Wadsworth. " Edinburgh LCF." LNCS 78, Springer-Verlag, 1979.

[Gordon 82] M. J. Gordon. "Representing a Logic in the LCF Metalanguage." In tools and Notions for Program Construction (ed. D. Neel), Cambrige University Press, 1982, pp. 163-185.

[Guttag 78] J. V. Guttag, J. J. Horning. "The Algebraic Specification of Abstract Data Types." Acta Informatica, Vol. 10, 1, 1978, pp. 27-52.

[Guttag 81] J. V. Guttag, J. J. Horning, J. H. Williams. "FP with Data Abstraction and Strong Typing." Proc. of the 1981 Conf. on the Functional Programming Languages and Computer Architecture, ACM, Portsmouth (New Hamsphire), 1981, pp. 25-32.

[Huet 80a] G. Huet, D. Open. "Equations and Rewrite Rules : A survey." Formal Languages : Perspectives and Open Problems (R. Book,ed.), Academic Press, 1980.

[Huet 80b] G. Huet, J.M. Hullot. "Proofs by Induction in Equational Theories with Constructors." 21st IEEE symposium on Foundations of Computer Science, 1980, pp. 96-107.

[Iverson 62] K. E. Iverson. "A Programming Language." Wiley, New-York, 1962.

[Iverson 79] K. E. Iverson. "Operators." ACM Trans. on Programming Languages and Systems, Vol. 1, 2, 1979, pp. 161-176.

[Jacquet 78] P. Jacquet. "Les types génériques : propositions pour un mécanisme d'abstraction dans les langaes de programmation." Thèse de 3eme cycle, USM Grenoble, (1978).

[Kapur 80] D. Kapur. "Towards a Theory of Abstract Data Types." MIT-LCS-TR-237, Cambrige, MA, 1980.

[Kapur 84] D. Kapur, D. R. Musser. " Proof by Consistency." Proceedings of an NSF Workshop on the Rewrite Rule Laboratory, Septembre 6-9, 1983, by J.V. Guttag, D. Kapur, and D.R. Musser Information Systems Laboratory, Report No. 84GEN008, Avril (1984).

[Kieburtz 81] R. B. Kieburtz, J. Shultis. "Transformations of FP Program Schemes." Proc. of the 1981 Conf. on the Functional Programming Languages and Computer Architecture, ACM, Portsmouth (New Hamsphire), 1981,

[Klaeren 84] H. A. Klaeren. "A Constructive Method for Abstract Algebraic Software Specification." TCS, 30, pp. 139-204 (1984).

[Kott 80] L. Kott. "Des substitutions dans les systèmes d'équations algébriques sur le magma. Application aux transformations de programmes et leur correction, thèse, université de Paris7 (1980).

[Lankford 80] D. S. Lankford. "Some Remarks on Inductionless Induction." MTP-11, Louisiana Tech. Univ., (1980).

[Lehmann 77] D. E. Lehmann, M. B. Smyth. "Data Types." Proc. of the 18th Symp. on

Foundations of Computer Science, pp. 7-12 (1977).

[Leroy 74] H. Leroy. "La fiabilité des programmes." Travaux de l'Institut d'Informatique de Namur, N0 3, (1974).

[Lindstrom 85] G. Lindstrom. "Functional Programming and the Logical Variables." 12th ACM symposium on Principles of Programming Languages, Janvier (1985).

[Mc Carthy 63] J. Mc Carthy. "A Basis for a Mathematical Theory of Computation." Computer Programming and Formal Systems, (1963).

[Manna 73] Z. Manna, S. Ness, J. Vuillemin. "Inductive Methods for Proving Properties of Programs, C. ACM, vol 16, 8 , pp. 491-502 (1973).

[Manna 75] Z. Manna, R. Waldinger. "Knowldge and reasoning in program synthesis". Artif. Intell. 6, 2, pp. 175-208 (1975).

[Manna 77] Z. Manna. "A New Approach to Recursive Programs." Perspectives on Computer Science, Anita K. Jones, Academic Press, INC, pp. 103-124 (1977).

[Manna 79] Z. Manna, R. Waldinger. "Synthesis : Dreams => Programs". IEEE Trans. Softw.. Eng. SE-5, 4, pp. 294-328 (1979).

[Morris 80] J. H. Morris, E. Schmidt, P. Wadler. "Experience With an Applicative String Processing Language." Proc. 7th ACM Symposium on Principles of Programming Languages, pp. 32-46 (1980) .

[Musser 80] D. R. Musser. "Proving Inductive Propreties of Abstract Data Types." Proc. 7th ACM Symposium on Principles of Programming Languages, (1980).

[Nordström 81] B. Nordström. " Programming in constructive set theory : some

examples". Proc. of the 1981 Conf. on the Functional Programming Languages and Computer Architecture, ACM, Portsmouth (New Hamsphire), 1981,

[Park 69] D. Park. "Fixpoint induction and Proofs of Program Properties." In Machine Intelligence 5, Meltzer and D. Michie (Eds.), Edinburgh Univ. Press, pp. 59-78 (1969).

[Schmidt 83] D. Schmidt. "A programming Notation for Tactical Reasoning". CSR, 141-83, Univ. of Edinburgh (1983).

[Turner 81] D. A. Turner. "Functional Programming and Proofs of Program Correctness." Internal Report, Univ. of Kent, Canterbury, England (1981).

[Turner 82] D. A. Turner. "Recursion Equations as a Programming Language". in Functional Programming and Its Applications, ed. Darlington et al., CUP 1982.

[Von Henke 75] F. W Von Henke. "On generating programs from data types : an approach to Automatic Programming." Colloques IRIA, Proving and Improving Programs, pp. 57-69 (1975).

[Wadler 81] P. Wadler. "Applicative Style Programming, Program Transformation, and list Operators." Proc. of the 1981 Conf. on the Functional Programming Languages and Computer Architecture, ACM, Portsmouth (New Hamsphire), 1981, pp. 11-24.

[Williams 82] J. Williams. "On the Development of the Algebra of Functional Programs." ACM TOPLAS, Vol. 4, 4, 1982, pp. 733-757.

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974

VU le rapport de présentation de Monsieur D. BERT, Chargé de recherche

Monsieur BENSALEM Saddek

est autorisé à présenter une thèse en soutenance en vue de l'obtention du titre de DOCTEUR de TROISIEME CYCLE, spécialité "Informatique".

Fait à Grenoble, le 6 décembre 1985

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,

RESUME :

Dans cette thèse, nous présentons un cadre qui associe la spécification algébrique de types à l'algèbre de programmes. La principale caractéristique de notre approche est fournie par les opérateurs génériques définissables par les utilisateurs qui donnent une grande puissance d'expression aux règles d'équivalence. En particulier, la structure de certains types est contenue implicitement dans des opérateurs génériques comme l'homomorphisme et l'homomorphisme "inverse". Les applications de cette algèbre de programmes typés incluent la preuve de programmes sans induction explicite, et les méthodes de transformation de programmes comme le "folding - unfolding".

Mots-clés :

types abstraits, généricité, langages fonctionnels, opérateurs génériques, algèbre de programmes, méthodes de preuves et de transformations de programmes.