La Programmation générique : construction de logiciel, spécification algébrique et vérification

Didier Bert

▶ **To cite this version:**

Didier Bert. La Programmation générique : construction de logiciel, spécification algébrique et vérification. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1979. tel-00289691

HAL Id: tel-00289691
https://theses.hal.science/tel-00289691

Submitted on 23 Jun 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée à

Université Scientifique et Médicale de Grenoble

pour obtenir le grade de
DOCTEUR ES- SCIENCES
Informatique

par

Didier BERT

LA PROGRAMMATION GENERIQUE

CONSTRUCTION DE LOGICIEL, SPECIFICATION ALGEBRIQUE
ET VERIFICATION.

Thèse soutenue le 26 Juin 1979 devant la commission d'examen

L.	BOLLIET	Président
C.	BOITET	
R.	BURSTALL	
P.	JORRAND	Examinateurs
M.	NIVAT	
C.	PAIR	

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

année scolaire 1980-1981

Président de l'Université : M. J.J. PAYAN

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS DE 1ère CLASSE

Mlle	AGNIUS DELORD Claudine	Biophysique
	ALARY Josette	Chimie analytique
MM.	AMBLARD Pierre	Clinique dermatologie
	AMBROISE THOMAS Pierre	Parasitologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	Physique nucléaire
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale A
	BEAUDOING André	Clinique pédiatrie et puériculture
	BELORISKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
Mme	BERIEL Hélène	Pharmacodynamie
M.	BERNARD Alain	Mathématiques pures
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZES Henri	Clinique chirurgicale & traumatologie
	BILLET Jean	Géographie
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET EYMARD Joseph	Clinique Hépato-gastro-entérologie
Mme	BONNIER Jane-Marie	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHET Yves	Anatomie
	BOUCHEZ Robert	Physique nucléaire
	BRAVARD Yves	Géographie

.../...

MM.	BUTEL Jean	Orthopédie
	CABANEL Guy	Clinique rhumatologie et hydrologie
	CARLIER Georges	Biologie végétale
	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHARACHON Robert	Clinique O.R.L.
	CHATEAU Robert	Clinique neurologique
	CHIBON Pierre	Biologie animale
	COEUR André	Chimie analytique et bromotologique
	COUDERC Pierre	Anatomie pathologique
	CRABBE Pierre	C.E.R.M.O.
	DAUMAS Max	Géographie
	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELOBEL Claude	M.I.A.G.
	DELORMAS Pierre	Pneumo-phtisiologique
	DENIS Bernard	Clinique cardiologique
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Electrochimie
	DODU Jacques	Mécanique appliquée IUT 1
	DOLIQUE Jean-Michel	Physique des plasmas
	DUCROS Pierre	Cristallographie
	FONTAINE Jean-Marc	Mathématiques pures
	GAGNAIRE Didier	Chimie physique
	GASTINEL Noël	Analyse numérique
	GAVEND Jean-Michel	Pharmacologie
	GEINDRE Michel	Electro-radiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	JEANNIN Charles	Pharmacie galénique
	JOLY Jean-René	Mathématiques pures
	KAHANE André	Physique
	KAHANE Josette	Physique
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	LACAZE Albert	Hermodynamique
	LACHARME Jean	Biologie cellulaire
	LAJZEROWICZ Joseph	Physique

.../...

Mme	LAJZEROWICZ Jeannine	Physique
MM.	LATREILLE René	Chirurgie thoracique
	LATURAZE Jean	Biochimie pharmaceutiques
	LAURENT Pierre	Mathématiques appliquées
	LE NOC Pierre	Bactériologie virologie
	LLIBOUTRY Louis	Géophysique
	LOISEAUX Jean-Marie	Sciences nucléaires
	LOUP Jean	Géographie
	LUU DUC Cuong	Chimie générale et minérale
	MALINAS Yves	Clinique obstétricale
Mlle	MARIOTTE Anne-Marie	Pharmacognostie
MM.	MAYNARD Roger	Physique du solide
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	NEGRE Robert	Mécanique IUT 1
	MOZIERES Philippe	Spectrométrie physique
	OMONT Alain	Astrophysique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY PEYROULA Jean-Claude	Physique
	PERRET Jean	Sémeiologie médicale (neurologie)
	PERRIER Guy	Géophysique
	PIERRARD Jean-Marie	Mécanique
	RACHAIL Michel	Clinique médicale B
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
Mme	RENAUDET Jacqueline	Bactériologie
M.	REVOL Michel	Urologie
Mme	RINAUDO Marguerite	Chimie CERMAV
MM.	DE ROUGEMONT Jacques	Neuro-chirurgie
	SARRAZIN Roger	Clinique chirurgicale B
Mme	SEIGLE MURANDI Françoise	Botanique et crytogamie
MM.	SENGEL Philippe	Biologie animale
	SIBILLE Robert	Construction mécanique IUT 1
	SOUTIF Michel	Physique
	TANCHE Maurice	Physiologie
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire

.../...

MM.	VAN CUTSEM Bernard	Mathématiques appliquées
	VAUQUOIS Bernard	Mathématiques appliquées
	VERAIN Alice	Pharmacie galénique
	VERAIN André	Biophysique
	VIGNAIS Pierre	Biochimie médicale

PROFESSEURS DE 2ème CLASSE

MM.	ARNAUD Yves	Chimie IUT 1
	AURIAULT Jean-Louis	Mécanique IUT 1
	BEGUIN Claude	Chimie organique
	BOITET Christian	Mathématiques appliquées
	BOUTHINON Michel	E.E.A. IUT 1
	BRUGEL Lucien	Energétique IUT 1
	BUISSON Roger	Physique IUT 1
	CASTAING Bernard	Physique
	CHARDON Michel	Géographie
	CHEHIKIAN Alain	E.E.A. IUT 1
	COHEN Henri	Mathématiques pures
	COHENADDAD Jean-Pierre	Physique
	COLIN DE VERDIERE Yves	Mathématiques pures
	CONTE René	Physique IUT 1
	CYROT Michel	Physique du solide
	DEPASSEL Roger	Mécanique des fluides
	DOUCE Roland	Physiologie végétale
	DUFRESNOY Alain	Mathématiques pures
	GASPARD François	Physique
	GAUTRON René	Chimie
	GIDON Maurice	Géologie
	GIGNOUX Claude	Sciences nucléaires
	GLENAT René	Chimie organique
	GOSSE Jean-Pierre	E.E.A. IUT 1
	GROS Yves	Physique IUT 1
	GUITTON Jacques	Chimie
	HACQUES Gérard	Mathématiques appliquées
	HERBIN Jacky	Géographie
	HICTER Pierre	Chimie
	IDELMAN Simon	Physiologie animale
	JOSELEAU Jean-Paul	Biochimie
	JULLIEN Pierre	Mathématiques appliquées
	KERCKOVE Claude	Géologie

.../...

MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gérard	Physique IUT 1
	KUPKA Yvon	Mathématiques pures
	LUNA Domingo	Mathématiques pures
	MACHE Régis	Physiologie végétale
	MARECHAL Jean	Mécanique
	MICHOULIER Jean	Physique IUT 1
Mme	MINIER Colette	Physique IUT 1
MM.	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique IUT 1
	OUDET Bruno	Mathématiques appliquées
	PEFFEN René	Métallurgie IUT 1
	PELMONT Jean	Biochimie
	PERRAUD Robert	Chimie IUT 1
	PERRIAUX Jean-Jacques	Géologie minéralogie
	PERRIN Claude	Sciences nucléaires
	PFISTER Jean-Claude	Physique du solide
	PIERRE Jean-Louis	Chimie organique
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	Mathématiques appliquées
	RICHARD Lucien	Biologie végétale
	ROBERT Gilles	Mathématiques pures
	ROBERT Jean-Bernard	Chimie physique
	ROSSI André	Physiologie végétale
	SAKAROVITCH Michel	Mathématiques appliquées
	SARROT REYNAUD Jean	Géologie
	SAXOD Raymond	Biologie animale
Mme	SOUTIF Jeanne	Physique
MM.	STUTZ Pierre	Mécanique
	VIALON Pierre	Géologie
	VIDAL Michel	Chimie organique
	VIVIAN Robert	Géographie

CHARGES D'ENSEIGNEMENT PHARMACIE

MM.	ROCHAS Jacques	Hygiène et hydrologie
	DEMENGE Pierre	Pharmacodynamie

PROFESSEURS SANS CHAIRE (médecine)

M.	BARGE Michel	Neuro-chirurgie

.../...

MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie
	CHAMBAZ Edmond	Biochimie (hormonologie)
	CHAMPETIER Jean	Anatomie
	COLOMB Maurice	Biochimie
	COULOMB Max	Radiologie
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	GROULADE Joseph	Biochimie A
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Gérontologie
	JALBERT Pierre	Histologie
	MAGNIN Robert	Hygiène
	PHELIP Xavier	Rhumatologie
	REYMOND Jean-Charles	Chirurgie générale
	STIEGLITZ Paul	Anesthésiologie
	VROUSOS Constantin	Radiothérapie

MAITRES DE CONFERENCES AGREGES (médecine)

MM.	BACHELOT Yvan	Endocrinologie
	BENABID Alim Louis	Médecine et chirurgie
	BERNARD Pierre	Gynécologie obstétrique
	CONTAMIN Charles	Chirurgie thoracique
	CORDONNIER Daniel	Néphrologie
	CROUZET Guy	Radiologie
	DEBRU Jean-Luc	Médecine interne
	DYON Jean-François	Chirurgie infantile
	FAURE Claude	Anatomie et organogènèse
	FAURE Gilbert	Urologie
	FLOYRAC Roger	Biophysique
	FOURNET Jacques	Hépato-gastro-entérologie
	GAUTIER Robert	Chirurgie générale
	GIRARDET Pierre	Anesthésiologie
	GUIDICELLI Henri	Chirurgie générale
	GUIGNIER Michel	Thérapeutique (réanimation)
	JUNIEN-LAVILLAUROY Claude	Clinique O.R.L.
	KOLODIE Lucien	Hématologie biologique
	MALLION Jean-Michel	Médecine du travail
	MASSOT Christian	Médecine interne
	MOUILLON Michel	Ophtalmologie

.../...

MM.	PARAMELLE Bernard	Pneumologie
	RACINET Claude	Gynécologie-Obstétrique
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
	SCHAEFER René	Cancérologie
	SEIGNEURIN Jean-Marie	Bactériologie-virologie
	SOTTO Jean-Jacques	Hématologie
	STOEBNER Pierre	Anatomie-pathologique

Il est une tradition à laquelle je me soumets avec plaisir, c'est celle de remercier ceux qui à des degrés divers ont participé à la création de cette thèse.

En premier lieu je veux citer les membres du jury.

Monsieur Louis Bolliet a bien voulu le présider ; je n'oublie pas qu'il m'a conseillé d'entrer au CNRS et de persévérer dans la recherche.

Monsieur le professeur Rod Burstall me fait l'amitié de venir d'Edinburgh ; j'éprouve une grande admiration pour son œuvre scientifique, d'autant plus que nos sujets de recherche se sont singulièrement rapprochés depuis deux ans, comme en témoigne cette thèse.

Monsieur le professeur Maurice Nivat est un observateur attentif de mon travail et je suis sensible à l'intérêt qu'il lui porte.

Christian Boitet a bien voulu examiner cette thèse avec un autre regard et les préoccupations

qui sont les siennes.

Je remercie monsieur le professeur Claude Pair à plus d'un titre ; dans chaque rencontre que j'ai eue avec lui depuis des années, j'ai pu apprécier la grande clarté de ses explications et la profondeur de son analyse ; ensuite les travaux de son équipe de Nancy n'ont pas été sans m'intéresser ni m'influencer ; enfin, et ce n'est pas le moindre, il a analysé la première version de cette thèse et m'a suggéré des modifications pertinentes.

J'aurais beaucoup à dire pour remercier Philippe Jorrand qui a été depuis 1971 à mes côtés, soit au centre scientifique IBM, soit à l'IMAG. Je lui dois le thème de mon travail et, au delà des conseils, des remarques et autres discussions, une certaine façon d'aborder les problèmes scientifiques, et pourquoi ne pas le dire, ceux de la vie.

Je ne veux pas oublier J.C. Boussard qui a été mon professeur à Grenoble et qui aurait dû faire partie de ce jury.

Enfin je dois citer mes collègues P. Jacquet et R. Soler qui m'entourent dans le travail d'équipe que j'ai entrepris.

Je ne voudrais pas terminer sans remercier aussi Mme Diaz, Geneviève Boulesteix, Mlle Naudin qui ont dactylographié cette thèse dans des délais record.

Didier Bert.

LA PROGRAMMATION GENERIQUE :

CONSTRUCTION DE LOGICIEL , SPECIFICATION ALGEBRIQUE ET VERIFICATION

0 - INTRODUCTION ---------- 4

1 - PROGRAMMATION GENERIQUE

1.1. Mécanismes d'abstraction

1.1.1. Abstraction fonctionnelle ---------- 9
1.1.2. Abstraction des types ---------- 10
1.1.3. Généricité des types ---------- 11
1.1.4. Généricité fonctionnelle ---------- 12

1.2. Outils de programmation

1.2.1. Modules de définition ---------- 14
1.2.2. Types génériques ---------- 18
1.2.3. Modules de représentation ---------- 19
1.2.4. Les propriétés ---------- 21
1.2.5. Types génériques et propriétés ---------- 24

1.3. Structures de contrôle programmables

1.3.1. Le traitement itératif ---------- 28
1.3.2. Actions d'itération ---------- 31
1.3.3. Décomposition d'itérateurs ---------- 34
1.3.4. Vérification des compositions d'itérateurs ---------- 43
1.3.5. Itérateurs multi-séquences ---------- 45

1.4. Méthodologie de programmation

1.4.1. Bibliothèque de types et procédures ---------- 48
1.4.2. Construction modulaire descendante ---------- 49
1.4.3. Abstraction d'algorithmes ---------- 50

2 - SPECIFICATION DES TYPES GENERIQUES

2.1. Modèle axiomatique

2.1.1. Présentation du modèle ---------- 5
2.1.2. Limites du modèle axiomatique ---------- 5

2.2. Modèle algébrique

2.2.1. Fondements mathématiques ---------- 6(
2.2.2. Spécification algébrique des types ---------- 6⁴
2.2.2.1. approche ADJ, Zilles ---------- 6⁴
2.2.2.2. approche de Guttag ---------- 7⁴
2.2.2.3. représentation d'un type abstrait ---------- 7⁹
2.2.2.4. synthèse de représentation d'un type abstrait ---- 9²
2.2.3. Spécification algébrique des types génériques
2.2.3.1. propriétés ---------- 9⁸
2.2.3.2. propriétés attachées à un type ---------- 10(
2.2.3.3. type abstrait paramétré ---------- 101
2.2.3.4. traitement des erreurs ---------- 111
2.2.3.5. exemples ---------- 115
2.2.4. Sémantique des fonctions génériques
2.2.4.1. spécification des fonctions par les constructeurs- 129
2.2.4.2. spécification des fonctions par leurs effets ----- 132
2.2.4.3. fonctions de types abstraits ---------- 134
2.2.4.4. actions répétitives ---------- 140
2.2.5. Etude de quelques propriétés
2.2.5.1. propriété scalaire ---------- 145
2.2.5.2. propriété égalité ---------- 146
2.2.5.3. propriété affectation ---------- 147
2.2.6. Equivalence des types algébriques et des clauses de Horn
2.2.6.1. syntaxe sémantique des clauses de Horn ---------- 149
2.2.6.2. inclusion des clauses de Horn dans les types algébriques ---------- 151
2.2.6.3. inclusion des types algébriques dans les clauses de Horn ---------- 154

3 - VERIFICATION DE PROGRAMMES SPECIFIES PAR EQUATIONS

3.1. Vérification ou déduction de programmes?

3.1.1. Position du problème de la vérification ---------- 156
3.1.2. Approche par constructeurs pré-spécifiés ---------- 156
3.1.3. Approche par affinements successifs ---------- 157
3.1.4. Classification des équations ---------- 160

3.2. Vérification à l'aide des assertions

3.2.1. Méthode générale ---------- 164
3.2.2. Application aux équations spécifiques ---------- 170
3.2.3. Application aux équations de propriétés ---------- 173

3.3. Autres méthodes de vérification

3.3.1. Assertions par résolution d'équations de programmes ------ 180
3.3.2. Comparaison de systèmes d'équations ---------- 183
3.3.3. Equivalence de programmes ---------- 187

4 - CONCLUSION ---------- 190

0 - INTRODUCTION

La programmation peut être considérée comme l'art de traduire l'énoncé d'un problème en programme [PAI78]. Dans cette traduction, les difficultés surgissent aux deux extrémités du processus : d'une part, il n'est pas simple de définir un énoncé ou une spécification d'un problème ; d'autre part, les langages de programmation savent bien refléter les caractéristiques logiques d'un ordinateur (mémoire, fichiers, modification d'un état), mais ne facilitent généralement pas l'expression des spécifications. De toutes façons, il ne semble pas qu'il y ait à l'heure actuelle (ni peut-être un jour, mais soyons prudents dans les prophéties) de formalisme qui permette à la fois de spécifier un problème et de l'exécuter "efficacement" sur ordinateur. Il est en effet bien connu que les langages de spécification que sont les langages mathématiques ne sont pas constructifs, cependant ils expriment des relations ou des propriétés statiques que l'esprit humain peut facilement appréhender. Par contre, les langages algorithmiques sont constructifs, mais ils expriment des modifications dynamiques d'états ; comme de plus, ces modifications sont très primitives, les programmes sont forcément longs et incompréhensibles. De nombreux travaux ont été réalisés ou sont en cours pour faciliter l'activité de programmation. On distinguera trois courants afin de situer le travail présenté dans cette thèse et de montrer dans quelle mesure il utilise ce qui a été fait et sur quels plans il essaie de l'améliorer.

- Les langages de spécification

Il existe un certain nombre de langages qui permettent à la fois de spécifier un problème à l'aide d'un formalisme particulier et, quelquefois moyennant certaines précautions, d'exécuter un programme utilisant ces spécifications. Voici une liste (non exhaustive) qui donne pour chacun de ces langages les notions mathématiques sous-jacentes.

LISP pur	λ-calcul, fonctions récursives
SETL	Théorie des ensembles
PROLOG	Clauses de Horn
LUCID	Suites numériques
CLEAR	Théories algébriques

- Les recherches méthodologiques

Ces recherches visent à donner une méthode pour construire et vérifier les programmes. La plupart d'entre elles s'appuient sur un langage de programmation particulier ou sur des extensions de langages existants. Ce courant s'est développé à partir de l'idée de la "programmation structurée" ou de la programmation descendante, qui n'est autre que la mise en application du proverbe : "diviser pour régner", ou plus philosophiquement, de la méthode d'analyse et de synthèse de Descartes [Discours de la Méthode, 1637]. L'apport le plus intéressant est celui des méthodes qui facilitent à la fois la construction des programmes et leur preuve. On peut citer :

- Spécification axiomatique (la littérature étant trop abondante, on ne cite que les articles de base [FLO67] [HOA69]) et synthèse de programmes (par exemple [DER75]).
- Ecriture systématique des programmes [WIR73][WIR76].
- Déduction de programmes d'après les assertions de sortie [DIJ76][HUC77].
- Déduction de programmes par transformations [ARS77].

- Les recherches en langages algorithmiques

Ce dernier courant, dans lequel je me situerai, étudie la manière dont les langages peuvent être adaptés pour que les programmes soient exprimés dans un formalisme qui soit le plus proche possible du problème, mais restent néanmoins algorithmiques et puissent être compilés avec les techniques habituelles. Il prend sa source avec SIMULA 67 et ALGOL 68, puis se développe avec les langages extensibles (sur lesquels je fais mes premières armes [JOR72][BER73]). Le grand tournant est celui de l'introduction de l'abstraction des types avec CLU [LIS75] puis ALPHARD [WUL76]. Aux avantages de l'extensibilité s'ajoutent ceux de l'abstraction, et de la sécurité, par oubli de la construction des nouveaux types.

Partant du principe que l'abstraction ne pouvait être que générique, c'est-à-dire accepter des types en paramètres, à condition que ces derniers soient déterminés à la compilation [BER77a], P.Jacquet et moi-même avons essayé dans

un premier temps de résoudre les problèmes de définition et de compilation que cela pouvait poser. Cela nous a conduits à introduire dès le début la notion de propriété, qui est un élément de résolution de ces problèmes et s'insère bien, de par sa nature syntaxique, au langage de types abstraits [BER78]. Cependant, le langage algorithmique auquel nous avons abouti, s'il est satisfaisant pour la construction de types et de programmes génériques, ne permet pas parfaitement de spécifier les types et les propriétés. A la suite des travaux de Guttag et de l'équipe ADJ, nous avons introduit la spécification algébrique dans le langage, de telle manière qu'elle ne soit qu'un commentaire pour le compilateur, mais qu'elle puisse servir à un démonstrateur qui serait associé au compilateur. De plus, il est possible (et souhaitable) d'utiliser la spécification algébrique pour construire les programmes ; c'est en ce sens que l'ensemble du projet apporte sa contribution aux recherches méthodologiques.

Avant de passer à la présentation des chapitres, on peut noter l'importance prise par la généricité dans les langages finalistes du projet du Département de la Défense Américaine[GRE79] [RED79], qui ont intégré dans un produit à caractère industriel les principaux éléments parus sur le sujet depuis ces dernières années.

Les différents chapitres de la thèse développent chacun un des aspects qui viennent d'être évoqués :

- le langage de programmation générique (ch.1),
- la spécification des types et des fonctions génériques (ch.2),
- les relations entre spécification et programme (ch.3).

Le chapitre 1 comprend une étude de l'abstraction et de la généricité dans les langages de programmation (§1.1.). Il décrit ensuite les principaux outils de la programmation générique, à savoir :

- la définition des régions génériques, c'est-à-dire paramétrées par des types ;
- les définitions syntaxiques des types ;
- les modules génériques de représentation des types, qui donnent l'implantation d'un objet d'un type abstrait et le corps des fonctions d'accès ;
- la spécification syntaxique des types formels (c'est la notion de propriété).

Cette présentation des constituants du langage générique (§1.2.) est à compléter par l'étude des problèmes de compilation exposés en [BER78][JAC78a] que nous n'aborderons pas dans cette thèse.

Au §1.3., on étudie les structures de contrôle associées aux types abstraits (actions d'itération) et on en donne une généralisation avec les conséquences de méthode et de vérification que cela entraîne. Enfin, le § 1.4. (pompeusement intitulé "méthodologie") est consacré aux applications possibles de la programmation générique.

Le chapitre 2, de loin le plus important, traite de la spécification des types et des fonctions. Le souci de spécification est apparu avec le désir de vérifier que des programmes sont "corrects", c'est-à-dire sont cohérents avec des spécifications exprimées dans un langage non algorithmique. Les langages de spécification des abstractions sont essentiellement les systèmes formels comme la logique du premier ordre (spécification axiomatique) ou les systèmes équationnels (spécification algébrique). On essaie de faire le point entre ces deux méthodes de spécification, notamment dans le cas des types et des fonctions associées à ces types. La présentation de la méthode axiomatique est faite rapidement au §2.1.1. La méthode algébrique est présentée au § 2.2.2. et une comparaison est faite entre l'approche algébrique de Guttag et celle de ADJ. L'application de la spécification algébrique aux types génériques est faite au §2.2.3. On traite ensuite au §2.2.4. de la spécification des fonctions génériques et des itérateurs qui ont été introduits au chapitre 1. Le paragraphe suivant est consacré à l'étude de quelques propriétés importantes en informatique et dans la sémantique des langages de

programmation, comme l'égalité et l'affectation. Enfin, on établit l'équivalence entre le système formel algébrique des types abstraits et celui des clauses de Horn en logique du premier ordre, §2.2.6.

Le chapitre 3 est consacré à l'étude des méthodes de vérification de programmes spécifiés algébriquement. Le sujet étant particulièrement difficile, on s'est contenté de donner quelques clés de solution. Le §3.1. aborde le sujet de la déduction des programmes à partir des spécifications et précise le cadre des vérifications que l'on peut faire. Le point important de ces vérifications est de passer d'une spécification opérationnelle (le programme) à une spécification statique ou dénotationnelle qui servira de base pour la comparaison avec la spécification algébrique. Le §3.2. propose de donner une spécification axiomatique du programme, puis d'utiliser cette spécification dans la vérification algébrique. Cette démarche est assez bonne et systématique, mais elle est assez lourde puisqu'elle exige deux sortes de spécifications et de vérifications. Pour pallier à cet inconvénient, on peut essayer (§3.3.) de remplacer le programme par un système d'équations de prédicats sur les états du programme. Si l'on sait résoudre ce système de prédicats, on obtient en particulier les assertions initiales et finales qui peuvent être utilisées pour valider les spécifications algébriques. Lorsqu'on ne sait pas résoudre exactement le système d'équations de prédicats, on peut essayer de comparer ce système lui-même avec la spécification algébrique mise sous forme adéquate. Enfin on discute de l'approche opposée qui consiste à substituer au système algébrique un système d'équivalence entre programmes qui peut être résolu par des techniques appropriées.

Dans la conclusion, on dégagera les lignes de travail qui doivent prolonger cette thèse et sont déjà définies, à savoir l'implantation expérimentale d'un langage générique et des recherches plus approfondies sur la sémantique des programmes génériques et sur les vérifications.

1. PROGRAMMATION GENERIQUE

1.1 - Mécanismes d'abstraction

Le principe de l'abstraction est (avec la composition fonctionnelle) l'un des points importants de l'écriture des programmes. Dès l'origine de la conception des langages de programmation, on voit apparaître la notion de procédure ou de fonction, qui permet, comme dans les lambda-expressions en LISP, de considérer des variables en tant que paramètres de l'abstraction fonctionnelle. Lors d'un appel particulier d'une procédure, il y a liaison entre les paramètres effectifs et les paramètres formels pour reconstituer une séquence d'instructions exécutables dans un contexte donné. La complexité et les différences entre les divers mécanismes de liaison des paramètres sont imputables à la définition (sémantique) du langage de programmation considéré. On rappelle ici les vertus de l'abstraction du point de vue de la conception des programmes, en suivant les idées exposées dans [JAC 78a] [JAC 78b].

1.1.1 - Abstraction fonctionnelle

L'abstraction fonctionnelle permet essentiellement de définir une action et de pouvoir la nommer ; c'est l'aspect abstraction proprement dit. On peut, de cette manière, oublier la façon dont se réalise l'action, par composition de petites actions élémentaires, pour ne retenir que la spécification de l'action au plus haut niveau. L'oubli de la définition concrète d'une procédure (la composition des instructions) au profit d'une définition abstraite (son effet global) est un apport essentiel pour la programmation, qui a été mis en évidence par la programmation "structurée", car il permet :

- une écriture et une lecture facile des programmes ;
- une vérification modulaire (à condition que certaines règles soient respectées) ;

- une indépendance entre la définition de la procédure et son utilisation (modification possible de la première sans avoir à changer la seconde, si on conserve le même effet).

Le deuxième aspect de l'abstraction fonctionnelle est la paramétrisation, c'est-à-dire le fait qu'un certain nombre de paramètres dans la procédure ne possèderont une valeur qu'au moment d'un appel particulier. Le résultat de la procédure dépend donc de ces valeurs. Comme les programmes intéressants sont toujours des programmes paramétrés par les données d'entrée qui sont lues pendant l'exécution du programme, on a assez facilement assimilé les procédures aux programmes, les "variables" jouant le rôle de paramètres pour les valeurs. Cet aspect des choses se retrouve dans la formalisation de la notion d'état de programme en sémantique dénotationnelle qui associe à chaque variable une valeur (les variables étant prises au sens étroit d'emplacement mémoire). Cette confusion entre l'idée de programme et celle de procédure, entre la notion de variable et celle de paramètre, a longtemps empêché d'étendre la notion de paramétrisation, et par là-même, de l'abstraction.

1.1.2 - Abstraction des types

L'idée de considérer les types comme "abstraction" est relativement récente ; elle est matérialisée par le terme de "type abstrait" [LIS 74]. Traditionnellement, on disait que les types servaient à trois choses :

- représenter des objets,
- spécifier des domaines de valeurs de variables,
- spécifier des domaines de paramètres et résultat de fonction.

Mais on s'est aperçu qu'ils n'étaient pas seulement des ensembles [MOR73] ; d'autre part, la nécessité de s'abstraire de la représentation est apparue pour les mêmes raisons que dans le cas de l'abstraction fonctionnelle. Donc, la seule chose qui était vraiment caractéristique d'un

type était les fonctions qui utilisaient ou qui retournaient des valeurs d'un type. L'apport des spécifications algébriques, dont nous reparlerons longuement (paragraphe 2.2) a été déterminant sur ce point.

Pour la programmation, la conséquence en est qu'une valeur d'un type donné ne peut être construite et utilisée qu'au travers d'un certain nombre de fonctions qui assurent l'abstraction de la représentation. On ne connaît donc pas les composants "concrets" d'un type et cela est à la base de la protection, donc de la fiabilité des programmes [LAM77] [DOD78] et de la multi-représentation [BER77b]. L'impact des types abstraits sur les méthodes de vérification de programmes est aussi très important [WUL76].

1.1.3 - Généricité des types

Depuis l'introduction systématique des constructeurs de type dans les langages de programmation [VW69], on sait qu'il est utile et possible de construire des types à partir d'autres types. Mais jusqu'à présent, à cause de difficultés de compilation [SOL78], on s'était refusé à considérer les constructeurs comme des types avec paramètres formels, c'est-à-dire utilisables dans un programme sans instance particulière de type effectif, bien que des essais aient été tentés [LIN74]. Les langages de types abstraits, pour leur part (CLU, ALPHARD), ont franchi allégrement le pas, car l'abstraction supprimait en particulier les problèmes d'équivalence de modes, mais sans trop mesurer a priori les conséquences d'une telle généralisation. En clair, si l'on sait parfaitement définir un type abstrait comme pile-ent, ensemble-car, matrice-bool, dans quelles conditions peut-on définir des types pile (t1), ensemble (t2), matrice (t3) de telle manière que les instances pile (ent), ensemble (car), matrice (bool) soient les mêmes types que précédemment ? Un autre point important est de savoir s'il est possible d'écrire des procédures ou des fonctions (à commencer par celles qui constituent le type abstrait) ayant des paramètres de type pile (t1) et donc des expressions de type formel t1. Evidemment, une première solution consiste à ne

considérer le type des paramètres que comme une définition de "macro" et d'engendrer un type effectif (ou une procédure) lors de chaque instance particulière, par un remplacement textuel, ce qui ramène à une définition de type abstrait. Cette solution, bien que pratique pour l'écrivain de compilateur est à rejeter, car elle n'apporte rien sur le plan de la méthode ni, a fortiori, de la vérification des programmes. La notion de généricité essaie de répondre à ces questions en remarquant que les types qui sont formels doivent être caractérisés par les "propriétés" qui sont nécessaires dans l'écriture du type ou de la fonction, mais cette caractérisation ne doit préciser que ce qui est strictement nécessaire. Par exemple, dans la spécification du type ensemble-car, on peut utiliser la fonction qui donne la valeur entière d'un caractère pour tester si un caractère appartient à un ensemble ; mais cette fonction n'est pas caractéristique de tous les objets qui peuvent être éléments d'ensembles ; par contre, l'égalité entre deux objets est caractéristique, et le type t2, pour ensemble (t2) devra avoir au minimum la propriété d'égalité entre deux objets ; cette égalité pour le type "caractère" peut alors être construite à partir de la fonction qui fournit un entier. On voit donc déjà s'esquisser une méthodologie de la programmation générique, qui sera de distinguer les propriétés minimum avec lesquelles un type peut être défini. Cela va s'appliquer aussi à la généricité fonctionnelle.

1.1.4 - Généricité fonctionnelle

Une fonction ou procédure est générique si les types qui spécifient les paramètres comprennent des types formels. C'est naturellement le cas pour les fonctions qui entrent dans la définition d'un type générique, puisque celui-ci admet des types formels en paramètre et que certains paramètres de fonction sont du type générique considéré. Mais on peut aussi vouloir écrire des algorithmes pour lesquels les types de certains paramètres ne sont caractérisés que par des propriétés, c'est-à-dire des fonctions particulières, et peuvent s'appliquer à tous les objets des types qui possèderont ces propriétés. Les exemples abondent ;

citons au hasard : le tri d'un tableau ou d'un fichier, si l'on connaît un ordre total sur les éléments ; la fermeture transitive d'une matrice, si deux opérations satisfont la distributivité et l'absorption ; la synthèse de propriétés dynamiques sur les programmes, pour un ensemble de valeurs ayant une structure de treillis fini, ou infini avec élargissement [COU78a], etc. Le procédé est parfaitement maîtrisé dans les démonstrations mathématiques, mais un outil équivalent n'était pas encore disponible dans les langages de programmation. [JAC78b] désigne cette généricité sous le nom de généricité structurale, par opposition à la généricité incrémentale qui est celle des langages traditionnels, où l'opérateur "+", par exemple, peut s'appliquer à la fois aux entiers, aux réels, aux chaînes de caractères, etc., et où le lien entre les différentes interprétations du + provient quelquefois, mais pas toujours, du rôle joué par cette opération dans le type considéré. C'est aussi en ce sens que CLU introduit la paramétrisation des types [LIS77] . Nous nous intéresserons uniquement, par la suite, à la généricité structurale ; la généricité incrémentale, sous certains de ces aspects, est incluse dans l'utilisation des propriétés.

1.2 - Outils de programmation

On ne va pas trouver ici, ni dans le reste de cette thèse, de définition d'un langage de programmation. Bien qu'un projet de langage de programmation générique soit en cours, on souhaite rester ici au plan des problèmes à résoudre et des solutions apportées, sans chercher à trop détailler. Les exemples seront notés dans un (ou plusieurs) langage (s) utilisant soit des notations algorithmiques proches de PASCAL, soit des instructions empruntées à CLU ou ALPHARD. Ce paragraphe expose les constructions qui seront utiles pour comprendre la méthodologie développée au paragraphe 1.3.

Les unités d'abstraction sont les types et les propriétés, mais les unités de programmation sont les modules de définition qui jouent un rôle central ; aussi il convient de commencer par introduire cet objet.

1.2.1 - Modules de définition

La forme de ces modules est inspirée du langage GREEN [GRE79] ; ici ils jouent en outre le rôle de régions génériques, c'est-à-dire paramétrées par des types. Un module de définition, du point de vue de la syntaxe abstraite est un objet qui, étant donné un environnement et des types effectifs fournit un état, ou moins concrétement, une structure d'information au sens de [PAI74] [REM74]. Les modules de définition sont utilisés pour écrire les corps des procédures et les représentations des types abstraits.

Un module de définition (abrégé m-d) comprend des déclarations d'objets (variables, constantes...), des déclarations de procédures et des instructions. Les types formels sont indiqués dans l'en-tête de la définition :

```
def <nom def> (<spécification des types formels>) ;
    | <déclarations d'objets>
    | <déclarations de procédures>
    |
début
    | <instructions>
  fdef
```

Des conventions d'accès permettent, à partir d'un m-d de "voir" d'autres m-d(s) pour pouvoir appeler des fonctions ou accéder à des objets déclarés dans ces m-d(s) ; à l'inverse, certains objets à l'intérieur d'un m-d sont privés, c'est-à-dire inaccessibles directement de l'extérieur.

Quand le système rencontre un m-d celui-ci est élaboré et rangé dans le système. Un m-d élaboré n'est exécutable que si tous ses paramètres de type sont liés à des types effectifs. L'exécution d'un m-d, notée : D consiste en

- la création du contexte local (en pratique, allocation des variables de la partie déclaration) ;
- exécution des instructions.

Le contexte local n'est pas perdu après l'exécution des instructions ; c'est ce qui permet de composer des contextes par déclarations, comme nous le verrons avec les types.

Un programme au sens classique du terme est :

```
: def  <déclarations d'objets>
   |   <déclarations de procédures>
début
   |   <instructions>
fdef
```

Si l'on veut conserver le contexte après exécution du m-d, ce contexte doit être nommé, sinon il est irrécupérable. Cela est possible en préfixant l'exécution du m-d par un identificateur qui doit être unique à ce niveau :

X : def . . . . . fdef

ou, si le m-d a déjà été élaboré dans le système sous le nom D,

X : D

Quand le m-d est générique, on doit lier les paramètres de type par des types effectifs qui peuvent à leur tour contenir éventuellement des types formels :

X : D(τ1,...τn)

Une procédure, dans ce langage, est constituée d'un en-tête qui donne son nom et caractérise ses paramètres, et d'un m-d qui représente son corps. L'appel d'une procédure consiste en :

- liaison des paramètres effectifs ;
- exécution du corps.

Les contextes associés aux appels de procédure ne peuvent pas être nommés. Si la procédure retourne une valeur, elle est utilisable directement à l'endroit de l'appel. Ainsi, la procédure se comporte exactement comme pouvait s'y attendre.

Voici deux exemples (qui selon l'adage font mieux qu'un long discours) :

```
def  complexe ;
  | re : reel ;
  | im : reel ;
  | module : reel ;
  | argument : reel ;
  |
  | proc  ranger-cart (r:reel,i:reel) ;
  |    début
  |      | re := r ; im := i ;
  |      | module := rac2 (r*r+i*i) ;
  |      | argument := si r≠0  alors  Arctg(i/r)
  |      |                 sinon  si  i ≥ 0 alors  pi/2
  |      |                    sinon    -pi/2 fsi  fsi ;
  |     fin
  |
  | proc  ranger-pol (m:reel , a:reel) ;
  |    début
  |      | module := m ;
  |      | argument := a  mod  2*pi ;
  |      | re := m * cos (argument) ;
  |      | im := m * sin (argument) ;
  |     fin
début
  | re := 0 ; im := 0 ;
  | module := 0 ; argument := pi/2
fdef
```

co les procédures ne contiennent pas de partie déclarations locales co

Utilisation de la définition :

```
Z1 : complexe ;
      co à l'exécution, Z1 est initialisé co
Z1 . ranger-cart (1.1) ;
      co après cette instruction, Z1 . module = rac2 (2) co
```

Le deuxième exemple permet de faire des calculs statistiques sur les éléments de vecteurs d'entiers :

```
def  stat ;
   max : entier ; min : entier ;
   nombre : entier ; somme : entier ;
   moyenne : reel ;
   proc  init ;
      début
       max := 0 ; min := max-entier ; nombre := 0 ;
       somme := 0 ; moyenne := 0 ;
      fin ;
   proc  calcul (v : vecteur-entier) ;
      début
         pour  i := 1  a  long (v)
         faire si v(i) > max  alors  max :=  v(i)  fsi ;
               si v(i) < min  alors  min  :=  v(i)  fsi ;
               somme := somme + v(i)
         ffaire ;
         nombre := nombre + long (v)
         moyenne := (reel(somme)/nombre)
       fin ;
   proc  écrire ;
      début
        ... écriture sur fichier des valeurs statistiques ...
       fin
début
   init
fdef
```

Utilisation pour des statistiques séparées :

```
S1 : stat ;
S2 : stat ;
v1 : vecteur-entier ;
v2 : vecteur-entier ;
  ....
S1 . calcul (v1) ;
S2 . calcul (v2) ;
```

Ou encore :

```
S : stat ;
  ....
S . calcul (v1) ;
S . écrire ;
S . init ;
S . calcul (v2) ;
S . écrire ;
  ....
```

Les statistiques seront cumulées si on fait :

```
S . calcul (v1) ;
S . calcul (v2) ;
   ::::
```

De tels modules de définition sont utilisables comme ceux du langage EUCLID [LAM77].

1.2.2. Types génériques

Les types sont des outils d'abstraction des objets. Comme les m-d(s), ils peuvent être paramétrés par des types formels. Un type est composé des en-têtes de procédures constituant les opérateurs d'accès élémentaires au type, et des déclarations de procédures "dérivées", utilisant ces fonctions d'accès. Il n'y a pas de partie instruction.

```
type   T  (<spécification de types formels>) ;
  | <en-tête de fonctions de construction et d'accès>
  | <déclaration de procédures dérivées>
 ftype
```

Dans l'en-tête du type, on peut spécifier des attributs nécessaires à la création des objets. On remarque que, contrairement à CLU ou à ALPHARD, la représentation ne fait pas physiquement partie de la déclaration du type.

Décrivons le type pile, pour satisfaire la tradition :

```
type  pile (t:tyfo) ;
  | proc  empiler (mod pile (t) , t) ou (dépassement) ;
  | proc  sommet (pile(t)) → t  ou (non-défini) ;
  | proc  depiler (mod pile (t)) ou  (sous-dépassement)
 extension
  | proc  remplacer (mod p : pile (t) , x:t) ;
      qd sous-dépassement alors  empiler (p,x)  fq
    début
        depiler (p) ; empiler (p,x) ;
     fin
ftype
```

Commentaires :

Le type résultat des procédures se trouve éventuellement après le signe → . La clause ou est suivie de un ou plusieurs identificateurs d'exception qui rendent la fonction totale en cas d'erreur. Ces exceptions sont traitées par une instruction qd apparaissant dans les déclarations d'une procédure appelant la procédure dans laquelle l'exception s'est produite. Dans la procédure remplacer, on a le traitement de l'exception "sous-dépassement" de dépiler. Le mot-clé mod signifie que le paramètre effectif est modifié par la procédure. L'extension contient les déclarations de procédures qui font partie du type mais peuvent être décrites en utilisant les procédures d'accès primitives, dites procédures de base.

1.2.3 - Modules de représentation

La représentation d'un type générique est fournie par un ou plusieurs modules de définition qui donnent une "interprétation" des procédures de base du type, c'est-à-dire les procédures qui sont uniquement décrites par leur en-tête. Une procédure de l'extension peut aussi être redéfinie dans une représentation particulière ; à ce moment-là, elle remplace (pour cette représentation) la procédure déclarée dans le type. L'en-tête du module indique le type qui est représenté et donne la représentation en fonction de types déjà existant.

Le type définissant peut être récursif, comme en témoigne rep-pile-2. Voici deux représentations du type pile ; dans la première, la pile est un couple (entier, tableau), dans la seconde, c'est une liste chaînée simple.

```
def  rep-pile-1 (t:tyfo)  attr  h:entier
représente  pile(t) par  r:struct(s:entier,tab:tableau(1..h,t)) ;
    | proc empiler (mod p:pile(t) , x:t)  ou (dépassement) ;
    |  début si  p.s = h(p)        alors dépassement
    |       sinon p.s := p.s+1 ; p.tab(p.s) := x  fsi
    |    fin ;
```

```
    | proc sommet (p:pile(t)) → t ou (non-défini) ;
      début si p.s=0 alors non-défini
            sinon retourner (p.tab(p.s)) fsi
      fin ;
    proc dépiler (mod p:pile(t)) ou (sous-dépassement) ;
      début si p.s=0 alors sous-dépassement
            sinon p.s := p.s-1 fsi
      fin
début
      r.s := 0
fdef ;
def rep-pile-2 (t:tyfo)
représente pile(t) par r:ptr(m)
    |     avec m = struct (val:t , d:ptr(m)) ;
      proc empiler (mod p:pile(t) , x:t) ou (dépassement) ;
          q : ptr(m) ;
          qd dépassement-mémoire alors dépassement fq ;
        début alloc(q) ;
            q.val := x ; q.d := p ;
          p := q ;
        fin ;
      proc sommet (p:pile(t)) → t ou (non-défini) ;
        début si p=nil alors non-défini
            sinon retourner (p.val) fsi
        fin
      proc dépiler (mod p:pile(t)) ou (sous-dépassement) ;
        début si p=nil alors sous-dépassement
            sinon p := p.d fsi ;
        fin ;
début
      r := nil
 fdef ;
```

La notation :

représente <type-abs> par idf : <type> ;

signifie que dans la portée du module de définition, <type-abs> est équivalent (au sens Algol68) à <type> et, d'autre part, un objet idf:<type> est alloué lors de l'exécution du module de définition. Le type abstrait <type-abs> n'a donc pas d'existence dans la portée du module de représentation. Par contre, l'objet idf, alloué pendant l'exécu-

tion de ce module est considéré comme appartenant au type abstrait, hors de ce module.

L'association entre une utilisation d'un type et un module de représentation est faite par un "interface" de représentation lorsqu'on veut compiler un programme. Les valeurs des attributs du module de représentation qui n'apparaissent pas dans la spécification du type abstrait doivent être initialisées par l'interface.

Soit la déclaration :

P1 : pile (entier) ;

Si la représentation choisie est la première avec h = 100, cette instruction est, en fait, équivalente à :

P1 : rep-pile-1(entier)(100) ;

avec allocation d'un objet de la représentation : P1.r
L'instruction : empiler (P1,1)
est équivalente à : P1.empiler (P1.r,1).
Les procédures qui peuvent être représentées sont donc des procédures unaires par rapport au type. Les procédures non-unaires doivent être définies comme extension dans le type, ou, si elles sont redéfinies, elles ne peuvent être applicables qu'à des objets ayant même représentation.

1.2.4 - Les propriétés

Une propriété est la donnée d'un ensemble de procédures (ou opérateurs) se rapportant à un type formel. Le nom de la propriété est caractéristique et a un triple rôle :

- identifier les relations qui existent entre les différentes procédures constituant la propriété. Ces relations ne peuvent pas, en général, être exprimées simplement dans le langage algorithmique. Elles seront, par contre, facilement décrites dans le langage algébrique (paragraphe 2.2.3).

- spécifier les types formels qui, dans un m-d ou un type générique, devront posséder un ensemble de procédures ou d'opérateurs satisfaisant de telles relations.

- permettre les liaisons correctes de types effectifs et surtout les résolutions d'appels formels ([BER77a]).

Les opérateurs sont identiques à des procédures, excepté pour l'appel qui peut être noté de manière infixée ; on ne rentrera pas ici dans les problèmes syntaxiques de priorité que cela peut poser. Donnons quelques exemples simples :

```
prop égalité (=,¬=) sur t:tyfo ;
 | op = (t,t) → bool ;
 extension
 | op ¬= (x,y:t) → bool
 | début retourner(¬(x=y)) fin
 fprop ;
prop scalaire (ord,card,g,prem,der,succ,pred,≤)
 |            sur t:tyfo ;
 | proc ord(t) → intervalle (entier) (1,card) ;
 | proc card → entier ;
 | privé proc g (intervalle (entier) (1,card))→ t ;
extension
 | proc prem → t ;
 |    début retourner (g(1))fin ;
 | proc der → t
 |    début retourner (g (card)) fin ;
 | proc succ (x:t) → t ou (indef) ;
 |    début si ord(x) = card alors indef
 |           sinon retourner g (succ (ord (x))) fsi
 |     fin ;
 | proc pred (x:t) → t ou (indef) ;
 |    début si ord(x) = 1 alors indef
 |           sinon retourner g(pred (ord (x))) fsi
 |     fin ;
 | op ≤ (x,y:t) → bool ;
 |    début retourner (ord(x) ≤ ord(y)) fin ;
fprop ;
```

Remarques

1. Dans l'exemple de scalaire, g est une procédure qui transforme un entier dans un objet du type, mais cette procédure ne doit pas être utilisable à l'extérieur du type pour des raisons de protection. Elle est donc privé(e). Les procédures habituelles dans un type ayant la propriété scalaire peuvent être déduites automatiquement à l'aide de ord, card et g. Il est possible néanmoins de les définir autrement dans un type particulier, mais cette définition comme extension est en quelque sorte une spécification (opérationnelle) de ces procédures.

2. Le type intervalle est un type générique lui-même scalaire, construit sur un autre type scalaire qui est, ici, le type entier. Les valeurs 1, card, sont les attributs de ce type intervalle (cf. 1.2.2). Les opérateurs pred, succ, ≤ sont définis en fonction de pred, succ, ≤ du type scalaire entier.

Puisque les propriétés déterminent des classes de types, on peut se demander s'il existe une propriété qui caractérise la classe de tous les types. Cela est évidemment nécessaire et c'est même la seule classe qui soit définie explicitement dans les autres langages, en général sous le nom de "any" (CLU) (cf. aussi Triv en [BUR77]). Cette propriété ne contient aucune procédure, voilà pourquoi tous les types la possèdent. Elle est identifiée par le mot-clé tyfo (type formel) dans le langage. A partir de cette remarque, il est clair qu'il existe un ordre partiel d'inclusion dans les propriétés, c'est-à-dire que si un type possède une propriété il possède aussi toutes les propriétés qui sont incluses (ex : tyfo est incluse dans toutes les propriétés). On reviendra au paragraphe 2.2.3.1.5 sur une définition précise de l'inclusion. [JAC78a] donne une manière de déclarer les inclusions ; on en donne ici une autre qui évite d'introduire des notations supplémentaires. Par exemple, on sait qu'une propriété d'ordre-total implique automatiquement une propriété d'égalité. Cela va s'exprimer par :

```
prop  ordre-total (≤,<,>,≥, =,¬=) sur  t:égalité (=,¬=) ;
  | op ≤ (t,t) → bool ;
extension
  | op > (x,y:t) → bool
  |   début  retourner (¬(x≤y))  fin ;
  | op = (x,y:t) → bool
  |   début  retourner (x≤y∧y≤x)  fin ;
  | op ≥ (x,y:t) → bool ;
  |   début  retourner (x>y∨x=y)  fin ;
  | op < (x,y:t) → bool ;
  |   début  retourner (¬(x≥y))  fin ;
fprop ;
```

L'opérateur ¬= étant une extension de l'opérateur =, il n'est pas nécessaire, sauf pour optimisation, de le redéfinir dans la propriété ordre-total. Si plusieurs propriétés sont incluses de plusieurs manières dans une propriété plus large, on écrira :

```
prop  P( ... )  sur t:P1 ( ... ) & P2 ( ... ) & ...
```

On a alors : P1 ⊆ P ∧ P2 ⊆ P ...
tyfo est l'élément neutre de l'opération & entre propriétés.

1.2.5. Types génériques et propriétés

Les éléments de ce chapitre ont été exposés initialement en [BER77a] ; les points à retenir sont :

- les types formels doivent être spécifiés par une propriété qui indique les opérations ou procédures qui sont nécessaires pour la définition de la région générique à laquelle ils sont liés.
- les types possèdent des procédures en propre. Dans l'en-tête du type, on précise la manière par laquelle ces procédures confèrent au type certaines propriétés. Le type est alors introduit dans la classe des types qui ont ces propriétés.
- un type muni d'une propriété P peut être passé en paramètre à la place d'un type formel spécifié avec une propriété P' qui est incluse dans P. En effet, dans ce cas, les procédures utilisées dans la région générique auront une interprétation dans le type effectif.

- lorsqu'un type ne possède pas une propriété suffisante pour être paramètre effectif, il est toujours possible de définir, pour cette liaison particulière, les procédures qui confèrent à ce type la propriété exigée. Cependant, cette propriété n'est valable que localement pour ce type.

Donnons là encore des exemples :

1.2.5.1 - Utilisation de types formels avec propriété

Soit le module de définition de minimum-maximum :

```
def  min-max  (t:ordre-fini (≤,minimum,maximum)) ;
   |    max:t ; min:t ;
   |   proc  init ;
   |      début  max := minimum ; min := maximum  fin ;
   |   proc  calcul (v:vecteur(t)) ;
   |      début
   |         |  pour  i := 1  a  long(v)
   |         |  faire  si ¬(v(i) ≤ max)  alors  max := v(i)  fsi ;
   |         |         si ¬(min ≤ v(i))  alors  min := v(i)  fsi
   |         |    ffaire ;
   |       fin ;
   |   proc  écrire ;
   |       début ... fin ;
début
   init
 fdef ;
```

Une occurrence de min-max pour des entiers positifs est :

MM : min-max (entier:ordre-fini(≤,0,max-entier)) ;

à moins que la propriété ordre-fini ne soit associée au type.

La propriété ordre-fini est naturellement :

```
prop  ordre-fini (≤,m1,m2)  sur  t : tyfn ;
    op ≤ (t,t) → bool ;
    proc  m1 → t ;
    proc  m2 → t
fprop ;
```

Spécification de types formels dans une définition de types :

```
type  ensemble  (t:égalité(=,¬=)) ;
    proc  insérer (t,mod ensemble(t)) ;
    proc  enlever (t,mod ensemble(t)) ;
    proc  appartient (t,ensemble(t)) → bool
    proc  vide (ensemble(t)) → bool ;
ftype ;
```

Pour définir le type ensemble, il est nécessaire que le type des éléments possède une opération d'égalité (et d'inégalité). La déclaration

E-ent : ensemble (entier) ;

est valide. Par contre,

EE-ent : ensemble (ensemble(entier)) ;

n'est pas valide car le type ensemble(entier) n'a pas l'opération d'égalité.

1.2.5.2 - Appartenance d'un type à une propriété

Pour résoudre le problème ci-dessus, on doit déclarer que le type ensemble satisfait la propriété d'égalité :

```
type  ensemble (t:égalité(=,¬=)) : égalité (eq-ens,neq-ens) ;
    proc ...
     ...
    proc  eq-ens (ens(t) , ens(t)) → bool ;
    proc  neq-ens(ens(t) , ens(t)) → bool
ftype
```

Un type donné peut appartenir à plusieurs propriétés par des familles de procédures qui ne sont peut-être pas disjointes, c'est-à-dire qu'une même procédure peut apparaître dans plusieurs propriétés.

Une extension d'un type peut définir une portée générique plus restrictive du point de vue des propriétés que celle des procédures de base. Prenons par exemple le type vecteur, où les extensions permettent de doter les vecteurs de certaines structures dépendant des propriétés des éléments :

```
type  vecteur (t:tyfo)  attr  long:entier ;
   proc  ranger (mod  vecteur(t), entier, t) ou  erreur ;
   proc  val (vecteur(t), entier) → t  ou  erreur ;

extension vecteur (t:monoïde (0,+)) : monoïde (v0,v+) ;
   proc  v0(ℓ) → vecteur(t) (ℓ)
      v : vecteur(t) (ℓ) ;
      début  pour  i  de 1  a  ℓ
           faire  ranger (v,i,0)  ffaire ;
           retourner (v)
       fin ;

   proc  v+ (v1,v2 : vecteur(t)) → vecteur(t) (long(v1)) ou  erreur ;
       v : vecteur(t) (long(v1))
       début  si  long(v1) ≠ long(v2) alors  erreur  fsi ;
          pour  i  de 1  a  long(v1)
          faire  ranger (v,i,val(v1,i)+val(v2,i))  ffaire ;
          retourner (v)
       fin ;

   proc  reduc (v : vecteur(t)) → t ;
       s:t ;
       début  s := 0 ;
          pour i  de  1  a  long(v)
          faire  s := s+v(i)  ffaire ;
          retourner (s)
       fin ;

extension  vecteur (t:égalité (=,¬=)) : égalité(v=,v¬=) ;

   proc  v = (v1,v2 : vecteur(t)) → bool ;
       b : bool ;
       début  b := vrai ;
          si  long(v1) = long(v2)
          alors  pour  i  de  1  a  long(v1)
                 faire b := b ∧ val(v1,i) = val(v2,i)  ffaire
          sinon  retourner (faux)  fsi ;
          retourner (b)
       fin

ftype ;
```

Si le type des éléments d'un vecteur possède à la fois les propriétés de monoïde et d'égalité, alors ce vecteur contient toutes les procédures décrites dans le type et ses extensions. Des extensions peuvent constituer des unités indépendantes du type en vue de constituer des bibliothèques pour des applications particulières.

1.3 - Structures de contrôle programmables

1.3.1 - Le traitement itératif

La composition des actions qui est à la source du traitement automatique de l'information a été longuement étudiée dans les langages de programmation. Les trois modes de composition qui se sont imposés sous différentes formes sont la composition séquentielle simple (";"), la composition conditionnelle et la répétition. Cette dernière, en particulier, est toujours l'objet de travaux, essentiellement d'ordre pédagogique [PEY78] , car les problèmes de définition axiomatique et sémantique sont résolus depuis longtemps. La relation entre le traitement itératif et les types est apparue par l'intermédiaire des tableaux, avec la boucle "pour", et aussi plus subtilement, avec le traitement des fichiers.

L'abstraction des types a conduit à la notion d'abstraction des structures de contrôle. Prenons un exemple simple : en Algol60 ou en Fortran, l'énumération des éléments d'un tableau passe par les valeurs successives des indices :

```
pour  i := 1  pas  1  a  N
faire   S(T[i]) ;
```

Si l'ordre des éléments n'est pas significatif dans S, une première abstraction va consister à prendre les valeurs de i dans l'intervalle [1,N] :

```
pour  i  dans  1 ..  N
faire   S(T[i]) ;
```

Si, maintenant, ce sont les valeurs des éléments du tableau qui intéressent, la variable i n'a aucune utilité ; on peut donc énumérer directement les éléments du tableau (dans un ordre quelconque) :

```
pour  x  dans  T
faire   S(x) ;
```

Etudions rapidement les abstractions de structures itératives en CLU et ALPHARD, puis nous donnerons une généralisation de la notion d'itération.

1.3.1.1. Générateurs en ALPHARD

En Alphard [SHA76a], les générateurs sont des "forms", (l'équivalent des types) qui contiennent deux procédures distinguées : &init et &next. Ces procédures ne peuvent être appelées d'une manière implicite qu'à l'intérieur d'une construction syntaxique particulière qui est l'itération. Il y a deux formes de l'itération :

- la boucle for ;

for x : gen while β(x) do ST(x) ;

dont la sémantique opérationnelle en pseudo-Alphard est :

```
begin  x : gen ; c : bool ;
   |   c := x.&init ;
   |   while  c  et  β(x)
   |    do (ST(x) ; c := x.&next)
end
```

&init et &next agissent par effet de bord sur x et produisent un résultat booléen. Le connecteur booléen et est séquentiel.

- la boucle first :

first x : gen suchthat β(x) then $S_1(x)$ else S_2 ;

dont l'équivalent est :

```
begin   x: gen ;  c: bool ;
   |    c:=  x.&init ;
   |    while  c  et  ¬β(x)
   |       do  c :=  x.&next ;
   |    if  β(x)  then  S1(x)
   |    else  S2
end
```

Voici les éléments de la déclaration d'une "form" qui permet de parcourir les valeurs d'un intervalle. On n'a conservé que les instructions intéressantes pour la comparaison avec CLU :

```
form   upto  (lb, ub : int) ;
   representation  k:int ;
   function &init  (u:upto)  returns  (b:bool) ;
     (b := u.ub ≥ u.lb ;
      if  b  then  u.k := u.lb) ;
   function &next (u:upto)  returns  (b:bool) ;
     (b := u.ub > u.k ;
      if  b  then  u.k := u.k+1)
endform
```

Une utilisation de cette "form" est, par exemple :

```
sum := 0 ;
for  j : upto (1,n)
do     sum :=  sum+j ;
```

Un autre exemple est celui de l'égalité de deux vecteurs à l'aide du first ; on remarque au passage l'introduction de fonctions attachées à des types formels pour écrire le corps de la fonction générique. On donne au paragraphe 2.1 une explication plus précise de la paramétrisation ainsi qu'une critique du mécanisme de généricité en Alphard.

```
function  eq-vect (A,B : vect(t:<≠>)) returns  (eq:bool) ;
          first  i : upto (A.lb,A.ub)
          suchthat  A[i] ≠ B[i]
          then  eq := false
          else  eq := true
   end ;
```

On note qu'une construction répétitive est composée de la boucle de l'itération et de la boucle de l'énumération qui est éclatée en deux fonctions. Cela facilite la détermination des conditions de preuve, mais l'écriture du générateur n'est peut-être pas des plus naturelles.

1.3.1.2. Itérateurs en CLU

La solution proposée par CLU est plus élégante, bien que les méthodes de preuve n'aient pas été développées comme en Alphard. Le langage donne la possibilité de déclarer des actions, autres que des procédures, appelées itérateurs ; ce sont en fait des coroutines. Les itérateurs fournissent une séquence d'éléments qui peuvent être utilisés un par un dans une instruction répétitive for. Reprenons l'exemple de l'itérateur d'intervalle ; upto sera défini par :

```
upto = iter  (lb,ub:int)  yields  (int) ;
   | k : int := lb ;
   | while  k ≤ ub
   | do  yield (k) ;
   |     k := k + 1
   | end ;
end  upto ;
```

L'utilisation est assez semblable à celle d'Alphard :

```
sum := 0 ;
for  j : int  in  upto (1,n)
do  sum := sum + j  end
```

La terminaison peut se produire de deux façons :

- soit par la fin de l'itérateur ;
- soit par une sortie explicite de la boucle for (instruction break) qui termine automatiquement la boucle de l'itérateur.

1.3.2. - Actions d'itération.

1.3.2.1. Les itérateurs émetteurs .

Si l'on cherche à banaliser la notion d'itérateurs de CLU, on peut remarquer que la valeur résultat qui est produite répétitivement et transmise par yield à la boucle for peut être considérée comme une séquence temporaire d'objets. Dans cette séquence temporaire, on n'a accès qu'au premier de la séquence, et c'est l'exécution de l'itérateur qui fournit le suivant (on reviendra en détail sur l'utilisation du type séquence dans la spécification des itérateurs : § 2.2.4.4.).

Un itérateur est donc une action qui produit une séquence temporaire d'objets d'un type τ.

Comme c'est le type des objets produits qui sera utile dans les vérifications statiques, on note un en-tête d'itérateur émetteur par :

proc idf (liste de paramètres) → seqt τ ;

Il est possible d'avoir plusieurs valeurs résultat de mode seqt qui seront alors identifiées à la manière de paramètres de mode sortie :

proc idf (... ; sortie seqt x : τ1, seqt y : τ2,...) ;

Le corps de l'itérateur doit contenir, à l'intérieur d'une action répétitive une ou plusieurs instructions qui fournissent une valeur à la séquence temporaire :

donner (expr1, expr2...)
ou donner (expr1 a idf1,...)

Lorsque l'émetteur s'arrête, la valeur donnée est une valeur indéfinie spéciale, appelée "eof" par analogie avec les traitements de fichier. Dans le programme qui utilise les valeurs de l'itérateur, la condition fin d'itération peut être récupérée par une instruction d'exception :

qd eof alors ... fq ;

Ecrivons l'itérateur qui produit une séquence d'éléments de n1 à n2

```
proc interv (n1,n2 : entier) → seqt entier ou (eof) ;
    k : entier :
    début k := n1
          tantque k ≤ n2
          faire donner (k) ;
          k := k+1
          ffaire ;
          eof
    fin ;
```

1.3.2.2. Les itérateurs récepteurs

Avec les langages CLU et ALPHARD, les procédures produisant des séquences temporaires (resp. itérateurs ou générateurs) sont toujours liées à des structures de contrôle particulières (for, first). En réalité, ces structures de contrôle jouent le rôle de coroutines qui reçoivent des valeurs de séquence temporaire. Il suffit donc que le langage permette de déclarer des

actions qui admettent des séquences temporaires en paramètres d'entrée. On appelle ces actions des itérateurs récepteurs :

proc idf (seqt <paramètres>) → résultat ;

Le récepteur, comme les autres procédures, peut avoir ou non un résultat. Evidemment, on peut avoir des actions qui soient à la fois récepteur et émetteur, c'est-à-dire qui acceptent en entrée des séquences temporaires et fournissent des séquences temporaires en sortie.

Pour accéder à un élément suivant de la séquence d'entrée, le récepteur dispose de l'instruction suivant qui s'applique à un paramètre et fournit l'élément suivant s'il existe. A l'entrée du récepteur, le paramètre est initialisé par le premier élément de la séquence.

Voici quelques exemples :

1. Somme d'éléments à l'aide d'un récepteur (générique) ;

```
def  S (t : monoïde(0,+)) ;
|  proc  somme (seqt  el : t) → t
|  |  x : t ;
|  |  qd  eof  alors  retourner (x) fq
|  début  x := 0 ;
|  |     faire  x := x + el ;
|  |           suivant  (el)
|  |     ffaire
|  fin
fdef ;
```

2. Récepteur-émetteur qui, étant donné une séquence temporaire d'indices, fournit la séquence des éléments positifs d'un tableau d'entiers :

```
type  tab-entier (index : intervalle (entier) (1,n)) ;
   ...
   proc  elempos (T : tab-entier, seqt i : entier) → (seqt entier) ;
   |  qd  eof  alors  eof  fq
   |        co  propagation de la fin de séquence  co
   début
   |  faire  si  val (T,i) ≥ 0  alors  donner (val(T,i))  fsi ;
   |         suivant(i)
   |  ffaire
   fin ;
```

Un programme peut donc être construit par composition d'émetteurs et de récepteurs à condition que les types soient compatibles. On peut composer les actions données en exemple pour calculer la somme des éléments positifs d'un tableau d'entiers TE de 1 à n :

```
y := somme (elempos (TE, interv(1,n))) ;
```

(Note : le m-d S ne contenant que des déclarations de procédures, n'a pas besoin d'être exécuté explicitement).

Remarque : la structure de programme introduite par ces actions est totalement abstraite, c'est-à-dire que l'on ne définit absolument pas la manière concrète d'évaluer l'expression (processus séquentiel, parallèle avec synchronisation, pipe-line, etc...). Il va de soi qu'une implantation particulière peut tenir compte de la configuration du système d'exécution pour optimiser le calcul. Il est aisé d'écrire les instructions d'itération classiques en terme de récepteur sans résultat. Si l'itérateur It est de la forme :

```
proc  It (<parms>) → seqt τ  ou  (eof) ;
```

l'instruction :

```
pour  x  dans  It (...)
faire  S(x) ffaire ;
```

est mise pour la composition : RS(It(...))

où RS est le récepteur défini par :

```
proc  RS (seqt x : τ) ;
      qd  eof  alors  retourner  fq
début
      faire  S(x) ; suivant (x)  ffaire
fin ;
```

Lorsqu'un récepteur termine (par une instruction retourner), le ou les émetteurs associés sont aussi terminés (même si on n'a pas atteint la fin de séquence).

1.3.3. Décomposition d'itérateurs

Etudions le problème inverse de celui que l'on vient d'évoquer, à savoir le passage d'une expression avec itérateurs en un programme classique.

On appelle cette opération une décomposition, car on perd totalement la structure abstraite du programme initial ; elle s'inspire de la méthode de programmation par affinements successifs [WIR71], mais la décomposition est plus complexe et elle n'est pas unique, ce qui conduit naturellement à des programmes d'inspiration différente. On se limite, pour aborder cette étude, aux itérateurs qui ne manipulent qu'une seule séquence temporaire d'objets. Le principe de la décomposition consiste à :

1. choisir une des boucles d'itération pour être l'itération finale principale ;

2. décomposer les émetteurs-récepteurs liés à cette boucle en procédures ;

3. remplacer textuellement ces procédures dans le corps de la boucle ;

4. faire les optimisations évidentes.

On obtient ainsi d'une manière systématique un programme équivalent au premier, comme on va voir à la fin du paragraphe. Cela peut être un moyen de compiler les compositions d'itérateurs. Comme les possibilités de choix sont en nombre fini pour une expression donnée, il est possible que le compilateur prévoie ou calcule l'efficacité des codes résultants et choisisse automatiquement le meilleur ; c'est l'amorce d'un compilateur "intelligent".

Voici comment passer de la notation d'itération à celle de procédure puis, par substitution, à celle de programme. Dans tous les cas, on va considérer que les variables locales deviennent des variables globales initialisées (après renommage éventuel). Ensuite, il y a deux cas à considérer :

1. La décomposition est dirigée par l'émetteur :
   - donner (x) devient un appel de procédure sans résultat où le paramètre est x ;
   - suivant (x) devient la fin de procédure : retourner.

Il y a à construire autant de membres de procédure qu'il y a de chemins dans un récepteur qui aboutissent à suivant ; les points de reprise (quant il y en a plusieurs), doivent être mémorisés dans de nouvelles variables globales.

2. La décomposition est dirigée par le récepteur :
   - les émetteurs sont découpés en une procédure d'initialisation et une procédure d'élément suivant qui retournent un résultat ;

- suivant devient l'appel à la procédure d'élément suivant ;

- donner (x) devient retourner (x).

On retrouve la décomposition en "init" et "next" des générateurs d'Alphard. Justification de la décomposition : on ne donne pas une démonstration rigoureuse de l'équivalence des programmes avant et après décomposition, mais simplement une justification basée sur la suite des instructions du programme complet :

Soit l'itérateur émetteur :

```
proc  E (...) → seqt τ  ou (eof) ;
   début  α1 ;
      |    faire  β1 ; donner (x) ; γ1 ffaire ;
      |    eof
   fin
```

et un récepteur :

```
proc  R (seqt y :τ)
    qd  eof  alors  δ2  fq
    début  α2 ;
       |   faire  β2 ;
       |         suivant (y) ;
       |        γ2
       |   ffaire
    fin
```

On rappelle que toutes les variables sont rapportées à un niveau global. La composition R(E) est équivalente à la séquence d'instructions :

début α1 ; β1 ; α2 ; β2 ;
γ1 ; β1 ; γ2 ; β2 ; ... γ2 ; β2 ; γ1; δ2 fin

où α2 ; β2 utilisent un paramètre calculé par α1 ; β1 et γ2 ; β2 un paramètre calculé par le groupe γ1 ; β1 précédent.

Dans la décomposition dirigée par E, le récepteur R comprend deux parties, puisqu'on distinguera le cas du traitement de l'exception eof du cas général :

```
proc  R (x : τ) ;
   début  si  initR  alors  α2 ; β2 ; initR  := faux ;  retourner
          sinon  γ2 ; β2 ; retourner  fsi
   fin

proc  eofR ;
   début  δ2  fin ;
```

Le programme déduit de l'émetteur E est simplement :

```
début  α1 ;
   faire  β1 ; R(x) ; γ1  ffaire ;
   eofR
 fin
```

La variable initR est initialisée à vrai avant l'appel de E. Par substitution de R et eofR dans E on obtient un programme avec une seule boucle dont la suite des opérations est la même que celle avec coroutines.

Dans la décomposition dirigée par R, on a deux procédures pour l'émetteur E :

```
proc  init-E (...) → τ
   début  α1 ; β1 ; retourner (x)  fin

proc  suivant-E (...) → τ  ou  (eof) ;
   début  γ1 ; eof  ou  β1 ; retourner (x)  fin
```

La notation eof ou β1 signifie que l'on a, soit le cas où le faire de l'émetteur E termine, soit le retour au début de la boucle avec β1.

Le programme déduit du récepteur R est :

```
   qd  eof  alors  δ2 fq
   début  y := init-E (...)
      |    α2 ;
      |    faire  β2 ;  y := suivant-E (...) ;
      |          γ2
      |    ffaire
   fin
```

Par substitution des corps des procédures appelées, on obtient un programme avec une seule boucle où l'on retrouve toujours la même suite d'instructions.

Note : la valeur transmise entre l'émetteur et le récepteur est affectée à la variable y (paramètre dans la notation du récepteur R).

Traitons les trois cas de décomposition de l'instruction :

y := somme (elempos(TE,interv(1,n)))

Exemple 1

Décomposition dirigée par l'itérateur interv :

programme initial :

```
| k := 1 ;
| tantque  k ≤ n
| faire  som-elempos (TE,k) ;
|        k := k+1
| ffaire ;
| y := som-elempos-eof ;
```

avec :

```
proc  som-elempos (T : tab-entier, i : entier) ;
   début
      | si  val (T,i) ≥ 0  alors  somme (val(T,i))  fsi ;
      | retourner
   fin ;

proc  som-elempos-eof → entier
   début  retourner (som-eof)  fin ;
                co  propagation de la fin d'intervalle  co

proc  somme (el : entier) ;
   début  x := x + el ; retourner  fin ;

proc  som-eof → entier :
   début  retourner (x)  fin ;
```

Note : L'initialisation x := 0 a été repoussée à l'extérieur.

Le programme obtenu est donc :

```
(P1) x : entier ;
     k : entier ;

début
  | x := 0 ;
  | k := 1 ;
  | tantque  k ≤ n
  | faire  si  val(TE,k) ≥ 0  alors  x := x + val(TE,k)  fsi ;
  |        k := k+1
  | ffaire ;
  | y := x ;
fin ;
```

Exemple 2 :

Découpage dirigé par le récepteur somme :

- programme initial :

```
   x : entier ; el : entier ;
   qd  eof  alors  y := x  fq ;
   début  x := 0 ; el := init-elempos (TE) ;
     | faire  x := x + el ;
     |        el := suiv-elempos (TE) ;
     | ffaire
   fin ;

proc  init-elempos (T : tab-entier) → entier  ou  (eof) ;
  | i : entier ;
  | qd  eof  alors  eof  fq
début
  | i := init-interv (1,n) ;
  | faire
  | si  val (T,i) ≥ 0  alors  retourner  (val(T,i))  fsi ;
  |     i := suiv-interv (1,n)
  | ffaire
fin ;
```

```
proc  suiv-elempos (T : tab-entier) → entier  ou  (eof) ;
   i : entier ;
   qd  eof  alors  eof  fq ;
   début
   |  i := suiv-interv (1,n) ;
   |  faire
   |     si  val(T,i) ≥ 0  alors  retourner (val(T,i))  fsi ;
   |     i := suiv-interv (1,n)
   |     ffaire
   fin ;

proc  init-interv (n1,n2 : entier)   entier  ou  (eof) ;
   début
   |  k := n1 ;
   |  si  k ≤ n2  alors  retourner (k)
   |  sinon  eof  fsi
   fin ;

proc  suiv-interv(n1,n2 : entier) → entier  ou  (eof) ;
   début  k := k+1 ;
   |     si  k ≤ n2  alors  retourner (k)
   |     sinon  eof  fsi
   fin ;
```

Note : k est une variable globale.

Un premier programme est obtenu en substituant init-elempos et suiv-elempos ; on remplace les instructions retourner par des affectations suivies de sortie qui permet de quitter la boucle.

```
   x : entier ; el : entier ; i : entier ;
   qd  eof  alors  y := x  fq

début  x := 0 ;
|  i := init-interv (1,n) ;
|  faire
|     si  val(TE,i) ≥ 0  alors  el := val (TE,i) ; sortie  fsi ;
|     i := suiv-interv (1,n)
|  ffaire ;
```

```
    faire  x := x + el ;
           i := suiv-interv (1,n) ;
           faire  si  val (TE,i) ≥ 0 alors  el := val (TE,i) ; sortie  fsi ;
                  i := suiv-interv (1,n)
           ffaire
    ffaire
fin
```

Le programme final obtenu par cette méthode est :

(P2) x, el, i, k : entier ;

```
début  x := 0 ;
    k := 1 ;
    si  k ≤ n  alors  i := k  sinon  allera  f  fsi ;
    faire
        si  val(TE,i) ≥ 0  alors  el := val(TE,i) ; sortie  fsi ;
        k := k + 1 ;
        si  k ≤ n  alors  i := k  sinon  allera  f  fsi
    ffaire ;
    faire  x := x + el ;
        k := k + 1 ;
        si  k ≤ n  alors  i := k  sinon  allera  f  fsi ;
        faire
            si  val (TE,i) ≥ 0  alors  el := val (TE,i) ; sortie  fsi ;
              k := k + 1 ;
              si  k ≤ n  alors  i := k  sinon  allera  f  fsi
        ffaire
    ffaire ;
f : y := x
fin
```

Les occurrences de eof ont été remplacées par un branchement au traitement de l'exception (ici y := x).

Exemple 3

Découpage par l'émetteur-récepteur elempos ;

- programme initial :

```
i : entier ; qd  eof  alors  y := som-eof  fq
début
    i := init-interv (1,n) ;
    faire
        si  val (TE,i) ≥ 0  alors  somme (val(TE,i))  fsi ;
        i := suiv-interv (1,n)
    ffaire
fin ;
```

Maintenant, toutes les procédures utilisées sont déjà définies dans les exemples précédents ; on a donc un programme complet :

(P3)
```
i, k, x : entier ;
début  x := 0 ;
    k := 1 ;
    si  k ≤ n  alors  i := k  sinon  allera  f  fsi ;
    faire
        si  val (TE,i) ≥ 0  alors  x := x + val (TE,i)  fsi ;
        k := k + 1 ;
        si  k ≤ n  alors  i := k  sinon  allera  f  fsi
    ffaire ;
    f : y := x
fin ;
```

Les programmes P1, P2, P3, pourtant issus d'un même programme très simple, présentent des différences de structure très visibles. Ils sont évidemment équivalents, par construction, et on ne peut manquer de souligner l'analogie entre ces différentes formes d'un même programme et les versions successives de transformations de programmes dans le système d'Arsac [ARS79]. On peut penser que la programmation à l'aide de composition d'itérateurs est beaucoup plus simple et "structurée" qu'avec les structures de contrôle et procédures habituelles. Il y a là un travail d'application à compléter.

Note : des éléments de ce paragraphe ont été étudiés également par J.P. PAILLARD (Assistant IUT II - GRENOBLE).

1.3.4 - Vérification des compositions d'itérateurs

On propose, dans ce paragraphe, de calculer les conditions de vérification d'un programme utilisant les itérateurs en suivant la méthode axiomatique. La difficulté est que les itérateurs n'ont pas une forme fixe à la manière du tantque-faire ou même des constructions d'Alphard ; aussi, chaque définition d'itération doit être accompagnée de son système de preuve. De plus, puisqu'on ne connaît pas a priori quelle va être l'utilisation d'un itérateur, on doit calculer des assertions assez fortes (peut-être optimales [COU77]) pour pouvoir représenter une axiomatisation de l'itérateur et en déduire les assertions finales d'une composition d'itérateurs. Supposons que l'on ait l'itérateur interv du § 1.3.3 ; une assertion invariante de la boucle associée à l'instruction donner (x) est :

$$x \geq n1 \wedge \text{suivant}(x) \neq x$$

où suivant (x) est mis pour la fonction qui détermine le x au pas suivant. Cependant, cette assertion n'est pas assez forte pour prouver des théorèmes du genre :

$$\text{somme (interv (1,n))} = \frac{n(n+1)}{2}$$

Il y a donc lieu de bien choisir les assertions qui vont caractériser une abstraction de structure de contrôle. Donnons les principes de la vérification que l'on appliquera à un exemple (sur ce sujet, voir aussi [DAH77]).

1.3.4.1 - Assertions associées à un itérateur-émetteur

Les assertions associées à un émetteur que l'on suppose à un seul résultat noté y sont au nombre de quatre. Il faut déterminer :

- init(y) : assertion qui fixe la valeur initiale de y ;
- inv (y) : assertion associée au(x) point(s) donner (y) de l'itérateur ;
- suivant(y) : expression d'un élément de la séquence temporaire en fonction du précédent ;
- eof(y) : assertion qui fixe la dernière valeur de la séquence temporelle, si elle existe ; si la séquence est infinie, eof(y) ≡ faux.

Exemple : Soit l'iter. déjà donné : interv (n1,n2) ; on a :

init(y) ≡ y = n1

inv(y) ≡ n1 ≤ y ≤ n2

suivant(y) = y+1

eof(y) ≡ y = n2

1.3.4.2 - Système de preuves associé à un récepteur

Le principe consiste à établir la preuve comme pour tout programme itératif, en construisant un nombre fini de chemins dans le graphe du programme entre deux assertions, mais en introduisant les trois prédicats et la fonction suivant de l'émetteur qui lui sera associé en paramètres de la vérification. Prenons un récepteur identifié par R est défini par :

```
proc R (seqt y : ...) ;
     qd eof alors δ fq ;
début α ;
   | faire β ;
   |   |   suivant (y) ;
   |   |   γ
   | ffaire
fin ;
```

Le système de preuve est alors :

$$\frac{P \wedge init(y)\ \{\alpha\ ;\ \beta\}\ P_0,\quad P_0 \wedge inv(y)\ \{\gamma\ ;\ \beta\}_y^{suivant(y)}\}\ P_0]_y^{suivant(y)},\quad P_0 \wedge eof(y)\ \{\delta\}\ P_1}{P\ \{R(I)\}\ P_1}$$

où $]_y^{suivant(y)}$ dénote la substitution uniforme de y par suivant(y) et où les assertions init(y), inv(y), eof(y) sont celles de l'émetteur I.

Appliquons ces règles au récepteur "somme" du § précédent, nous obtenons :

$$\frac{P \wedge init(el)\ \{x := 0\ ;\ x := x+el\}\ P_0,\quad P_0 \wedge inv(el)\ \{x := x+suivant(el)\}\ P_0]_{el}^{suivant(el)},\quad P_0 \wedge eof(el) \supset P_1}{P\ \ \{somme(I)\}\ P_1}$$

La vérification du programme :

vrai {k := somme (interv(1,n))} $k = \frac{n(n+1)}{2}$

dépend de la découverte de P_0 (invariant du récepteur) qui est trivial ici :

$$P_0 \equiv x = \frac{el(el+1)}{2}$$

En effet :

$$el = 1 \ \{x:=0 \ ; \ x:=x+el\} \ x=1 \wedge el=1 \supset x = \frac{el(el+1)}{2}$$

$$x = \frac{el(el+1)}{2} \wedge 1 \leq el \leq n \ \{x:=x+(el+1)\} \ x = \frac{(el+1)(el+2)}{2}$$

$$x = \frac{el(el+1)}{2} \wedge el = n \supset x = \frac{n(n+1)}{2}$$

1.3.5 - Itérateurs multi-séquences

On a indiqué au § 1.3.2.1 que les itérateurs pouvaient avoir plusieurs paramètres "séquence temporelle". Si le principe de définition est identique à celui des actions d'itération plus simples, telles que celles que nous avons étudiées, l'utilisation peut poser quelques problèmes. Soient les exemples suivants d'itérateurs qui font l'éclatement en monotonies et la fusion ordonnée d'éléments de type t, avec la propriété ordre-total :

```
def m (t : ordre-total (≤)) ;
  | proc éclatement (sortie seqt s1 : t ou (eof), seqt s2 : t ou (eof),
  |                  entrée seqt u : t) ;
  | qd eof alors (eof a s1, s2) fq ;
  | x : t ; b : bool ;
début b := vrai ; x := u ; suivant(u) ;
  | faire si b
  |     | alors si u ≤ x
  |     |       alors donner (u a s1)
  |     |       sinon b := ¬ b ; donner (u a s2) fsi
  |     | sinon si u ≤ x
  |     |       alors donner (u a s2)
  |     |       sinon b := ¬ b ; donner (u a s1) fsi
  |     | fsi ;
  |     | x := u ; suivant (u)
  | ffaire
fin ;
```

```
proc fusion (seqt x, y:t) → (seqt t) ou (eof) ;
    qd x ⇒ eof ∧ y ⇒ eof alors eof
     | x ⇒ eof ∧ ¬ y ⇒ eof alors faire donner (y) ; suivant(y) ffaire
     | ¬ x ⇒ eof ∧ y ⇒ eof alors faire donner (x) ; suivant(x) ffaire fq
  début faire si x ≤ y
     |     | alors donner (x) ; suivant (x)
     |     | sinon donner (y) ; suivant (y) fsi
     |  ffaire
  fin ;
fdef ;
```

On donne pour compléter l'exemple les itérateurs de la lecture et de l'écriture des fichiers :

```
type fichier (t : tyfo) ;
 | proc éléments-de-fichier (f : fichier(t))→ (seqt t) ou (eof) ;
 |    qd eof alors fermer (f) ; eof fq
 |  début ouvrir (f) ;
 |    | faire donner (lire (f)) ffaire
 |  fin ;
 | proc faire-fichier (mod f : fichier (t), seqt x:t) ;
 |    qd eof alors fermer (f) fq ;
 |  début ouvrir (f) ;
 |    | faire écrire (f,x) ; suivant (x) ffaire
 |  fin
ftype ;
```

Ecrivons quelques programmes à l'aide de ces actions d'itération. Lorsque les itérateurs sont liés par plusieurs paramètres, on peut introduire la composition par le symbole ⋈ ; si l'on a :

```
proc g (sortie seqt x : t, y : τ) ;
proc f (seqt x : t) ;
```

la composition des deux actions s'écrira :

```
seqt a : t ;
   b : τ ;
     f (a) ⋈ g (a,b) ;
```

1. Lecture et écriture de fichier

```
     F1, F2 : fichier (τ) ;
début
     faire-fichier (F2, éléments-de-fichier (F1))
fin ;
```

2. Eclatement d'un fichier

```
        F1, FS1, FS2 : fichier(τ) ;
        seqt fs1, fs2 : τ ;
    début
        (faire-fichier (FS1, fs1), faire-fichier (FS2, fs2) ⋈
            éclatement (fs1, fs2, éléments-de-fichier (F1))
    fin ;
```

3. Fusion de deux fichiers

```
        F2, FS1, FS2 : fichier (τ) ;
    début
        faire-fichier (F2, fusion (éléments-de-fichier (FS1), éléments-de-fichier
                                                                (FS2)))
    fin ;
```

4. Supposons que l'on essaie d'écrire un seul programme qui fait à la fois l'éclatement et la fusion sans passer par des fichiers intermédiaires, mais simplement par des séquences temporaires intermédiaires :

```
faire-fichier (F2, fusion (fs1, fs2))
        ⋈ éclatement (fs1, fs2, éléments-de-fichier (F1)) ;
```

On s'aperçoit aisément que le programme ne peut marcher car il se peut que fusion attende un suivant de fs1 alors que l'éclatement ne peut fournir qu'un élément sur fs2. Une telle situation de blocage "logique" peut se produire lorsque plusieurs séquences sont construites à partir d'une séquence unique (ici : éléments-de-fichier (F1)). Une solution possible est que le compilateur prenne en charge de rompre la dépendance en transformant des séquences temporelles en objets et de réutiliser ces objets comme paramètres d'itérateurs pour poursuivre la chaîne de traitement.

Les conditions pour qu'une chaîne de traitement puisse être rompue sont qu'il existe un type d'objets qui puisse représenter une séquence. Cela peut être spécifié par des propriétés :

```
prop cvseq (f,g) sur t1, t2 : tyfo ;
    proc f(t1) → seqt t2 ;
    proc g(seqt t2) → t1 ;
```

Cette propriété établit que le type t1 est convertible en séquence temporelle de t2, telle que :

g(f(x)) = x

pour tout objet x : t1.

Mais cette propriété est insuffisante pour notre propos, car on veut passer d'une séquence temporaire à une autre séquence temporaire identique, via un objet. On définit donc :

prop isoseq (f,g) sur t1, t2 : tyfo ;

où f et g ont la même fonctionnalité que précédemment, mais de plus, f(g(x)) = x pour toute séquence temporaire x, considérée comme séquence spaciale. On a donc cvseq(f,g) ⊂ isoseq(f,g). D'une manière générale, on peut donc transformer un émetteur-récepteur en émetteur ou en récepteur à l'aide de f et g :

proc h (seqt t) → (seqt t)

peut devenir le receveur : g ⋉ h,

ou l'émetteur : h ⋉ f

et l'expression β ⋉ h ⋉ α

peut être rompue en : x : t1 ;

x : g ⋉ h ⋉ α ; β ⋉ h ⋉ f(x) ;

Dans notre exemple, on a la possibilité d'utiliser la propriété isoseq (éléments-de-fichier, faire-fichier) sur les fichiers, et le programme 4 peut être rompu en produisant le programme 2 suivi du programme 3.

1.4 - Méthodologie de programmation

1.4.1 - Bibliothèques de types et procédures

La première application de la programmation générique est la constitution de bibliothèques de types et de procédures. Il est aisé de voir que toutes les structures de données classiques (queue, pile, arbre, liste, fichier, etc) ainsi que de nombreux objets mathématiques (vecteurs, polynômes, matrices, ensembles, etc) peuvent être déclarés sous forme de types génériques munis de leurs accès élémentaires. Le mécanisme d'extension des types permet, de plus, de définir des ensembles de procédures sur un type en vue d'applications particulières (ex : extension de matrices pour le traitement des graphes, pour la résolution de systèmes d'équations, etc). Il peut être utile aussi de définir des extensions sur plusieurs types combinés.

1.4.2 - Construction modulaire descendante

Les méthodes de programmation descendante ont fait l'objet de nombreux articles et il est totalement inutile d'y revenir. On peut simplement constater que la plupart des langages existants ne permettent pas de décrire les programmes de façon rigoureusement descendante, car on ne peut étendre que les actions, et les langages (excepté Algol68) ne permettent pas de remplacer partout un identificateur par un appel de procédure (ex : en partie gauche d'affectation).

Un point important d'analyse descendante qui n'est pas suffisamment développé est celui de la programmation descendante des objets. Cela est assez bien résolu par l'introduction des types abstraits et des accès notés fonctionnellement. Là, Algol68 est insuffisant car il faut connaître le mode (struct, array, ...) pour pouvoir programmer les accès aux objets et le principe d'équivalence des modes ne convient pas non plus. En résumé, les deux principes de la programmation descendante sont :

1. Orthogonalité de la composition des actions ;
2. Abstraction des types et accès fonctionnels (principe de référence uniforme).

D'autre part, pour une véritable assistance de programmation descendante par le compilateur, il faut avoir la possibilité de spécifier syntaxiquement les objets à affiner avant de les programmer. Ainsi les modules de programmes de haut niveau peuvent être écrits et compilés avant que les modules de plus bas niveau ne soient eux-mêmes écrits. La seule spécification demandée par le compilateur est l'interface de compilation qui doit contenir :

1. Les fonctionnalités des procédures et des fonctions ;
2. La représentation des types (et des constantes) et la fonctionnalité des accès.

On voit que le deuxième point qui permet la compilation séparée descendante est en contradiction avec la conception abstraite descendante, car, puisqu'on ne connaît pas le corps des actions, on ne veut pas connaître non plus (ni anticiper le choix de) la représentation des objets. Le langage GREEN [GRE79] pour lequel la compilation séparée est un objectif primordial exige la

spécification de la représentation des types. Nous avons choisi pour notre part de ne pas spécifier la représentation avec le type car cela rendrait impossible la multi-représentation. Un interface de représentation doit donc être précisé pour pouvoir compiler un programme.

Le dernier point général relatif à la programmation descendante est celui de l'efficacité. Il est certain que des accès à des champs de structures (enregistrements) compilés via un appel de fonction sont tout à fait coûteux à l'exécution. La réponse à cet argument est d'optimiser les accès par une incorporation du code de la procédure appelée dans la procédure appelante, selon des techniques classiques. Mais là encore, on ne peut avoir à la fois le beurre et les canons, c'est-à-dire que le choix d'un code efficace par inclusion va à l'encontre de l'objectif de compilation (et surtout recompilation) modulaire des programmes, qui peut être très important dans le cadre de gros systèmes.

L'apport de la généricité n'est pas essentiel dans la méthode de programmation descendante, puisqu'on connaît dès le début du projet le type d'application que l'on veut traiter. Par contre, comme on vient de le voir, l'abstraction est fondamentale, ainsi que la multi-représentation des types qui est une forme privilégiée de la multiplicité des algorithmes pour réaliser une action.

1.4.3 - Abstraction d'Algorithmes

Ce dernier aspect de méthode de programmation n'a pas été beaucoup développé, par défaut de support en logiciel. On peut le résumer ainsi : étant donné un algorithme A qui calcule une fonction ϕ sur un domaine $\mathcal{D}$.

1. Trouver les conditions suffisantes C dans le domaine $\mathcal{D}$ pour que A calcule ϕ dans tout domaine $\mathcal{D}'$ qui satisfait C ;

ou bien

2. Trouver des conditions d'abstraction A telles que A(A) calcule A(ϕ) sur $\mathcal{D}$.

Le premier cas correspond aux cas classiques de généricité comme les problèmes de tri sur tous les domaines qui admettent un ordre total, etc. On trouvera des exemples de cette abstraction tout au long de cette thèse. Le deuxième

cas rejoint davantage les préoccupations de synthèse d'algorithme et l'on en trouve une présentation dans [DER75]. Je vais reprendre l'exemple de cet article dans le cadre de la généricité comme abstraction d'algorithmes.

Soit le programme de recherche dichotomique dans un tableau trié. Les assertions sont données en commentaire :

```
proc recherche (n : entier, x : réel, T : tableau (1..n , réel)) → entier ;
   {(∃k) (1 ≤ k ≤ n) (x = T(k)) ∧
    (∀k, ℓ) (1 ≤ k ≤ ℓ ≤ n) (T(k) ≤ T(ℓ))}
     z, y, h : entier ;
   début (z,y) := (1,n) ;
      tantque (x ≥ T(z) ∧ x ≤ T(y) ∧ 1 ≤ z ≤ y ≤ n)
               z < y
      faire h := (z+y) ÷ 2 ;
           si x ≤ T(h) alors y := h
           sinon z := h+1 fsi
      ffaire
   retourner (z) {x ≥ T(z) ∧ x ≤ T(z) ∧ 1 ≤ z ≤ n}
   fin ;
```

Les conditions de vérification du programme sont :

$x \geq T(1),\ x \leq T(n)$ (initialisation)

$(\forall z,y)\ (x \geq T(z) \wedge x \leq T(y) \wedge 1 \leq z < y \leq n)$

$\wedge \neg (x \leq T((z+y) \div 2)) \supset x \geq T((z+y) \div 2+1)$ (boucle).

L'abstraction consiste ici à remplacer les prédicats

$x \geq T(i)$ par $p(i)$

$x \leq T(j)$ par $q(j)$

et de remplacer la condition de vérification de la boucle en :

$(\forall u)\ \neg q(u) \supset p(u+1)$.

Le programme abstrait est donc :

```
proc abstrait (ℓ, n : entier) → entier ;
    {ℓ ≤ n ∧ p(ℓ) ∧ q(n) ∧ (∀u) ¬ q(u) ⊃ p(u+1)}
     z,y,h : entier ;
  début (z,y) := (1,n) ;
    | tantque {p(z) ∧ q(y) ∧ ℓ ≤ z ≤ y ≤ n} z < y
    | faire h := (z+y) ÷ 2 ;
    |      si q(h) alors y := h
    |      sinon z := h+1 fsi
    | ffaire ;
    | retourner (z) {p(z) ∧ q(z) ∧ ℓ ≤ z ≤ n}
  fin ;
```

Les conditions de validité de l'abstraction sont exprimées par l'assertion d'entrée sur les prédicats p et q. Ce sont exactement des propriétés telles que celles du § 1.2.4. Syntaxiquement, on a :

```
prop inv (p,q) ;
   proc p (entier) → bool
   proc q (entier) → bool
fprop ;
```

Cette spécification est insuffisante ; la spécification axiomatique de p et q est donnée par l'assertion d'entrée ; on pourrait aussi donner une spécificatio algébrique (cf. 2.2.3.1) par :

```
p(ℓ) = vrai
q(n) = vrai
ℓ ≤ n = vrai
p(u+1) = si ¬ q(u) alors vrai fsi
```

Utilisons cette abstraction pour calculer rac2(x) pour $x \geq 0$ avec l'assertion de sortie :

$$z^2 \leq x \wedge x < (z+1)^2.$$

On identifie aisément $z^2 \leq x$ à p(z) et $x < (z+1)^2$ à q(z), ce qui donne le programme (en notation lambda) :

```
rac2 = λx. abstrait (0,x)
       avec inv[λz.z² ≤ x, λz.x < (z+1)²]
```

(Note : La clause avec indique quels prédicats devront être utilisés à la place de p et q ([BER77a])).

On vient de traver le cadre d'un langage de programmation générique. Dans le chapitre suivant, on va étudier les méthodes de spécification adaptées à cette forme de programmation, ce qui permettra de faire le point sur les méthodes de spécification en général.

2 - SPECIFICATION DES TYPES GENERIQUES

2.1. Modèle axiomatique

2.1.1. Présentation du modèle

La spécification axiomatique des programmes a eu son point de départ avec les articles de Floyd [FLO67] et de Hoare [HOA69]. La spécification et la vérification des programmes utilisant des structures de données complexes ont été étudiées par [HOA72][SPI75]. Le langage ALPHARD [WUL76] essaie d'intégrer les notions de types abstraits et de spécifications axiomatiques. Dans cette première partie, nous allons donner les caractéristiques de la spécification axiomatique en les illustrant à l'aide du langage ALPHARD ; dans une deuxième partie, nous montrerons les limites de la spécification axiomatique en ce qui concerne la généricité.

Un type abstrait ("form" en ALPHARD), comprend trois parties :

- la spécification ;
- la représentation ;
- l'implantation.

La spécification donne les éléments nécessaires à l'utilisation du type et à la vérification des programmes qui déclarent des objets de ce type, à savoir :

- les conditions sur les paramètres,
- l'invariant abstrait,
- les conditions initiales,
- la représentation abstraite des objets du type,
- la spécification des fonctions d'accès (fonctionnalité, pré et post-conditions).

On remarque que l'on a besoin de spécifier une "représentation abstraite" ; en effet, il faut exprimer l'effet des fonctions à l'aide d'un univers de référence comprenant des structures de données "de base" et des primitives

d'accès prédéfinies. On voit déjà là un point faible du système qui est l'existence de domaines d'objets prédéfinis, non spécifiables dans le langage.

Supposons que l'on dispose du type prédéfini séquence: seq, s'appliquant sur n'importe quel ensemble d'objets D, alors syntaxiquement, les séquences sont définies par :

<> : →seq(D) (séquence vide)

∿ : (seq(D),D) → seq(D)

La notation <d1,d2 ...,dn> est mise pour

<> ∿ <d1> ∿ <d2> ∿ ... ∿ <dn>

Les relations suivantes sont satisfaites sur les séquences :

x∿(y∿z) = (x∿y) ∿ z

premier (<d>) = d

x ≠ <> ⊃ premier (x∿<d>) = premier (x)

reste (<d>) = <>

longueur (<>) = 0

longueur (x∿<d>) = 1+longueur(x)

Si V est de type vecteur (D), on note seq(V,n,m) la séquence construite de V par:

n > m ⊃ seq(V,n,m) = <>

n ≤ m ⊃ seq(V,n,m) = <V(n), V(n+1),... V(m)>

On peut maintenant spécifier le type file d'attente (FIFO) dans le formalisme d'ALPHARD :

```
form  FIFO (T:form <:=>, N:integer) =
      ! commentaire: T est le type des éléments de la file sur
      ! lequel l'affectation doit être possible,
beginform
specifications
      requires N>0;                    ! condition sur les paramètres
      let FIFO = <...xj...> where xj is T ;
      ! représentation abstraite de la FIFO comme séquence
      invariant  0 ≤ longueur(FIFO) ≤ N ;
      initially  FIFO = <>;
      function
         ENQ (q:FIFO,x:T)
```

```
            pre 0 ≤longueur(q) < N
            post q=q' ∿ x
      !fonction d'entrée d'un élément x dans la file q
      DEQ (q:FIFO) returns x:T
            pre 0 ≤ longueur(q) ≤ N
            post x=premier(q') ∧ q=reste(q')
      ! fonction de sortie d'un élément ;
      ! la notation q' indique la valeur de q avant l'exécution
      ! de la fonction.
```

D'après cette spécification, on voit donc qu'un objet FIFO est assimilé à un objet seq, pour lequel on restreint l'ensemble des fonctions qui lui sont applicables.

La représentation indique::

- la représentation concrète des objets,
- la fonction d'abstraction (au sens de Hoare),
- l'invariant concret,
- des états typiques de l'objet.

Continuons l'exemple :

```
représentation
      V : vecteur (T,0,N-1)
      F,B,NUM : integer
            ! variables qui représentent une FIFO
   init F := 0 ; NUM := 0 ; B := N-1 ;
   rep (V,F,B,NUM) =
      if NUM=0 then <>
      else if F ≤ B then seq(V,F,B)
            else seq(V,F,N-1) ∿ seq(V,0,B) ;
            ! séquence définie sur la représentation
   invariant (0 ≤ NUM ≤ N) ∧ (0 ≤ F ≤ N-1) ∧ (0 ≤ B ≤ N-1)
      ∧(NUM=0 ∧ N=((B+N-F) mod N)+1
            ∨NUM > 0 ∧ NUM = ((B+N-F) mod N)+1) ;
   states  mt  when NUM=0
           normal when 0 < NUM < N
           full when NUM=N
           err otherwise ;
```

Si l'on note I_a et I_c respectivement les invariants abstraits et concrets, x un élément du domaine concret et rep(x) l'objet abstrait représenté , les conditions de vérification de la représentation sont :

$$I_c(x) \supset I_a(rep(x))$$

La vérification de l'initialisation, avec β_{req} , β_{init} les prédicats requires et initially et "init" l'instruction d'initialisation :

$$\beta_{req} \{init\} \quad \beta_{init}(rep(x)) \wedge I_c(x)$$

Enfin, la partie implantation donne le corps de chaque fonction et les assertions d'entrée et de sortie concrètes.

```
implementation
   body ENQ                          ! fonctionnalité dans la spécification
        out(q.B=(q.B'+1) mod N ∧ q.V=q.V'[q.B/x]
                 ∧q.NUM = q.NUM'+1)
        = mt, normal :: (q.B:=(q.B+1) mod N ;
                 q.V[q.B]:=x ;
                 q.NUM:=q.NUM+1) ;
          otherwise :: FAIL ;
   body DEQ
        out(q.F=(q.F'+1) mod N∧ x=q.V'[q.F']∧ q.NUM=q.NUM'-1)
        = normal, full::(x:=q.v[q.F];
                         q.F:=(q.F+1) mod N ;
                         q.NUM:=q.NUM-1) ;
          otherwise :: FAIL ;
endform
```

Pour chaque fonction, on note β_{in} et β_{out} les assertions d'entrée et de sortie concrètes, C, le corps et β_{pre} et β_{post} les pré et post conditions abstraites. Les théorèmes à démontrer pour assurer la validité des fonctions sont :

1 - $\beta_{in}(x) \wedge I_c(x) \{C\} \quad \beta_{out}(x) \wedge I_c(x)$

2 - $I_c(x) \wedge \beta_{pre}(rep(x)) \supset \beta_{in}(x)$

3 - $I_c(x) \wedge \beta_{pre}(rep(x')) \wedge \beta_{out}(x) \supset \beta_{post}(rep(x))$

Par induction sur l'initialisation et l'ensemble des fonctions d'un type, on démontre que $I_c(x)$ est toujours vrai dans tous les programmes qui utilisent le type abstrait, et donc $I_a(rep(x))$ est toujours vrai. D'autre part, les théorèmes 2 et 3 assurent que la précondition implique l'assertion d'entrée et

que d'autre part la post-condition est toujours vraie si l'implantation est correcte.

Les preuves sont donc bien séparées entre les vérifications internes au type abstrait qui font intervenir l'invariant concret I_c et les vérifications de programme hors type abstrait qui n'utilisent que les pré et post conditions des fonctions, sans souci de la représentation.

Voyons maintenant les limites de la spécification axiomatique, notamment par rapport à la généricité.

2.1.2. Limites du modèle axiomatique

2.1.2.1. Définition des types

Comme nous l'avons déjà signalé, le modèle axiomatique n'est pas constructif, c'est-à-dire qu'il ne permet de définir de nouveaux types que comme restriction de types existants (cf. [BER77a]). Les types abstraits primitifs (dans l'exemple : séquence) ne peuvent pas être décrits dans le même formalisme. De ce fait, deux solutions sont possibles :

1 - Avoir un ensemble de constructeurs abstraits prédéfinis, à partir desquels il doit être possible de définir tous les autres types, mais cette approche conduit à l'alternative suivante :

a - l'ensemble des constructeurs est réduit, mais les fonctions d'accès sont complexes, ce qui rend complexes leur axiomatisation et les preuves basées sur ces constructeurs dans le domaine abstrait (exemple : dans [SHA76b], le type abstrait arbre-binaire est défini abstraitement par restriction sur le constructeur prédéfini graphe avec arcs étiquetés) ;

b - prédéfinir un très grand nombre de constructeurs standard , couvrant toutes les structures de données classiques, mais alors on n'a plus besoin d'un système puissant de définition de types par l'utilisateur, et l'apprentissage du langage devient particulièrement long et difficile.

2 - Avoir un langage de spécification de constructeurs de base. Ce langage existe, c'est le langage algébrique que nous utiliserons au §2.2. Mais il est

clair que l'introduction d'un deuxième langage de spécification rend l'ensemble extrêmement lourd, d'autant plus que le langage algébrique est suffisant comme nous le verrons. Seul le problème des preuves de programmes demeure avec le formalisme algébrique, mais il sera résolu au chapitre 3.

2.1.2.2. Spécification des opérations formelles

Lorsqu'on a des types formels, il faut presque toujours avoir des opérations sur ces types pour pouvoir les utiliser. Or la spécification axiomatique exprime l'effet de l'opération et non ses caractéristiques. Aussi une spécification axiomatique (au sens de Hoare, c'est-à-dire en logique du premier ordre) ne peut pas être utilisée pour exprimer les propriétés d'une classe de fonctions, indépendamment de leur effet particulier. La seule caractérisation possible est en général une spécification du second ordre:

$$\text{associative}(f) =_{def} (f:(t,t) \to t) \wedge (\forall x,y,z:t)\ f(f(x,y),z) = f(x,f(y,z))$$

$$\text{égalité}(g) =_{def} (g:(t,t) \to \text{bool}) \wedge (\forall x,y,z:t)\ (g(x,x) \wedge (g(x,y) \supset g(y,x)) \wedge (g(x,y) \wedge g(y,z) \supset g(x,z)))$$

$$\text{affectation}\ (:=) =_{def} (:= :(\underline{\text{mod}}\ t,t))\ (\forall \text{ predicat } P)\ wp\ (x := y,P) = P^{y}_{x}$$

(où wp est la plus faible précondition qui implique que P est vrai à la sortie de l'affectation si celle-ci termine).

Cette caractérisation peut servir pour montrer qu'une fonction dont on connaît la spécification axiomatique satisfait un prédicat. Par exemple, soit la spécification :

$$\text{eq-vecteur}(v,w) =_{def} (\text{longueur}(v) = \text{longueur}(w)) \wedge (\forall i)\ (1 \leq i \leq \text{longueur}(v))\ (v(i) = w(i))$$

Si l'égalité sur les éléments des vecteurs satisfait le prédicat "égalité", alors on peut démontrer qu'il en est de même pour eq-vecteur, par substitution de g par

eq-vecteur, et du type t par le type vecteur. Les propositions de spécification de fonctions par pré et post conditions ne s'appliquent pas aux fonctions formelles et donc un nouveau formalisme par prédicats du second ordre devrait être introduit. Voila pourquoi la généricité en ALPHARD a toujours été sommaire, par défaut de solution générale.

D'autres problèmes plus techniques tels que ceux soulevés en [BER78] ne peuvent pas non plus être résolus en ALPHARD. En particulier, la liaison de fonctions effectives aux fonctions formelles exige théoriquement une démonstration de théorème, et les ambiguïtés éventuelles ne peuvent pas être résolues.

En conclusion, le formalisme axiomatique est agréable, car on dispose avec lui d'un mécanisme puissant de preuves de programmes, mais il n'est pas adapté à l'extensibilité des types ni à la généricité. Dans la suite, nous présentons le modèle algébrique de spécification des types qui sera approfondi pour englober notamment les propriétés et les itérateurs.

2.2. Modèle algébrique

2.2.1. Fondements mathématiques

La spécification algébrique des types abstraits se base essentiellement sur l'idée qu'un type est constitué d'un ensemble d'opérations, et que la "valeur" d'un objet d'un type est construite à partir de la séquence des opérations qui s'appliquent sur lui. L'abstraction provient du fait que l'on n'utilise, ni une représentation particulière pour les objets, ni même un domaine "abstrait", comme dans l'approche axiomatique, dans lequel on exprimerait l'effet des fonctions. Nous allons présenter d'abord les définitions de base, en renvoyant aux ouvrages classiques d'algèbre pour l'exposé formel de ces notions et les démonstrations [COH65] [BIR70].

eq-vecteur, et du type t par le type vecteur. Les propositions de spécification de fonctions par pré et post conditions ne s'appliquent pas aux fonctions formelles et donc un nouveau formalisme par prédicats du second ordre devrait être introduit. Voila pourquoi la généricité en ALPHARD a toujours été sommaire, par défaut de solution générale.

D'autres problèmes plus techniques tels que ceux soulevés en [BER78] ne peuvent pas non plus être résolus en ALPHARD. En particulier, la liaison de fonctions effectives aux fonctions formelles exige théoriquement une démonstration de théorème, et les ambiguïtés éventuelles ne peuvent pas être résolues.

En conclusion, le formalisme axiomatique est agréable, car on dispose avec lui d'un mécanisme puissant de preuves de programmes, mais il n'est pas adapté à l'extensibilité des types ni à la généricité. Dans la suite, nous présentons le modèle algébrique de spécification des types qui sera approfondi pour englober notamment les propriétés et les itérateurs.

2.2. Modèle algébrique

2.2.1. Fondements mathématiques

La spécification algébrique des types abstraits se base essentiellement sur l'idée qu'un type est constitué d'un ensemble d'opérations, et que la "valeur" d'un objet d'un type est construite à partir de la séquence des opérations qui s'appliquent sur lui. L'abstraction provient du fait que l'on n'utilise, ni une représentation particulière pour les objets, ni même un domaine "abstrait", comme dans l'approche axiomatique, dans lequel on exprimerait l'effet des fonctions. Nous allons présenter d'abord les définitions de base, en renvoyant aux ouvrages classiques d'algèbre pour l'exposé formel de ces notions et les démonstrations [COH65] [BIR70].

Définitions

2.2.1.1. Une présentation est la donnée d'un ensemble d'identificateurs de domaines : S (anglais : sort) et d'un ensemble d'opérateurs Σ sur S. A chaque opérateurs $\sigma\in\Sigma$ est associé une fonctionnalité $\phi(\sigma)$ qui est une chaîne x,s où $x \in S^*$ et $s \in S$. Les opérateurs dont la fonctionnalité $\phi(\sigma)$ est de la forme λ,s sont les constantes (opérateurs sans paramètres) du domaine S.

2.2.1.2. Etant donnée une présentation $<S,\Sigma>$, une $<S,\Sigma>$-algèbre A (ou algèbre d'opérateurs Σ) est la donnée d'une famille de domaines $\{A_s | s \in S\}$ et d'opérateurs σ_A associées à tout $\sigma\in\Sigma$ telles que, si $\phi(\sigma) = s_1 \ldots s_n, s$ alors σ_A est une application : $A_{s_1} x \ldots x A_{s_n} \rightarrow A_s$

2.2.1.3. Un homomorphisme $h : A \rightarrow B$ où A et B sont des $<S,\Sigma>$-Algèbres est une famille d'applications $\{h_s : A_s \rightarrow B_s | s \in S\}$ qui préserve les opérations:

$\forall\sigma$: $\phi(\sigma) = \lambda,s$ alors $h_s(\sigma_A) = \sigma_B$

$\forall\sigma$: $\phi(\sigma) = s_1 \ldots s_n, s$ et $a_i \in A_{s_i}$, $\forall i \in [1,n]$

alors $h_s(\sigma_A(a_1, \ldots a_n)) = \sigma_B(h_{s_1}(a_1^i), \ldots h_{s_n}(a_n))$.

2.2.1.4. La $<S,\Sigma>$-algèbre des mots ou algèbre sur l'univers de Herbrand (notée $T_{S,\Sigma}$) est la famille de langages $\{L_s | s \in S\}$ définis par les grammaires hors contexte G_s de la manière suivante :

$VT = \{\sigma_T | \sigma \in \Sigma\} \cup \{"(", ",", ")"\}$

$VN = \{N_s | s \in S\}$

$P = \{N_s \rightarrow \sigma_T | \phi(\sigma) = \lambda,s\} \cup$

$\{N_s \rightarrow \sigma_T(N_{s_1}, \ldots N_{s_n}) | \phi(\sigma) = s_1 \ldots s_n, s\}$

et pour chaque G_s l'axiome est N_s.

2.2.1.5. Soit X une famille d'ensembles disjoints: $\{X_s | s \in S\}$.

On considère $x \in X_s$ comme un opérateur tel que $\phi(x) = \lambda,s$. Alors l'algèbre $T_{S,\Sigma'}$, où $\Sigma' = \Sigma \cup X$ est appelée algèbre des mots sur les générateurs X notée $T_{S,\Sigma}(X))$ ou algèbre librement engendrée par X.

Les ensembles X_s sont appelés les ensembles de variables dans les équations (§ 2.2.1.10).

Théorèmes

Voici quelques résultats qui seront utiles dans les paragraphes suivants. Les démonstrations se trouvent essentiellement dans les références [ADJ76] [ADJ78b] et dans les articles originaux.

2.2.1.6.

Les $<S,\Sigma>$-algèbres et leurs homomorphismes forment une catégorie : $ALG(S,\Sigma)$ dans laquelle $T_{S,\Sigma}$ est initiale ([HIG63] [BIR70]).

2.2.1.7.

Si A est une $<S,\Sigma>$-algèbre et f, une application : $X \to A$, alors il existe un homomorphisme unique $f^{\#} : T_{S,\Sigma}(X) \to A$ tel que $f^{\#}$ est une extension de f, c'est-à-dire : $\forall x \in X : f^{\#}(x)=f(x)$.

2.2.1.8. Exemples

1 . Dans les hypothèses du 2.2.1.7., si A est $T_{S,\Sigma}$, alors l'application f est une substitution et $f^{\#}$ est l'application d'une substitution à un terme de $T_{S,\Sigma}(X)$.

2 . Soit la $<S,\Sigma>$-algèbre X de l'ensemble des variables avec comme opérations Σ, l'union des ensembles, et soit $v : X \to X$ définie par : $v_s(x) =$ <u>si</u> $x \in X_s$ <u>alors</u> $\{x\}$ <u>sinon</u> $\emptyset$. $v^{\#} : T_{S,\Sigma}(X) \to X$ est alors la fonction var qui à tout terme t de $T_{S,\Sigma}(X)$ associe l'ensemble des variables qui se trouvent dans t.

Définitions

2.2.1.9.

Une $<S,\Sigma>$-congruence Ξ sur une $<S,\Sigma>$-algèbre A est une famille de relations d'équivalences $\{\Xi_s \mid s \in S\}$ possédant la propriété de substitution : pour tout $\sigma \in \Sigma$ tel que $\phi(\sigma)=s_1 \ldots s_n,s$ si $a_i,b_i \in A_{s_i}$ et si $a_i \Xi_{s_i} b_i$, $\forall i \in [1,n]$, alors

$$\sigma_A(a_1,\ldots a_n) \Xi_s \sigma_A(b_1,\ldots b_n).$$

2.2.1.10.

Une S,Σ-équation dans s:e est définie par $e_1 = e_2$ où $e_1, e_2 \in T_{S,\Sigma}(X)$ dans s. Une S,Σ-algèbre A satisfait e ssi pour toute application θ : var(e) → A, on a $\theta^{\#}(e_1) = \theta^{\#}(e_2)$.

Une S,Σ-algèbre A satisfait un ensemble d'équations E si elle satisfait toute $e \in E$.

2.2.1.11.

Si A est une S,Σ-algèbre et Ξ une S,Σ-congruence sur A, alors A/Ξ, algèbre quotient est une S,Σ-algèbre, et il existe un homomorphisme canonique (unique) h : A → A/Ξ.

2.2.1.12. **Proposition**

Soit A une S,Σ-algèbre, alors les relations de congruence sur A forment un treillis complet avec :

$\perp$: la relation d'identité ;

$\top$: la relation universelle ;

$\cup$: la congruence engendrée par fermeture de la propriété de substitution sur l'union des relations ;

$\cap$: intersection des relations.

$\subseteq$: inclusion des relations.

2.2.1.13. **Définition**

Soient une S,Σ-présentation et un ensemble d'équations E sur $T_{S,\Sigma}(X)$. On définit Ξ_E, la relation de congruence engendrée par E sur $T_{S,\Sigma}$ comme l'intersection de toutes les relations de congruences qui contiennent la relation $E(T_{S,\Sigma}(X))$ où $E(T_{S,\Sigma}(X))$ est l'ensemble des relations d'équivalence obtenues de E par substitution. On note $T_{S,\Sigma,E}$ l'algèbre quotient $T_{S,\Sigma}/\Xi_E$.

2.2.1.14. **Théorème**

$T_{S,\Sigma,E}$ est initiale dans la catégorie ALG(S,Σ,E) des S,Σ-algèbres qui satisfont E [HIG63] [BIR70] [ADJ73].

2.2.1.15. Proposition

Soit $h : A \to B$ un homomorphisme de S,Σ-algèbres, et $\equiv_h$ définie par

$$a \equiv_h a' \text{ ssi } h(a) = h(a')$$

alors $\equiv_h$ est une S,Σ-congruence et si h est surjective, alors $A/\equiv_h$ est isomorphe à B.

2.2.2. Spécification algébrique des types

2.2.2.1. Approche de ADJ [ADJ76], Zilles [ZIL74]

Définitions

2.2.2.1.1.

Un type abstrait défini par un triplet $<S,\Sigma,E>$ est la classe d'équivalence des algèbres initiales dans la catégorie $ALG(S,\Sigma,E)$.

Les relations entre algèbres dans $ALG(S,\Sigma,E)$ étant les homomorphismes, la classe d'équivalence des algèbres initiales est constituée des algèbres isomorphes à $T_{S,\Sigma,E}$, d'après 2.2.1.14.

2.2.2.1.2.

Soient τ_1 : $<S,\Sigma,E>$ un type abstrait et $<S \cup S',\Sigma',E'>$ une présentation avec $S \cap S' = \Sigma \cap \Sigma' = \emptyset$. Posons $\overline{S}=S \cup S'$, $\overline{\Sigma}=\Sigma \cup \Sigma'$, $\overline{E}=E \cup E'$, alors :

$<\overline{S},\overline{\Sigma},\overline{E}>$ est une <u>extension</u> de τ_1 si $T_{S,\Sigma,E}$ est isomorphe à $T_{\overline{S},\overline{\Sigma},\overline{E}}$ en tant que $<S,\Sigma>$-algèbre.

Si $S'=\emptyset$, alors $<S,\overline{\Sigma},\overline{E}>$ est un <u>enrichissement</u> de τ_1.

2.2.2.1.3. Exemples

1 - Le type des entiers naturels <u>nat</u> est défini par :

$<S = \{nat\}, \Sigma = \{0, succ\}>$

et les fonctionnalités : $\phi(0) = \lambda, nat$

$\phi(succ) = nat, nat.$

(par la suite, on définira les opérateurs et leur fonctionnalité en une seule écriture par :

$0 : \rightarrow nat$

$succ : nat \rightarrow nat$)

Par convention, un type aura le même nom que celui du domaine "principal" souligné.

Les classes d'équivalence dans T_{nat} sont :

$\{0\}, \{succ(0)\}, \{succ(succ(0))\} \ldots$

On peut"enrichir" nat à l'aide des opérateurs :

$\Sigma' = \{pred : nat \rightarrow nat$

$+ : nat \times nat \rightarrow nat$

$\dot{-} : nat \times nat \rightarrow nat\}$

et des équations :

(E) $pred(0) = 0$

$pred(succ(N)) = N$

$N+0 = 0$

$N+succ(N') = succ(N+N')$

$N\dot{-}0 = N$

$N\dot{-}succ(N') = pred(N\dot{-}N')$

Le type enrichi $T_{\overline{nat}}$ possède maintenant les classes de termes :

$\{0, pred(0), pred(succ(0)), 0+0, \ldots\},$

$\{succ(0), pred(succ(succ(0))), succ(0)+0, \ldots\}$

On peut vérifier qu'il s'agit d'un enrichissement en considérant l'application: $h : T_{nat} \rightarrow T_{\overline{nat}}$ telle que $h(N) = [N]$ (classe d'équivalence de N). Comme $T_{\overline{nat}}$ est une $\langle S,\Sigma\rangle$-algèbre, h est un homomorphisme (d'après 2.2.1.6.). Il reste à montrer que h est bijective :

- h est surjective, car tout terme de $T_{\overline{nat}}$ est congru à un terme de T_{nat} ; en effet, par induction, on a :

$pred(0) \equiv 0$

si $N \in T_{nat}$ et $N \equiv M_{nat}$ alors

$pred(succ(N)) \equiv N$

$pred(succ(N)) \equiv N_{nat}$

donc $pred(N) \equiv N'_{nat}$ si $N \equiv M_{nat}$ (1)

De même pour les autres opérations :

Si $N \equiv M_{nat}$ alors $N+0 \equiv N$

$N+0 \equiv M_{nat}$

Si $N+N' \equiv M_{nat}$ alors $N+succ(N') \equiv succ(N+N')$

$N+succ(N') \equiv succ(M_{nat})$

$N+succ(N') \equiv M'_{nat}$

Si $N \equiv M_{nat}$ alors $N \dot{-} 0 \equiv N$

$N \dot{-} 0 \equiv M_{nat}$

Si $N \dot{-} N' \equiv M_{nat}$ alors $N \dot{-} succ(N') \equiv pred(N \dot{-} N')$

$N \dot{-} succ(N') \equiv pred(M_{nat})$

et d'après (1) $N \dot{-} succ(N') \equiv M'_{nat}$

- h est injective car si $N \not\equiv M$ dans T_{nat}, alors $h(N) \neq h(M)$, c'est-à-dire $[N] \neq [M]$. Autrement dit, deux termes de T_{nat} ne peuvent pas être congrus dans la relation engendrée par les équations (E).

2 - Considérons le type $\overline{nat}$ précédemment défini auquel on ajoute la constante naterr et le type pile :

Σ_{pile} = {creerp : → pile
empiler: pile x nat → pile
sommet : pile → nat
depiler: pile → pile
pilerr : → pile
naterr : → nat}

et les équations :

sommet(creerp) = naterr

sommet(empiler(P,N)) = N

depiler (creerp) = pilerr

depiler(empiler(P,N)) = P

Le type $\overline{\text{nat}}$ contient les éléments :

{natterr, sommet(creerp), ...}

mais aussi:

{sommet(pilerr)} {sommet(depiler(pilerr))} ...

qui modifient les classes de congruence de $\overline{\text{nat}}$. La solution au problème de l'introduction des erreurs est donnée au §2.2.2.1.6.

2.2.2.1.4. Définitions

Un type τ : <S,Σ,E> est protégé dans une spécification $<\overline{S},\overline{\Sigma},\overline{E}>$, si $<\overline{S},\overline{\Sigma},\overline{E}>$ est une extension de τ .

2.2.2.1.5. Axiomes conditionnels

Dans toute spécification algébrique, le type bool joue un rôle déterminant. Il permet en particulier de définir les axiomes conditionnels. Le type bool ne contient que deux classes, celles des constantes "vrai" et "faux": S={bool}

$$\Sigma = \{ \text{vrai} : \rightarrow \text{bool}$$
$$\text{faux} : \rightarrow \text{bool}$$
$$\neg : \text{bool} \rightarrow \text{bool}$$
$$\wedge : \text{bool} \times \text{bool} \rightarrow \text{bool}$$
$$\vee : \text{bool} \times \text{bool} \rightarrow \text{bool}$$
$$\Leftarrow : \text{bool} \times \text{bool} \rightarrow \text{bool}\}$$

$$E = \{ \neg\text{vrai} = \text{faux}$$
$$\neg\text{faux} = \text{vrai}$$
$$\text{vrai} \wedge b = b$$
$$\text{faux} \wedge b = \text{faux}$$
$$b \wedge b' = b' \wedge b$$
$$b \vee b' = \neg(\neg b) \wedge (\neg b'))$$
$$b \Leftarrow b' = b \vee \neg b'\}$$

Pour tout type t (y compris bool), il est possible d'étendre t+bool par l'opération si :

$$\text{si} : \text{bool} \times t \times t \rightarrow t$$

telle que $\text{si}(\text{vrai},x,y) = x$

$\text{si}(\text{faux},x,y) = y$

Une équation dans t de la forme :

L = si b alors R1 sinon R2 fsi

est mise pour :

L = si(b,R1,R2)

Une équation dans t de la forme :

L = si b alors R fsi

est mise pour

L = si(b,R,L).

[ADJ77] [ADJ78a] montrent qu'il existe des types que l'on ne peut pas spécifier sans axiomes conditionnels. Ces axiomes augmentent donc le pouvoir d'expression des systèmes équationnels.

2.2.2.1.6. Traitement des erreurs [ADJ76]

La solution du problème posé au 2.2.2.1.3. ne peut pas se résoudre en propageant directement les erreurs par :

depiler(pilerr) = pilerr

empiler(pilerr,N) = pilerr

sommet(pilerr) = naterr

En effet on peut alors déduire :

N = sommet(empiler(pilerr,N))
  = sommet(pilerr)
  = naterr

et tous les éléments de nat sont congrus à naterr (nat n'est pas protégé).

Il convient de distinguer les éléments qui sont des "erreurs" des éléments "ok" à l'aide d'équations conditionnelles. Pour spécifier le type pile correctement, on écrira,:

okpile : pile → bool

oknat : pile → bool

$siok_1$: pile x nat x nat → nat

$siok_2$: pile x nat x pile → pile

et les équations générales sur les "ok" :

okpile(pilerr) = faux

okpile(creerp) = vrai

okpile(empiler(P,N)) = okpile(P) ∧ oknat(N)

oknat(0) = vrai

oknat(succ(N)) = oknat(N)

oknat(naterr) = faux

etc. pour les opérations de nat

$siok_1(P,N,M)$ = si(okpile(P) ∧ oknat(N),M,naterr)

$siok_2(P,N,P')$ = si (okpile(P) ∧ oknat(N),P',pilerr)

équations spécifiques :

sommet(creerp) = naterr

$siok_1(P,N,sommet(empiler(P,N)))$
= $siok_1(P,N,N)$

depiler(creerp) = pilerr

$siok_2(P,N,depiler(empiler(P,N)))$
= $siok_2(P,N,P)$

propagation d'erreurs :

empiler(pilerr,N) = pilerr

empiler(P,naterr) = pilerr

depiler(pilerr) = pilerr

sommet(pilerr) = naterr

Evidemment la description complète d'un type abstrait avec traitement d'erreurs est très fastidieuse ; aussi il est possible de spécifier simplement, pour un type donné, les éléments et équations "ok", les éléments et équations d'erreurs, et d'en déduire automatiquement le système algébrique complet. Ici on aurait :

ok-opérations : {créerp, empiler, depiler, sommet}

err-opérations : {naterr, pilerr}

ok-equations : depiler(empiler(P,N)) = P
sommet(empiler(P,N)) = N

err-équations : depiler(creerp) = pilerr
sommet(creerp) = naterr

La déduction des équations exactes se fait par les étapes suivantes :

a - construire les équations spécifiques en introduisant les siok convenables : une équation ok est valide si ses variables sont ok.

b - engendrer les équations de propagation d'erreur : un terme est une erreur si l'un des arguments est erreur.

c - engendrer les équations ok : un terme est ok si ses arguments sont ok, excepté pour les opérateurs qui retournent des erreurs.

exemple : pour la pile, on n'a pas :

okpile(depiler(P)) = okpile(P)

car on pourrait déduire :

okpile(depiler(creerp)) = okpile(creerp) = vrai

et :

okpile(depiler(creerp)) = okpile(pilerr) = faux

Ce traitement des erreurs n'est pas très satisfaisant, car il multiplie les équations, oblige à tenir compte des erreurs dans les équivalences de types ; d'autre part, toutes les erreurs sont confondues par propagation, et ce qui est plus grave, dans une implantation qui serait proche de la spécification, les opérations (non erreur) devraient tester systématiquement si leurs arguments sont erreur ou ok. Pour résoudre quelques uns de ces problèmes on va résumer maintenant la méthode de [GOG77b] et on donnera au §2.2.3.4. notre propre conception du traitement des erreurs.

Une présentation avec erreurs est la donnée de $P = \langle S, \Sigma, \Xi, E, F \rangle$ où $\langle S, \Sigma, E \rangle$ est une présentation, Ξ l'ensemble des opérateurs d'erreur et F l'ensemble des équations d'erreur. Chaque domaine est divisé en valeurs-ok et valeurs-erreur (V_E, V_F) disjointes. Le domaine des valeurs-ok est engendré par les opérateurs-ok; mais le domaine des valeurs-erreur est engendré par les opérateurs-erreur et par les opérateurs-ok si un de leurs arguments est erreur. A l'inverse de l'approche précédente où les valeurs-erreur devaient être explicitement équivalentes à une constante erreur, on a tout un domaine de termes d'erreur (qui peut rester libre).

Dans l'exemple de la pile, le domaine ok est engendré par créerp, empiler, depiler, et sommet donne alors des termes congrus à des valeurs-ok de nat. Le domaine erreur est engendré par pilerr, empiler, depiler pour pile; naterr, sommet et les opérations de nat (sauf 0) pour nat ; les éléments identifiés entre les deux sortes de domaines sont dépiler(créerp) et pilerr, sommet (créerp) et naterr.

On évite les contradictions (ou incohérences) évoquées au début du paragraphe en imposant que les équations-ok (E) ne s'appliquent que si la partie droite et la partie gauche sont des termes-ok et à l'inverse, les équations-erreur (F) s'appliquent si l'une ou l'autre des deux parties est un terme-erreur. Une conséquence de ce principe d'application des équations est qu'un système formel qui veut traiter les erreurs de cette manière doit choisir une technique de réduction (d'évaluation) remontante (par valeur) [GOG77a]. L'avantage de la méthode est que l'on peut spécifier par les équations-erreur la façon dont on veut reconnaître les erreurs :

- pas d'équations autres que l'introduction des erreurs donne une erreur différente pour chaque terme d'erreur ;
- des équations : empiler (S,I) = pilerr, etc... identifient toutes les erreurs aux constantes d'erreur ;
- des équations : empiler (S,I) = S, etc... permettent de conserver l'état de l'erreur à travers les réductions suivantes.

Reste à redéfinir la notion de type protégé puisque l'utilisation d'un type peut conduire à l'introduction de valeurs-erreur dans ce type. On dira donc qu'un type est protégé (au sens large) si ses valeurs-ok sont protégées (au sens précédent). La notion d'homomorphisme entre algèbre d'erreur est aussi modifiée, mais nous ne rentrons pas dans les détails, renvoyant à [GOG77b].

2.2.2.1.7. Vérifications des spécifications

Etant donné une spécification d'un type A dans un formalisme quelconque et sa spécification algébrique τ : <S,Σ,E>, montrer l'équivalence de A et de τ revient à montrer que A en tant que <S,Σ,E>-algèbre est isomorphe à $T_{S,\Sigma,E}$. [ADJ76] montre l'équivalence du type nat avec IN défini à l'aide des axiomes de Peano. On peut traiter en exercice l'équivalence de nat avec le type int spécifié comme co-domaine de rétraction dans le modèle de Scott [SCO76].

Rappelons brièvement les définitions nécessaires à l'établissement de la preuve : une rétraction r est une application de Pw dans Pw (parties de IN) telle que $r \circ r = r$. On définit dans Pw les opérations :

$$\overline{0} = \{0\}$$

$$\text{plusun}(X) = \{x+1 \mid x \in X\}$$

$$\text{moinsun}(X) = \{x \mid x+1 \in X\}$$

$$\perp = \{\ \}\ ,\ \top = Pw$$

$Z \overline{\supset} X, Y$ a le sens suivant :

si $Z = \perp$ alors $\perp$

si $Z = \overline{0}$ alors X

si $Z \neq \perp$ et $0 \notin Z$ alors Y

si $Z \neq \overline{0}$ et $0 \in Z$ alors $\top$

Alors int est le co-domaine de la rétraction int :

$$\text{int} = \lambda X.\ X \overline{\supset} \overline{0}\ ,\ (\text{int}(\text{moinsun}(X)) \overline{\supset} X,\ X)$$

autrement dit, les éléments de Pw tels que int(X)=X.

1 - int est une algèbre avec les opérateurs : $\overline{0}$ et plusun.

En effet $\text{int}(\overline{0}) = \overline{0} \overline{\supset} \overline{0},\ (\text{int}(\text{moinsun}(\overline{0})) \overline{\supset} \overline{0},\ \overline{0})$

$= \overline{0}$

Si $X \in$ int, alors plusun (X) $\in$ int :

$\text{int}(\text{plusun}(X)) = \text{plusun}(X) \overline{\supset} \overline{0},\ (\text{int}(\text{moinsun}(\text{plusun}(X))) \overline{\supset} \text{plusun}(X),\ \text{plusun}(X))$

si plusun(X) = $\perp$ alors $\perp$

plusun(X) = $\top$ alors $\top$

sinon, comme $(\forall X)$, plusun(X) $\neq \overline{0}$ (dans P_W)

on a :

int(plusun(X)) = int(moinsun(plusun(X))) $\overrightarrow{\supset}$ plusun(X),plusun(X))

= int(X) $\overrightarrow{\supset}$ plusun(X), plusun(X)

= si $X \neq \perp$ et $X \neq \top$ alors plusun(X).

Donc on a :

$X = \perp$ alors plusun(X) = $\perp$

$X = \top$ alors plusun(X) = $\top$

$X \in$ int - $\{\perp,\top\}$ alors plusun(X) $\in$ int - $\{\perp,\top\}$

2 - Il y a homomorphisme entre int -$\{\perp,\top\}$ et T_{nat}.

Soit f : int - $\{\perp,\top\} \rightarrow T_{nat}$

telle que f = $\lambda X.X \overline{\supset} 0$, succ(f(moinsun(X)))

f est un homomorphisme :

$f(\overline{0}) = 0$

f(plusun(X)) = succ (f(moinsun(plusun(X))))

= succ(f(X))

Tous les éléments de int - $\{\perp,\top\}$ ont une image par f.

Comme par 2.2.1.6., il existe un homomorphisme unique entre T_{nat} et int - $\{\perp,\top\}$, cette dernière est initiale dans la catégorie de T_{nat} et donc isomorphe à T_{nat}.

Remarque : On pourrait montrer que $T_{nat} \cup \{naterr\}$ est isomorphe à int - $\{\top\}$ en identifiant naterr et $\perp$.

Revenons plus en détail aux techniques de vérifications ou équivalences de spécifications.

Soient un type τ : $\langle S,\Sigma,E\rangle$ et une $\langle S,\Sigma\rangle$-algèbre A.

- Méthode de vérification 1 ([ZIL79]) :

  a/ Montrer que A satisfait les équations E, ce qui implique, d'après 2.2.1.14 qu'il existe un homomorphisme unique $h : T_{S,\Sigma,E} \rightarrow A$;

  b/ Montrer que h est une bijection.

- Méthode de vérification 2 ([ADJ76]) :

  a/ Définir une application $h : A \rightarrow T_{S,\Sigma,E}$ en associant à chaque élément de A une classe de congruence, ou mieux, un représentant (§ 2.2.2.2.7.) de chaque classe ;

  b/ Montrer que h est un homomorphisme surjectif.

Note : On utilisera ces techniques pour vérifier la cohérence d'une représentation d'un type, et on comparera alors avec la méthode de Guttag (§ 2.2.2.3.).

2.2.2.2. Approche de Guttag [GUT76a] [GUT78]

La présentation de Guttag diffère sensiblement de celle que nous avons adoptée jusqu'ici. Néanmoins, il est possible de décrire la sémantique des types abstraits "selon Guttag" en utilisant le même formalisme. On soulignera au passage les différences avec l'approche précédente.

Notons d'abord que les définitions de types se font d'une manière hiérarchique: On doit spécifier un nouveau type, appelé type d'intérêt (ti) par rapport à un ensemble de types déjà définis, la base B qui est spécifiée par <B,Σ,E>. Initialement, B contient le type bool, comprenant deux constantes, vrai et faux, et les opérations habituelles.

2.2.2.2.1. Définitions

Parmi l'ensemble Σ des opérateurs associés à un type d'intérêt ti, on distingue :

- les générateurs (G) : opérateurs dont le co-domaine est ti ;
- les accès (A) : opérateurs dont le co-domaine est dans B ;

  note : les accès du type ti sont au moins un paramètre dans ti.

Les générateurs sont les opérateurs qui permettent de construire le domaine de T_{ti}.

Exemple : Soit le type ens (ensemble d'entiers).

Les opérateurs Σ sont :

créer : → ens

inserer : nat x ens → ens

enlever : nat x ens → ens

appartient : nat x ens → bool

La base B est {bool , nat}

Les équations E sont :

enlever(N,créer) = créer

enlever(N,inserer (M,S)) = si eq(N,M)
alors enlever(N,S)
sinon inserer (M,enlever(N,S)) fsi

appartient(N,créer) = faux

appartient(N,inserer (M,S)) = si eq(N,M)
alors vrai
sinon appartient(N,S) fsi

Les opérateurs générateurs sont : {créer,inserer ,enlever} et l'opérateur d'accès est appartient.

L'opérateur eq est obtenu par extension du type nat avec :

le : nat x nat → bool

eq : nat x nat → bool

et les équations:

le(0,N) = vrai

le(succ(N),0) = faux

le(succ(N), succ(M)) = le(M,N)

eq(N,M) = le(N,M) ∧ le(M,N)

fin exemple.

2.2.2.2.2. Définition

A chaque opérateur d'accès a∈A on peut faire correspondre la famille de termes où a est la racine à une seule variable du type d'intérêt ti , appelés accès unaires. On note cette famille de termes: au = {ν(x)}, où x est le paramètre de type ti.

On définit la relation de congruence Ξ_{au} dans T_{ti}, déduite de au, par :

$$t1 \ \Xi_{au} \ t2 \iff (\forall \nu \in au) \quad \nu(t1) \equiv \nu(t2)$$

où ≡ est la congruence utilisée pour le type du résultat de a.

- Congruence image des accès : Ξ_A dans T_{ti}

Pour un ensemble de fonctions d'accès A dont les accès unaires sont {au}, on a : $\Xi_A = \cap(\Xi_{au})$.

Par défaut, si A=∅, Ξ_A est la relation universelle sur T_{ti}.

2.2.2.2.3. Proposition

Soient le type τ construit sur la base B, A les accès de τ dans B et E les équations dans T_τ , alors : B est protégée par τ ssi $\Xi_E \subseteq \Xi_A$.

Preuve :

a/ (⟸) Soient T_B l'algèbre de Herbrand des termes de la base et $T_{B+\tau}$ la nouvelle algèbre de Herbrand. Si l'on considère dans $T_{B+\tau}$ les classes de congruence de T_B, celles-ci sont étendues des éléments de A, mais d'après l'hypothèse, elles ne sont pas modifiées par la relation Ξ_E. Donc $T_{B+\tau}$ est isomorphe en tant que B-algèbre à T_B et B est protégée.

b/ (⟹) Supposons qu'il existe x et y dans T_τ tels que $x \Xi_E y$ et $x \not\equiv_A y$. Alors par définition de Ξ_A, (∃au), ∃(ν∈au) tels que :

$$\nu(x) \not\equiv \nu(y)$$

Mais par congruence dans E :

$$\nu(x) \equiv \nu(y)$$

et B n'est pas protégée.

2.2.2.2.4. Catégorie GALG(S,Σ,E,B)

On considère la classe des algèbres <S,Σ,E> qui sont des extensions de B, notée GALG(S,Σ,E,B). Alors, GALG(S,Σ,E,B) est une sous-catégorie pleine de ALG(S,Σ,E), dans laquelle $T_\Sigma/\equiv_E$ est une algèbre initiale et $T_\Sigma/\equiv_A$ est une algèbre terminale.

2.2.2.2.5.

Un type abstrait (selon Guttag) définit par <τ+B,Σ,E>, où τ est le type d'intérêt, est la catégorie GALG(τ+B,Σ,E,B).

Exemples :

1 - Le type nat défini en 2.2.2.1.3. n'a pas de fonctions d'accès ni d'équations ; donc il peut être représenté par l'une quelconque des algèbres de la catégorie ALG(nat, {0,succ},∅), c'est-à-dire aussi bien T_{nat} qui est initiale que l'algèbre ne contenant qu'un seul élément, qui est terminale.

2 - Si on étend nat sur la base {bool} avec les opérateurs le et eq (cf. exemple, § 2.2.2.2.1.), alors le type nat : <nat+bool, {0,succ,le,eq}, E> est la catégorie GALG(nat+bool,Σ,E,bool) qui ne contient plus que les algèbres isomorphes à T_{nat}.

3 - Le type ens (§ 2.2.2.2.1.) définit une catégorie GALG(ens+{nat,bool}, Σ,E,bool) où l'algèbre initiale est l'algèbre des ensembles avec répétition et où l'ordre des insertions est significatif.

exemple :

inserer(0,créer) est différent
de inserer(0,inserer(0,créer)) ,

de même :

inserer(succ(0), inserer(0,créer)) est distinct
de inserer(0,inserer(succ(0),créer))

Par contre, dans l'algèbre terminale de cette catégorie, les éléments de ces paires sont congrus, puisque :

(VN,Vθ) appartient(N,θ(inserer(0,créer)))

= appartient (N, (inserer(0,inserer(0,créer)))).

où θ est une composition d'opérateurs inserer et enlever ; et identiquement pour la deuxième paire.

fin exemples.

2.2.2.2.6.

Dans ce paragraphe nous allons donner quelques définitions et résultats trouvés dans [GUT78] pour clore cette partie introductive sur les spécifications algébriques.

Définitions

1 - Une spécification $<\tau+B,\Sigma,E>$ est suffisamment complète pour τ ssi pour tout terme dans $T_{\tau+B}$: $a(w_1,w_2,\ldots)$ où $a \in A$, il existe un théorème dérivable de E de la forme

$$a(w_1,w_2,\ldots) = u$$

où u est un terme de T_B.

2 - Etant donnée la spécification d'un type τ : $<S,\Sigma,E>$, et ensemble de générateurs G de τ ; un opérateur f tel que $f \notin G$ est dit convertible à G si pour tout terme $f(w_1,\ldots,w_n)$, il existe un théorème dérivable de E de la forme:

$$f(w_1,\ldots,w_n) = z$$

où z ne contient que des opérateurs de G.

Exemple :

Dans ens, enlever est convertible à {créer,inserer}.

3 - Si G est un ensemble de générateurs, on considère les sous-ensembles C de G tels que $\forall f \in G-C$, f est convertible à C. Un sous-ensemble C de G est un ensemble de constructeurs ssi $\forall f \in C$, f n'est pas convertible à C-{f}. En général, pour un type donné, l'ensemble des constructeurs n'est pas unique.

2.2.2.2.7. Proposition

Pour tout type ti : <ti+B,Σ,E> il existe une algèbre normale des termes T^n_{ti} qui est initiale, où les termes ne sont composés que d'opérateurs constructeurs ($T^n_{ti} \subseteq T_{ti}$).

Cette proposition est une conséquence directe des définitions. Chaque terme d'une algèbre normale est un représentant des classes de congruence de T_{ti}. Si chaque classe peut être caractérisée par un représentant unique, alors on parlera de représentant canonique.

2.2.2.3. Représentation d'un type abstrait

Lorsqu'un type a été spécifié dans un formalisme algébrique par les méthodes du 2.2.2.1. ou 2.2.2.2., il arrive le plus souvent que l'on désire "représenter" ce type par un ou plusieurs autres types plus "primitifs". C'est, en tous cas, le processus de la programmation par affinements successifs (par décomposition) jusqu'aux types du niveau machine.

Le problème qui se pose est de vérifier que la représentation d'un type est "correcte", c'est-à-dire qu'elle satisfait les spécifications du type initial. D'un point de vue formel, la vérification d'une représentation n'est pas différente de la vérification de l'équivalence de deux spécifications, néanmoins, il y a quelques aspects particuliers que l'on va étudier dans ce paragraphe.

2.2.2.3.1. Méthode de ADJ [ADJ76]

La méthode est directement inspirée de la vérification des spécifications (§ 2.2.2.1.7.). Soit un type τ, spécifié par <S,Σ,E> et soit R : <S',Σ',E'> l'algèbre de la représentation. L'idée est de construire une algèbre dR, dérivée de R qui soit isomorphe à $T_{S,\Sigma,E}$. En général l'isomorphisme n'est pas immédiatement obtenu ; on doit considérer une relation de congruence Ξ et on montre que $T_{S,\Sigma,E}$ est isomorphe à un sous-ensemble de dR/Ξ.

Traitons complètement l'exemple de la représentation des ensembles d'entiers par des tableaux d'entiers.

```
type tableau :
   tabvide : →tableau
   ranger : tableau x nat x nat → tableau
   val : tableau x nat → nat
équations :
   val(tabvide,J) = naterr
   val(ranger(TAB,I,M),J)=
         si eq(I,J) alors M sinon val(TAB,J) fsi
   ranger (ranger(TAB,I,M),J,N) = si ¬eq(I,J)
                                  alors ranger(ranger(TAB,J,N),I,M) fsi
type ens (ensemble d'entiers) :
   créer : → ens
   ins   : nat x ens → ens
   enl   : nat x ens → ens
   app   : nat x ens → bool
équations :
   ins(N,ins(M,S)) = si eq(N,M) alors ins(M,S)
         sinon ins(M,ins(N,S)) fsi
   enl(N,créer) = créer
   enl(N,ins(M,S)) = si eq(N,M) alors enl(N,S)
         sinon ins(M,enl(N,S)) fsi
   app(N,créer) = faux
   app(N,ins(M,S)) = si eq(N,M) alors vrai
         sinon app(N,S) fsi
```

Si t = <t+S,Σ,E> est le type à représenter dans R : <S,Σ',E'>, un dériveur est une application f de t dans S et une application d de Σ dans $T_{S,\Sigma',E'}$ qui respecte f.

(Note: [FIN78] définit un dériveur comme l'application des symboles fonctionnels du système source dans l'ensemble des schémas fonctionnels du système cible).

Dans notre exemple, le dériveur choisi est :

f(ens) = <tableau,nat>

où < >est le constructeur de tuple.

Les opérations dérivées sont :

```
d(créer) = <tabvide,0>
d(ins)(X,<TAB,N>) =
   si dans (X,TAB,N)
   alors <TAB,N>
   sinon <ranger(TAB,succ(N),X),succ(N)> fsi
d(enl)(X,<TAB,N>) =
   si dans (X,TAB,N)
   alors <ranger(TAB,indice(X,TAB,N), val(TAB,N)), N-1>
   sinon <TAB,N>  fsi
d(app)(X,<TAB,N>) = dans (X,TAB,N)
```

Pour définir ce dériveur, on a utilisé deux opérateurs auxiliaires sur les tableaux que l'on peut spécifier comme extensions :

```
dans : nat x tableau x nat → bool
indice : nat x tableau x nat → nat

équations :
   dans(X,tabvide,N) = faux
   dans(X,ranger(TAB,I,Y),N) =
         si eq(X,Y) ∧ le(I,N)
         alors vrai sinon dans (X,TAB,N) fsi
   indice(X,tabvide,N) = 0  (choix arbitraire)
   indice(X,ranger(TAB,I,Y),N)=
         si eq(X,Y) ∧ le(I,N)
         alors I sinon indice (X,TAB,N) fsi
```

Définissons l'application suivante :

```
h : <tableau,nat> → ens
```

par les règles :

```
(∀ TAB)   h(<TAB,0>) = créer
(∀TAB,N≠0)   h(<TAB,N>) = ins(val(TAB,N), ins(val(TAB,N-1), ...
                                    ins(val(TAB,1),créer) ...))
```

D'après la méthode ADJ, h doit être un homomorphisme surjectif; avant de vérifier cela, il est utile d'établir un lemme sur la représentation. Il est facile de démontrer, d'après la définition de "dans" et "indice" que :

```
dans (X,TAB,N) ≡ vrai
   ⊃(∃i)(1 ≤ i ≤ N) : indice (X,TAB,N) = i ∧ val(TAB,i) = X
```

La partie intéressante du lemme de représentation (LR), est que, si <TAB,N> est la représentation d'un ensemble, alors l'indice i est unique :

Lemme (LR) :

(VS : ens) : S = h(<TAB,N>)
alors dans (X,TAB,N) ≡ vrai
⊃ (∃i unique) (1 ≤ i ≤ N) : indice(X,TAB,N) = i ∧ val(TAB,i) = X.

Démonstration - (induction structurale)

1 - S = créer = h(<TAB,0>)
d'après la définition de "dans" et "le"
dans(X,TAB,0) ≡ dans(X,tabvide,0)
≡ faux

2 - Soit S = h(<TAB,N>) pour lequel LR est vrai, alors :
h(S') = h(ins(Y,S))
= d(ins)(Y,<TAB,N>)
a) = <TAB,N> si dans (Y,TAB,N) ≡ vrai
b) = <ranger(TAB(succ(N),Y), succ(N)> = <TAB',succ(N)>
si dans (Y,TAB,N) ≡ faux ;
donc (∀i) (1 ≤ i ≤ N) : indice (X,TAB,N) ≠ i ∧ val(TAB,i) ≠ Y ; (1)
⇒ val(TAB',succ(N))= Y
∧ indice(X,TAB',succ(N)) = indice(X,TAB',N) (∀X≠Y)
∧ indice(Y,TAB',succ(N)) = succ(N)
⇒ le lemme est vrai pour h(S') dans les cas a) et b).

3 - Soit S = h(<TAB,N>) pour lequel LR est vrai :
h(S') = h(enl(Y,S))
= d(enl)(Y,<TAB,N>)
a) = <TAB,N> si dans (Y,TAB,N) ≡ faux
b) = <ranger(TAB,indice(Y,TAB,N), val(TAB,N)),N-1> = <TAB',N-1>
si dans'(Y,TAB,N) ≡ vrai ;
donc (∃i unique) (1 ≤ i ≤ N) : indice(Y,TAB,N) = i ∧ val(TAB,i)=Y
⇒ si i=N, alors dans (Y,TAB',N-1) ≡ faux
∧ (∀i)(1 ≤ i ≤ N-1) val(TAB,i) = val(TAB',i)
si i<N, alors val(TAB',i) = val(TAB,N)
mais, (∀i)(1 ≤ i ≤ N-1) val(TAB,i) ≠ (TAB,N)

donc, $(\forall i,j)(1 \leq i,j \leq N\dot{-}1)$

$val(TAB',i) \neq (TAB',j)$ si $i \neq j$

$\Rightarrow$ le lemme est vrai pour h(S') dans les cas a) et b).

note(1) : On peut conserver $\wedge$ car les termes de la conjonction sont liés.

□

L'application h est un homomorphisme :

1 - h(d(créer)) = <tabvide,0>

= créer

2 - h(d(ins)(X,<TAB,N>))

= h(si dans(X,TAB,N) alors <TAB,N>
sinon <ranger(TAB,succ(N),X), succ(N)> fsi)

= si dans(X,TAB,N) alors h(<TAB,N>)
sinon h(<ranger(TAB,succ(N),X), succ(N)>) fsi

= si dans (X,TAB,N) alors h(<TAB,N>)
sinon ins(X,h(<TAB,N>)) fsi

Il reste à montrer que si dans (X,TAB,N) $\equiv$ vrai

alors ins(X,h(<TAB,N>)) $\equiv$ h(<TAB,N>) ;

en effet par LR, $(\exists i)1 \leq i \leq N$ tel que val(TAB,i)=X

$\wedge(\forall j \neq i)$ val(TAB,j)$\neq$X

donc :

ins(X,h(<TAB,N>))
= ins(X,ins(val(TAB,N),...ins(val(TAB,i),...,créer...)...))
= ins(val(TAB,N),ins(X,...ins(val(TAB,i),...,créer...)...))
= ins(val(TAB,N),...ins(X,ins(val(TAB,i),... créer...))...)
= ins(val(TAB,N),...ins(val(TAB,i),... créer ...)...)
= h(<TAB,N>)

d'où h(d(ins)(X,<TAB,N>)) = ins(X,h(<TAB,N>))

3 - h(d(enl)(X,<TAB,N>)

= h(si dans(X,TAB,N)
alors <ranger(TAB,indice(X,TAB,N), valeur(TAB,N)),N-1>
sinon <TAB,N> fsi)

= si dans(X,TAB,N)
alors h(<ranger(TAB,indice(X,TAB,N),val(TAB,N)),N-1>)
sinon h(<TAB,N>) fsi

Montrons :

a) Si dans(X,TAB,N) ≡ faux

alors enl(X,h(<TAB,N>)) ≡ h(<TAB,N>) ;

en effet :

(Vi)(1≤ i ≤N) val(TAB,i)≠X

donc:

enl(X,ins(val(TAB,N),... créer ...))
= ins(val(TAB,N), enl(X, ... créer ...))
= ins(val(TAB,N), ...enl(X,créer) ...)
= ins(val(TAB,N),... créer ...)
= h(<TAB,N>)

b) Si dans(X,TAB,N) ≡ vrai,
alors enl(X,h(<TAB,N>))≡ h(<ranger(TAB,indice(X,TAB,N), val(TAB,N)),N-1>)

Soit i : 1≤ i ≤N : indice(X,TAB,N)=i

alors (∀j≠i) val(TAB,j)≠X

b1 : i=N :

enl(X,ins(val(TAB,N), ... créer ...))
= enl(X,ins(X,ins(val(TAB,N-1),... créer ...)))
= enl(X,ins(val(TAB,N-1),... créer ...))
= ins(val(TAB,N-1),enl(X,... créer ...))
= ins(val(TAB,N-1) ... enl(X,créer) ...)
= ins(val(TAB,N-1), ... créer ...)
= h(<TAB,N-1>)
= h(<ranger(TAB,N,X),N-1>)
= h(<ranger(TAB,indice(X,TAB,N),val(TAB,N)),N-1>)

b2 : i<N :

enl(X,ins(val(TAB,N), ...ins(val(TAB,i)... créer...) ...))
= ins(val(TAB,N), enl(X,...ins(val(TAB,i), ... créer ...) ...))
= ins(val(TAB,N), ...ins(val(TAB,succ(i)), enl(X,ins(val(TAB,i),
ins(val(TAB,i-1), ... créer ...)))) ...)
= ins(val(TAB,N), ...ins(val(TAB,succ(i)), ins(val(TAB,i-1),
enl(X, ... créer ...)))...)

= ins(val(TAB,N), ...ins(val(TAB,succ(i)), ins(val(TAB,i-1), ...
enl(X,créer) ...)) ...)
= ins(val(TAB,N), ...ins(val(TAB,succ(i)), ins(val(TAB,i-1),
... créer ...)) ...)
= ins(val(TAB,N-1), ins(val(TAB,N), ... créer ...))
= ins(val(TAB,N-1), ...(val(TAB,succ(i)), ins(val(TAB,N),
ins(val(TAB,i-1), ... créer ...))) ...)
= h(<ranger(TAB,i,val(TAB,N)),N-1>)
= h(<ranger(TAB,indice(X,TAB,N), val(TAB,N)), N-1>)

Finallement h(d(enl)(X,<TAB,N>)) = enl(X,h(<TAB,N>))

4 - d(app)(X,<TAB,N>) = app(X,h(<TAB,N>))

a) d(app)(X,<TAB,N>) ≡ vrai

⊃ dans (X,TAB,N) ≡ vrai

d'après LR, (∃i)1 ≤ i ≤ N : val(TAB,i) = X

donc app(X,ins(val(TAB,N), ... ins(val(TAB,i), ... créer ...) ...))

= app(X,ins(val(TAB,i), ... créer ...))

= vrai

b) d(app)(X,<TAB,N>) ≡ faux ⊃ dans(X,TAB,N) ≡ faux

(LR) ⇒ (∀i)1 ≤ i ≤ N : val(TAB,i)≠X

donc app(X,ins(val(TAB,N), ... créer ...))

= app(X,créer)

= faux

L'homomorphisme <u>h est surjectif</u>.

Tout élément de ens peut être mis sous la forme

ins(X_1,ins(X_2,... créer ...))

tels que tous les X_i soient distincts (la preuve par induction est analogue à celle du lemme LR. Donc tout élément de ens peut être représenté par <TAB,N> et h est surjectif.

<u>Relation de congruence</u> engendrée par h :

On note $\equiv_h$ la relation dans <tableau,nat> définie par :

<TAB1,N1> $\equiv_h$ <TAB2,N2>

<=> h(<TAB1,N1>) ≡ h(<TAB2,N2>)

Une telle congruence n'est pas triviale ; si on prend

ET1 = <ranger(ranger(tabvide,1,0),2,1),2>

et

ET2 = <ranger(ranger(tabvide,1,1),2,0),2>

où 1 = succ(0), 2 = succ(succ(0)) ;

on a :

h(ET1) = ins(1,ins(0,créer))

qui est congru dans ens à :

h(ET2) = ins(0,ins(1,créer))

alors que ET1 et ET2 ne sont pas équivalents.

Représentation de ens :

Soit Rens l'algèbre sur <tableau,nat>avec les opérations { d(créer), d(ins), d(enl), d(app)}, alors, par la proposition 2.2.1.15., Rens/$\equiv_h$ est isomorphe à T_{ens} et constitue une représentation du type ens à l'aide de tableau et nat.

2.2.2.3.2. Méthode de ZILLES/GUTTAG [ZIL79] [GUT76a]

Reprenons les mêmes spécifications de ens et tableau du § 2.2.2.3.1., ainsi que la définition de l'algèbre Rens.

On définit sur Rens l'équivalence suivante :
(interprétation de l'égalité) notée IE :

<TAB1,N1> IE <TAB2,N2>

<==> N1 = N2 ∧ (∀X) dans (X,TAB1,N1) ≡ dans (X,TAB2,N2)

De plus, on restreint Rens aux couples <TAB,N> qui satisfont LR, c'est-à-dire :

(∀i,j)1 ≤ i,j ≤ N, val(TAB,i) ≠ val(TAB,j) ssi i≠j.

Soit LR(Rens) ce sous-ensemble. Alors on considère AR = LR(Rens)/$\equiv_{IE}$ (où $\equiv_{IE}$ est la relation de congruence engendrée par IE).

1 - AR est fermée pour d(ins), d(enl)

Ce résultat s'appuie directement sur la démonstration de LR.

2 - AR avec les opérations d(créer), d(ins), d(enl), d(app) satisfait les équations de la spécification de ens : L'exercice est long mais sans grandes difficultés ; montrons par exemple que AR satisfait la première équation des ensembles :

d(ins)(N,d(ins)(M,<TAB,K>) =
si eq(N,M) alors d(ins)(M,<TAB,K>)
sinon d(ins)(M,d(ins)(N,<TAB,K>)) fsi

(cas 1) si dans(N,d(ins)(M,<TAB,K>))
alors d(ins)(M,<TAB,K>)
sinon <ranger (d(ins)(M,<TAB,K>),succ(α),N),succ(α)>fsi
= si eq(N,M) alors d(ins)(M,<TAB,K>)
sinon d(ins)(M,d(ins)(N,<TAB,K>)) fsi
(où α est mis pour K ou succ(K))

⇒ si eq(N,M) alors
si dans(N,d(ins)(M,<TAB,K>))
alors d(ins)(M,<TAB,K>)
sinon <ranger(d(ins)(M,<TAB,K>),succ(α),N),succ(α)> fsi
= d(ins)(M,<TAB,K>)
fsi

⇒ d'après les définitions de dans et d(ins),
eq(N,M) ≡ vrai ⊃ dans(N,d(ins)(M,<TAB,K>)) ≡ vrai
d'où d(ins)(M,<TAB,K>) = d(ins)(M,<TAB,K>)

(cas 2) si ¬eq(N,M) alors
si dans(N,d(ins)(M,<TAB,K>))
alors d(ins)(M,<TAB,K>)
sinon <ranger(d(ins)(M,<TAB,K>), succ(α),N) succ(α)> fsi
= d(ins)(M,d(ins)(N,<TAB,K>))
fsi

(2.1.) Supposons dans (M,TAB,K) ≡ vrai
∧ dans (N,d(ins)(M, TAB,K)) ≡ vrai
d(ins)(M,<TAB,K>)
= <TAB,K> def. d(ins)
= d(ins)(N,<TAB,K>) id.
= d(ins)(M,d(ins)(N,<TAB,K>)) id.

(2.2.) Supposons:dans(M,TAB,K) ≡ faux

$\wedge$dans(N,d(ins)(M,<TAB,K>)) ≡ vrai

alors, par $\neg$eq(N,M) on a dans(N,<TAB,K>) ≡ vrai

d(ins)(M,<TAB,K>)

= d(ins)(M,d(ins)(N,<TAB,K>))

(2.3.) Supposons : dans(M,TAB,K) ≡ vrai

$\wedge$dans(N,d(ins)(M,<TAB,K>)) ≡ faux

alors dans(N,TAB,K) ≡ faux

<ranger(d(ins)(M,<TAB,K>),succ(K),N),succ(K)>

=<ranger(TAB,succ(K),N),succ(K)> def. d(ins)

= d(ins)(N,<TAB,K>) id.

= d(ins)(M,d(ins)(N,<TAB,K>)) id.

(2.4.) Supposons : dans(M,TAB,K) ≡ faux

$\wedge$ dans(N,d(ins)(M,<TAB,K>)) ≡ faux

alors dans(N,TAB,K) ≡ faux

<ranger(ranger(TAB,succ(K),M),succ(succ(K)),N),succ(succ(K))>

= <ranger(ranger(TAB,succ(K),N),succ(succ(K)),M),succ(succ(K))>,IE,

= d(ins)(M,<ranger(TAB,succ(K),N),succ(K)>) def.d(ins)

= d(ins)(M,d(ins)(N,<TAB,K>)) id.

3 - D'après le théorème 2.2.1.14., il existe un homomorphisme unique $f : T_{ens} \rightarrow AR$. On doit montrer de plus que f est une bijection. Pour cela, choisissons des représentants canoniques de T_{ens} et AR que l'on fait correspondre par f :

f(créer) = <tabvide,0>

$f(ins(X_n,...ins(X_1,créer)...) =$

$<ranger(...ranger(tabvide,succ(0),X_1),...,succ^n(0),X_n),succ^n(0)>$

avec $X_1 < X_2 ... < X_n$

Ce qui termine la démonstration de la représentation du type ens par l'algèbre AR.

2.2.2.3.3. Discussion sur la méthode de vérification de la représentation selon Guttag

On considère le type ens du § 2.2.2.3.1. mais dans lequel on a supprimé l'équation sur "ins". Soit Gens ce nouveau type. Alors, cette spécification selon Guttag définit l'ensemble des types compris entre T_{Gens} et T_{Gens}/Ξ_A qui est, en fait, T_{ens}. La relation Ξ_A est déduite des équivalences de la forme:

$$\text{Si } (\forall N)(\forall \theta)\, app(N,\theta(S)) \Xi_{bool} app(N,\theta(S')) \text{ alors } S \Xi_A S'$$

La représentation est définie comme en 2.2.2.3.2. par le quotient d'un sous-ensemble d'une algèbre par la relation de congruenee déduite de l'interprétation de l'égalité. Par exemple, ici :

- Rens spécifie un domaine de représentation ;
- LR(Rens) est le sous-ensemble de Rens fermé par les opérations de l'algèbre ; il existe une application surjective de T_{Gens} dans LR(Rens), mais LR(Rens) ne satisfait pas les équations de Gens et ne constitue pas une représentation.
- $LR(Rens)/\Xi_{IE}$ par contre, satisfait les équations ; donc il existe un homomorphisme surjectif $h : T_{Gens} \twoheadrightarrow LR(Rens)/\Xi_{IE}$.

En appliquant la proposition 2.2.1.15, on trouve que l'algèbre de représentation est isomorphe à T_{Gens}/Ξ_h. La clé de la représentation est donc, non seulement la recherche du domaine de représentation, mais surtout l'interprétation de l'égalité qui doit fournir une congruence suffisante pour vérifier les axiomes, mais telle que les types de la base soient protégés (T_{Gens}/Ξ_h doit être une algèbre de la catégorie GALG où T_{Gens} est initiale et T_{ens} est terminale).

Il semble cependant qu'il est inutile de définir explicitement Ξ_{IE}. En effet, le mécanisme qui permet de définir Ξ_A peut être appliqué à la représentation.

Définitions

Soient les Σ-algèbres A1 et $A1/\Xi_A$ définies selon Guttag et soit une Σ-algèbre de représentation B telle que $\exists$ un homomorphisme $j : A1 \twoheadrightarrow B$ qui respecte

les accès (j est l'identité pour les types de la base) :

$(\forall au)\ (\forall v \in au)\ (\forall v \in ti)$

$$\nu(v) \equiv j(\nu(v))$$
$$\equiv j(\nu)(j(v))$$

Soit dans j(A1) la congruence engendrée par d(a) :

$$x \equiv_{d(a)u} y \iff$$

$$\forall \nu' \in d(a)u,\ \ \nu'(x) \equiv \nu'(y)$$

et $\equiv_{A'} = \cap(\equiv_{d(a)u})$ (∀ accès unaire d(a)u dans B).

Proposition 1

$j(A1)/\equiv_{A'}$ satisfait les équations de A1.

Démonstration

Pour toute équation e:e1 = e2 dans A1, on a:

1/ e est une équation dans les termes des types de la base, et on a immédiatement:
∀σ' substitution dans j(A1), ∃σ dans A1 tel que σ'=j(σ) (image par j de chaque terme de la substitution), d'où :

$$\sigma' e1 \equiv j(\sigma)e1 \equiv j(\sigma e1) \equiv j(\sigma e2)$$
$$\equiv j(\sigma)e2 \equiv \sigma' e2$$

2/ e est une équation dans le type d'intérêt, d'où :
on doit avoir : $\sigma' e1 \equiv_{A'} \sigma' e2$,
c'est-à-dire $\forall d(a), \forall \nu' \in d(a)u : \nu'(\sigma' e1) \equiv \nu'(\sigma' e2)$

$$\nu'(\sigma' e1) \equiv j(\nu)(j(\sigma' e1))$$
$$\equiv j(\nu)(j(\sigma)e1) \qquad \text{par homorphisme}$$
$$\equiv j(\nu)(j(\sigma e1))$$
$$\equiv j(\nu)(j(\sigma e2)) \qquad \text{car e est vérifiée par A1}$$
$$\equiv \nu'(\sigma' e2)$$

Proposition 2

$j(A1)/\Xi_{A'}$ est isomorphe à $A1/\Xi_A$

Démonstration

Soit $h : A1 \rightarrow j(A1)/\Xi_{A'}$, l'homomorphisme surjectif entre algèbres qui satisfont les équations de A1. Si l'on examine la relation Ξ_h dans A1 définie par :

$$x \Xi_h y \iff h(x) \Xi_{A'} h(y)$$

on a:

$$\forall d(a),\ \forall v' \in d(a)u,\ v'(h(x)) \equiv v'(h(y))$$

d'où

$$v(x) \equiv v(y),\ \forall v \in au$$

ce qui définit $x \Xi_A y$;

donc $\Xi_h = \Xi_A$ et $j(A1)/\Xi_{A'}$ est isomorphe à $A1/\Xi_A$.

Etant donné une Σ,E-algèbre A1 définie par ses accès A, alors si B est une Σ-algèbre telle qu'il existe un homomorphisme $j : A1 \rightarrow B$ qui respecte A, alors $j(A1)/\Xi_{A'}$ est une représentation de A1.

(note : c'est même la "meilleure" représentation).

Dans l'exemple des ensembles, on peut choisir pour B l'algèbre Rens et la relation de congruence est alors :

$$\langle TAB1,N1\rangle \equiv \langle TAB2,N2\rangle \Longleftrightarrow (\forall\theta)(\forall X)\ d(app)(X,\theta(\langle TAB1,N1\rangle)) \equiv d(app)(X,\theta(\langle TAB2,N2\rangle)$$

Dans ce cas $\Xi_{A'} = \Xi_{IE}$ puisque $T_{Gens}/\Xi_A \tilde{=} T_{ens} \tilde{=} LR(Rens)/\Xi_{IE}$.

Comme le remarque [FIN78], on n'a besoin de définir j que sur les objets de la forme normale et donc, si on s'en tient au type défini par ses équations, on ne donne que l'image des constructeurs de A1. L'image des discriminateurs et des accès peut être obtenue quasi automatiquement en utilisant les équations de A1 qui doivent être satisfaites (méthode constructive). On peut aussi utiliser directement la condition de représentation des accès qui peut conduire à une synthèse plus simple des discriminateurs, mais d'une manière non constructive. L'algèbre de représentation B n'est donc pas forcément une Σ-algèbre.

(Voir aussi au §2.2.4.3. comment on peut définir la sémantique des discriminateurs dans un domaine d'interprétation qui peut être le domaine de la représentation).

2.2.2.4. Synthèse de représentation d'un type abstrait

Le problème a été étudié spécialement par [GAU78]. Il est intéressant de voir ici comment la sémantique de la spécification d'un type peut influer sur les choix automatiques d'une représentation. Dans la méthode du 2.2.2.1., on est conduit à choisir une algèbre où chaque classe d'équivalence sera représentée par un élément. Cela est simple si chaque classe a un seul représentant sous forme normale (ex. : nat avec 0 et succ).

Dans les autres cas, deux solutions sont envisageables :

1 - Trouver une forme canonique sur les constructeurs qui tienne compte des propriétés de l'équivalence dans les termes de la forme normale : (ex.: décrire des ensembles à l'aide de créer et ins, mais où les éléments sont tous distincts). Ensuite, comme précédemment, trouver une algèbre isomorphe aux termes canoniques (ex. : liste des éléments ordonnés).

2 - Utiliser la méthode de définition par les accès comme au § 2.2.2.2. L'idée est de dire que si "au" est un accès unaire de la forme au(x,p) où x est l'objet du type et p la liste des autres paramètres, la représentation consiste à associer à chaque p la valeur au(x,p). Dans le cas des ensembles, app(n,S) est un booléen, donc une représentation possible est l'ensemble des couples <nat,bool>. On reconnaît là l'idée de base de la représentation des ensembles par des vecteurs booléens.

L'inconvénient de cette dernière méthode est que l'introduction d'une nouvelle fonction d'accès peut conduire à modifier la représentation. Par exemple, une pile avec les opérations créerp, empiler, depiler et l'accès pilevide peut être représentée par un entier (équivalence des piles de même hauteur). Si l'on ajoute l'accès sommet, la représentation peut être un couple <entier,tableau> avec entier comme précédemment et tableau pour mémoriser les valeurs du sommet, ou liste, représentation qui contient la valeur de l'entier implicitement par la fonction longueur de la liste.

2.2.3. Spécification algébrique des types génériques

2.2.3.1. Propriétés

D'un point de vue algébrique, une propriété est la donnée d'une présentation : <S, Σ, E> dans laquelle une ou plusieurs sortes sont distinguées. On note, comme pour le type d'intérêt : $S = t_1, t_2 \ldots + S'$.
A la différence des types algébriques qui sont entièrement déterminés par leurs opérateurs, une propriété n'a pas la notion de constructeur, générateur, etc... En particulier, on ne peut pas définir une algèbre de Herbrand pour les sortes distinguées d'une propriété, ou plutôt, l'algèbre des termes est composée des objets des domaines t_1, t_2, ... considérés comme des constantes, sur lesquels on définit les opérations de Σ(§2.2.3.3.1.1.). Il y a "oubli" de la structure qui a permis de construire les termes. Voilà pourquoi, dans notre formalisme, les types d'intérêt d'une propriété sont appelés types formels, en ce sens qu'ils ne sont pas entièrement déterminés.

2.2.3.1.1. Définitions

1 - Une algèbre A satisfait une propriété
P : <S,Σ, E> ssi ilexiste une sous-algèbre de A qui soit une S,Σ-algèbre satisfaisant E (§2.2.1.2., 2.2.1.10).

2 - Un type abstrait τ : $<\tau + \overline{S}, \overline{\Sigma}, \overline{E}>$ satisfait une propriété P : <t + S, Σ, E ssi $S \subseteq \overline{S}$, $\Sigma \subseteq \overline{\Sigma}$ et l'algèbre $P(\tau)$, où le domaine associé à t est T_τ, est une t+S,Σ-algèbre qui satisfait E (on dit aussi que le type possède la propriété P).

2.2.3.1.2. Proposition

Si un type τ satisfait une propriété P, alors toute extension ou enrichissement de τ satisfait P.

Démonstration : immédiate d'après la définition du § 2.2.2.1.2.

On dira qu'un type τ peut satisfaire une propriété P s'il existe une extension de τ qui satisfait P.

Note : ici, comme pour définir une représentation d'un type par un autre type, se placer dans le cadre des théories algébriques plutôt que celui des algèbres simplifierait les définitions [BUR77][EHR78].

2.2.3.1.3. Exemples

Soit la propriété monoïde définie sur t par

$$0 : \rightarrow t$$

$$+ : t \times t \rightarrow t$$

et les équations :

$$0 + x = x$$

$$x + 0 = x$$

$$(x + y) + z = x + (y + z)$$

Alors le type bool possède la propriété monoïde avec la constante vrai et l'opérateur $\wedge$ (§ 2.2.2.1.5.).
En effet, la fonctionnalité des opérations est respectée et les équations sont satisfaites :

$vrai \wedge x = x$ (axiome)

$x \wedge vrai = vrai \wedge x$ (commutativité)

$= x$

$(x \wedge vrai) \wedge vrai = x \wedge vrai = x$ (traitement par cas).

$x \wedge (vrai \wedge vrai) = x \wedge vrai = x$

$(x \wedge vrai) \wedge faux = x \wedge faux = faux$

$x \wedge (vrai \wedge faux) = x \wedge faux = faux$

$(x \wedge faux) \wedge vrai = faux \wedge vrai = faux$

$x \wedge (faux \wedge vrai) = x \wedge faux = faux$

$(x \wedge faux) \wedge faux = faux \wedge faux = faux$

$x \wedge (faux \wedge faux) = x \wedge faux = faux$

donc x, y, z : bool $\quad (x \wedge y) \wedge z = x \wedge (y \wedge z)$

On n'a fait la vérification de la propriété que sur les représentants du type bool, et non sur tous les termes de l'algèbre ; aussi, l'équation générale ne peut être induite dans le type que si les types qui possèdent des accès à résultat booléen "protègent" le type bool (ou en d'autres termes, si les spécifications qui utilisent bool sont cohérentes (consistent) et suffisamment complètes [GUT78]).

Le type nat satisfait aussi la propriété monoïde avec la constante 0 et l'opérateur + : nat x nat → nat (§2.2.2.1.3.).
C'est-à-dire, pour chaque x_0, y_0, z_0 donné dans T_{nat}, on peut trouver un équivalent sous forme canonique : $succ^k(0)$ et ces équivalents : x_1, y_1, z_1 satisfont :

$$x_1 + 0 = 0$$

$$0 + x_1 = 0$$

$$(x_1 + y_1) + z_1 = x_1 + (y_1 + z_1)$$

L'intérêt est de pouvoir démontrer que les équations données pour spécifier une propriété peuvent être déduites (moyennant un renommage des domaines et des opérateurs) des équations du type qui satisfait cette propriété. Cette démonstration peut se faire, le plus souvent par induction structurale, à partir des opérateurs constructeurs :

1 - N + 0 = N (axiome)

2 - 0 + 0 = 0

si 0 + M = M alors 0 + succ(M) = succ(0 + M)
= succ(M)

d'où 0 + N = N

3 - (N1 + N2) + 0 = N1 + N2
= N1 + (N2 + 0)

si (N1 + N2) + N3 = N1 + (N2 + N3)

alors (N1 + N2) + succ(N3) = succ((N1 + N2) + N3)
N1 + (N2 + succ(N3)) = N1 + succ(N2 + N3)
= succ(N1 + (N2 + N3))

et par hypothèse d'induction, le résultat est démontré.

2.2.3.1.4. Comme, en général, un type peut satisfaire une propriété de plusieurs manières, on devra toujours indiquer l'association entre les opérateurs du type et ceux de la propriété.

ex : bool : monoïde (vrai, ∧)

nat : monoïde (0, +)

L'ensemble des types qui satisfont une propriété P : <S, Σ, E> forment une catégorie de S, Σ-algèbres que l'on désigne sous le nom de classe de P : C(P). Un type peut évidemment appartenir à plusieurs classes.

2.2.3.1.5. Treillis des propriétés

2.2.3.1.5.1. Définitions

Une propriété P1 est incluse dans une propriété P2 (P1 $\subseteq$ P2) ssi toute algèbre qui satisfait P2 satisfait aussi P1.

Deux propriétés P1 et P2 sont équivalentes ssi
P1 $\subseteq$ P2 $\wedge$ P2 $\subseteq$ P1.

. Pour un ensemble d'algèbres $\dot{A}$, la plus grande propriété commune est la propriété P satisfaite par toutes les algèbres, et telle que pour toute propriété P' non équivalente à P et P $\subseteq$ P', P' n'est pas satisfaite par $\dot{A}$.

On note P1 & P2 la plus grande propriété commune des algèbres qui satisfont à la fois P1 et P2.

On note P1 inter P2 la plus grande propriété commune de l'union des algèbres qui satisfont P1 et de celles qui satisfont P2.

2.2.3.1.5.2. Proposition.

L'ensemble des propriétés forme un inf-demi treillis avec tyfo : élément minimum : <t, $\emptyset$, $\emptyset$ >,

$\subseteq$: ordre partiel,

&, inter

car : P1 $\subseteq$ P2 $\Rightarrow$ P1 & P2 = P2 et P1 inter P2 = P1.

D'après la définition restrictive de l'inclusion donnée au § 2.2.3.1.5.1., il est facile de voir que si P1 $\subseteq$ P2, alors P2 est une extension (ou enrichissement) de P1. D'un point de vue pratique, on aura $\Sigma 1 \subseteq \Sigma 2$, et si les ensembles d'opérateurs sont égaux, alors les équations de P1 sont redondantes dans celles de P2.

2.2.3.1.5.3. Exemples.

1 - Soit ordre-total : sur t + bool, avec les opérateurs spécifiques de t :

$\leq$: t x t $\rightarrow$ bool

eq : t x t $\rightarrow$ bool

et les équations :

$x \leq x$ = vrai

$x \leq y \wedge y \leq z \wedge \neg x \leq z$ = faux

eq(x, y) = $x \leq y \wedge y \leq x$

Soit maintenant la propriété égalité sur t + bool, l'opérateur eq' : t x t $\rightarrow$ bool

et les équations :

1) $eq'(x, x) = vrai$

2) $eq'(x, y) = eq'(y, x)$

3) $eq'(x, y) \wedge eq'(y, z) \wedge \neg eq'(x, z) = faux$

Pour montrer que égalité (eq') $\subseteq$ ordre-total ($\leq$, eq), on doit :

- exhiber dans ordre-total les opérateurs de l'égalité ;
- montrer que les équations de l'égalité peuvent se déduire des équations d'ordre-total (et de bool) :

Ici, l'opérateur eq est identifié à eq' et les équations de eq' sont satisfaites par eq, en effet :

1) $eq(x, x) = x \leq x \wedge x \leq x$
$= vrai \wedge vrai$
$= vrai$

2) $eq(x, y) = x \leq y \wedge y \leq x$
$= y \leq x \wedge x \leq y$
$= eq(y, x)$

3) $eq(x, y) \wedge eq(y, z) \wedge \neg eq(x, z)$
$= x \leq y \wedge y \leq x \wedge y \leq z \wedge z \leq y \wedge \neg(x \leq z \wedge z \leq x)$
$= (x \leq y \wedge y \leq z \wedge \neg x \leq z)$
$\quad v(y \leq x \wedge z \leq y \wedge \neg z \leq x)$
$= faux \vee (z \leq y \wedge y \leq x \wedge \neg z \leq x)$
$= faux \vee faux$
$= faux$

2 - Soient les propriétés : groupe 1 (0, +, inv, -) et groupe 2 (0', +', inv', -') dont les fonctionnalités des opérateurs sont :

$0, 0' : \rightarrow t$

$+, +', -, -' : t \times t \rightarrow t$

$inv, inv' : t \rightarrow t$

et les équations de groupe 1 :

$0 + x = x$

$x + 0 = x$

$(x + y) + z = x + (y + z)$

$inv(x) = 0 - x$

$x + (y - z) = (x + y) - z$

$x - x = 0$

celles de groupe 2 :

$$0' +' x = x$$
$$x +' 0' = x$$
$$(x +' y) +' z = x +' (y +' z)$$
$$inv'(x) +' x = 0'$$
$$x +' inv'(x) = 0'$$
$$x -' y = x +' inv'(y)$$

Pour montrer l'équivalence de groupe 1 et groupe 2, il suffit d'associer les opérateurs op et op' et de montrer que chaque ensemble d'équations se dérive de l'autre :

- les trois premières équations sont identiques.

- groupe 1 $\subseteq$ groupe 2 :

$$\begin{aligned} inv'(x) &= 0' +' inv'(x) \\ &= 0' -' x \end{aligned}$$

$$\begin{aligned} x -' x &= x +' inv'(x) \\ &= 0' \end{aligned}$$

$$\begin{aligned} x +'(y -' z) &= x +'(y +' inv'(z)) \\ &= (x +' y) +' inv'(z) \\ &= (x +' y) -' z \end{aligned}$$

- groupe 2 $\subseteq$ groupe 1

$$\begin{aligned} x + inv(x) &= x + (0 - x) \\ &= (x + 0) - x \\ &= x - x \\ &= 0 \end{aligned}$$

$$\begin{aligned} x - y &= (x + 0) - y \\ &= x + (0 - y) \\ &= x + inv(y) \end{aligned}$$

$$\begin{aligned} inv(x) + x &= inv(x) + (x + (inv(x) - inv(x))) \\ &= inv(x) + ((x + inv(x)) - inv(x)) \\ &= inv(x) + (0 - inv(x)) \\ &= inv(x) + inv(inv(x)) \\ &= 0 \end{aligned}$$

2.2.3.1.6. Fonctions de base et extensions

Le paragraphe précédent utilise la notion d'équivalence forte entre les propriétés. On peut affaiblir l'inclusion par la définition :
une propriété P1 est faiblement incluse dans P2($P1 \leq P2$) ssi toute algèbre qui satisfait P2 peut satisfaire P1.

La notion d'équivalence : $P1 \lesssim P2$ et $P2 \lesssim P1$ est trop faible pour pouvoir être exploitée, mais on peut préciser la pseudo-équivalence :

$$P1 \leq P2 = P1 \subseteq P2 \wedge P2 \lesssim P1$$

Proposition : si $P1 \leq P2$, alors P2 est équivalente à une propriété P3 comprenant les opérateurs de P1 et des opérateurs dérivés de ceux de P1.

La preuve est immédiate d'après la définition :
comme $P1 \subseteq P2$, alors les opérateurs de P1 sont un sous-ensemble des opérateurs de P2. Mais $P2 \lesssim P1$, alors il existe une extension d'opérateurs de P1 qui satisfont P2 ; donc, cette extension ne dépend que des opérateurs de P1, et sont des opérateurs dérivés de ceux de P1. Cette extension est alors équivalente à P2.

Définition : si pour une propriété P2, $\exists$ une propriété P1 telle que $P1 \leq P2$, alors les opérateurs de P1 sont appelés la base de P2 et les opérateurs dérivés seront appelés extension de P2.

Exemples :

1 - soit la propriété :

groupe (0, +, inv)

$0 : \rightarrow t$

$+ : t \times t \rightarrow t$

$inv : t \rightarrow t$

et les équations $0 + x = x$

$x + 0 = x$

$(x + y) + z = x + (y + z)$

$inv(x) + x = 0$

$x + inv(x) = 0$

Alors on a groupe (0, + , inv) ≤ groupe 1 (0, +, inv, -) et la propriété P3 de la proposition précédente est évidemment groupe 2 (0, +, inv, -).

2 - La propriété égalité (eq) du § 2.2.3.1.5.3peut être considérée comme la base de : égalité 1(eq, neq) et $neq : t \times t \rightarrow bool$

avec $neq(x, y) = \neg eq(x, y)$.

Remarque : La séparation entre base et extension est utile pour la programmation, mais comme la base d'une propriété n'est pas unique, sa détermination revêt un caractère arbitraire, justifié le plus souvent par l'habitude et la simplicité.

2.2.3.2. Propriétés attachées à un type

Un type spécifie une algèbre, et donc un type abstrait satisfait une propriété de la manière indiquée au § précédent. Généralement, on peut associer à un type une union de propriétés P1 & P2 & ...

Exemple : type nat : monoïde (0, +) & égalité (égal, dif),

où les opérateurs 0 : → nat

+ : nat x nat → nat

égal : nat x nat → bool

dif : nat x nat → bool

sont ensuite définis dans le type.

Le fait de déclarer qu'un type τ possède une propriété $P : \langle t + S, \Sigma, E\rangle$ signifie :

1) que les équations de la propriété sont vérifiées pour le type (via la substitution de T_τ à t, et des noms des opérateurs), autrement dit, que les axiomes de la propriété sont des théorèmes du type.

Cette condition doit évidemment être vérifiée pour que la déclaration du type soit valide.

2) que le type τ appartient, d'une manière standard, à la classe de P

3) que le type τ possède toutes les propriétés incluses ($\subseteq$) dans P.

Une remarque intéressante est de voir que si une propriété P est composée d'opérateurs de base B et d'opérateurs dérivés (extension) D, si le type τ ne possède que l'interprétation des opérateurs de base B, on peut synthétiser automatiquement l'interprétation des opérateurs dérivés D. En effet, la définition d'un opérateur dérivé est vraie pour toutes les interprétations des opérateurs de base et donc en particulier pour celle du type τ.

Exemple : Raprenons la propriété ordre-total (§ 2.2.3.1.5.3.) dans laquelle, eq et neq sont des extensions :

ordre-total ($\leq$, eq, neq) sur t + bool

$\leq$, eq, neq : t x t → bool

$x \leq x$ = vrai

$x \leq y \wedge y \leq z \wedge \neg x \leq z$ = faux

eq(x, y) = $x \leq y \wedge y \leq x$

neq(x, y) = $\neg$ eq(x, y)

Soit le type nat muni de

0 : → nat

succ : nat → nat

le : nat x nat → bool

le(0, N) = vrai

le(succ(N), 0) = faux

le(succ(N), succ(M)) = le(N, M)

alors, si l'on déclare :

nat : ordre-total (le, eq-nat, neq-nat), cela signifie que "le" satisfait la base de la propriété et que eq-nat, neq-nat : nat x nat → bool sont définis par :

eq-nat(N, M) = le(N,M) ∧ le(M, N)

neq-nat(N,M) = ¬eq-nat(N,M)

2.2.3.3. Type abstrait paramétré

Un type générique, tel que défini au chapitre 1 peut être à la fois un type abstrait "simple" ou un type abstrait possédant des paramètres de type (exemple : pile(t), table(t1, t2), ...). Le § 2.2.2. définit plusieurs sémantiques pour les types abstraits simples. Nous choisirons par la suite la sémantique de ADJ, car elle est plus constructive, mais en suggérant que la spécification des types soit fortement complète [ZIL79], c'est-à-dire que, pour un type : $<\tau + B, \Sigma, E>$ dont les accès sont A, T_τ soit isomorphe à $T_\tau / \equiv_A$ (ce qui ramène la catégorie GALG (τ+ B, Σ, E, B) à une seule algèbre).

Les types abstraits paramétrés (TAP) sont algébriquement d'une autre nature que les types abstraits simples et les paragraphes suivants étudient leurs principales caractéristiques.

2.2.3.3.1. Spécification algébrique des TAP

Un TAP est la donnée d'une présentation : $<S, \Sigma, E>$ où S se décompose en $\tau + t_1, t_2, \ldots + S'$ avec τ, le type d'intérêt, t_1, t_2, ... des types formels spécifiés par des propriétés P1, P2, ... et S' un ensemble de types de base.

Σ est l'ensemble des opérateurs spécifiques de τ, de ceux des propriétés P_i et des types de base et E est l'ensemble des équations associées. On note le type d'une telle présentation : $\tau(t_1, t_2, \ldots)$.

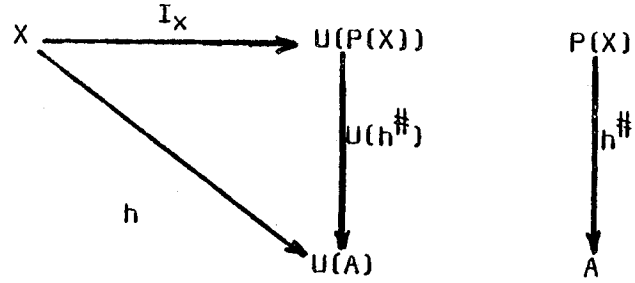
On étend maintenant la définition donnée au § 2.2.1.5. des ensembles de générateurs en une forme plus générale.

2.2.3.3.1.1. Définitions

On note Ens la catégorie des ensembles munis de leurs applications. Soit P une propriété et U le foncteur d'oubli de C(P) dans Ens qui associe à chaque algèbre son ensemble sous-jacent et à chaque homomorphisme l'application correspondante. Si X est un objet de Ens, alors (P(X), I_X) où P(X) $\in$ C(P) et I_X est un morphisme de Ens d'origine X, est le couple attaché par U à X tel que :

$\forall$ application $h : X \to U(A)$, $\exists$ un prolongement unique $h^{\#} : P(X) \to A$ dans C(P) (homomorphisme) et $h(X) = U(h^{\#}) \circ I_X(X)$

Sous forme de diagramme :



On dit que P(X) est l'algèbre libre de la propriété P sur l'ensemble générateur X.

(Note : techniquement, U est le foncteur adjoint à droite par rapport aux couples (P(X), I_X) pour tout X).

De même, on note h_τ l'homomorphisme dans C(P), pour tout type τ dans C(P), qui applique P(U(τ)) dans τ considéré comme une P-algèbre.

2.2.3.3.1.2. Soit un TAP : $\underline{\tau}$: <τ + t_1, t_2, ... + S, Σ, E>
où $t_1 : P_1$, ..., $t_n : P_n$; alors, pour tout ensemble de générateurs donnés : $X_1, X_2, \ldots X_n$ associés aux $t_1, \ldots t_n$, l'algèbre définie par la présentation τ est le quotient de l'algèbre des termes d'opérateurs Σ, librement engendrée par $X_1, \ldots X_n$, par la relation de congruence $\equiv_E$.
On la notera $\tau(X_1, \ldots X_n)$.
Un TAP est valide si sa présentation τ(t1,...tn) est une extension de la combinaison des présentations des P_i.

(Note : L'algèbre libre d'une propriété P, peut ne pas protéger les types de base ; par contre, $P_1(X_1), \ldots P_n(X_n)$ doivent être protégés par $\tau(X_1, \ldots X_n)$).

Un type abstrait paramétré définit donc une classe de types, de la même manière que les propriétés. On la notera $C(\tau)$ pour le type τ . C'est une catégorie où les objets sont les algèbres, et les morphismes sont les familles d'homomorphisme h_s pour chaque domaine s, qui sont l'identité pour les domaines de τ et des types de base.

2.2.3.3.1.3. Foncteur de TAP

Si $\underline{\underline{C1}}$, $\underline{\underline{C2}}$, ... $\underline{\underline{Cn}}$ sont des catégories, on peut construire la catégorie $\underline{\underline{C1}} \times \underline{\underline{C2}} \times \ldots \times \underline{\underline{Cn}}$ correspondant au n-uple des catégories composantes.

Soit un TAP τ (t1 : P1, ... tn : Pn), alors il existe un foncteur F_τ de $C(P1) \times \ldots \times C(Pn)$ dans $C(\tau)$ tel que :

$|F_\tau|$: objet $(C(P1) \times \ldots \times C(Pn)) \rightarrow$ objet $(C(\tau))$

$\forall X_1, X_n$ dans objet $(\underline{\underline{Ens}})$,

$[P_1(X_1), P_2(X_2), \ldots, P_n(X_n)] \mapsto \tau(X_1, X_2, \ldots, X_n)$

$|F_\tau|$ est l'application qui, étant donné des présentations P_i et $\tau(t_1{:}P_1, \ldots t_n{:}P_n)$ associe aux algèbres $P_i(X_i)$, l'algèbre $\tau(X_1, \ldots X_n)$ -

et F_τ : $fl(C(P1) \times \ldots C(Pn)) \rightarrow fl(C(\tau))$

associe à tout n-uple d'homomorphismes $[h_{p1}, \ldots h_{pn}]$ où h_{pi} est une flèche dans $C(P_i)$, l'homomorphisme dans $C(\tau)$ qui est la famille : id pour la sorte τ et h_{pi} pour les sortes $t_i{+}S$ de chaque paramètre.

2.2.3.3.2. Instanciation d'un TAP

Soit τ un TAP que l'on considère sans perte de généralité à un paramètre t:P. Alors, soit $\tau 1$ un type tel que $\tau 1 \in C(P)$ par l'intermédiaire d'opérateurs Σp de $\tau 1$, noté $\tau 1$: $P(\Sigma p)$. Le type défini par l'instanciation de t par $\tau 1$ et noté $\tau(\tau 1)$ est obtenu dans $C(\tau)$ par :

$F_\tau(h_{\tau 1})\ (|F_\tau|\ (P(U(\tau_1))))$

$= |F_\tau|(h_{\tau 1}(P(U(\tau 1))))$ d'après la propriété des foncteurs.

$= |F_\tau|(\tau 1{:}P(\Sigma_p))$

Autrement dit, $\tau(\tau 1)$ est l'extension par τ de $\tau 1$ considéré comme P-algèbre à l'aide des opérateurs Σp.

Il est temps de donner quelques exemples de tout ce formalisme.

Exemple 1 : Classe de propriété.

Soit la propriété monoïde donnée au § 2.2.3.1.3., et le type nat muni des opérateurs {0, succ, pred, +} , et des équations associées. L'ensemble sous-jacent est fourni par les représentants des classes d'équivalences :

$$\{\{0\}, \{succ(0)\}, \ldots \{succ^n(0)\}\ldots \} = U(nat)$$

Le monoïde libre engendré par U(nat) avec la constante 0 et l'opérateur + est une algèbre où l'on trouve les termes de U(nat) et les termes composés avec +, et les classes d'équivalences déduites des équations du monoïde, c'est-à-dire :

{0, 0 + 0, ...},

{succ(0),0+succ(0),...}, ...

mais dans cette algèbre monoïde(U(nat)), on ne peut pas identifier succ(succ(0)) et succ(0) + succ(0).

Ce qui permet de faire l'inclusion, c'est l'homomorphisme h_{nat} dans C(monoïde) qui plonge le monoïde nat dans l'algèbre nat. Les vérifications de propriétés du § précédent démontrent que cette inclusion est possible.

Exemple 2 : Type abstrait paramétré.

Soit un type vecteur pour lequel le paramètre formel appartient à la classe des monoïdes :

type vecteur (t : monoïde (0,+))

créerv : → vecteur (t)

étendre : vecteur (t) x t → vecteur (t)

restreindre : vecteur (t) → vecteur (t)

valeur : vecteur (t) → t

eqns

restreindre (créerv) = créerv

restreindre (étendre (v,x)) = v

valeur (créerv) = 0

valeur (étendre (v,x)) = valeur (v) + x

Pour tout ensemble de variables x, y, z... du domaine de t, on peut utiliser les "propriétés" ou les équivalences du monoïde libre engendré avec 0, + sur x, y, z ...

Ainsi :

valeur (étendre(étendre(étendre(créerv,x),y),z))

= valeur(étendre(étendre(créerv,x),y)) + z

= (valeur(étendre(créerv,x)) + y) + z

= ((valeur(créerv) + x) + y) + z

= ((0 + x) + y) + z

= (x + y) + z

= x + (y + z)

= (0 + x) + ((0 + y) + z)

= valeur(étendre(étendre(créerv,x), valeur(étendre(étendre(créerv,y),z))))

Dans le cas d'une instanciation de type :

vecteur (nat),

les équations sur vecteur restent vrai, mais on peut de plus utiliser les résultats spécifiques de nat. On sait dans vecteur(t) que

valeur(étendre(étendre(créerv,x),y)) = x + y

d'autre part, dans nat : succ(0) + $succ^2$(0) = $succ^3$(0)

d'où, par substitution (image de $h_{monoïde}$ par F_τ) :

valeur(étendre(étendre(créerv,succ(0)),$succ^2$(0))) = $succ^3$(0)

2.2.3.3.3. Composition générique

Un type abstrait paramétré $\tau(t_1,...t_n)$ satisfait une propriété P ssi, pour tout ensemble de générateurs $X_1,...X_n$, $\tau(X_1,...X_n)$ satisfait la propriété P.

On doit donc envisager la possibilité d'instancier un type abstrait paramétré par un autre type abstrait paramétré. C'est ce que l'on appelle la composition générique. Cette composition peut s'exprimer simplement par la composition des foncteurs.

Soit un TAP τ(t1:P1,...tn:Pn) et un TAP τ'(t1':P1',...tk':Pk') tel que $\tau' \in C(Pi)$.

Le type τ(t1,...ti-1, τ'(t1',...tk'),ti+1,...tn) est un TAP défini par le foncteur :

$$F_{\tau(...\tau'...)} : [C(P1),...C(Pi-1), C(P1'),... C(Pk')... C(Pn)] \rightarrow C(\tau)$$

tel que les constituants de $F_{\tau(...\tau'...)}$ sur les composantes, sont :

$$F_{p1},...F_{pi-1},F_{pi} \circ F_{p1},...,F_{pi} \circ F_{pk},F_{pi+1},...F_{pn}$$

Traitons là encore un petit exemple :

On modifie le type vecteur(t) du § 2.2.3.3.2. par :

type vecteur (t : monoïde (0,+) & égalité (eq-t))

créerv : → vecteur(t)

étendre : vecteur(t) x t → vecteur(t)

restreindre : vecteur(t) → vecteur(t)

valeur : vecteur(t) → t

eqv1 : vecteur(t) x vecteur(t) → bool

eqv2 : vecteur(t) x vecteur(t) → bool

les équations de restreindre et valeur sont conservées et on les complète par (le type bool étant supposé défini ailleurs) :

eqv1 (v1,v2) = eq-t (valeur(v1),valeur(v2))

eqv2 (créerv,créerv) = vrai

eqv2 (créerv,étendre(v,x)) = faux

eqv2 (étendre(v,x),créerv) = faux

eqv2 (étendre(v1,x1),étendre(v2,x2)) = eqv2(v1,v2) ∧ eq-t (x1,x2)

On peut facilement montrer que vecteur (t) : égalité (eqv1) et vecteur (t) : égalité (eqv2), sont vérifiées, avec l'hypothèse de t : égalité (eq-t).

On déclare aussi le TAP ens comme au § 2.2.2.3.1., mais avec un type formel satisfaisant la propriété d'égalité :

type ens (t' : égalité (eq))

créer : → ens (t')

ins : t' x ens (t') → ens (t')

enl : t' x ens (t') → ens (t')

app : t' x ens (t') → bool

ins (x,ins(y,S)) = si eq (x,y) alors ins (y,S)
sinon ins (y,ins(x,S)) fsi

enl (x,ins(y,S))= si eq (x,y) alors enl(x,S)
sinon ins (y,enl(x,S)) fsi

app (x,créer) = faux

app (x,ins(y,S)) = si eq (x,y) alors vrai
sinon app (x,S) fsi

enl(x,créer) = créer

Avec ces définitions de types, on peut construire le nouveau type générique ens (vecteur (t : monoïde (0,+) & égalité (eq-t)))
où vecteur (t) : égalité (eqv2) - (On a de même un ens (vecteur (t)) où vecteur (t) : égalité (eqv1)). Ces deux compositions génériques ne sont évidemmen pas identiques ;
Par exemple, dans le premier cas :

ins (étendre(étendre(créerv,x),y),ins(étendre(créerv,x+y),créer))
et ins (étendre(créerv,x+y),ins(étendre(étendre(créerv,x),y),créer))
sont différents, tandis qu'ils sont équivalents dans le second.

On obtient une présentation de ens (vecteur (t)) en concaténant la présentation de vecteur (t) à celle de ens (t') dans laquelle t' est substitué uniformément par vecteur (t), et eq par eqv2 (premier cas), les variables dans le domaine de t' devenant des variables dans vecteur(t).
De plus, on ajoute les présentations des types de base utilisés (ici bool).

Unicité de la composition.

On peut se demander si le type ens(vecteur(t)) où t est substitué par nat est identique à ens (t') où t' est substitué par vecteur (nat). La réponse, affirmative, se base sur :

- unicité de la composition des foncteurs de TAP ;
- le fait que l'instanciation de t' par vecteur (nat) n'utilise que les propriétés génériques de vecteur et non des propriétés plus fortes de vecteur (nat).

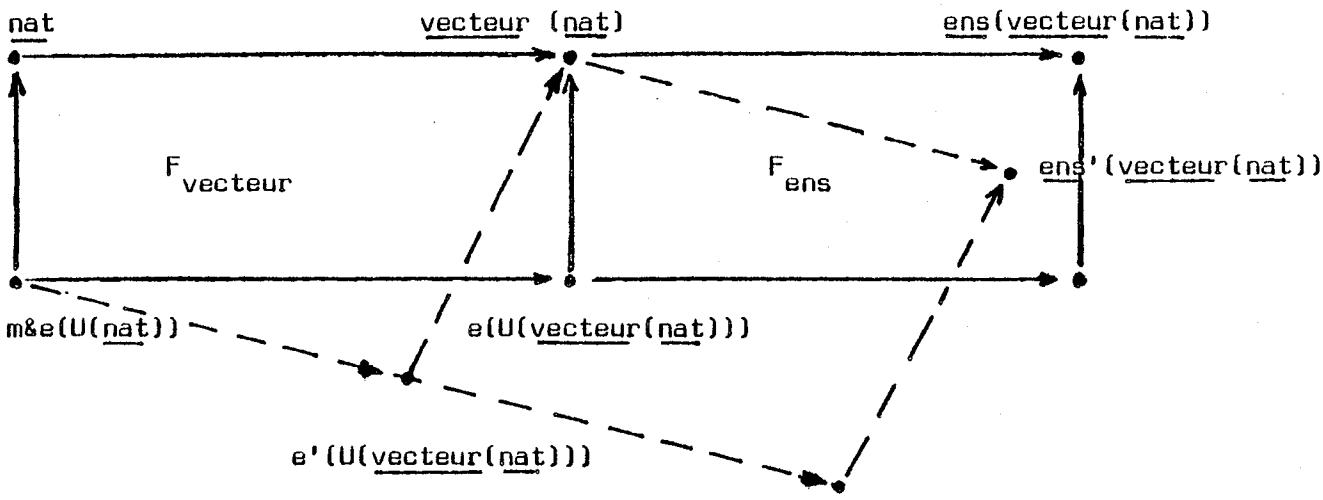
En termes algébriques, soit $\underline{\underline{m \,\&\, e}}$ la catégorie de monoïde & égalité et $\underline{\underline{e}}$ la catégorie de l'égalité. On note $\underline{\underline{A}}\ (X,Y)$ la flèche dans A entre X et Y, et $F_{vecteur}$ et F_{ens}, les foncteurs des TAP. La deuxième condition ci-dessus se traduit par :

$$F_{vecteur}\ (\underline{\underline{m\&e}}\ (m\&e(U(\underline{nat})),\underline{nat}))$$
$$= \underline{\underline{e}}\ (|F_{vecteur}|\ (m\&e(U(\underline{nat}))), |F_{vecteur}|\ (\underline{nat}))$$
$$= \underline{\underline{e}}\ (e(U(\underline{vecteur}(\underline{nat}))), \underline{vecteur}(\underline{nat}))$$

d'où : $\underline{ens}\ (\underline{vecteur}(\underline{nat})) = |F_{ens} \circ F_{vecteur}|\ \underline{nat}$

$$= |F_{ens} \circ F_{vecteur}|\ (\underline{\underline{m\&e}}\ (m\&e(U(\underline{nat})),\underline{nat})\ (m\&e(U(\underline{nat}))))$$
$$= ((F_{ens} \circ F_{vecteur})\ (\underline{\underline{m\&e}}\ (m\&e(U(\underline{nat})),\underline{nat})))\ (|F_{ens} \circ F_{vecteur}|\ m\&e(U(\underline{nat}))$$
$$= F_{ens}\ (\underline{\underline{e}}\ (e\ (U(\underline{vecteur}(\underline{nat}))), \underline{vecteur}(\underline{nat})))\ (|F_{ens}|\ e(U(\underline{vecteur}(\underline{nat}))))$$
$$= (F_{ens})\ (\underline{\underline{e}}\ (e(U(\underline{vecteur}(\underline{nat}))), \underline{vecteur}(\underline{nat}))\ e(U(\underline{vecteur}(\underline{nat}))))$$
$$= |F_{ens}|\ \underline{vecteur}(\underline{nat})$$

Ce qui s'exprime dans le diagramme :



Remarque : en pointillés, on a noté l'effet de la liaison avec une autre opération d'égalité dans vecteur.

2.2.3.3.4. Représentation et inclusion de TAP

2.2.3.3.4.1. Deux TAP : τ_1 (t1 : P1,...t_n : P_n) et
τ_2(t'1, : P'1,...t'_n : P'_n) sont isomorphes ($\tau_1 \cong \tau_2$) ssi

$(\forall X_1,...X_n)$ ensembles de générateurs, $\tau_1 (X_1,...X_n) \cong \tau_2(X_1,...X_n)$

Cette définition impose que le nombre de types formels soit le même dans τ_1 et τ_2 et que les propriétés soient deux à deux équivalentes. Toutes les techniques données au § 2.2.2. sont utilisables pour démontrer l'isomorphisme des TAP. De même, construire une représentation d'un TAP, c'est trouver un nouveau TAP utilisant les types déjà construits et tel qu'il soit isomorphe au TAP que l'on veut représenter.

2.2.3.3.4.2. Treillis des représentations

La condition de représentation que l'on vient de donner est un peu trop rigide car elle oblige à avoir des représentations qui ne tiennent compte que des propriétés les plus faibles exigées sur les paramètres. Ce qui était un avantage dans la composition générique, devient un inconvénient dans la représentation, car une instance de type paramétré peut être très mal représentée en utilisant la représentation la plus générale. Par exemple, si l'on définit

le type table (t1,t2) où la propriété de t1 est simplement l'égalité, afin de pouvoir rechercher une valeur de t2 à l'aide d'une valeur de t1, il est évident que le choix des représentations de table est limité et que l'on n'a à sa disposition que des techniques de recherche séquentielle. Un type tel que table (nat,valeur) est donc valide puisque nat possède l'égalité, mais alors que, par exemple, l'ordre total pour t1 peut être sans intérêt pour la spécification abstraite (et de plus, limite les types qui peuvent être liés à t1), il en va tout-à-fait différemment lorsqu'on choisit une représentation, où les particularités de t1 influent grandement sur l'efficacité de la représentation. Ici, un rangement ordonné et une recherche dichotomique seraient mieux adaptés. En termes plus généraux, ce cas se produit lorsque l'invariant de représentation utilise des "propriétés" au sens général des types en paramètres. Mais il se trouve que ces propriétés au sens général s'expriment presque toujours par des propriétés telles que celles du § 2.2.3.1., c'est-à-dire qu'elles font intervenir des opérateurs, des constantes et des relations. On a vu la propriété d'ordre total ; on peut aussi, dans le cas de la table, utiliser la propriété dispersion (h,n) sur t :

h : t → intervalle (nat) (1,n)

n : → nat

où h est une fonction surjective de t dans [1,n] , ou, mieux encore, la propriété "scalaire" de PASCAL (type index dans [TEN77]), qui peut s'exprimer soit par les primitives de PASCAL (§ 2.2.5.), soit par :

scalaire (h,n,g) sur t :

h : t → intervalle (nat) (1,n)

n : → nat

g : intervalle (nat) (1,n) → t

g(h(x)) = x

h(g(y)) = y

On a donc la possibilité de définir des représentations τ' (t : P') d'un TAP τ(t : P) telles que $P \subseteq P'$.

L'ensemble des représentations est donc ordonné par les propriétés sur les paramètres formels, l'élément minimum correspondant à l'ensemble des représentations qui exigent exactement les propriétés du TAP.

Lors d'une liaison du paramètre avec un type effectif de propriété P1, un premier critère de choix de représentation est de choisir la représentation r exigeant une propriété P2 $\subseteq$ P1 et telle qu'il n'existe pas dans le demi-treillis d'autres représentations exigeant P3 telle que P3 $\neq$ P2 et P2 $\subseteq$ P3 $\subseteq$ P1.

Ce critère de choix n'est pas suffisant, car en général, il peut y avoir plusieurs représentations exigeant des propriétés distinctes qui satisfont cette condition, de même qu'il peut y avoir plusieurs représentations exigeant la même propriété. Néanmoins, dans beaucoup de cas simples, ce critère peut déterminer une seule représentation. La même idée de sélection de représentation a été proposée, mais sans utiliser explicitement des propriétés, dans [MIT77].

2.2.3.3.4.3. Inclusion de TAP

L'idée de l'inclusion des représentations peut être reprise pour un TAP, au niveau de ses fonctions dérivées (extension). Un exemple intéressant est celui du type matricecarrée (t), qui n'exige aucune propriété particulière sur t si l'on s'en tient aux fonctions d'accès comme tableau à deux dimensions, mais qui exige t : monoïde si l'on veut réaliser l'addition de deux matrices ; t : corps, si l'on veut réaliser la multiplication, l'inversion, etc... ; t : pseudo-treillis si l'on veut réaliser la fermeture transitive, etc... Rien n'empêche évidemment de définir des types distincts dans chacun des cas, mais il est davantage dans l'esprit de la généricité de profiter au maximum d'un même cadre de définition et de jouer sur la spécification des types formels. Le mécanisme de définition est alors le suivant : le TAP τ est déclaré avec un type formel qui exige la propriété P minimum (celle des fonctions de base). Une fonction dérivée f peut exiger pour le type formel d'un de ses paramètres (de type τ) une propriété P' avec P $\subseteq$ P'. Cette fonction f n'est définie pour une instance $\tau(\tau')$ que si τ' satisfait la propriété P'. Dans une telle définition, le TAP $\tau(t)$ contient plus ou moins de fonctions suivant les propriétés du type effectif lié à t. De la même manière, le type τ peut satisfaire des propriétés à l'aide de ses fonctions. On donne à τ la (ou les) propriété(s) satisfaite(s) comme si le type possédait toutes ses fonctions. Dans une instance particulière, le type τ ne possède effectivement que la sous-propriété déduite de la propriété de τ et des fonctions définies pour cette instance. Les propriétés des types effectifs liés à des types formels de τ permettent donc de déterminer :

- l'ensemble des fonctions valides pour cette instance ;
- la plus grande propriété que satisfait cette instance. On verra au § 2.2.3.5. quelques exemples de spécification de TAP et de représentations mettant en oeuvre les idées de tout ce paragraphe.

2.2.3.4. Traitement des erreurs

Les algèbres d'erreur (§ 2.2.2.1.6.) résolvent bien tous les problèmes théoriques que l'on peut se poser au sujet du traitement des erreurs, mais il faut encore un peu y revenir pour que l'on puisse passer de la spécification à la réalisation pratique. Les problèmes à résoudre sont :
- ne pas tester dans les opérations-ok si les arguments sont erreur ou non ;
- pouvoir passer d'un état erreur à un certain niveau à un état non-erreur à un autre niveau d'abstraction (récupération d'erreur) ;
- avoir une sémantique standard du traitement des erreurs qui puisse être implantée par une structure de contrôle spécifique.

Le premier argument, qui pourrait être utilisé dans un système formel de réduction avec algèbre d'erreur, suggère qu'il y ait deux modes de fonctionnement : le mode normal dans lequel les équations-ok sont valides, le mode erreur, dans lequel ce sont les équations-erreur. Aussi, pourquoi ne pas faire de ce mode d'erreur un type d'erreur, (ou méta-type prédéfini) qui comprend tous les termes d'erreur qui peuvent être engendrés. Pour pouvoir être exploités, ces termes d'erreur doivent être reconnus par des prédicats portant sur l'opérateur qui a pu produire une erreur (équivalent au test dynamique de type union). Le fait d'introduire un méta-type d'erreur résout tout aussi bien le deuxième point, puisqu'il suffit de tester une valeur erreur d'un type erreur, et suivant le résultat, retourner une valeur de n'importe quel type. Enfin, la sémantique standard des erreurs est celle d'être conservées à travers tous les appels de fonction, exceptés les prédicats d'erreur :

$(\forall f)\quad f \neq \text{prédicat d'erreur} \quad f(X,\ldots) = X.$

ssi X est un terme d'erreur.

Le seul point à résoudre est de savoir comment une fonction qui retourne des erreurs peut être décrite syntaxiquement puisque le type du résultat n'est pas unique. Là encore, il suffit d'utiliser l'union de type des langages algo-

rithmiques traditionnels, en se limitant aux unions d'un type algébrique avec le type erreur.

Reprenons l'exemple de la pile ; on note E le type erreur et son domaine :

type pile ;

créerp : → pile
empiler : pile x nat → pile
dépiler : pile → pile ou E
sommet : pile → nat ou E
errpile : → E

Les équations de pile sont données par :

sommet (créerp) ⇒errpile
sommet (empiler(P,N)) = N
dépiler (créerp) =>errpile
dépiler (empiler(P,N)) = P

Le symbole =>signifie que la relation n'est pas symétrique, ce qui fait que l'on peut avoir la même constante d'erreur pour sommet (créerp) et dépiler (créerp) sans que l'on puisse en déduire sommet (créerp) = dépiler (créerp). Dans E on a maintenant les prédicats :

E-errpile (errpile) = vrai
E-errpile (X) = faux ssi X ≠ errpile

L'intérêt de ces prédicats est qu'ils peuvent être utilisés pour construire des opérateurs dérivés. Définissons par exemple l'opérateur :

remplacer : pile x nat → pile ou E

tel que remplacer (créerp,M) ⇒errpile
remplacer (empiler(R,N),M) = empiler(R,M)

Si on veut définir remplacer comme opérateur dérivé, on s'aperçoit que l'on a :

remplacer (P,M) = empiler (dépiler(P),M)

sauf dans le cas où dépiler (P) =>errpile, pour lequel le résultat est errpile. On peut penser que la règle de transitivité des erreurs s'applique et que l'on obtient bien le résultat cherché. Mais ce n'est pas le cas, puisqu'on déduirait :

remplacer (créerp,M) = empiler (dépiler(créerp),M)
= empiler (errpile,M)
= errpile

ce qui n'est pas exactement ce que l'on veut, puisqu'on ne veut pas l'égalité. On introduit donc une notation particulière qui permet de réaliser ce concept, c'est la condition d'erreur (ou condition d'exception). Sa forme générale est :

```
L = R  ou
    qd  t-erreur1  alors  R1 |
        t-erreur2  alors  R2 |
        ..... fq
```

t-erreur$_i$ sont des termes d'erreur pouvant provenir de la réduction de R et R_i est un terme qui peut être une erreur ou non. S'il peut y avoir ambigüité dans les termes d'erreur, ceux-ci seront de la forme :

L_i =>t-erreur$_i$ où L_i contient des opérateurs et variables de R.

La signification de cette équation générale est :

```
L = si  E-t-erreur1 (R)  alors [=>] R1
    sinonsi  E-t-erreur2 (R)  alors [=>] R2
      :
    sinon  R  fsi
```

Le signe => est introduit après le alors lorsque R_i est un terme d'erreur, ce qui étend les conditions aux conditions retournant erreur :

L = si C alors => R1 sinon => R2 fsi

est équivalent à :

L => si C alors R1 sinon R2 fsi

De même :

L = si C alors => R1 sinon R2 fsi

est mis pour l'équation L = R2 lorsque C ≡ faux
et L => R1 lorsque C ≡ vrai
La définition de remplacer est alors :

```
remplacer (P,M) = empiler (dépiler (P),M)  ou
                  qd  errpile  alors  errpile  fq
```

La forme étendue de cette condition d'erreur est :

```
remplacer (P,M) = empiler (dépiler (P),M) ou
        qd  dépiler (P,M) ⇒ errpile  alors  errpile  fq
```

On peut, si l'on veut, définir un opérateur dérivé remplacer qui "récupère" l'erreur :

```
remplacer (P,M) = empiler (dépiler (P),M)  ou
                  qd  errpile  alors  empiler (P,M)  fq
```

Ce nouvel opérateur a la fonctionnalité :

remplacer : pile x nat → pile

Dans ce deuxième cas, remplacer (créerp,M) et empiler (créerp,M) sont équivalents.

Remarque : lorsqu'on ne définit pas explicitement de condition d'erreur pour des opérateurs dont les calculs peuvent produire des erreurs, c'est la règle de propagation standard des erreurs qui est prise par défaut et qui ramène l'erreur jusqu'au premier niveau (en remontant la chaîne d'appels) où elle est traitée. De ce fait, tous les opérateurs, implicitement, sont de la forme : op : param → result ou E .

Néanmoins, pour simplifier l'écriture et mettre en évidence les cas de "création" d'erreur, on n'utilise cette notation que pour les opérateurs qui sont définis explicitement avec des équations possédant des parties droites :

⇒ idf-erreur.

2.2.3.5. Exemples

Pour clore ce chapitre, voici quelques exemples de définition de types et de propriétés, ainsi qu'un exemple de représentation.

2.2.3.5.1. Définition de propriétés

```
prop  monoïde  (0,+)  sur  t : tyfo ;
   |  0 : → t
   |  + : t x t → t
   eqns
   |  0 + x = x
   |  x + 0 = x
   |  (x + y) + z = x + (y + z)
fprop ;

prop  groupe (0,+,opp,-)  sur  t : monoïde (0,+) ;
   |  opp : t → t
   |  -   : t x t → t
   eqns
   |  opp (x) + x = 0
   |  x + opp (x) = 0
   |  x - y = x + opp(y)
fprop ;

prop  gr-com (0,+,opp,-)  sur  t : groupe (0,+,opp,-) ;
   eqns  x + y = y + x
fprop ;

prop  anneau-unit (0,+,opp,-,1,*)
   |     sur  t : gr-com (0,+,opp,-)  &  monoïde (1,*)
   eqns  x * (y + z) = x * y + x * z
   |     (x + y) * z  = x * z + y * z
   |
   |     sur  t : dioïde (0,+,1,*)
   eqns  opp (x) + x = 0
   |     x + opp (x) = 0
   |     x - y = x + opp (y)
fprop ;
```

```
prop  corps (0,+,opp,-,1,*,inv,/)
   |          sur  t : anneau-unit (0,+,opp,-,1,*)
   |      inv : t → t  ou  E
   |       /  : t x t → t  ou E
   |  eqns  inv (0) ⇒ div-par-zéro
   |        inv (x) * x = 1
   |        x * inv (x) = 1
   |        x/y = x * inv (y)  ou
   |              qd  div-par-zéro  alors  div-par-zéro  fq
fprop ;

prop  dioïde (0,+,1,*)  sur  t : monoïde (0,+) & monoïde (1,*)
   eqns  x + y = y + x
   |     x * 0 = 0
   |     0 * x = 0
   |     x * (y + z) = x * y + x * z
   |    (x + y) * z = x * z + y * z
fprop ;

prop  pseudo-treillis (0,+,1,*)  sur  t : dioïde (0,+,1,*) ;

   eqns  x + x = x
   |     x + x * y = x
   |     x + y * x = x
fprop ;
```

Note : on n'a pas voulu décrire ici un memento d'algèbre, (qui serait d'ailleurs en partie incorrect), mais simplement donner la façon de construire un arbre d'inclusion de propriétés. La seule vérification à faire est que les équations respectent la fonctionnalité des opérateurs. Les opérateurs standards sont décrits dans les propriétés ou les types abstraits de base, et leur priorité est fixée une fois pour toutes. Ainsi, par exemple, si un type possède une propriété anneau-unit où + et * sont implantés par de nouveaux opérateurs, ceux-ci acquièrent les priorités de + et *, à moins que cela crée des incohérences de définition. Il n'y a pas d'inconvénients à ce que des types distincts aient des noms d'opérateurs identiques, car ces noms sont pseudo-locaux, c'est-à-dire qu'ils sont locaux dans la définition de propriété, et par défaut, comme opérateurs formels de propriété exigée, s'il n'y a pas ambigüité (cf. 2.2.3.5.2.).

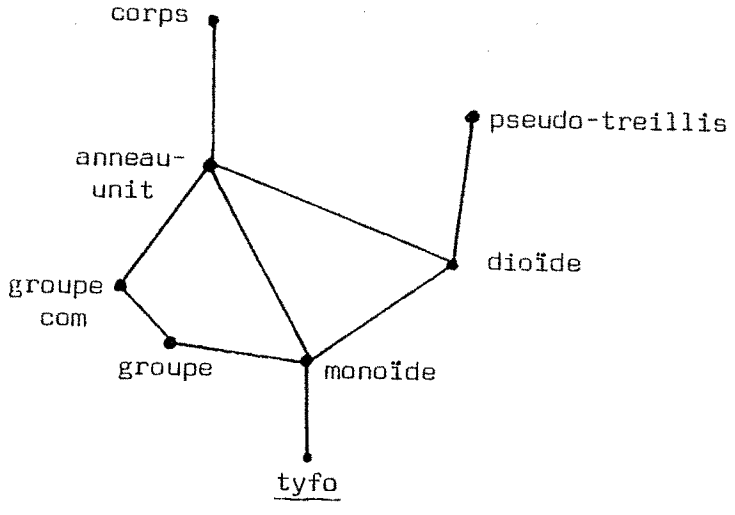
On aurait pu tout aussi bien écrire :

prop dioïde (0,v,1,∧) sur t : monoïde (0,v) & monoïde (1,∧) ;

eqns ...

La propriété anneau-unit a deux sous-propriétés et deux manières équivalentes d'être défini.

L'arbre d'inclusion de ces propriétés issu de ces définitions est :



2.2.3.5.2. Type générique des matrices carrées d'ordre n

On définit préalablement le type vecteur de longueur l qui sera utile dans la définition des matrices.

type vect (t : tyfo) attr l : nat ;
indefv : → E
créerv : → vect(t)
ranger : vect(t) x nat x t → vect(t) ou E
val : vect(t) x nat → t ou E

eqns

ranger (v,i,a) = si ¬(le(succ(0),i) ∧ le(i,l(v)))
alors ⇒indefv fsi

ranger (ranger(v,i,a),j,b) = si ¬ eq-nat (i,j)
alors ranger (ranger(v,j,b),i,a) fsi

val (créerv,j) ⇒ indefv

val (ranger(v,i,a),j) = si eq-nat (i,j) alors a
sinon val (v,j) fsi

```
extension  vect (t : monoïde) : monoïde (v0,comp) ;
      v0 : → vect (t)
      comp : vect (t) x vect (t) → vect (t)  ou E
      reduc : vect (t) → t
   privé  init0 : vect (t) x nat → vect (t)
          compel : vect (t) x vect (t) x nat → vect (t)
          reducel : vect (t) x nat → t
 eqns
   v0 = init0(créerv,l(v0))
   init0 (v,0) = v
   init0 (v,succ(i)) = ranger (init0(v,i),succ(i),0)
   comp (v1,v2) = si ¬ eq-nat (l(v1),l(v2))
              alors⇒ indefv  sinon  compel (v1,v2, l(v1))  fsi
   compel (v1,v2,0) = créerv
   compel (v1,v2,succ(i)) = ranger (compel(v1,v2,i),succ(i),val(v1,succ(i)) +
                                                        val(v2,succ(i)))
   reduc (v) = reducel (v,l(v))
   reducel (v,0) = 0
   reducel (v,succ(i)) = reducel (v,i) + val (v,succ(i))
ftype ;

type  mc (t : tyfo)  attr  ordre : nat ;
      créerm : → mc (t)
      ranger : mc (t) x nat x nat x t → mc (t)  ou E
      val : mc (t) x nat : nat → t  ou E
      ligne : mc (t) x nat → vect (t)
      col : mc (t) x nat → vect (t)
      transpo : mc (t) → mc (t)
      valide : mc (t) → bool
      indefm : → E
  privé
     v : mc (t) x nat x nat → bool
     init-el : mc (t) x nat x nat → bool
  eqns
  ranger (m,i,j,a) = si ¬(le(succ(0),i) ∧ le (i,ordre(m))
           ∧ le(succ(0),j) ∧ le(j,ordre(m)))  alors ⇒ indefm fsi
```

```
    ranger (ranger(m,i1,j1,a),i2,j2,b) =
            si ¬(eq-nat(i1,i2) ∧ eq-nat(j1,j2))
            alors  ranger (ranger(m,i2,j2,b),i1,j1,a)  fsi

    val (créerm,i,j) ⇒indefm
    val (ranger(m,i1,j1,a),i2,j2,) =
            si  eq-nat (i1,i2) ∧ eq-nat(j1,j2)  alors  a
            sinon  val (m,i2,j2)  fsi

    ligne (créerm,k) = créerv
    ligne (ranger(m,i,j,a),k) =
            si  eq-nat (i,k)  alors  ranger(ligne(m,k),j,a)
            sinon  ligne(m,k)  fsi

    col (créerm,k) = créerv
    col (ranger(m,i,j,a),k) =
            si  eq-nat(j,k)  alors  ranger(col(m,k),i,a)
            sinon  col(m,k)  fsi

    transpo (créerm) = créerm
    transpo (ranger(m,i,j,a)) = ranger(transpo(m),j,i,a)

    valide (m) = v(m,ordre(m),ordre(m))

    v (m,0,0) = vrai

    v (m,succ(i),0) = v (m,i,ordre(m))

    v (m,succ(i),succ(j)) =
                v (m,succ(i),j) ∧ init-el (m,succ(i),succ(j))

    init-el (créerm,i,j) = faux

    init-el (ranger(m,i1,j1,a),i2,j2) =
             si  eq-nat (i1,i2) ∧ eq-nat(j1,j2)
             alors  vrai  sinon  init-el (m,i2,j2)  fsi

extension  mc(t : dioïde) : dioïde (mc0,add,mc1,mult) ;

    mc0 : → mc(t)
    mc1 : → mc(t)
    add : mc(t) x mc(t) → mc(t)  ou  E
    mult : mc(t) x mc(t) → mc(t)  ou  E
```

```
privé
    faire-mc0 : mc(t) x nat x nat → mc(t)
    faire-mc1 : mc(t) x nat x nat → mc(t)
    add-par-el : mc(t) x mc(t) x nat x nat → mc(t)
    mult-par-el : mc(t) x mc(t) x nat x nat → mc(t)

eqns
  mc0 = faire-mc0 (créerm,ordre(mc0),ordre(mc0))
  faire-mc0 (m,0,0) = m
  faire-mc0  (m,succ(i),0) = faire-mc0(m,i,ordre(m))
  faire-mc0 (m,succ(i),succ(j)) =
      faire-mc0 (ranger(m,succ(i),succ(j),0),succ(i),j)
  mc1 = faire-mc1 (créerm,ordre(mc1),ordre(mc1))
  faire-mc1 (m,0,0) = m
  faire-mc1 (m,succ(i),0) = faire-mc1(m,i,ordre(m))
  faire-mc1 (m,succ(i),succ(j)) =
      faire-mc1 (ranger(m,succ(i),succ(j),
              si eq-nat(i,j) alors 1 sinon 0 fsi),succ(i),j)
  add (m1,m2) = si  eq-nat (ordre(m1),ordre(m2)) ∧
      valide (m1) ∧ valide (m2)
      alors  add-par-el (m1,m2,ordre(m1),ordre(m1))
      sinon ⇒ indefm  fsi

  add-par-el (m1,m2,0,0) = créerm
  add-par-el (m1,m2,succ(i),0) = add-par-el (m1,m2,i,ordre(m1))
  add-par-el (m1,m2,succ(i),succ(j)) =
      ranger (add-par-el(m1,m2,succ(i),j),succ(i),succ(j),
              val(m1,succ(i),succ(j)) + val(m2,succ(i),succ(j)))
  mult(m1,m2) = si  eq-nat (ordre(m1),ordre(m2)) ∧
      valide (m1) ∧ valide (m2)
      alors  mult-par-el (m1,m2,ordre(m1),ordre(m1))
      sinon ⇒ indefm  fsi

  mult-par-el (m1,m2,0,0) = créerm
  mult-par-el (m1,m2,succ(i),0) = mult-par-el (m1,m2,i,ordre(m1))
  mult-par-el (m1,m2,succ(i),succ(j)) =
      ranger(mult-par-el(m1,m2,succ(i),j),succ(i),succ(j),
        reduc[t : monoïde(0,+)]  (comp[t : monoïde(1,*)]
              (ligne(m1,succ(i)),col(m2,succ(j)))))
```

```
extension  mc(t : corps) : anneau-unit (mc0,add,opp-m,moins,mc1,mult)

    opp-m : mc(t) → mc(t)
    moins : mc(t) x mc(t) → mc(t)  ou E
    det   : mc(t) → t
    inv   : mc(t) → mc(t)  ou E

  eqns
opp-m (créerm) = créerm
opp-m (ranger(m,i,j,a)) = ranger (opp-m(m),i,j,opp(a))
moins (m1,m2) = add(m1,opp-m(m2))  ou
                qd  indefm  alors  indefm  fq
det(m) = ... calcul du déterminant ...
inv(m) = ... calcul de l'inverse de m (s'il existe) ...

extension  mc (t : pseudo-treillis) ;
        fermeture : mc(t) → mc(t)  ou E
        fermeture-reflex : mc(t) → mc(t)   ou E
    privé
        fm : mc(t) x mc(t) x nat → mc(t)
  eqns
    fermeture (m) = si valide (m)  alors  fm (m,m,ordre(m))
                                  sinon  ⇒ indefm  fsi
    fm (m1,m2,0) = m1
    fm (m1,m2,succ(i)) = fm (mult(add(mc1,m1),m2),i)
    fermeture-reflex (m) = add (mc1,fermeture(m))  ou
                    qd  indefm  alors  indefm  fq
ftype ;
```

Commentaires :

1. La structure générale est celle d'un noyau de fonctions du type suivi d'extensions. Chaque extension détermine une portée où les propriétés des types formels peuvent être plus fortes que dans le noyau. Les extensions sont ordonnées, au même titre que les propriétés. Elles pourraient être définies séparément

du type initial, car chaque extension est justifiée par une application particulière. Il est peut-être possible d'étendre plusieurs types à la fois, mais ce point reste encore à préciser syntaxiquement (cf. theory module" dans le "i system", qui est à plusieurs égards très proche de nos concepts - ex : category modules et propriétés - [NAK77]).

2. On a introduit presque toujours des opérations "privées" ; celles-ci ne font pas partie de la signature extérieure du type et ne seront probablement pas implantées, mais sont utiles dans la spécification des opérateurs "visibles". Elles pourront servir également dans l'implantation pour la découverte d'invariants de programmes (cf. 3.2.3.).

3. L'opération mult, dans mult-par-el, donne un bon exemple d'une liaison locale d'opérateurs formels grâce aux propriétés : reduc et comp exigent que le type formel ait la propriété de monoïde. Or, dans cette utilisation, le type utilisé (qui est encore formel), a deux fois la propriété de monoïde. On lève l'ambigüité à l'appel en indiquant entre [ ] quel monoïde on veut utiliser.

4. La transmission des valeurs attributs n'est pas très claire. Pratiquement, l'idée est très facile à mettre en oeuvre. Il suffit de donner explicitement les valeurs d'attributs à chaque occurrence de définition d'un objet, c'est-à-dire les déclarations et les résultats de fonctions, mais cela n'est pas aussi simple dans la spécification algébrique, à moins que l'attribut ne soit explicite partout.

5. Cet exemple présente une utilisation non systématique du traitement des erreurs. Si l'on veut "garder" chaque opération par des prédicats (par exemple, valide(m)), il paraît important que ceux-ci soient mis en tant que préconditions et non comme tests explicites, afin de pouvoir supprimer automatiquement les tests inutiles dans les compositions d'opérations.

6. Les vérifications des propriétés se font par induction comme indiqué au § 2.2.3.1.3. Montrons par exemple le théorème :

$$(\forall m1,m2,m3 : mc(t : dioïde)) \wedge ordre(m1) = ordre(m2) = ordre(m3) \wedge valide(m1) \wedge valide(m2) \wedge valide(m3),$$
$$mult(add(m1,m2),m3) = add(mult(m1,m3),mult(m2,m3))$$

```
(induction) => mult-par-el(add(m1,m2),m3,succ(i),succ(j))
      = add-par-el(mult(m1,m3),mult(m2,m3),succ(i),succ(j))
   => reduc[t : monoïde(0,+)] (comp[t : monoïde(1,*)]
        (ligne(add(m1,m2),succ(i)),col(m3,succ(j))))
      = val(mult(m1,m3),succ(i),succ(j)) + val(mult(m2,m3),succ(i),succ(j))
```

pour simplifier, notons les opérations génériques respectivement reduc[+] et comp[*] . On veut montrer :

```
   => reduc[+] (comp[*] (ligne(add(m1,m2),succ(i)),col(m3,succ(j))))
      = reduc[+](comp[*] (ligne(m1,succ(i)),col(m3,succ(j))))
       + reduc[+](comp[*](ligne(m2,succ(i)),col(m3,succ(j))))
```

Pour cela, il faut démontrer les lemmes suivants (que l'on ne détaille pas) :

```
lemme 1 : (∀m,m' : mc(t:dioïde))(∀k : nat ∧ 1 ≤ k ≤ ordre(m))
     ∧ ordre(m) = ordre(m') ∧ valide(m) ∧ valide(m'),
     ligne(add(m,m'),k) = comp[+] (ligne(m,k),ligne(m',k))

lemme2 : ∀ v1,v2,v3 : vect(t : dioïde) ∧ l(v1) = l(v2) = l(v3),
     comp[*](comp[+] (v1,v2),v3)
         = comp[+] (comp[*](v1,v3),comp[*](v2,v3))

lemme 3 : ∀ v1,v2 : vect(t : monoïde) ∧ l(v1) = l(v2),
     reduc[+] (comp[+] (v1,v2))
         = reduc[+] (v1) + reduc[+] (v2)
```

2.2.3.5.3. Type table et ses implantations

Ce paragraphe montre des exemples d'implantations plus ou moins complexes pour un type donné, en fonctions des propriétés des types formels. On ne donne qu'une seule implantation dans chaque cas, mais à cause de l'inclusion des propriétés, un type dont les paramètres satisfont les propriétés les plus fortes, peut être implanté de plusieurs manières.

Résumons d'abord les propriétés qui seront utilisées :

```
prop  égalité (==)  sur  t : tyfo ;
     == : t x t → bool

  eqns
       x == x = vrai
       x == y = y == x
       x == y ∧ y == z  ∧¬(x == z) = faux
fprop ;
```

prop ordre-total (≤,==) sur t : égalité (==) ;
≤ : t x t → bool

eqns
x ≤ x = vrai
x ≤ y ∧ y ≤ z ∧ ¬(x ≤ z) = faux
x ≤ y ∧ y ≤ x = x == y
fprop ;

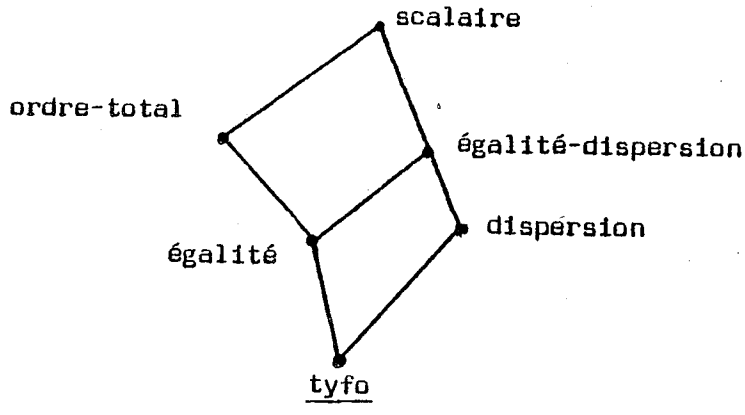
prop dispersion (h,n) sur t : tyfo
h : t → intervalle (nat) (1,n)
n : → nat
fprop ;

prop scalaire (h,n,g) sur t : ordre-total (≤) & dispersion (h,n) ;
g : intervalle (nat) (1,n) → t

eqns g (h(x)) = x
h(g(y)) = y
x ≤ y = le(h(x),h(y))
fprop ;

Remarques : - on note 1 = succ(0) ;
- dispersion est une propriété sans équations ; on ne s'intéresse qu'à sa spécification syntaxique.

Arbre des propriétés :



Le type que l'on veut implanter est :

```
type  table (t1: égalité(=),t2 : tyfo) ;
   |    créert : → table (t1,t2)
   |    ins : t1 x t2 x table (t1,t2) → table (t1,t2)  ou E
   |    trouver : t1 x table (t1,t2) → t2  ou E
   |    app : t1 x table (t1,t2) → bool
   |    double-def : t1 → E
   |    non-def : t1 → E
   |  eqns
   |  ins (x,a,t) = si  app (x,t)  alors⇒ double-def (x)  fsi
   |  ins (x,a,ins(y,b,t)) = ins(y,b,ins(x,a,t))
   |  trouver (x,créert) ⇒ non-def(x)
   |  trouver (x,ins(y,a,t)) = si  x = y  alors  a
   |                            sinon  trouver (x,t)  fsi
   |  app (x,créert) = faux
   |  app (x,ins(y,a,t)) = si  x = y  alors  vrai  sinon  app(x,t)  fsi
ftype ;
```

Voici, d'autre part, les types qui permettront de faire l'implantation :

```
type  liste (t : tyfo) ;
   |    nil : → liste(t)
   |    cons : t x liste (t) → liste (t)
ftype ;
```

```
type  [t1 : tyfo,t2 : tyfo]  ;  co  le type tuple est confondu avec son
   |                                constructeur co
   |    [,] : t1 x t2 → [t1,t2]
   |  premier : [t1,t2] → t1
   |  second : [t1,t2] → t2
   |  eqns
   |  premier ([x,y]) = x
   |  second ([x,y]) = y
ftype ;
```

```
type  tableau (t : tyfo) ;
   |  tabvide : → tableau (t)
   |  ranger : tableau (t) x nat x t → tableau (t)
   |  val : tableau (t) x nat → t  ou E
   |  indef : → E

   eqns
   |  ranger (ranger(t,i,a),j,b) = si  eq-nat (i,j)
   |  alors  ranger  (t,j,b)  sinon  ranger (ranger(t,j,b),i,a)  fsi
   |  val (tabvide,i)⇒ indef
   |  val (ranger(t,i,a),j) = si  eq-nat (i,j)  alors  a
   |                      sinon  val(t,j)  fsi
ftype ;
```

On donne quatre cas de représentation, chacun d'eux correspondant à une propriété différente de t1. Les représentations sont faites à l'intérieur de modules de définition (def).

```
def  rep-table-1 (t1 : égalité(=),t2 : tyfo)
   représente  table (t1,t2)  par  liste ([t1,t2]) ;
      d(créert) : → liste ([t1,t2])
      d(ins) : t1 x t2 x liste ([t1,t2]) → liste ([t1,t2])  ou E
      d(trouver) : t1 x liste ([t1,t2]) → t2  ou E
      d(app) : t1 x liste ([t1,t2]) → bool
   privé
      dans : t1 x liste ([t1,t2]) → bool
      trouverliste : t1 x liste ([t1,t2]) → t2  ou  E
  eqns
   d(créert) = nil
   d(ins) (x,y,t) = si  dans (x,t)  alors ⇒ double-def(x)
                    sinon  cons ([x,y],t)  fsi
   d(trouver) (x,t) = trouverliste (x,t)
   d(app) (x,t) = dans (x,t)
   dans(x,nil) = faux
   dans(x,cons ([x1,y1],l)) = si  x = x1  alors  vrai
              sinon  dans (x,l)  fsi
   trouverliste(x,nil) ⇒ non-def(x)
   trouverliste(x,cons ([x1,y1],l)) = si  x = x1  alors  y1
                  sinon  trouverliste (x,l)  fsi
fdef ;
```

```
def  rep-table-2 (t1 : ordre-total (≤,=),t2 : tyfo)
   représente  table (t1,t2)  par [nat,tableau ([t1,t2])] ;
      d(créert) : → [nat,tableau ([t1,t2])]
      d(ins) : t1 x t2 x [nat,tableau ([t1,t2])] → [nat,tableau ([t1,t2])]  ou E
      d(trouver) : t1 x [nat,tableau ([t1,t2])] → t2  ou E
      d(app) : t1 x [nat,tableau ([t1,t2])] → bool
   privé
      décaler : tableau (t) x nat x nat → tableau (t)
      ind : t1 x tableau ([t1,t2]) x nat x nat → nat
   eqns
   d(créert) = [0,tabvide]
   d(ins) (x,y,[n,t]) = si  premier (val,(t,ind(x,t,1,n))) = x
                        alors⇒  double-def (x)
            sinon [succ(n),ranger(décaler(t,succ(ind(x,t,1,n)),n),
                        succ(ind(x,t,1,n)),[x,y])]  fsi
   d(trouver) (x,[n,t]) = si  premier (val(t,ind(x,t,1,n))) = x
                     alors  second (val(t,ind(x,t,1,n)))
                     sinon ⇒ non-def(x)  fsi
   d(app) (x,[n,t]) = premier (val(t,ind(x,t,1,n))) = x
   décaler(t,i,j) = si  le (j,i)
         alors  si  eq-nat (i,j)  alors  ranger (t,succ(j),val(t,j))
                sinon  t  fsi
         sinon  décaler (ranger(t,succ(j),val(t,j)),i,pred(j))  fsi
   ind(x,t,i,j) = si  le (j,i)  alors  si  premier (val(t,i))≤ x  alors  i
                                       sinon  pred(i)  fsi
                  sinon  si  premier (val(t,div2(i+j))) ≤ x
                     alors  si  premier (val(t,div2(i+j))) = x
                            alors  div2 (i+j)
                            sinon  ind(x,t,succ(div2(i+j)),j)  fsi
                     sinon  ind(x,t,i,pred(div2(i+j)))
                     fsi
                  fsi
fdef ;
```

Cette représentation fait un rangement ordonné et une recherche par dichotomie. L'opérateur ind donne l'indice k du tableau t (ou 0) tel que val(t,k) ≤ x ∧ x < val(t,succ(k)) (ou x < val(t,1)).

La troisième représentation utilise la fonction de dispersion et les opérateurs dans et trouverliste de rep-table-1.

```
def  rep-table-3 (t1 : égalité(=) & dispersion (h,n),t2 : tyfo)
   représente  table (t1,t2)  par  tableau (liste ([t1,t2])) ;
      d(créert) : → tableau (liste ([t1,t2]))
      d(ins) : t1 x t2 x tableau (liste ([t1,t2])) → tableau (liste ([t1,t2]))  ou E
      d(trouver) : t1 x tableau (liste ([t1,t2])) → t2  ou E
      d(app) : t1 x tableau (liste ([t1,t2])) → bool
   privé
      mettre-a-nil : tableau (liste (t)) x nat → tableau (liste (t))
 eqns
   d(créert) = mettre-a-nil (tabvide,n)
   d(ins) (x,y,t) = si  dans (x,val(t,h(x)))
                    alors ⇒double-def(x)
                    sinon  ranger(t,h(x),cons ([x,y],val(t,h(x))))  fsi
   d(trouver) (x,t) = trouverliste(x,val(t,h(x)))
   d(app) (x,y) = dans (x,val(t,h(x)))
   mettre-a-nil (t,0) = t
   mettre-a-nil (t,succ(i)) = mettre-a-nil (ranger(t,succ(i),nil),i)
fdef ;
```

La dernière représentation utilise le type t1 comme index d'un tableau de t2, de longueur n.

```
def  rep-table-4 (t1 : scalaire(h,n,g),t2 : tyfo)
    représente  table (t1,t2) par  tableau (t2) ;
    d(créert) : → tableau (t2)
    d(ins) : t1 x t2 x tableau (t2) → tableau (t2)  ou E
    d(trouver) : t1 x tableau (t2) → t2  ou E
    d(app) : t1 x tableau (t2) → bool
   privé
      def : nat x tableau (t2) → bool
```

```
eqns
d(créert) = tabvide
d(ins) (x,y,t) = si  def (h(x),t)
                 alors ⇒ double-def(x)
                 sinon  ranger(t,h(x),y)  fsi
d(trouver) (x,t) = si  def(h(x),t)
                   alors  val(t,h(x))  sinon ⇒ non-def(x)  fsi
d(app) (x,t) = def (h(x),t)
def(i,tabvide) = faux
def(i,ranger(t,j,a)) = si  eq-nat(i,j)  alors  vrai
                       sinon  def(i,t)  fsi
fdef ;
```

2.2.4. Sémantique des fonctions génériques

2.2.4.1. Spécification des fonctions par les constructeurs.

Dans la spécification algébrique, on a distingué (§ 2.2.2.2.) les constructeurs d'un type, qui sont les opérateurs avec lesquels on peut "construire" tous les objets. Ils sont une base pour les démonstrations par induction structurale. De ce fait, une façon de spécifier les autres opérateurs du type (discriminateurs) est de les décrire pour les objets mis sous forme normale, le résultat étant lui-même décrit en termes de constructeurs. Un exemple simple est celui du type ens(t) où les constructeurs sont créer et ins, et l'opération enl s'exprime par :

```
enl (x,créer) = créer
enl (x,ins(y,S)) = si  eq(x,y)  alors  enl(x,S)
                   sinon  ins(y,enl(x,S))  fsi
```

Bien que ces équations puissent être lues de manière parfaitement symétrique, l'idée est d'énumérer dans la partie gauche tous les cas possibles de construction du 2° paramètre (de type ens) et dans la partie droite, d'avoir une composition de constructeurs ou d'expressions "réduites" par rapport à la partie gauche. C'est la notion d'évaluation ou de règles de réécriture qui est sous-jacente à cette description ; on l'a utilisée systématiquement dans les exemples du §2.2.3.5.

GUTTAG et HORNING [GUT79] ont formalisé syntaxiquement de tels axiomes pour pouvoir prouver qu'un tel système est suffisamment complet et que les opérateurs discriminateurs sont convertibles aux constructeurs. Les avantages d'une telle spécification sont :

- méthode systématique (et vérifiable) d'écriture d'axiomes ;
- facilité de manipulation formelle ;
- implantation possible d'un système de réduction ;
- description constructive utilisable pour une implantation.

Les inconvénients sont, par le fait même, que la spécification a besoin d'être très détaillée, pour prévoir tous les cas, et que cela peut encombrer la description. De plus, on risque, déjà au niveau de la spécification, de faire des choix dans la spécification des fonctions, qu'il sera difficile d'oublier dans une implantation. On influence donc d'une certaine manière l'implantation, ce qui peut avoir pour conséquence que les réalisations sophistiquées seront très difficiles à vérifier.

On doit faire le rapprochement entre cette approche et la méthode de spécification de fonctions récursives dans le système Burstall-Darlington [DAR73] [BUR77a]. La différence essentielle entre les deux est évidemment que dans la spécification algébrique, on a des égalités, alors que dans le BD-système, ce sont des "implications". De ce fait, dans ce dernier cas, si on remplace une partie droite par une expression qui l'implique ("unfolding"), on ne modifie pas la sémantique de la fonction, par contre, cela peut être le cas si on la remplace par une expression gauche ("folding") qui est peut-être moins forte. Ici, on n'a que des égalités, donc tous les remplacements sont possibles, mais en même temps, on perd certaines possibilités de transformations où les remplacements ne sont pas égaux. Il faut remarquer quand même que l'égalité d'expressions récursives se ramène à une égalité d'objets infinis pour lesquels certaines précautions sont à prendre (cf. 2.2.4.3.).

Appliquons les principes du BD-système à la transformation de la fonction d(ins) de la définition de rep-table-4 (§ 2.2.3.5.3.) :

d(ins)(x,y,t) = si def(h(x),t) alors ⇒ double-def(x)
sinon ranger(t,h(x),y) fsi

- par cas de construction de t :

1) d(ins)(x,y,tabvide) = si def(h(x),tabvide) alors⇒ double-def(x)
sinon ranger(tabvide,h(x),y) fsi
= ranger(tabvide,h(x),y)

2) d(ins)(x,y,ranger(t,i,z))
= si def(h(x),ranger(t,i,z)) alors ⇒ double-def(x)
sinon ranger(ranger(t,i,z),h(x),y)) fsi
= si eq-nat(h(x),i) alors ⇒ double-def(x)
sinon si def(h(x),t) alors ⇒ double-def(x)
sinon ranger(ranger(t,i,z),h(x),y) fsi
fsi
= si eq-nat(h(x),i) alors ⇒ double-def(x)
sinon si def(h(x),t) alors ⇒ double-def(x)
sinon ranger(ranger(t,h(x),y),i,z) fsi
fsi
= si eq-nat(h(x),i) alors ⇒ double-def(x)
sinon ranger(
si def(h(x),t) alors ⇒ double-def(x)
sinon ranger(t,h(x),y)fsi),i,z) fsi
= si eq-nat(h(x),i) alors ⇒ double-def(x)
sinon ranger(d(ins)(x,y,t),i,z) fsi

On a supprimé dans ce cas la fonction intermédiaire def. Un programme de démonstration peut être basé sur le même système : soit à prouver, dans l'exemple du § 2.2.3.5.2. de traitement de matrices :

val(transpo(m),i,j) = val(m,j,i)

- par cas, on a :

1) val(transpo(créerm),i,j) = val(créerm,i,j)
⇒ indefm
val(créerm,j,i)⇒ indefm

A proprement parler on n'a d'égalité qu'en tant que équation d'erreur.

2) val(transpo(ranger(m,i1,j1,a)),i2,j2)
= val(ranger(transpo(m),j1,i1,a),i2,j2)
= si eq-nat(j1,i2) ∧ eq-nat(i1,j2) alors a
sinon val(transpo(m),i2,j2) fsi

= si eq-nat(j1,i2) ∧ eq-nat(i1,j2)
alors val(ranger(m,i1,j1,a),j2,i2)
sinon val(m,j2,i2) fsi
= val(ranger(m,i1,j1,a),j2,i2)

L'avant-dernière ligne, qui est l'utilisation de l'hypothèse d'induction dans le cas 2, peut être vue comme l'application d'un "folding" du théorème à démontrer. La règle d'application de ce "folding" d'induction semble être la même que celle de la validité des transformations du BD-système [KOT78] ; de même que dans l'exemple précédent, on ne pourrait pas déduire :

d(ins)(x,y,ranger(t,i,z)) = d(ins)(x,y,ranger(t,i,z))

comme une définition de d(ins) par application directe du "folding" (bien que l'égalité soit évidemment vérifiée), de même, on ne peut appliquer directement le théorème à démontrer.

2.2.4.2. Spécification des fonctions par leurs effets.

Cette manière de spécifier les fonctions est issue des idées de Parnas [PAR72]. Comme la plupart de ces travaux sont descriptifs, il est difficile de définir formellement ce genre de spécification. L'idée de base, qui est à rapprocher de la spécification des types abstraits de Guttag, est de définir syntaxiquement les opérations à résultat dans le type (G) et les opérateurs d'accès (A) mais de ne donner l'effet de G que d'une manière indirecte à l'aide des résultats de A. Dans cette méthode, on ne se ramène pas forcément à une forme normale. Par rapport à la méthode précédente, la spécification est moins détaillée, donc plus "abstraite" en quelque sorte, mais, si elle se prête bien aux vérifications, elle est moins constructive, et il n'existe pas de méthode pour s'assurer que les axiomes sont complets d'une certaine manière. Pour illustrer notre propos, donnons une partie de la spécification des matrices carrées d'ordre n, par l'effet de la fonction val. Une des caractéristiques des spécifications par les effets est l'utilisation des implications logiques : A ⊃ B où ⊃ est un méta-opérateur au même titre que le = de la spécification équationnelle. Quelques auteurs (en particulier [KRI78]) utilisent la méta-condition :

si A alors B fsi pour A ⊃ B,

mais cela peut créer des confusions avec l'opérateur si booléen :

si a alors b = c fsi

n'est pas équivalent à

si a alors b fsi = c

On définit syntaxiquement les axiomes A comme le plus petit ensemble satisfaisant (1) (2) (3) (4).

(1) si t1 et t2 sont des termes, t1 = t2 $\in$ AS, t1 $\neq$ t2 $\in$ AS

(2) si A1 $\in$ AS, A2 $\in$ AS alors A1 $\wedge$ A2 $\in$ AS

(3) si A1 $\in$ AS alors A1 $\in$ A

(4) si A1 $\in$ A, A2 $\in$ AS alors A1 $\supset$ A2 $\in$ A et A1 $\equiv$ A2 $\in$ A

Sous-ensemble des fonctions du type mc :

ranger : mc(t) x nat x nat x t $\rightarrow$ mc(t) ou E

val : mc(t) x nat x nat $\rightarrow$ t ou E

transpo : mc(t) $\rightarrow$ mc(t)

valide : mc(t) $\rightarrow$ bool

indefm : $\rightarrow E$

mc0 : $\rightarrow$ mc(t)

mc1 : $\rightarrow$ mc(t)

add : mc(t) x mc(t) $\rightarrow$ mc(t)

mult : mc(t) x mc(t) $\rightarrow$ mc(t)

index : nat x nat x mc(t) $\rightarrow$ bool

Spécification.

index (i,j,m) = le(1,i) $\wedge$ le(1,j) $\wedge$ le(i,ordre(m)) $\wedge$ le(j,ordre(m))

$\forall$ (i,j)(index(i,j,m) = vrai $\supset$ (val(m,i,j) $\neq$ indefm)) $\equiv$ valide(m) = vrai

valide(mc0) = vrai

valide(mc1) = vrai

valide(add(m1,m2)) = valide(m1) $\wedge$ valide(m2)
$\wedge$ eq-nat (ordre(m1),ordre(m2))

valide(mult(m1,m2)) = valide(m1) $\wedge$ valide(m2)
$\wedge$ eq-nat(ordre(m1),ordre(m2))

valide(transpo(m)) = valide(m)

index(i,j,m) = vrai $\supset$ val(ranger(m,i,j,a),i,j) = a

val(m,i1,j1) = a $\wedge$ (eq-nat(i1,i2) $\wedge$ eq-nat(j1,j2)) = faux
$\supset$ val(ranger(m,i2,j2,b),i1,j1) = a

index(i,j,m) = vrai $\supset$ ($\exists$a) val(m,i,j) = a v val(m,i,j) = indefm

index(i,j,m) = vrai $\supset$ val(transpo(m),i,j) = val(m,j,i)

index(i,j,mc0) = vrai $\supset$ val(mc0,i,j) = 0

(index(i,j,mc1) $\wedge \neg$ eq-nat(i,j)) = vrai $\supset$ val(mc1,i,j) = 0

(index(i,j,mc1) $\wedge$ eq-nat(i,j)) = vrai $\supset$ val(mc1,i,j) = 1

index(i,j,m1) = vrai $\supset$ val(add(m1,m2),i,j)
= val(m1,i,j) + val(m2,i,j)

index(i,j,m1) = vrai $\supset$ val(mult(m1,m2),i,j)
= reduc[*] (comp[+] (ligne(m1,i),col(m2,j)))

2.2.4.3. Fonctions de types abstraits

2.2.4.3.1. Fonctions simples

Dans un type abstrait, et a fortiori une propriété, on a considéré les opérateurs (ou symboles fonctionnels) comme des constructeurs d'objets. Les domaines considérés sont des domaines de Herbrand (typés), et les équations engendrent des classes d'équivalence entre les termes. La façon de donner une signification à ces symboles fonctionnels est de fournir une interprétation qui soit une application fonctionnelle. On peut aussi les interpréter par des programmes et le passage d'un programme à une application se fera par la sémantique dénotationnelle. Nous allons simplement évoquer ici quelques idées qui permettent de mieux comprendre la notion de programme générique et sa justification théorique.
Les références sur le sujet peuvent être trouvées principalement dans [NIV75] [COU78] [ADJ77b] [ADJ78c]. Nous nous en tiendrons principalement à quelques exemples.

Soit le type nat décrit algébriquement et d'autre part le domaine des entiers : w. Donnons une interprétation des opérateurs de nat sous forme de lambda-expressions :

0 : nat est interprété par	0 : w
succ : nat → nat	S : λx.x+1 : w → w
pred : nat → nat	P : λx.x$\dot{-}$1 : w → w
eq : nat x nat → bool	E : λxy.x = y : w x w → bool

Lorsqu'on écrit une expression à l'aide des symboles fonctionnels de nat, on sait qu'il existe un prolongement unique de cette expression dans son domaine d'interprétation (§ 2.2.1.7.). Par exemple, le terme :

succ(si eq(x,0) alors succ(x) sinon y fsi)

est interprétée homomorphiquement par :

λxy.(si x = 0 alors x + 1 sinon y fsi) + 1
= λxy.(si x = 0 alors x + 2 sinon y + 1 fsi)

La sémantique d'un opérateur défini par une expression (opérateur dérivé simple) est obtenue de la même manière.

2.2.4.3.2. Fonctions récursives

On considère que les opérateurs qui définissent un type abstrait sont divisés en constructeurs (c) et discriminateurs (§2.2.2.2.6.). De même les équations sont

divisées en équations sur les constructeurs (eq-c) et en équations sur les discriminateurs (eq-d). On n'étudie pas l'effet des équations eq-c.

Les équations eq-d sont de la forme :

$f(c(\alpha)) = \beta$

où f est l'opérateur discriminateur que l'on étudie, $c(\alpha)$ est une expression où n'apparaissent que des constructeurs, β une expression avec si alors sinon fsi, constructeurs et f. (On peut, comme pour les définitions de procédures mutuellement récursives, étendre la définition à un ensemble d'opérateurs f1, f2, ...fn). [GUT78] montre que ces restrictions ne diminuent pas le pouvoir d'expression des systèmes d'équations si on autorise l'utilisation des opérateurs privé(s). Les constructeurs d'un type engendrent un domaine C de termes, et on considère les opérateurs discriminateurs comme des applications $C^n \rightarrow C$, pour des C associés aux types des paramètres et du résultat.

Exemple : le type nat possède un domaine

C_{nat} : $\{0, succ(0), succ^2(0)...\}$

Les opérateurs pred : nat → nat

plus : nat x nat → nat

représentent des applications : $C_{nat} \rightarrow C_{nat}$

$C_{nat} \times C_{nat} \rightarrow C_{nat}$

La sémantique d'un opérateur discriminateur peut être représentée (dénotée) d'une manière classique à l'aide de la relation entre les paramètres et le résultat. On note $\mathcal{D}$ le domaine de cette relation et D(f) la sémantique de f ($D(f) \in \mathcal{D}$). D'autre part, f est définie par ses équations eq-f. Le système eq-f peut être considéré comme une transformation $T_f : \mathcal{D} \rightarrow \mathcal{D}$. Cette transformation, quoique assez complexe à définir, généralise la notion de fonctionnelle habituelle. Etudions d'abord la notion d'évaluation dans un domaine C à l'aide des sémantiques $D(f_i)$ des opérateurs f_i.

Evaluation

L'évaluation d'une expression E consiste à calculer le ou les termes $\in C$ s'ils existent, équivalents à E. Soient $f_1, f_2...$ définis par leur sémantique $D(f_1), D(f_2)...$, l'évaluation est : eval : E → C

telle que : eval(c) = c ssi $c \in C$

eval(f(c)) = c' ssi $\langle c, c' \rangle \in D(f)$

eval(si cond alors α_1 sinon α_2 fsi)

$$= \begin{cases} eval(\alpha_1) & \text{ssi } eval(cond) = vrai \\ eval(\alpha_2) & \text{ssi } eval(cond) = faux \end{cases}$$

$eval(f_1(\alpha)) = eval(f_1(eval(\alpha)))$

Exemple : soit $D(f) = \{<0, succ^2(0)>, <succ(0), 0>, <succ^2(0), succ^3(0)>\}$

eval(si eq(f(succ(0)),0) alors f(f(0)) sinon succ(0))

⇒ eval(eq(f(succ(0)),0))

= eval(eq(eval(f(succ(0))),0))

= eval(eq(0,0))

= vrai

d'où l'expression

= eval(f(eval(f(0))))

= eval(f($succ^2$(0)))

= $succ^3$(0)

Définition de T_f

Soient {v} l'ensemble des variables d'une équation et $\{\alpha_v\}$ l'ensemble des substitutions qui affectent à chaque variable x un élément du domaine des constructeurs de son type : C_x. Si D_0 est une définition sémantique de f, et $f(\alpha) = \beta$ une équation, alors : $\forall\, \sigma \in \{\alpha_v\}$ tel que eval($\beta\sigma$) existe, on a :

$$<\alpha\sigma, eval(\beta\sigma)> \in T_f(D_0)$$

Exemple : soit la définition :

(D_2) f(0) = 0

f(succ(x)) = succ(succ(f(x)))

si l'on donne :

$D_0(f) = \{<0, succ(0)>, <succ^2(0), succ(0)>\}$

$T_f(D_0) = \{<0,0>, <succ(0), succ^3(0)>, <succ^3(0), succ^3(0)>\}$

$T_f^2(D_0) = \{<0,0>, <succ(0), succ^2(0)>, <succ^2(0), succ^5(0)>, <succ^4(0), succ^5(0)>\}$

Définition par point fixe de D(f).

On définit D(f) à partir de la transformation T_f déduite des équations eq-f par :

$$D(f) = \cap \{D \mid T_f(D) = D\}$$

D(f) est le plus petit point fixe de $T_f : \mathcal{D} \to \mathcal{D}$.

Comme T_f est monotone : $D_1 \subseteq D_2 \supset T_f(D_1) \subseteq T_f(D_2)$

et continue : $T_f(\cup D_i) = \cup T_f(D_i)$

les théorèmes classiques de Tarski et Kleene sur les plus petits points fixes s'appliquent :

$$1 - D(f) = \cap \{D \mid T_f(D) \subseteq D\}$$

$$2 - D(f) = \bigcup_{n=0}^{\infty} T_f^n(\emptyset)$$

avec la propriété : $T_f(D(f)) = D(f)$

Exemple : D_2 est la limite des itérés :

$T^1(\emptyset) = \{<0,0>\}$

$T^1 \cup T^2 = \{<0,0>, <succ(0), succ^2(0)>\}$

$T^1 \cup T^2 \cup T^3 = \{<0,0>, <succ(0), succ^2(0)>, <succ^2(0), succ^4(0)>\}$

Propositions (§2.2.2.2.6.)

1 - L'axiomatisation eq-f est cohérente (anglais : "consistent")
ssi $D(f)$ définit le graphe d'une fonction.

2 - l'axiomatisation eq-f est suffisamment complète
ssi $D(f)$ définit le graphe d'une fonction totale.

Les preuves se déduisent immédiatement des définitions. On s'intéresse maintenant aux interprétations fonctionnelles des opérateurs dont l'axiomatisation est cohérente.

Interprétation fonctionnelle d'un domaine C.

L'interprétation fonctionnelle d'un domaine C dans un domaine A est la donnée :

- $\forall$ c constructeur de C : $c : C^n \to C$

1. d'une fonction : $I(c) : A^n \to A$ (interprétation des constructeurs) ;
2. d'un prédicat : $P_c : A \to bool$ tel que :
$P_c(I(\alpha)) = vrai$ si $\alpha \in C$ est de la forme $c(\beta)$
3. de fonctions inverses : $I(c^{-1})$
$P_c(I(\alpha)) = vrai \supset \alpha = c(\beta) \wedge I(\beta) = I(c^{-1})\ I(\alpha)$

Note : si les fonctions inverses sont multivaluées, $I(c^{-1})$ est défini par une famille de fonctions.

Exemple : soit nat : {0,succ} ; w est une interprétation fonctionnelle de nat avec :

$I(0) = 0$

$I(succ) = \lambda x.x+1$

$P_0(x) = \lambda x.x = 0$

$P_{succ}(x) = \lambda x.x>0$

$I(succ^{-1}) = \lambda x.x-1$

Interprétation d'une relation de D(f)

Une relation $<\alpha,\beta> \in D(f)$ est interprétée naturellement dans $A^n \times A$ par :

$<I(\alpha), I(\beta)>$

D'où : $I(D(f)) = \{<I(c_1),I(c_2)>\}$ $\forall <c1, c2> \in D(f)$

Interprétation d'une axiomatisation eq-f

Soit une équation : $f(\alpha) = \beta \in$ eq-f. Cette équation est transformée dans l'interprétation fonctionnelle en

$I(f)(x) = \phi(\alpha,\beta) \in$ eq-I(f).

On note {x} les variables de eq-I(f)

{v} les variables de eq-f.

On remarque que puisque eq-f est cohérente, D(f) est le graphe d'une fonction et cela entraîne que toutes les variables {v} apparaissent dans α. On calcule donc, à l'aide des interprétations inverses, v en fonction de x à l'aide de α, noté $v = I^{-1}(\alpha)$.

Alors, $\phi(\alpha,\beta) =$ si $\cup P_{c_i}(I^{-1}(\alpha))$ alors $I(\beta)]_{v}^{I^{-1}(\alpha)}$

Explications :

- Le terme $\cup P_{c_i}$ impose les conditions sur les variables x pour qu'il soit possible d'avoir un α tel que $I(\alpha) = x$ pour toute valeur de x dans A^n.

D'autre part, le deuxième terme est l'interprétation de β dans laquelle les variables v sont substituées par $I^{-1}(\alpha)$. On obtient donc une équation de la forme :

$I(f)(x) = F(x)$ si f n'apparaît pas dans β

$I(f)(x) = F(I(f),x)$ si f apparaît dans β.

Exemple : Interprétation de l'axiomatisation D_2.

$$I(f)(0) = 0$$

$$I(f)(y) = \underline{si}\ \ P_0 \vee P_{succ}(I^{-1}(succ(x)))$$

$$\underline{alors}\ \ I(succ(succ(f(x))))]_{x}^{I^{-1}(succ(x))}$$

Calcul de $I^{-1}(succ(x))$

$$y = I^{-1}(succ(x)) \Rightarrow x = I(succ^{-1})\ (y)$$

$$\Rightarrow x = y - 1$$

d'où : $I(f)(y) = \underline{si}\ \ y-1 \geq 0\ \ \underline{alors}\ \ I(f)(y-1) + 2.$

Le système de I(f) est :

$$I(f)(0) = 0$$

$$I(f)(y) = \underline{si}\ \ y \geq 1\ \ \underline{alors}\ \ 2 + I(f)\ (y-1)$$

Sémantique d'une interprétation

Puisqu'on est dans un domaine fonctionnel, le système d'équations récursives eq-I(f) détermine une transformation de $A^n \times A \rightarrow A^n \times A$ (relation d'entrée-sortie) ; soit $T_{I(f)}$ cette transformation. Là encore, on détermine la sémantique de I(f) par le plus petit point fixe de la transformation.

$$D(I(f)) = \cap \{D \mid T_{I(f)}(D) = D\}$$

Théorème ([ADJ77b])

$$I(D(f)) = D(I(f))$$

Preuve : Elle se base sur le fait que D(f) est initiale et que l'interprétation I admet un prolongement I(D(f)) qui est un morphisme continu.

2.2.4.3.3. Fonctions génériques

On aimerait étendre ce mécanisme d'équivalence d'interprétations aux fonctions génériques qui admettent des objets de types formels en paramètres. Cela signifie : si une fonction f est définie avec les types t formels (caractérisés par des propriétés), et θ est une substitution de t par des types effectifs (τ) qui satisfont les propriétés, (on note $I(\theta)$ la substitution : $t \leftarrow I(\tau)$).

Les questions que l'on se pose, sont :

1 - comment comparer D(f), θD(f), D(θf) ?

2 - quelles relations a-t-on entre :

I(θD(f)), I(D(θf)), D(I(θf)), I(θ) I(D(f)) ?

Autrement dit, la sémantique des interprétations des fonctions génériques dépend-elle des liaisons des types effectifs ? On aimerait répondre par la négative, ce qui permettrait de définir une sémantique "générique" des fonctions génériques et donc des programmes génériques.
Il semble que les classes d'interprétation ([COU78b]) permettent de répondre à ces questions, mais nous laissons ce travail très technique pour plus tard.

2.2.4.4. Actions répétitives

Les actions répétitives (cf. § 1.3.) ont des paramètres de type séquence temporelle ; du point de vue de la spécification algébrique, les séquences temporelles sont assimilées aux séquences abstraites. [BEL79] utilise le terme de "listes logiques". On va donc définir d'abord le type générique séquence et montrer son utilisation dans la spécification des itérateurs.

2.2.4.4.1. Type séquence

```
type  seq (t : tyfo) ;
      nil : → seq (t)
       ~ : t x seq (t) → seq (t)
      premier : seq(t) → t  ou E
      reste : seq(t) → seq(t)
      null : seq(t) → bool
      eof : → E
 . eqns  premier (nil) ⇒ eof
      premier (x ~ s) = x
      reste (nil) = nil
      reste (x ~ s) = s
      null (nil) = vrai
      null (x ~ s) = faux
 ftype ;
```

On dispose aussi des propriétés isoseq et cvseq qui s'appliquent aux opérateurs

sur les séquences :

```
prop  cvseq(f,g)  sur  t1,t2 : tyfo ;
        f : t1 → seq(t2)
        g : seq(t2) → t1
    eqns  g (f(x)) = x
fprop ;

prop  isoseq (f,g)  sur  t1,t2 : cvseq(f,g) ;
    eqns      f(g(x)) = x
fprop ;
```

Algébriquement, un itérateur est un opérateur avec résultat ou paramètres de type seq(t). Sa spécification fait donc intervenir les objets séquence.

2.2.4.4.2. Spécification algébrique des itérateurs.

Prenons le type ens(t) (cf. § 2.2.3.3.3.) dans lequel on définit un émetteur qui produit la séquence des éléments de l'ensemble, et un récepteur qui, étant donné une séquence, construit un ensemble :

```
type  ens (t : égalité) : cvseq (elem-de,faire-ens) ;
    ...
      elem-de : ens(t) → seq(t)
      faire-ens : seq(t) → ens(t)
   eqns
       elem-de(créer) = nil
       elem-de(ins(x,s)) = x ~ elem-de (enl(x,s))
       faire-ens(nil) = créer
       faire-ens(x ~ l) = ins(x,faire-ens(l))
ftype ;
```

On a donné au § 1.3.5. des programmes itérateurs de fusion et d'éclatement ; en voilà la spécification algébrique :

Soit le type t : ordre-total (≤) ;

fusion : seq(t) x seq(t) → seq(t)

éclatement : seq(t) → [seq(t),seq(t)]

privé éclat1 : seq(t) x t x bool → [seq(t),seq(t)]

où [ ] est le constructeur de tuple.

```
eqns
   fusion(x~S1,y~S2) = si x≤y
                       alors  x ~ fusion(S1,y~S2)
                       sinon  y ~ fusion(x ~ S1,S2) fsi
   fusion (nil,y ~ S2) = y ~ S2
   fusion (x ~ S1,nil) = x ~ S1
   fusion (nil,nil) = nil

   éclatement (nil) = [nil,nil]
   éclatement (x ~ S) = [x ~ S1,S2] où [S1,S2] = éclat1(S,x,vrai)
   éclat1 (y ~ S,x,b) =
        si  b  alors  si  y ≤ x
            alors  [y ~ S1,S2]  où [S1,S2] = éclat1 (S,y,b)
            sinon [S1,y ~ S2]  où [S1,S2] = éclat1 (S,y,¬b) fsi
        sinon  si  y ≤ x
            alors  [S1,y ~ S2]  où [S1,S2] = éclat1 (S,y,b)
            sinon  [y ~ S1,s2]  où [S1,S2] = éclat1 (S,y,¬ b) fsi
        fsi

   éclat1 (nil,x,b) = [nil,nil]
```

Note : avec le résultat tuple, on est obligé de nommer les composants (ici à l'aide de la clause "où").

2.2.4.4.3. Dérivation de programmes itératifs

Il semble à première vue que la spécification abstraite des itérateurs soit assez éloignée de la programmation, à cause notamment des constructeurs explicites de séquence. On va montrer dans ce paragraphe qu'elle est au contraire une base à partir de laquelle on peut, quasi automatiquement construire des procédures d'itération, en supprimant toute notion explicite de séquence.

La transformation se base sur la contraction syntaxique suivante : si une opération abstraite avec un paramètre seq (τ) devient un récepteur, ce paramètre est traduit en seqt τ , où le type du paramètre est celui des éléments de la séquence abstraite. De même, un résultat seq(τ) (pour un émetteur) devient seqt τ ou (eof). On va voir pourquoi le "eof" apparaît dans le résultat. En résumé :

```
      em : paramètres → seq(t)
devient  proc  em(paramètres) → seqt  t  ou  (eof) ;
         rec : seq(t) → résultat
devient  proc  rec(seqt t) → résultat ;
```

Le premier pas consiste à traduire la spécification sous forme d'opérateur dérivé. Prenons l'exemple de l'émetteur-récepteur "fusion" :

```
fusion (l1,l2) = si  premier(l1) ≤ premier(l2)
        alors  premier(l1) ~ fusion (reste(l1),l2)
        sinon  premier(l2) ~ fusion (l1,reste(l2)) fsi
        ou  qd  premier(l1) ⇒ eof ∧ premier(l2)⇒ eof  alors  eof
            | premier(l1) ⇒ eof ∧¬ premier(l2) ⇒ eof
                        alors  premier(l2) ~ reste(l2)
            |¬premier(l1) ⇒ eof ∧ premier(l2) ⇒ eof
                        alors  premier(l1) ~ reste(l1)  fq
```

Ensuite, on passe à la forme itérative en englobant les instructions (sauf les exceptions) par faire ... ffaire et on remplace les paramètres par x et y où :

x = premier (l1)

y = premier (l2)

De plus :

- un résultat doit être de la forme a ~ α qui devient :

  "donner (a) ; traduction de α"

- α doit être de la forme : I (paramètres), où l'itérateur I a le même type de résultat que l'itérateur que l'on définit. La traduction de α est : - calcul des paramètres (par invocation de suivant, si le paramètre est de la forme reste(l)) ;

  - appel de l'itérateur I si celui-ci est différent de celui que l'on définit (sinon retour de boucle).

Dans le programme de fusion, les résultats du cas normal correspondent à un appel au même itérateur ; par contre, dans les exceptions, il faut introduire un appel intermédiaire d'itérateur. Cet itérateur est l'itérateur d'identité :

```
ident : seq(t) → seq(t)
eqns
ident (l) = l
```

ou bien, en décomposant le corps :

```
ident (l) = premier(l) ~ ident (reste(l))
        ou  qd  eof  alors  eof  fq
```

Sous forme algorithmique :

```
proc ident (seqt u : t) → seqt t ou (eof) ;
   qd eof alors eof fq
début faire donner (u) ; suivant (u) ffaire
fin ;
```

L'itérateur fusion devient :

```
proc fusion (seqt x,y : t) → seqt t ou (eof) ;
   qd x ⇒eof ∧ y ⇒ eof alors eof
    | x ⇒ eof ∧¬y ⇒ eof
                alors donner (y) ; suivant (y) ; ident (y)
    |¬x ⇒ eof ∧ y=> eof
              alors donner (x) ; suivant (x) ; ident (x) fq ;
   début faire si x ≤ y
      |       alors donner (x) ; suivant (x)
      |       sinon donner (y) ; suivant (y) fsi
      |    ffaire
   fin ;
```

Par inclusion textuelle de ident et optimisation, on obtient le programme du § 1.3.5.

Autres cas :

1 - L'itérateur "éclatement" peut être développé de la même manière ; dans la procédure du § 1.3.5., on a fait aussi l'inclusion de éclat1 dans le corps de éclatement.

2 - l'itérateur elem-de peut être dérivé en utilisant les opérateurs auxiliaires :

prem-ens : ens(t) → t

reste-ens : ens(t) → ens(t)

tels que : prem-ens (ins(x,s)) = x

reste-ens (ins(x,s)) = enl(x,s)

L'itérateur elem-de est alors :

```
proc elem-de (S : ens(t)) → seqt t ou E ;
   début tantque S ≠ créer
         faire donner (prem-ens(S)) ;
              | S := reste-ens (S)
           ffaire ;
              eof
   fin ;
```

3 - L'émetteur faire-ens est a priori plus compliqué à dériver, puisque la séquence temporelle d'entrée est utilisée pour construire l'ensemble résultat dans l'ordre inverse d'arrivée. Les solutions sont celles de la transformation de programmes récursifs en itératifs :

- utilisation d'une pile (qui permet de remplacer la séquence temporelle en séquence spaciale, puis d'inverser cette séquence) ;

- garder la même séquence si les opérations ne dépendent pas de l'ordre des éléments dans la séquence.

Ici, on est dans le deuxième cas, puisque :

ins (x,ins(y,S)) = ins(y,ins(x,S))

d'où :
```
proc  faire-ens (seqt  x : t) → ens(t) ;
          S : ens(t),
          qd  eof  alors  retourner (S)  fq ;
     début  S := créer ;
        |   faire  S := ins(x,S) ;
        |      |     suivant (x)
        |   ffaire
     fin ;
```

2.2.5. Etude de quelques propriétés

On a vu, tout au long de ces chapitres, l'importance des propriétés dans la programmation générique. On revient ici sur trois d'entre elles qui sont fondamentales en programmation et pour lesquelles il y a encore beaucoup à faire ; il s'agit de la propriété "scalaire", de l'égalité et de l'affectation.

2.2.5.1. Propriété "scalaire".

Le nom de propriété scalaire provient du langage PASCAL ; (voir aussi "index" en [TEN77]). Nous l'utilisons ici pour tous les types dont les éléments peuvent servir à indexer des tableaux. On peut donner plusieurs définitions de cette propriété ; la plus "mathématique" est de donner la bijection entre l'ensemble des éléments du type et un intervalle d'entiers ; c'est ce qui est fait au §2.2.3.5.3. De plus, l'ordre sur les éléments est hérité automatiquement de l'ordre sur les entiers. Cette formulation a aussi l'avantage de fournir un codage des objets du type.

La deuxième formulation est celle qui comporte les fonctions succ, pred, premier, dernier, etc... Un type énuméré est un exemple d'une telle formulation, où les fonctions peuvent être déduites sous forme de table. Soit le type énuméré :

type couleur = (jaune, bleu, rouge) ;
les fonctions associées sont :

premier (de couleur) = jaune

dernier (de couleur) = rouge

	jaune	bleu	rouge
succ	bleu	rouge	—
pred	—	jaune	bleu

ordre : jaune < bleu ∧ bleu < rouge

Les relations entre les deux formulations sont données au § 1.2.4. à l'aide du mécanisme des extensions.

2.2.5.2. Propriété égalité

On a utilisé abondamment cette propriété dans les exemples. La spécification est celle d'une relation d'équivalence, et l'on doit, lorsque l'égalité est définie dans un type t, y ajouter la propriété de substitutivité :

- pour toute fonction f qui admet un paramètre de type t, si a, b : t et a = b

alors f(...a...) = f(...b...)

dans le type résultat de f (les autres paramètres étant constants).

La question que l'on peut se poser est :
pourquoi l'égalité n'est-elle pas définie automatiquement pour tous les types, puisque les axiomes fournissent précisément une équivalence entre les termes ? Autrement dit, pourquoi ne pas poser, pour tout type t :

eq-t : t x t bool

eq-t (x,y) = si x ≡ y alors vrai .
sinon faux fsi

où ≡ signifie que x et y sont congrus dans t. On voit tout de suite que l'on utilise comme prédicat booléen dans l'équation, le méta-prédicat d'équivalence. Or, cette séparation entre le système formel et le type bool est essentielle pour

éviter les paradoxes puisqu'on pourrait écrire :

x = si ¬(x ≡ y) alors y fsi

On en déduit alors que l'égalité est exprimée par un opérateur qui doit être défini pour chaque type qui possède cette propriété et qu'il doit être total sur ses paramètres. De plus, à cause des contraintes de protection de bool (§ 2.2.2.2.3.), on doit avoir :

$x \equiv_t y \supset$ eq-t (x,y) ≡ vrai

Si on a de plus :

$x \equiv_t y$ ssi eq-t (x,y) ≡ vrai

alors l'opérateur eq-t est une implantation de l'équivalence entre les termes. La synthèse automatique de l'opérateur eq-t à partir des équations, présente les mêmes difficultés dans la détermination des valeurs "faux" que la synthèse du prédicat ¬P d'après la définition de P en logique du premier ordre [ROU72]. En conclusion, la méthode la plus simple que l'on propose est la détermination par le programmeur de l'opérateur d'égalité s'il désire que le type possède la propriété d'égalité, à charge pour le système de démonstration de vérifier les axiomes de l'égalité sur la définition ainsi donnée.

2.2.5.3. Propriété affectation.

Dans un système de programmation algébrique (à la Guttag, CLEAR, OBJ), on n'a pas besoin de l'opérateur d'affectation puisqu'on n'utilise que des expressions et des fonctions. Cependant, si on veut descendre au niveau de programmation des algorithmes, il est nécessaire d'introduire cet opérateur. La déclaration syntaxique est : proc := (mod cible : t, source : t) ;
La spécification la plus couramment admise est [RED79] : après l'affectation x := y où x est une variable et y une expression, si y' est la valeur de y avant l'affectation, on a :

(AFF) valeur (x) = y'

Là encore, on fait référence à l'égalité qui est celle fournie par le programmeur avec le type. Les problèmes proviennent de deux choses :

1 - Après une affectation, on ne connaît l'effet véritable de l'instruction que si l'on connaît exactement les relations de désignation et de repérage (au sens Algol 68), entre les autres variables du programme et la variable x. Comme certaines de ces relations sont dynamiques, on comprend la difficulté de la spécification de l'affectation. Par contre, si l'affectation est

uniquement un moyen de construire et de modifier des relations de désignation, on peut en tenir compte dans un système de calcul statique approché ou par un calcul dynamique des objets construits [BER76].

2 - Puisque, lorsqu'on est au niveau de la programmation, il peut y avoir plusieurs définitions de l'égalité (dépendant en particulier de l'implantation : cf. dans CLU les fonctions equal, similar1, similar), il devrait y avoir, pour respecter la règle (AFF) plusieurs sortes d'affectation possibles. On peut relier d'ailleurs ces sortes d'affectation aux modes de passage de paramètres : copie, partage, indirection, etc... On obtient donc, pour les types abstraits (qui ne doivent pas dépendre de la représentation), une quasi-impossibilité de spécifier proprement l'affectation, sauf si l'on accepte qu'il n'y ait aucune synonymie de désignation entre les objets ("non-aliasing") et que les affectations soient des copies de valeur. Mais une telle sémantique n'est pas concevable actuellement pour un langage de programmation.

2.2.6. Equivalence des types abstraits et des clauses de Horn.

Pour terminer ce deuxième chapitre consacré essentiellement aux types algébriques, on va comparer la sémantique des types abstraits avec celle des clauses de Horn en logique. Il est nécessaire d'abord de rappeler ce que sont les clauses de Horn (C.H.).

2.2.6.1. Syntaxe et sémantique des C.H. [KOW74] [EMD76].

Du point de vue syntaxique, un programme est un ensemble fini de C.H. On notera un tel programme Γ .
Une clause est une disjonction de littéraux : $L_1 \vee L_2 \ldots \vee L_n$;
un littéral L est une formule atomique $P(t_1,\ldots,t_k)$ où P est un symbole de prédicat à k paramètres et $t_1,\ldots t_k$ sont des termes. Un littéral peut être positif (affirmation du prédicat) ou négatif : $\overline{L}$ (négation du prédicat).
Les clauses de Horn sont des clauses de la forme suivante :

$A \vee \bar{B}_1 \vee \bar{B}_2 \ldots \vee \bar{B}_n$	assertion conditionnelle
A	axiome
$\bar{B}_1 \vee \bar{B}_2 \vee \ldots \bar{B}_n$	but à démontrer

Suivant PROLOG, on écrira :

$A \leftarrow B_1 \quad B_2 \ldots B_n$

$A \leftarrow$

$\leftarrow B_1 \quad B_2 \ldots B_n$

Un terme est soit une variable, soit une expression $f\ (t_1, \ldots t_m)$ où f est un symbole fonctionnel à m paramètres et $t_1,\ldots t_m$ sont des termes. Les constantes sont des symboles fonctionnels sans paramètres. Les variables dans une clause sont quantifiées universellement.

On notera (sauf indications particulières) :

- x,y,... : variables
- a,b,... : constantes (opérateurs 0-aire)
- f,g,... : symboles fonctionnels (opérateurs n-aires)
- t,u,... : termes
- p,q,P,Q : symboles de prédicats (opérateurs booléens).

Exemple : si les termes 0, s(0), s(s(0)), ... représentent les entiers, on peut spécifier la multiplication (m), l'addition (a) et la factorielle (f) par

l'ensemble des clauses :

(Γ_f) (1) f (0,s(0)) ← {fact(0) = 1}

(2) f (s(x),y) ← f(x,z) m(s(x),z,y)

qui peut se lire : si fact (x) = z ∧ (x+1) * z = y alors fact(x+1) = y ;
de même pour m :

(3) m (0,y,0) ←

(4) m (s(x),y,z) ← m (x,y,w) a (y,w,z)

avec (5) a (0,y,y) ←

(6) a (s(x),y,s(z)) ← a (x,y,z)

où m (x,y,z) est mis pour x * y = z
et a (x,y,z) pour x + y = z

Avant de définir un modèle de sémantique pour un ensemble de clauses (Γ), donnons la méthode de déduction opérationnelle sur l'exemple. La preuve d'un but : $\leftarrow B_1 \ldots B_n$ est équivalente à trouver les valeurs des variables (s'il y en a) telles que la négation de la clause $(\bar{B}_1 \vee \bar{B}_2 \ldots \vee \bar{B}_n)$ est en contradiction avec le système Γ. Pratiquement, la découverte de la contradiction se fait par déduction de la clause vide : $\square$.

Exemple : (suite)
Prouver le but : ← f(s(0),x)

(2) $x = y_1$: ← $f(0,z_1)$ m $(s(0),z_1,y_1)$

(1) $z_1 = s(0)$: ← $m(s(0),s(0),y_1)$

(4) $y_1 = z_2$: ← $m(0,s(0),w_1)$ $a(s(0),w_1,z_2)$

(3) $w_1 = 0$: ← $a(s(0),0,z_2)$

(6) $z_2 = s(z_3)$: ← $a(0,0,z_3)$

(5) $z_3 = 0$: $\square$

D'où par composition : x = s(0) et $\overline{f(s(0),s(0))}$ est contradictoire avec le système (Γ_f) donc f(s(0),s(0)) est un théorème du système Γ_f.

Voyons précisément comment on définit l'ensemble des théorèmes qui peuvent être déduits d'un ensemble Γ de C.H. Pour cela, on ne rappelle que la notion de "model-theoretic semantics" de [EMD76] qui sera utilisée dans les paragraphes suivants. Cette sémantique se base sur les interprétations de Herbrand d'un ensemble de clauses.

Rappelons quelques définitions :

- l'univers de Herbrand H est l'ensemble des termes sans variables ;
- la base de Herbrand $\hat{H}$ d'un ensemble de clauses Γ est l'ensemble des formules $P(t_1,\ldots t_m)$ où P est un symbole de prédicat et $t_1,\ldots t_m$ des termes de H.
- une interprétation de Herbrand I est un sous-ensemble de $\hat{H}$.

Pour un ensemble de clauses Γ, on dit que :

1 - une formule atomique P(...) est <u>vraie</u> dans I ssi
$P(\ldots) \in I$; $\bar{P}(\ldots)$ est vraie ssi $P(\ldots) \notin I$.

2 - une clause sans variable L1 v L2 v ... Ln est vraie si l'un des L_i est vrai.

3 - une clause générale C est vraie ssi chaque instance de substitution $C\sigma$ est vraie.

4 - un ensemble de clauses Γ est vrai dans I ssi chaque clause est vraie dans I.

Si Γ est vrai dans I, on dit que I est un modèle de Herbrand de Γ . On note $M(\Gamma)$ l'ensemble des modèles de Herbrand de Γ , alors $\cap M(\Gamma)$ est elle-même un modèle de Herbrand. Un prédicat est un théorème de Γ

$$\Gamma \models P(t_1,\ldots t_n) \quad \text{ssi} \quad P(t_1\ldots t_n) \in \cap M(\Gamma).$$

On dit aussi que la dénotation de P est :

$$D(P) = \{(t_1\ldots t_n) \mid P(t_1\ldots t_n) \in \cap M(\Gamma)\}$$

2.2.6.2. Inclusion de C.H. dans TA (types algébriques).

On prend la définition des types au sens de ADJ (§ 2.2.2.1.).

2.2.6.2.1. Définition

A tout système Γ de C.H. on peut faire correspondre d'une façon unique un type τ_Γ par la construction suivante :

$$\tau_\Gamma = \langle \{\text{terme, bool}\}, \Sigma , E\rangle$$

tel que : $\forall$ a : constante dans Γ : $\tilde{a} : \rightarrow \text{terme} \in \Sigma$

$\forall$ f : n-aire : $\tilde{f} : \text{terme}^n \rightarrow \text{terme} \in \Sigma$

$\forall$ symbole de prédicat n-aire P : $\tilde{p} : \text{terme}^n \rightarrow \text{bool} \in \Sigma$

Les équations E sont :

si $P(t_1 \ldots t_n) \leftarrow \in \Gamma$ alors $\tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) = \text{vrai} \in E$

si $P(t_1 \ldots t_n) \leftarrow Q_1(\ldots)\ Q_2(\ldots) \ldots Q_k(\ldots) \in \Gamma$

alors $\tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) =$ si $\tilde{q_1}(\ldots) \wedge \ldots \wedge \tilde{q_k}(\ldots)$ alors vrai fsi $\in E$

On complète τ_Γ par les équations booléennes classiques (§ 2.2.2.1.5.).

Par exemple Γ_f devient :

Σ : 0 : → terme

s : terme → terme

f : terme x terme → bool

m,a : terme x terme x terme → bool

E : f(0,s(0)) = vrai

f(s(x),y) = si f(x,z) ∧ m(s(x),z,y) alors vrai fsi

m(0,y,0) = vrai

m(s(x),y,z) = si m (x,y,w) ∧ a(y,w,z) alors vrai fsi

a (0,y,y) = vrai

a (s(x),y,s(z)) = si a (x,y,z) alors vrai fsi

+ équations booléennes.

2.2.6.2.2. Théorème

Soit Γ un ensemble de H.C., alors :

$\Gamma \vdash P(t_1 \ldots t_n)$ ssi $\tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) \equiv$ vrai dans τ_Γ

Démonstration :

Par construction, à chaque interprétation de Herbrand I qui est un modèle de Γ, on peut associer une algèbre (de Herbrand) A_I qui satisfait E de τ_Γ (cf.2.2.1.10) ; c'est-à-dire, si $P(t_1 \ldots t_n) \in I$ alors on a $\tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) \equiv$ vrai dans A_I et $P(t_1 \ldots t_n) \notin I$ implique $\tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) \equiv$ faux. En effet, pour une clause composée d'un littéral $A \leftarrow$, le résultat est évident ; pour une clause $A \leftarrow B$ (où B peut être une disjonction), $A \vee \bar{B}$ doit être vrai ; donc, si B est vrai, alors obligatoirement, $A \in I$ (sinon I n'est pas un modèle). En passant de l'interprétation I à l'algèbre A_I, il est clair que l'équation "A = si B alors vrai fsi" est satisfaite.

Il reste à montrer que l'interprétation $\cap M(\Gamma)$ est effectivement associée à T_{τ_Γ} (noté $T_{\tau(\Gamma)}$) qui est ici le quotient de l'algèbre des termes et des

prédicats par la relation de congruence engendrée par E. Mais on ne s'intéresse qu'à la classe de congruence de "vrai". Donc, pour chaque $\tilde{p}$ dans A_I, on considère la relation n-aire dans les termes de la manière suivante :

$$\{(t_1 \ldots t_n) \mid \tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) \equiv \text{vrai dans } A_I\}$$

est équivalente à :

$$\{(t_1 \ldots t_n) \mid P(t_1 \ldots t_n) \in I\}$$

D'après la sémantique de P, on a :

$$\{(t_1 \ldots t_n) \mid P(t_1 \ldots t_n) \in \cap\, M(\Gamma)\}$$

$$= \{(t_1 \ldots t_n) \mid \tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) \equiv \text{vrai dans toute } A_I\}$$

$$= \{(t_1 \ldots t_n) \mid \tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) \equiv \text{vrai dans } T_{\tau(\Gamma)}\}$$

Inversement, si $\tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) \equiv$ vrai, on doit montrer que $\overline{P(t_1 \ldots t_n)} \notin \cap\, M(\Gamma)$.

Par définition de $\equiv$, on sait que ($\exists$ substitution σ)
telle que : $\tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) = \tilde{p}(\tilde{u_1} \ldots \tilde{u_n})\,\sigma$
et : $\tilde{p}(\tilde{u_1} \ldots \tilde{u_n}) = \text{vrai} \in E$

ou bien : $\tilde{p}(\tilde{u_1} \ldots \tilde{u_n}) =$ <u>si</u> $\tilde{q_1}(\,) \wedge \tilde{q_2}(\,) \ldots \wedge \tilde{q_k}(\ldots)$ <u>alors</u> vrai <u>fsi</u> $\in E$

Dans le cas 1, $P(u_1 \ldots u_n)\,\sigma \in \cap\, M(\Gamma)$ par définition d'un modèle de Herbrand de Γ ; dans le cas 2, on sait que $[P(u_1 \ldots u_n) \vee \bar{Q}_1(\,) \vee \bar{Q}_2(\,) \ldots \vee \bar{Q}_k(\ldots)]\sigma$ est vraie dans $\cap\, M(\Gamma)$, donc :

- soit : $P(u_1 \ldots u_n)\,\sigma \in \cap\, M(\Gamma)$;
- soit : $(\exists i)(1 \leq i \leq k)\ \bar{Q}_i(\ldots)\,\sigma \in \cap\, M(\Gamma)$;

mais dans le système équationnel, $\tilde{q_i}(\ldots)\sigma \equiv$ vrai $(\forall i)$.
Donc, par récurrence sur la longueur de la dérivation de la clause $\leftarrow P(t_1 \ldots t_n)$, on en déduit que :

$$\tilde{p}(\tilde{t_1} \ldots \tilde{t_n}) \equiv \text{vrai} \supset \Gamma \models P(t_1 \ldots t_n)$$

2.2.6.3. Inclusion de TA dans C.H.

2.2.6.3.1. Définition

A tout type abstrait $\tau = \langle S, \Sigma, E \rangle$, on peut faire correspondre un ensemble de clauses Γ_τ de la manière suivante :

- soit $S = \{s_1, \ldots s_n\}$, $\Sigma = \{f_1 \ldots f_k\}$

$E = \{L_1 = R_1, \ldots, L_i = \underline{si}\ C_i\ \underline{alors}\ R_i\ \underline{fsi}, \ldots\}$;

$\forall s_i \in S$, on introduit un symbole de prédicat binaire E_{s_i}

$\forall f_i \in \Sigma$, on fait correspondre un symbole de fonction $\hat{f}_i$.

Toutes les expressions construites à l'aide des constantes et des symboles fonctionnels de τ sont considérées comme des termes en Γ_τ. Les prédicats E_{s_i} sont définis par :

$(\forall s_i \in S)\ E_{s_i}(x,x) \leftarrow$ (axiomes de l'égalité formelle)

$E_{s_i}(x,y) \leftarrow E_{s_i}(y,x)$

$E_{s_i}(x,z) \leftarrow E_{s_i}(x,y)\ E_{s_i}(y,z)$

$\forall f : s_1\ s_2 \ldots s_n \rightarrow s_{n+1} \in \Sigma$

$E_{s_{n+1}}(\hat{f}(x_1, \ldots x_n), \hat{f}(y_1, \ldots y_n))$

$\leftarrow E_{s_1}(x_1, y_1) \ldots E_{s_n}(x_n, y_n)$

$\forall (L = R) \in E$, si L dans s_i :

$E_{s_i}(\hat{L}, \hat{R}) \leftarrow$

$\forall (L = \underline{si}\ C\ \underline{alors}\ R\ \underline{fsi})$ et L dans s_i :

$E_{s_i}(\hat{L}, \hat{R}) \leftarrow E_{bool}(\hat{C}, vrai)$

$\forall (L = \underline{si}\ C\ \underline{alors}\ R_1\ \underline{sinon}\ R_2\ \underline{fsi})$:

$E_{s_i}(\hat{L}, \hat{R}_1) \leftarrow E_{bool}(\hat{C}, vrai)$

$E_{s_i}(\hat{L}, \hat{R}_2) \leftarrow E_{bool}(\hat{C}, faux)$

Note : le type bool est lui aussi exprimé à l'aide du prédicat E_{bool} :

$E_{bool}(\neg vrai, faux) \leftarrow$

$E_{bool}(\neg faux, vrai) \leftarrow$

$E_{bool}(vrai \wedge x, x) \leftarrow$

etc...

2.2.6.3.2. Théorème

Soit τ un type abstrait défini par équations, alors :

$$\forall (t_1, t_2 \in s_i)\ (\forall s_i \in S) \quad t_1 \equiv_{s_i} t_2 \ \underline{\text{ssi}}\ \Gamma_\tau \models E_{s_i}(\hat{t}_1, \hat{t}_2)$$

Démonstration :

Elle est immédiate ; d'après la définition algébrique des types, $\equiv_{s_i}$ est la plus petite relation de congruence engendrée par les équations, c'est-à-dire l'intersection de toutes les relations de congruence qui contiennent toutes les instances de substitution de E (§2.2.1.13).

Soit $M = \{E_{s_i}(\hat{t}_1, \hat{t}_2) \mid t_1 \equiv_{s_i} t_2,\ \forall (s_i \in S)\}$,

alors il est évident que M est un modèle de Γ_τ.

Par un raisonnement analogue à la réciproque de 2.2.6.2.2., $\nexists\ s_i \in S,\ (\hat{t}_1, \hat{t}_2) \in s_i$ tels que $E_{s_i}(\hat{t}_1, \hat{t}_2) \in M \wedge E_{s_i}(\hat{t}_1, \hat{t}_2) \notin \cap M(\Gamma_\tau)$

donc $M \subseteq \cap M(\Gamma_\tau)$ et comme $\cap M(\Gamma_\tau) \subseteq M$, on termine la démonstration, puisque :

$$\cap M(\Gamma_\tau) = \{E_{s_i}(\hat{t}_1, \hat{t}_2) \mid t_1 \equiv s_i\ t_2, \forall (s_i \in S)\}$$

Remerciements :

Les idées de ce paragraphe §2.2.6. ont germé à la suite de discussions avec Henri SAYA (Laboratoire IMAG) et Philippe ROUSSEL (Luminy, Marseille et Université Simon Bolivar, Caracas).

3 - VERIFICATION DE PROGRAMMES SPECIFIES PAR EQUATIONS

3.1. Vérification ou déduction de programmes ?

3.1.1. Position du problème de la vérification

Le problème de la vérification de programmes spécifiés par équations doit être soigneusement posé. En effet, on a déjà parlé de vérification de représentation (au § 2.2.2.3.) et de propriétés (au § 2.2.3.1.), mais il s'agissait de faire des démonstrations de théorèmes à l'intérieur d'un même système formel, le système équationnel. C'est cette approche qui est prise par Guttag, de même que dans les langages OBJ et CLEAR, où le langage de spécification est aussi le langage de programmation. L'univers des objets est l'univers de Herbrand et les "calculs" se ramènent à des transformations d'arbres. L'avantage est que le formalisme est unique et que les démonstrations utilisent essentiellement l'induction structurale ou l'induction par les générateurs. L'inconvénient majeur est que l'on n'utilise pas les structures de données déjà fournies par le matériel des ordinateurs : la constante 3 résulte du calcul succ(succ(succ(0))), l'indexation d'un tableau (§ 2.2.2.3.1.) passe par une définition récursive de l'opérateur val, etc... L'exécution de tels programmes est donc terriblement inefficace. Il y a donc lieu de chercher d'autres solutions d'implantation sans perdre le bénéfice de la spécification algébrique.

3.1.2. Approche par constructeurs pre-spécifiés

Une première solution (qui mérite d'être étudiée davantage) est de considérer que le langage de programmation contient des structures de données classiques

(genre ALGOL 68) qui sont spécifiées algébriquement et pour lesquelles on a fait une implantation efficace (qui satisfait les spécifications). Le programmeur réalise ses programmes et ses nouvelles définitions dans le langage algébrique. Ainsi les preuves de programmes se font comme précédemment tandis que l'implantation des objets de base et des fonctions d'accès élémentaires est traitée efficacement. Le module de représentation "rep-table-2" est un bon exemple de programmation algébrique avec structures de données classiques (tuple :[ ] et tableau). Les seuls problèmes à résoudre sont les suivants :

1 - Lorsqu'on donne la représentation d'un type abstrait à l'aide d'autres types abstraits non de base, on doit assurer une "transitivité" des représentations pour conserver l'efficacité. Par exemple, dans rep-table-1, le type de représentation est liste ; il faudra donc remplacer le type liste et les constructeurs abstraits (cons,nil) par leur représentation (et ainsi de suite jusqu'aux constructeurs de base).

2 - Les structures de contrôle du langage algébrique sont la composition fonctionnelle et le si-alors-sinon-fsi. Pour obtenir une itération, on est forcé de passer par des opérateurs auxiliaires et des définitions récursives (cf. type mc : §2.2.3.5.2.). Il s'ensuit une relative simplicité de définition, mais une lourdeur certaine de l'exécution. On peut penser optimiser ces fonctions récursives, soit en les transformant automatiquement sous forme itérative, soit par la recherche (peu automatisable) de plus petits points fixes (cf. [GAU78]).

3.1.3. Approche par affinements successifs[FIN78]

Cette méthode consiste à spécifier algébriquement des types et des fonctions d'accès, puis à transformer progressivement les opérateurs intervenant dans la spécification pour se ramener à un programme algorithmique. Cette transformation est quelquefois automatique, quelquefois manuelle (lorsque des décisions d'implantation doivent être prises), et le programme peut être guidé dans son travail par un système conversationnel. Là encore, un tel système demande un

travail approfondi de recherche, que l'on n'aborde pas ici. Néanmoins, il peut facilement s'insérer dans le cadre que l'on a choisi : le langage de spécification est celui utilisé dans le chapitre 2 ; le langage de programmation est celui du chapitre 1. On a systématiquement fait en sorte que les connexions entre les deux formalismes soient apparentes.

Rappelons deux exemples :

Exemple 1 : Le type pile générique, d'après le § 2.2.3.4. est spécifié par :

```
type pile (t:tyfo) ;
     créerp : → pile(t)
     empiler : pile(t) x t → pile(t)
     depiler : pile(t) → pile(t) ou E
     sommet : pile(t) → t ou E
     remplacer : pile(t) x t → pile(t)
     non-défini : → E

eqns depiler(créerp) ⇒ non-défini
     depiler(empiler(p,x)) = p
     sommet(créerp) ⇒ non-défini
     sommet(empiler(p,x)) = x
     remplacer(p,x) = empiler(depiler(p),x)
          ou qd non-defini alors empiler(p,x) fq
ftype ;
```

La spécification du type pile (t) dans le langage algorithmique du chapitre 1 (§ 1.2.2.) est obtenue en faisant les choix suivants (on a supprimé la condition de dépassement qui n'apparait pas dans la spécification) :

- créerp n'est pas implanté car il est fourni par les instructions d'initialisation du module de représentation (transformation standard) ;
- empiler est une procédure dans laquelle le premier paramètre est modifié ; ce qui donne :
  - syntaxe abstraite: empiler : pile(t) x t → pile(t)
  - syntaxe concrète : proc empiler (mod pile(t),t)

- de même, depiler modifie son paramètre :

  syntaxe abstraite : depiler : pile(t) → pile(t) ou E

  syntaxe concrète : proc depiler (mod pile(t)) ou (non-défini)

- sommet n'est pas modifié ;

- remplacer étant un opérateur dérivé, on peut le reprendre quasi textuellement de la spécification (au moins dans une première version), en choisissant de modifier le premier paramètre. Les changements sont :

  . les cas d'erreur sont écrits dans la partie déclaration ;

  . la composition de fonction est remplacée par une séquence d'appels, d'après la sémantique classique du ";" :

  $S(a;b)\sigma = S(b)(S(a)\sigma)$

  où S est la fonction sémantique et σ l'état de la mémoire. Ici les seules modifications de σ sont celles du paramètre p, ce qui donne :

```
proc remplacer (mod p : pile(t),x: t );
     qd non-défini alors empiler (p,x) fq
début depiler(p) ; empiler(p,x)
fin ;
```

La déduction des corps des fonctions empiler, depiler, sommet sur la base de la représentation et de la spécification algébrique peut être conduite comme en [GAU78] mais en tenant compte des modifications syntaxiques.

Exemple 2 : Les itérations.

Le passage de la forme syntaxique abstraite à la forme concrète est exposé en détail au § 2.2.4.4.3. Lorsqu'on spécifie un opérateur avec des paramètres seq(t), on ne présuppose pas qu'il sera implanté sous forme d'itérateur, il peut être implanté à l'aide du type déduit de seq(t), et qui fait une représentation spaciale des éléments. Néanmoins, dans la plupart des cas, il sera avantageux de passer par des séquences temporelles, car l'écriture des itérateurs à partir des spécifications est facile, et l'implantation peut être efficace en espace (par rapport aux séquences spaciales) et en temps, à cause des optimisations par décomposition d'itérations.

3.1.4. Classification des équations

Dans les paragraphes suivants, on va s'intéresser plus spécialement aux méthodes de vérification proprement dites, c'est-à-dire que l'on se donne a priori un programme algorithmique et des spécifications algébriques, et l'on essaie de démontrer que le programme satisfait les équations. S'il s'agit d'un type, la représentation doit satisfaire les équations du type (à l'aide des fonctions de représentation et de l'invariant concret).

On distingue plusieurs sortes d'équations à démontrer, non pas dans leur nature, mais plutôt dans leur forme syntaxique. Ce sont :

1 - Les équations de propriété.

Elles caractérisent des classes de fonctions, donc elles ne tiennent pas compte de l'effet particulier de chaque fonction, mais essentiellement de relations entre les fonctions (structure algébrique).

ex. :
```
prop el-neutre (0,+) sur t : tyfo ;
     0 : → t
     + : txt → t
eqns     x+0 = x
         0+x = x
fprop
```

qui peut s'appliquer, aussi bien aux entiers, aux chaînes de caractères avec λ et la concaténation, ou encore à l'instruction vide et à la composition d'instruction (;).

2 - Les équations spécifiques.

Ce sont essentiellement les équations de définition des types ; elles font aussi intervenir des relations entre les fonctions, mais elles caractérisent des effets.

ex. :
```
depiler(empiler(p,x)) = p
ins(x,ins(y,S)) = ins(y,ins(x,S))
x * (y+1) = x * y+x
```

3 - Les équations dérivées.

Elles sont de la forme f(x) = expression, où x représente des variables. Ces équations dérivées décrivent une fonction (simple ou récursive) de la même manière que dans le langage algorithmique, puisque la partie gauche constitue l'en-tête, et la partie droite le corps. Aussi nous ne nous intéresserons pas à ce genre d'équations (bien qu'on puisse le traiter comme dans le cas 2), puisque l'équivalence entre une telle équation et un programme se ramène à des travaux connus :

1/ transformation récursif/itératif,

2/ synthèse de fonction récursive à partir de programmes ou de spécifications de ces programmes,

3/ comparaison de fonctions récursives [BUR77a].

Illustrons ces différents cas :

Soit la spécification algébrique de l'opérateur décaler de la définition rep-table-2 (§ 2.2.3.5.3.).

(a) décaler : tableau(t)x nat x nat → tableau(t)

```
décaler(T,i,j) = si le(j,i)
     alors si eq-nat(i,j) alors ranger(T,succ(j),valeur(T,j))
           sinon T fsi
     sinon décaler(ranger (T,succ(j),val(T,j)),i,pred(j)) fsi
```

On donne aussi le programme d'implantation de cet opérateur ; c'est une procédure qui modifie son premier paramètre :

(b)
```
proc décaler (mod T: tableau(t),i,j:nat) ;
début
     si j ≥ i
     alors faire T[j+1] := T[j];
               si i=j alors sortie fsi ;
               j := j-1
          ffaire
     fsi
fin ;
```

Le premier pas consiste à donner les interprétations des opérateurs abstraits dans le domaine de programmation en transformant éventuellement la syntaxe des opérateurs pour introduire les modifications. Ici, on a :

```
le : natxnat → bool    devient ≤   :opérateur infixé ;
eq-nat : natxnat → bool    devient   = : opérateur infixé ;
ranger : tableau(t) x nat x t → tableau(t)
     devient   := (mod T : tableau(t),a:nat, b:t)
     avec la notation  T[a] := b  ;

val : tableau(t) x nat → nat
     possède concrètement la même syntaxe,mais
     val(T,a) s'écrit T[a];

succ(j)   devient   j+1 ;
pred(j)   devient   j-1 ;
```

Moyennant ces transformations syntaxiques, la version (a) de l'opérateur décaler devient la procédure récursive :

(a1)
```
proc décaler (mod T : tableau(t), i,j:nat) ;
début si j ≤ i
      alors si i=j alors T[j+1] := T[j] fsi
      sinon T[j+1] := T[j] ; décaler (T,i,j-1)
      fsi
fin ;
```

Appliquons la méthode 1 de transformation de programme récursif (a1) en itératif :

(a2) par mise en facteur de T[j+1] := T[j] dans les cas i=j et j>i :

```
début si j ≥ i
      alors T[j+1] := T[j] ;
            si i<j alors décaler (T,i,j-1)  fsi
      fsi
fin ;
```

La transformation d'une récursivité finale en itération donne immédiatement :

```
(a3)    début  E : si j ≥ i
                  alors T[j+1] := T[j] ;
                  si i < j alors j := j-1 ; allera E fsi
                  fsi
        fin ;
```

Au point allera E, l'assertion {j≥i} est vraie, donc le test est inutile au retour ; ce qui donne :

```
(a4)    début  si j ≥ i
               alors E : T[j+1] := T[j] ;
                     si  i < j alors j := j-1 : allera E fsi
               fsi
        fin ;
```

Et en remplaçant le allera par faire ffaire, avec le cas de sortie quand (¬(i < j) ∧ j ≥ i) ≡ (i=j), on a le programme (b).

La deuxième méthode consiste à utiliser le programme itératif (b) et à le transformer :

(b1) : déploiement de la boucle :

```
début si j ≥ i
      alors T[j+1] := T[j] ;
            si i≠j alors j := j-1 ;
                        faire T[j+1] := T[j] ;
                              si i=j alors sortie fsi ;
                              j := j-1
                        ffaire ;
            fsi
      fsi
fin ;
```

On peut faire apparaître le corps (b) initial :

```
(b2)    début si j ≥ i
              alors T[j+1] := T[j] ;
                    si i≠j alors j := j-1 ;
                                  (si j ≥ i
                                   alors faire ... ffaire fsi)
                    fsi
              fsi
        fin ;
```

Ce qui donne, par substitution de l'appel (repliement) :

```
(b3)    début si j ≥ i
              alors T[j+1] := T[j] ;
                    si i≠j alors j := j-1 ; décaler(T,i,j) fsi
              fsi
        fin ;
```

On obtient ainsi, à quelques variantes près, le programme (a2).

Un exemple de la troisième méthode a été donné au § 2.2.4.1., avec l'opérateur de représentation d(ins) défini de deux manières différentes.

On reviendra aux vérifications par transformation au § 3.3.3. pour montrer des équivalences de programmes. Auparavant, on essaie de ramener une démonstration d'équation à une démonstration de théorème, ou à une équivalence de relations.

3.2. Vérification à l'aide des assertions

3.2.1. Méthode générale

Les programmes étant des actions dynamiques dont l'effet est de modifier un environnement, il n'est pas facile de démontrer qu'ils satisfont des équations statiques. La méthode que l'on propose est donc de remplacer le programme par

une spécification statique et de démontrer que celle-ci satisfait les équations algébriques. La première spécification statique à laquelle on pense est celle des assertions. L'avantage est que les preuves de programmes par assertions sont bien connues et les méthodes bien répertoriées. Le pas supplémentaire que l'on propose ici est d'utiliser les assertions initiales et finales des programmes pour démontrer les théorèmes dérivés des équations. Cette méthode est aussi proposée par [NAK77] et utilisée dans [ZIL79] [DAR78]. Elle associe à la fois les techniques de [GUT76a] et celles de [WUL76]. Comme en ALPHARD, la représentation d'un type abstrait (générique) doit être caractérisée par un interface de spécifications axiomatiques. Cet interface comprend :

1 - la fonction de représentation :

rep : type abstrait A → type de la représentation R.

2 - l'invariant concret $I_c(x)$ qui exprime les conditions pour qu'un objet du type de la représentation soit un représentant du type abstrait, c'est-à-dire :

si type(y) = A ∧ type(x) = R

alors rep(y)=x <==> $I_c(x)$

(note : on utilise ≡ pour l'équivalence à l'intérieur des types et <==> pour des équations de prédicats).

3 - l'interprétation de l'égalité du type abstrait A :

$égal_A(x_1, x_2)$

qui exprime l'équivalence au sens de la définition algébrique de y_1 et y_2 de type A tels que $rep(y_1)=x_1$ et $rep(y_2)=x_2$

4 - les pré et post conditions des fonctions concrètes f exprimées en termes de I_c et des assertions $entrée_f$ et $sortie_f$:

$pré_f <==> I_c \wedge entrée_f$

$post_f <==> I_c \wedge sortie_f$

Les conditions de vérification du module de représentation sont celles de ALPHARD, à savoir :

1 - Si <init> représente les instructions du module de représentation et r l'objet construit (de type R), la preuve de l'initialisation est :

vrai { <init> } $sortie_{créer} \wedge I_c(r)$

2 - Pour toute procédure P définie dans la représentation, si x dénote les paramètres, on a :

$$I_c(x) \wedge entrée_P(x) \quad \{P(x)\} sortie_P \wedge I_c(x)$$

La condition sortie des procédures peut être exprimée en fonction de :

- res_p si P a un résultat,
- x' : valeur des paramètres à l'entrée de la procédure,
- x : valeur des paramètres à la sortie.

Pour les paramètres qui ne sont pas de mode mod on a évidemment x=x'.

3 - La vérification de $égal_A$ exige un traitement particulier ; on doit vérifier les axiomes de l'égalité :

$$\forall(x,y,z)$$
$$I_c(x) \supset égal_A(x,x)$$
$$(I_c(x) \wedge I_c(y)) \supset (égal_A(x,y) \supset égal_A(y,x))$$
$$(I_c(x) \wedge I_c(y) \wedge I_c(z)) \supset$$
$$((égal_A(x,y) \wedge égal_A(y,z)) \supset égal_A(x,z))$$

De plus, $\forall$ proc P dans A, tel que P : $t_1 x t_2 \ldots t_n \rightarrow t$,
On pose $x = x_1, x_2 \ldots, x_n$ et $y = y_1, \ldots y_n$, alors le théorème à démontrer est :

$$(I_c(x'_i) \wedge I_c(y'_i)(\forall i : type(x'_i)=R))$$
$$\supset [(E(x'_i,y'_i) \wedge entrée_p(x') \wedge entrée_p(y') \wedge sortie_p(res_p^1,x,x')$$
$$\wedge sortie_p(res_p^2,y,y')) \supset (E(res_p^1,res_p^2) \wedge E(x,y))]$$

où

$E(x_i,y_i)$ est mis pour $x_i \equiv y_i$ ssi $type(x_i) \neq R$

et $égal_A(x_i,y_i)$ ssi $type(x_i) = type(y_i) = R$

Cette définition du prédicat $égal_A$ et les démonstrations que l'on fait sur les axiomes du type impliquent que :

$$\forall x,y : type(x) = type(y) = A,\ x \equiv y \supset égal_A(rep(x),rep(y))$$

Donc on assure que toute démonstration faite à l'extérieur du type en utilisant les axiomes algébriques est aussi vraie sur l'implantation en utilisant le prédicat $égal_A$.

Lorsque les vérifications que l'on vient de décrire valident les spécifications axiomatiques du module de représentation, on peut démontrer que cette représentation satisfait les spécifications algébriques du type abstrait.

Les théorèmes à démontrer sont déduits des équations du type ; une équation est décrite syntaxiquement par

<e> ::= <pg> = <pd> | <pg> => <exception>
<pg> ::= <expr>
<pd> ::= si <expr> alors <pd> sinon <pd> fsi |
si <expr> alors <pd> fsi | <expr>
<expr> ::= <symb-f> [(<expr>[, <expr>]*] | <var>
<var> ::= id (variable)
<symb-f> ::= id (symbole fonctionnel)
<exception> ::= id (symbole d'erreur).

Si e est une équation, alors tr(e) est un théorème obtenu par les fonctions tr, spec, repc et rep ci-après :

tr(e) = spec(e) ⊃ repc(e) si e : <pg> = <pd>
tr(pg => exception) = spec(pg) ⊃ faux

Les transformations spec et repc sont définies par :

spec(pg=pd) = spec(pg) ∧ spec(pd)
spec(si expr alors pd_1 sinon pd_2 fsi)
= spec(expr) ∧ spec(pd_1) ∧ spec(pd_2)
spec(si expr alors pd fsi)
= spec(expr) ∧ spec(pd)

∀ symbole fonctionnel f :

$$\text{spec}(f(\alpha)) = \begin{cases} \text{spec}(\alpha) \wedge \text{entrée}_f(\text{rep}(\alpha)) \\ \qquad \wedge \text{post}_f(\text{res}_f, u, u') & \underline{\text{ssi}}\ f \in A\ , \\ \text{spec}(\alpha) & \text{sinon.} \end{cases}$$

α est mis pour la liste des paramètres, et donc,

$$\text{spec}(\alpha_1, \alpha_2 \ldots \alpha_k) = \text{spec}(\alpha_1) \wedge \text{spec}(\alpha_2) \wedge \ldots \wedge \text{spec}(\alpha_k)$$

Dans la post-condition de f, les paramètres u et u' sont obtenus des paramètres de la spécification x et x' par substitution uniforme de rep(α), paramètre d'entrée.

∀ variable x :

$$\text{spec}(x) = \begin{cases} I_c(\text{rep}(x)) & \text{ssi type}(x) = A \\ & \text{rep est ici la fonction de représentation.} \\ \text{vrai} & \text{sinon} \end{cases}$$

On a d'autre part :

$$\text{repc}(pg = pd) = \text{repc}(pg, pd)$$

$$\text{repc}(expr_1, \underline{\text{si}}\ expr_2\ \underline{\text{alors}}\ pd_1\ \underline{\text{sinon}}\ pd_2\ \underline{\text{fsi}}) = (\text{rep}(expr_2) \equiv_{bool} \text{vrai}) \supset \text{repc}(expr_1, pd_1) \wedge (\text{rep}(expr_2) \equiv_{bool} \text{faux}) \supset \text{repc}(expr_1, pd_2)$$

$$\text{repc}(expr_1, \underline{\text{si}}\ expr_2\ \underline{\text{alors}}\ pd\ \underline{\text{fsi}}) = (\text{rep}(expr_2) \equiv_{bool} \text{vrai}) \supset \text{repc}(expr_1, pd_1)$$

$$\text{repc}(expr_1, expr_2) = E(\text{rep}(expr_1), \text{rep}(expr_2))$$

où E est défini comme précédemment, c'est-à-dire :

$$E(x,y) = \begin{cases} \text{égal}_A(x,y) & \text{ssi type}(x) = \text{type}(y) = R \\ x \equiv_B y & \text{ssi type}(x) = \text{type}(y) = B \wedge B \neq A \end{cases}$$

∀ symbole fonctionnel f :

$$rep(f(\alpha)) = \begin{cases} res_f & \text{ssi f est une procédure de A avec résultat} \\ x & \text{ssi x est un paramètre modifié de f et } f \in A \\ f(rep(\alpha)) & \text{ssi } f \notin A \end{cases}$$

∀ variable x :

$$rep(x) = \begin{cases} rep(x) & \text{ssi } type(x) = A \\ x & \text{sinon} \end{cases}$$

On remarque que, pour les équations du genre pg=pd, l'application de tr est transparente si aucun opérateur n'est un opérateur du type A :

tr(u = v) devient $u \equiv v$

tr(u = si C alors v1 fsi) devient $(C \equiv_{bool} vrai) \supset u \equiv v1$

etc...

Cela signifie que tr va chercher dans le module de spécification associé à la représentation de A les éléments nécessaires à la démonstration, en conservant les axiomes des autres types. Une équation e est un axiome du type abstrait A représenté par R si tr(e) est un théorème démontrable à l'aide des axiomes des types utilisés par R. Le fait de considérer e comme un axiome provient uniquement de ce que l'on fait l'abstraction du module de spécification qui a servi à démontrer tr(e), de même que le type A est un nouveau type par abstraction de la représentation. Le mécanisme est à rapprocher du constructeur derive en CLEAR [BUR77].

[GUT78a] présente une méthode de vérification des axiomes à l'aide des post-conditions de procédures de représentation qui est approximativement la même que celle présentée ici, mais cet article ne m'était pas connu lorsque j'ai fait ce travail. Néanmoins, il y a quelques différences : on introduit ici les invariants concrets et on établir la relation entre les conditions d'entrée et la vérification des axiomes exprimant les cas d'erreur.

3.2.2. Application aux équations spécifiques

On illustre le principe de la vérification à l'aide du type pile dont a donné la spécification algébrique au § 3.1.3. Le module de représentation est semblable à celui du § 1.2.3. mais on utilise un tableau extensible pour représenter la pile. La spécification algébrique de ce type est :

```
type tablext (t:tyfo) ;
     creert : → tablext(t)
     ranger : tablext(t) x nat x t → tablext(t)
     val : tablext(t) x nat → t ou E
     indef : → E

eqns
     ranger(ranger(T,I,M),J,N) = si ¬eq-nat(I,J)
          alors ranger(ranger(T,J,N),I,M) fsi
     val(creert,J) ⇒ indef
     val(ranger(T,I,M),J) = si eq-nat(I,J) alors M
          sinon val(T,J) fsi
ftype ;
```

La représentation du type pile(t) est :

```
def rep-pile (t:tyfo)
représente pile(t) par r:struct(s:nat, tab:tablext(t)) ;
proc empiler (mod p:pile(t), x:t) ;
     début  p.s :=  p.s+1 ;  p.tab( p.s) := x fin ;
proc depiler (mod p:pile(t)) ou (non-defini) ;
     début si  p.s=0 alors non-defini
           sinon  p.s :=  p.s-1 fsi
     fin ;
proc sommet (p:pile(t)) → t ou(non-défini) ;
     début si  p.s=0 alors non-défini
           sinon retourner ( p.tab( p.s)) fsi
     fin ;
début r.s:=0
fdef ;
```

L'interface de spécification du module rep-pile est :

```
specif rep-pile (t:tyfo) ;
   rep : pile(t) → struct(nat,tablext(t))
         tel que rep(p)≡ < p.s,  p.tab> ;
     I_c(p) ⇐⇒ p.s ≥ 0 ∧ (∀i)(1 ≤ i ≤ p.s) ¬(val(p.tab,p.s) ⇒ indef) ;
egal_pile(p1,p2) ⇐⇒ eq-nat(p1.s, p2.s) ≡ vrai ∧
                    (∀i)(1 ≤ i ≤ p1.s) val(p1.tab,i) ≡_t val(p2.tab,i) ;
```

$entrée_{empiler}(p,x) \iff entrée_{creerp} \iff vrai$

$sortie_{empiler}(p,x,p') \iff p \equiv <succ(p'.s),ranger(p'.tab,succ(p'.s),x)>$;

$entrée_{depiler}(p) \iff entrée_{sommet}(p) \iff le(p.s,0) \equiv faux$

$sortie_{depiler}(p,p') \iff p \equiv <pred(p'.s),p'.tab>$;

$sortie_{sommet}(res,p,p') \iff res \equiv val(p.tab,p.s) \wedge p \equiv p'$;

$sortie_{creerp}(res) \iff res.s \equiv 0$;

fspecif ;

Notes :

- On a utilisé dans ces prédicats les opérateurs abstraits de tablext et de nat. Il va de soi que l'on peut réécrire ces opérateurs sous forme infixée avec modifications pour vérifier le module rep-pile.
- Dans $égal_{pile}$, on a utilisé la relation $\equiv_t$. Comme le type t est formel, on ne peut utiliser dans la démonstration que les axiomes de l'égalité formelle (sans jeu de mots), qui sont toujours vérifiés pour une congruence.
- Les conditions d'entrée sont les conditions sur les paramètres telles que le résultat est différent d'une erreur.

On ne donne pas la vérification du module rep-pile qui est ici particulièrement évidente. Appliquons plutôt la méthode de démonstration des équations spécifiques

1 - Démonstration de sommet (empiler(p,x))=x :

tr(sommet(empiler(p,x))=x)

$\Rightarrow$ spec(sommet(empiler(p,x))=x) $\supset$ repc(sommet(empiler(p,x))=x)

$\Rightarrow$ spec(sommet(empiler(p,x))$\wedge$spec(x) $\supset$ rep(sommet(empiler(p,x)))$\equiv_t$ x

Développons séparément :

rep(sommet(empiler(p,x))) $\Rightarrow res_{sommet}$;

spec(sommet(empiler(p,s))

$\Rightarrow$ spec(empiler(p,x)) $\wedge$ $entrée_{sommet}$(rep(empiler(p,x)))

$\wedge$ $post_{sommet}(res_{sommet},p_1,rep(empiler(p,x)))$

$\Rightarrow$ spec(empiler(p,x)) $\wedge$ $le(p_2.s,0) \equiv faux$

$\wedge$ $res_{sommet} \equiv val(p_1.tab,p_1.s) \wedge p_1 \equiv p_2 \wedge I(p_1)$;

spec(empiler(p,x))

$\Rightarrow$ spec(p) $\wedge$ spec(x) $\wedge$ entrée$_{empiler}$ (rep(p),rep(x))

$\wedge$ post$_{empiler}$ (p_2,rep(x),rep(p))

$\Rightarrow$ I_c(rep(p)) $\wedge$ vrai

$\wedge$ $p_2 \equiv$ <succ(p.s),ranger(p.tab,succ(p.s),x)>

$\wedge$ $I_c(p_2)$

En réunissant le tout, on obtient :

I_c(rep(p)) $\wedge$ $p_2 \equiv$ <succ(p.s),ranger(p.tab,succ(p.s),x)> $\wedge$ $I_c(p_2)$
$\wedge$ le(p_2.s,0) $\equiv$ faux $\wedge$ res$_{sommet} \equiv$ val(p_1.tab,p_1.s) $\wedge$ $p_1 \equiv p_2$
$\wedge$ $I_c(p_1) \supset$ res$_{sommet} \equiv_t$ x

Par substitution, la formule devient :

I_c(rep(p)) $\supset$ val(ranger(p.tab,succ(p.s),x),succ(p.s)) $\equiv$ x

$\Rightarrow$ x $\equiv$ x (par axiome 3 de tablext)

$\Rightarrow$ vrai

2 - Démonstration de dépiler(empiler(p,x))=p :

L'égalité est à démontrer dans le type, donc on utilise égal$_{pile}$.

tr(depiler(empiler(p,x))=p)

$\Rightarrow$ I_c(rep(p)) $\wedge$ vrai

$\wedge$ $p_2 \equiv$ <succ(p.s),ranger(p.tab,succ(p.s),x)> $\wedge$ $I_c(p_2)$

$\wedge$ le(p_2.s,0) $\equiv$ faux $\wedge$ $p_1 \equiv$ <pred(p_2.s),p_2.tab> $\wedge$ $I_c(p_1)$

$\supset$ égal$_{pile}$ (p_1,rep(p))

$\Rightarrow$ I_c(rep(p))

$\supset$ égal$_{pile}$(<pred(succ(p.s)),ranger(p.tab,succ(p.s),x)> ,rep(p))

$\Rightarrow$ I_c(rep(p)) $\supset$ égal$_{pile}$(<p.s,ranger(p.tab ,succ(p.s),x)> ,rep(p))

$\Rightarrow$ I_c(rep(p)) $\supset$ (eq-nat(p.s,p.s) $\equiv$ vrai)

$\wedge$ ($\forall$i)(1$\leq$i$\leq$p.s) val(ranger(p.tab,succ(p.s),x),i) $\equiv_t$ val(p.tab,i)

$\Rightarrow$ vrai

3 - Démonstration de sommet(creerp) => non-défini :

tr(sommet(creerp) => non-défini)

=> spec(sommet(creerp)) ⊃ faux

=> $I_c(res_{creerp}) \wedge res_{creerp}.s \equiv 0 \wedge le(res_{creerp}.s,0) \equiv faux$
$\wedge res_{sommet} \equiv val(p_1.tab,p_1.s) \wedge p_1 \equiv res_{creerp} \wedge I_c(p_1)$
⊃ faux

=> $\alpha \wedge le\ (0,0) \equiv faux \supset faux$

=> faux ⊃ faux

=> vrai.

Note : Il faut distinguer, syntaxiquement, les constantes du type booléen et les prédicats constants.

3.2.3. Application aux équations de propriétés

La vérification de propriétés de fonctions relève normalement du formalisme algébrique, ainsi qu'il a été montré au § 2.2.3. Cependant, si on connait la représentation du type, et la spécification de la représentation, il est quelquefois plus simple de démontrer la propriété comme un théorème de la représentation, puisqu'après tout , c'est la procédure de la représentation qui est utilisée dans les calculs. C'est aussi une méthode à utiliser si l'on n'est pas sûr que la spécification algébrique soit complète. Alors dans ce cas, les équations de propriété peuvent ne pas être dérivables de la spécification algébrique du type alors qu'ils peuvent l'être de la représentation. Pour l'utilisateur d'un type, cela revient à dire que le type sera spécifié à la fois par les équations spécifiques, mais aussi par les équations des propriétés du type. Ces dernières pourront quelquefois être plus utiles dans les preuves que les premières.

On va simplement appliquer les deux méthodes de démonstration au type vect(t) du § 2.2.3.5.2. Rappelons les éléments que l'on va utiliser :

```
extension vect(t:monoïde) : monoïde(vo,comp) ;
     comp : vect(t) x vect(t) → vect(t) ou E
privé compel : vect(t) x vect(t) x nat → vect(t)
eqns
     comp(v1,v2) = si ¬eq-nat(l(v1),l(v2))
          alors indefv sinon compel(v1,v2,l(v1)) fsi
     compel(v1,v2,0) = creerv
     compel(v1,v2,succ(i)) = ranger(compel(v1,v2,i),succ(i),
                              val(v1,succ(i))+val(v2,succ(i)))
```

On désire démontrer le théorème :

(ASS) comp(v1,comp(v2,v3)) = comp(comp(v1,v2),v3)

dans le cas où l'opération est définie, c'est-à-dire

l(v1) = l(v2) = l(v3) = l.

La méthode algébrique va consister à faire une induction sur l :

(ASS) ⇒ compel(v1,compel(v2,v3,l),l) = compel(compel(v1,v2,l),v3,l)

qui peut se démontrer par :

(l=0) et creer≡creerv

(l≥0) l'hypothèse d'induction et la propriété

TRUNC : compel(v1,v2,l) ≡ compel(v1/l,v2/l,l)

où v/l désigne le vecteur v composé des l premiers éléments.

```
⇒  compel(v1,compel(v2,v3,succ(l)),succ(l))
          = compel(compel(v1,v2,succ(l)),v3,succ(l))
⇒  (partie gauche)
   ranger(compel(v1,compel(v2,v3,succ(l)),l),succ(l),
        val(v1,succ(l))+val(compel(v2,v3,succ(l)),succ(l)))
 ≡ ranger(compel(v1,compel(v2,v3,l),l),succ(l),
        val(v1,succ(l))+val(ranger(compel(v2,v3,l),succ(l),
                          val(v2,succ(l))+val(v3,succ(l))),succ(l)))
 ≡ ranger(compel(v1,compel(v2,v3,l),l),succ(l),
        val(v1,succ(l))+(val(v2,succ(l))+val(v3,succ(l))))
```

$\Rightarrow$ (partie droite)

ranger(compel(compel(v1,v2,succ(1)),v3,1),succ(1),
val(compel(v1,v2,succ(1)),succ(1))+val(v3,succ(1)))

$\equiv$ ranger(compel(compel(v1,v2,1),v3,1),succ(1),
val(ranger(compel(v1,v2,1),succ(1),val(v1,succ(1))+val(v2,succ(1))))
+val(v3,succ(1)))

$\equiv$ ranger(compel(compel(v1,v2,1),v3,1),succ(1),
(val(v1,succ(1))+val(v2,succ(1)))+val(v3,succ(1)))

On a bien partie droite $\equiv$ partie gauche, par associativité de + et l'hypothèse d'induction.

Supposons maintenant que l'on ait une spécification axiomatique de comp(v1,v2) :

$entrée_{comp}(v1,v2) \Leftrightarrow l(v1) \equiv l(v2)$

$sortie_{comp}(res,v1',v2') \Leftrightarrow l(res_{comp}) \equiv l(v1')$
$\wedge\ (\forall i)(1 \leq i \leq l(res_{comp}))val(res_{comp},i) \equiv val(v1',i)+val(v2',i)$

$I_c(v) \Leftrightarrow (\forall i)(1 \leq i \leq l(v))\neg val(v,i) \Rightarrow indef$

$égal_{vect}(v1,v2) \Leftrightarrow l(v1) \equiv l(v2)$
$\wedge\ (\forall i)(1 \leq i \leq l(v1))val(v1,i) \equiv val(v2,i)$

Note : La représentation d'un vecteur est comme précédemment un tableau extensible : tablext ; pour simplifier, on a gardé les mêmes noms (val,ranger) pour les opérateurs qui ont les mêmes spécifications dans tablext et dans vect.

Dérivons la condition de vérification de l'associativité de comp à l'aide de la spécification axiomatique. On note r1,r2,r3,r4 les différentes occurrences de res_{comp} dans le théorème.

tr(comp(v1,comp(v2,v3)) = comp(comp(v1,v2),v3))

$\Rightarrow$ spec(comp(v1,comp(v2,v3)))$\wedge$ spec(comp(comp(v1,v2),v3))
$\supset$ $égal_{vect}(r1,r3)$

$\Rightarrow$ $(I_c(v1) \wedge I_c(v2) \wedge I_c(v3)$
$\wedge\ l(v2) \equiv l(v3) \wedge l(r2) \equiv l(v2)$
$\wedge\ (\forall i)(1 \leq i \leq l(r2))val(r2,i) \equiv val(v2,i)+val(v3,i)$
$\wedge\ l(v1) \equiv l(r2) \wedge l(r1) \equiv l(v1) \wedge I_c(r1)$
$\wedge\ (\forall i)(1 \leq i \leq l(r1))val(r1,i) \equiv val(v1,i)+val(r2,i))$
$\wedge\ (I_c(v1) \wedge I_c(v2) \wedge I_c(v3)$
$\wedge\ l(v1) \equiv l(v2) \wedge l(r4) \equiv l(v1)$
$\wedge\ (\forall i)(1 \leq i \leq l(r4))val(r4,i) \equiv val(v1,i)+val(v2,i)$
$\wedge\ l(r4) \equiv l(v3) \wedge l(r3) \equiv l(r4) \wedge I_c(r4)$
$\wedge\ (\forall i)(1 \leq i \leq l(r3))val(r3,i) \equiv val(r4,i)+val(v3,i))$
$\supset$ $égal_{vect}(r1,r3)$

$\Rightarrow$ $H \supset l(r1) \equiv l(r3)$
$\wedge\ (\forall i)(1 \leq i \leq l(r1))val(r1,i) \equiv val(r3,i)$

$\Rightarrow$ $H \supset$
$(\forall i)(1 \leq i \leq l(r1))val(v1,i)+(val(v2,i)+val(v3,i))$
$\equiv (val(v1,i)+val(v2,i))+val(v3,i)$

$\Rightarrow$ $H \supset$ vrai (par associativité de +)

$\Rightarrow$ vrai

Cet exemple montre que, dans les cas où la complexité est suffisante et où le programme et la spécification utilisent les mêmes schémas, l'équation algébrique sera décrite récursivement, tandis que le programme comportera des itérations. Si on examine les différentes étapes de la démonstration d'une équation de propriété, on s'aperçoit que plusieurs chemins sont possibles entre programmes, spécification axiomatique, spécification algébrique. La vérification de la spécification axiomatique à partir du programme passe par la découverte des invariants (Note : on peut dire aussi qu'ils sont nécessaires pour écrire le programme). La vérification de propriétés à partir de la spécification algébrique exige l'induction, tandis que la vérification de propriétés ou d'équations spécifiques à partir de spécification axiomatique n'exige que des substitutions et le calcul des propositions. On peut tracer le graphe (où les longueurs symbolisent les difficultés de démonstration) :

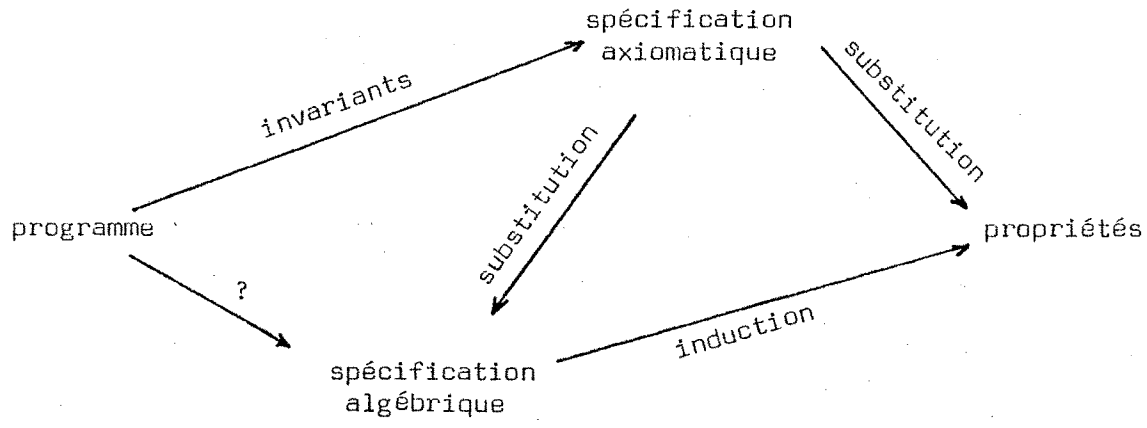


figure 1

On remarque que la substitution entre les spécifications axiomatiques et les spécifications algébriques n'est pas aussi simple qu'il paraît lorsque cette dernière utilise des opérateurs auxiliaires qui n'ont pas a priori de spécification axiomatique car ils ne sont pas implantés par un programme. Il doit y avoir alors une décomposition de la spécification axiomatique pour pouvoir caractériser axiomatiquement les opérateurs auxiliaires.

Appliquons le principe de transformation de spécification, à la manière de [MAN71] à la procédure compt, par l'intermédiaire de la spécification de compel :

$sortie_{comp}(res_{comp},v1',v2') \Longleftrightarrow sortie_{compel}(res_{comp},l,v1',v2') \wedge l \equiv l(v1)$

$entrée_{compel}(v1,v2,l) \Longleftrightarrow l(v1) \equiv l(v2) \wedge le(l(v1,l) \equiv faux.$

$sortie_{compel}(res_{compel}, l, v1', v2') \Longleftrightarrow le(l, l(v1')) \equiv vrai$

$\wedge (\forall i)\ (1 \leq i \leq l)\ \ val(res_{compel}, i) \equiv val(v1',i) + val(v2',i)$

Cette spécification permet de vérifier les équations de compel, et d'un autre côté, on peut dire qu'elle est à la base

a) de la spécification algébrique elle-même ;

b) de l'écriture du programme de comp.

Le point a) se montre aisément par cas [BER77c] :

$l \equiv 0$:

$sortie_{compel}(res_{compel}, 0, v1', v2') \Longleftrightarrow res_{compel} \equiv res_{créerv}$

$l \geq 0$:

$sortie_{compel}(res_{compel}, succ(l), v1', v2') \Leftrightarrow le\ (succ(l), l(v1')) \equiv vrai$

$\wedge\ (\forall i)\ (1 \leq i \leq succ(l))\ val(res_{compel}, i) \equiv val(v1', i) + val(v2', i)$

$\Leftrightarrow\ le\ (succ(l), l(v1')) \equiv vrai \wedge$

$(\forall i)\ (1 \leq i \leq l)\ val(res_{compel}, i) \equiv val(v1', i) + val(v2', i)$

$\wedge\ val(res_{compel}, succ(l)) \equiv val(v1', succ(l)) + val(v2', succ(l))$

$\Leftrightarrow\ le\ (succ(l), l(v1')) \equiv vrai \wedge sortie_{compel}(res_{compel}, l, v1', v2')$

$\wedge\ val(res_{compel}, succ(l)) \equiv val(v1', succ(l)) + val(v2', succ(l))$.

La dernière expression est à quelques notations près l'équation de compel dans laquelle on a résolu le problème suivant :

- sachant que $res1_{compel} \equiv compel(v1, v2, 1)$, trouver $res2_{compel}$

  tel que $(\forall i)\ (1 \leq i \leq l)\ val(res2_{compel}, i) \equiv val(res1_{compel}, i)$

  $\wedge\ val(res2_{compel}, succ(l)) \equiv val(v1, succ(l)) + val(v2, succ(l))$.

La réponse la plus simple est bien sûr :

$$res2_{compel} \equiv ranger(res1_{compel}, succ(l), val(v1, succ(l)) + val(v2, succ(l)))$$

Le point b) est un problème de synthèse où $sortie_{compel}$ est choisie comme assertion invariante :

```
proc  comp (v1, v2 : vect(t)) → vect(t) ;
      assert  l(v1) ≡ l(v2)
      res : vect(t) ;
      but  sortie_comp(res, v1, v2) ;
```

⇒ le but peut être réécrit :

```
l : nat ;
but  sortie_compel(res, l, v1, v2) ∧  l ≡ l(v1) ;
```

⇒ d'après [DER75], on peut résoudre ce but par une itération :

```
(1)  obtenir sortie_compel(res,l,v1,v2) ;
     faire assert sortie_compel(res,l,v1,v2) ;
           si l=l(v1) alors sortie fsi ;
           assert sortie_compel (res,l,v1,v2) ∧ l ≠ l(v1) ;
(2)        obtenir sortie_compel (res1,l1,v1,v2) ∧ res1 ≠ res ∧ l1 ≠ l
     ffaire ;
     assert sortie_comp (res,v1,v2)
```

⇒ à la ligne(1), res ≡ creerv, donc l ≡ 0 satisfait l'assertion

⇒ à la ligne(2), comme précédemment, on choisit :

$l_1 \equiv succ(l)$

$res_1 \equiv ranger(res, succ(l), val(v1, succ(l)) + val(v2, succ(l)))$

d'où le programme complet (en notations algorithmiques) :

```
⇒ proc comp (v1, v2 : vect(t)) → vect(t) ;
      assert l(v1) ≡ l(v2) ;
      res : vect(t) ;
      l : nat ;
  début l := 0 ;
  |   faire assert sortie_compel(res, l, v1, v2) ;
  |   |   si l = l(v1) alors sortie fsi ;
  |   |   l := l + 1 ;
  |   |   res (l) := v1(l) + v2(l)
  |   ffaire
  |   assert sortie_comp(res, v1, v2) ;
  |   retourner (res)
  fin ;
```

Remarque : comme la spécification initiale donne les équations de comp et compel en tant qu'extension du type vect, c'est-à-dire utilisant les primitives d'accès de vect, on a dérivé le programme dans vect plutôt que dans tablext (le résultat aurait été identique avec res : tablext(t)).

En conclusion, si l'on revient au schéma de vérification (fig.1), on peut lui faire correspondre un schéma de construction qui s'appuie sur les spécifications auxiliaires (ou Euréka), celles-ci étant le seul point d'invention (avec le choix de représentation) :

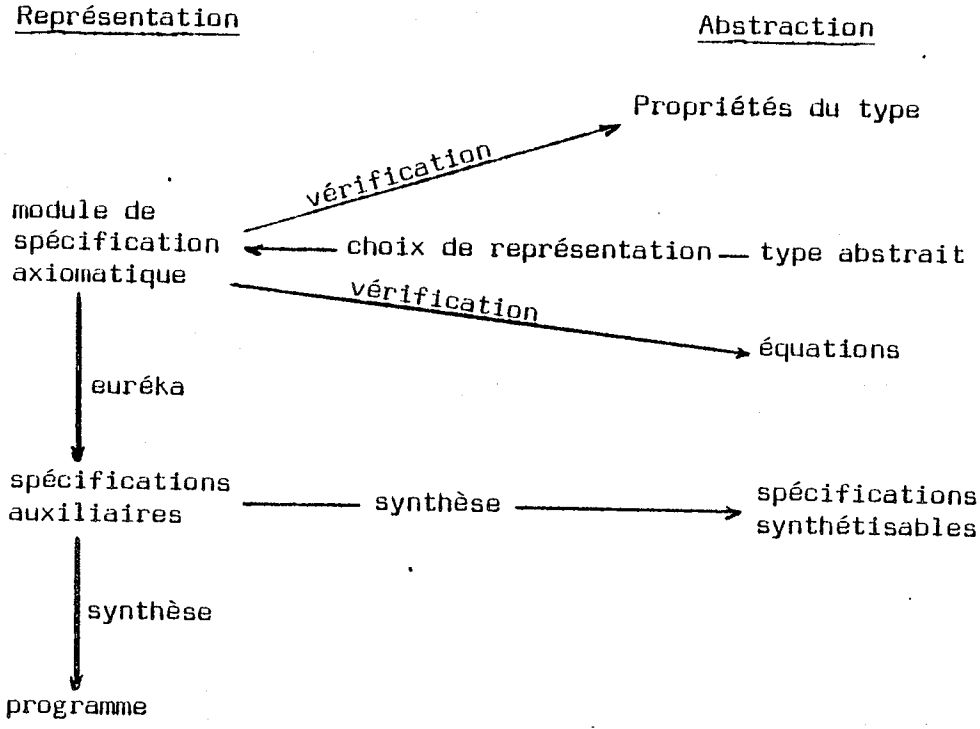


figure 2

3.3 Autres méthodes de vérification

3.3.1. Assertions par résolution d'équations de programmes.

Il peut apparaître que les méthodes de vérification présentées au §3.2. sont un peu lourdes, car elles supposent que l'on donne :

- les spécifications abstraites du type : T
- le module de représentation : R
- le module de spécifications axiomatiques : S

et que l'on fait les démonstrations :

R vérifie S (preuve de programme)

S implique T (théorème dans le système algébrique).

Plusieurs de ces éléments peuvent être redondants et donc calculés les uns à partir des autres par un système. On a déjà vu que le type et les spécifications axiomatiques permettent d'obtenir les programmes et quelques spécifications algébriques. Il est un autre cas que l'on va étudier ici, c'est celui où l'on se donne les programmes et la spécification algébrique, et l'on souhaite démontrer que les programmes satisfont les équations (sont donc une implantation correcte du type). C'est la flèche ? de la figure 1.

Dans les § 3.3.1. et 3.3.2., on étudie le cas où l'on transforme le programme en système d'équations, alors que dans le paragraphe 3.3.3., on étudiera le cas de la transformation des équations en programmes.

La représentation d'un programme sous forme d'un système d'équations où les inconnues sont des prédicats sur le vecteur d'état du programme (les variables) a été particulièrement étudiée par [COU78a], ch. 3. De même, la résolution automatique de ces systèmes est présentée en [COU77]. Illustrons cette méthode par l'exemple de comp §3.2.3.

```
proc  comp (v1, v2 : vect(t)) → vect(t) ;
   |  assert  P0(v1, v2)⇐⇒l(v1) ≡ l(v2) ;
   |  res : vect(t) ; l : nat ;
début  l := 0 ;
   |  faire  assert  P1 (res, l, v1, v2)
   |        si l = l(v1)  alors  sortie  fsi ;
   |         l := l + 1 ;
   |        res(l) := v1(l) + v2(l)
   |  ffaire
   |  assert  P2 (res, l, v1, v2)
   |  retourner (res)
fin  ;
```

Le système d'équations en avant est :

(AV) P_0 (v1, v2)⇐⇒l(v1) ≡ l(v2)

P_1 (res, l, v1, v2)⇐⇒(P_0(v1, v2)∧ l≡0∧ res ≡ creerv)
∨((∃res_1, l_1) P_1(res_1, l_1, v1, v2)∧ l≡succ(l_1) ∧
l_1 ≠ l(v1)∧ res≡ranger(res_1, l, val(v1,l) + val(v2, l)))

P_2 (res, l, v1, v2)⇐⇒P_1 (res, l, v1, v2)∧ l ≡ l(v1)

Le système d'équations en arrière est :

$\Longleftrightarrow$(AR) P_0 (v1, v2)$\Longleftrightarrow$l(v1) $\equiv$ l(v2) $\wedge$ P_1 (creerv, 0, v1, v2)

P_1 (res, l, v1, v2)$\Longleftrightarrow$(P_2(res, l, v1, v2) $\wedge$ l $\equiv$ l(v1)

$\vee$ P_1 (ranger (res, succ(l), val(v1, succ(l)) + val(v2, succ(l))),
succ(l), v1, v2))

La résolution dusystème (AV) donne les valeurs de prédicats :

P_0 (v1, v2)$\Longleftrightarrow$l(v1) $\equiv$ l(v2)

P_1 (res, l, v1, v2)$\Longleftrightarrow$l(v1) $\equiv$ l(v2) $\wedge$ 0 $\leq$ l $\leq$ l(v1)

$\wedge$ res $\equiv$ [ranger ()^{l} creerv, 1, val(v1, 1) + val(v2, 1))
, 2, val(v1, 2) + val(v2, 2))
, ...
, l, val(v1, l) + val(v2, l))

P_2 (res, l, v1, v2)$\Longleftrightarrow$l(v1) $\equiv$ l(v2) $\wedge$ l $\equiv$ l(v1)

$\wedge$ res $\equiv$ [ranger ()^{l}...)

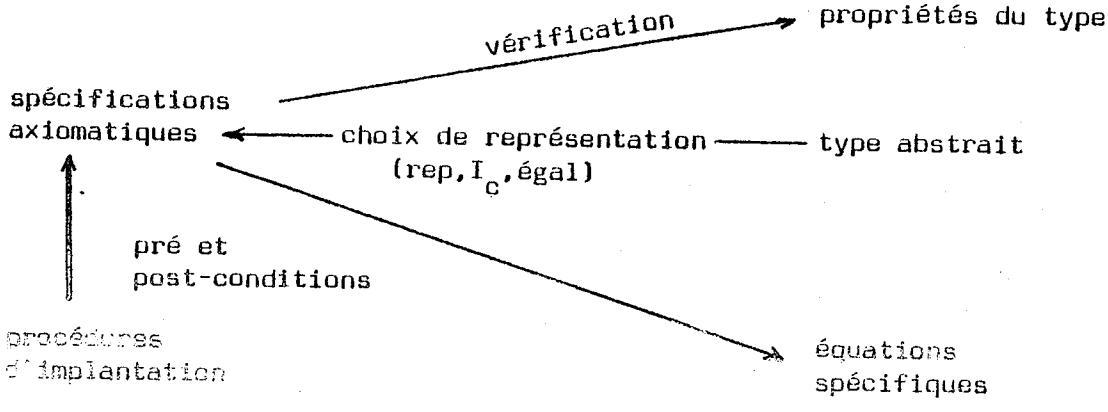
Ces assertions sont optimales et on a :

P_1 (res, l, v1, v2) $\supset$ sortie$_{compel}$(res, l, v1, v2)

P_2 (res, l, v1, v2) $\supset$ sortie$_{comp}$ (res, v1, v2)

En théorie, la résolution des équations du programme permet de donner les assertions de sorties, comme on vient de le voir, et les assertions d'entrée, qui sont les conditions pour lesquelles le programme termine. On obtient donc les pré et post-conditions des procédures d'implantation du type, qui, associées à la fonction de représentation, à l'invariant concret et au prédicat égal, constituent le module de spécification, avec lequel on peut prouver les théorèmes déduits des spécifications du type, comme il a été indiqué au § 3.2.

Le schéma qui illustre cette démarche est donné à la figure 3.



En pratique, cette méthode ne peut pas être utilisée, car la résolution exacte des équations de prédicats, même à la main, est quasi-impossible (bien que pour les cas simples que sont le plus souvent les procédures d'implantation des fonctions de types, elle puisse être envisageable). On doit donc essayer d'autres méthodes qui ne font pas intervenir la résolution des prédicats.

3.3.2. Comparaison de systèmes d'équations.

Les prédicats déduits d'un programme forment un système équationnel, de la même manière que les équations de la spécification d'un type. On peut se demander s'il est possible de comparer ces systèmes. Les équations de la spécification algébrique sont typées, aussi il faut ramener ces équations à des prédicats comprenant le résultat comme variable supplémentaire, et lier les résultats de la partie droite et de la partie gauche par ≡ ou égal, suivant que l'on est dans le type ou dans la représentation.

Reprenons la spécification de comp et compel ;
on note :

P_{comp}(res, v1, v2) ⟺ comp(v1, v2) ≡ res

P_{compel}(res, l, v1, v2) ⟺ compel(v1, v2, l) ≡ res

Les équations algébriques deviennent alors :

P_{comp}(res, v1, v2) ⟺ P_{compel}(res, l(v1), v1, v2)

P_{compel}(creerv, 0, v1, v2) ⟺ vrai

P_{compel}(res1, succ(l), v1, v2)

⟺ res1 ≡ ranger(res2, succ(l), val(v1, succ(l)) + val(v2, succ(l)))
∧ P_{compel}(res2, l, v1, v2)

Puisqu'il s'agit d'une équation spécifique, on veut montrer que P_2 et P_{comp} caractérisent les mêmes relations ; en effet, de (AR) et (AV) on déduit le système :

P_2 (res, l(v1), v1, v2) ⟺ P_1 (res, l(v1), v1, v2)

P_1 (ranger(res, succ(l), val(v1, succ(l)) + val(v2, succ(l))), succ(l), v1, v2)
⟺ P_1 (res, l, v1, v2)

P_1 (creerv, 0, v1, v2) ⟺ vrai

On obtient le même système que ci-dessus, avec :

$P_{compel}(res, 1, v1, v2) \Leftrightarrow P_1(res, 1, v1, v2)$

$P_{comp}(res, v1, v2) \Leftrightarrow P_2(res, 1(v1), v1, v2)$

Les systèmes d'équations étant identiques, ils caractérisent les mêmes relations. On a donc démontré que le programme satisfait une spécification algébrique sans avoir résolu le système d'équations de prédicats du programme.

De la même façon, on va essayer de démontrer une équation de propriété :

comp(v1, comp(v2, v3)) = comp(comp(v1, v2), v3)

Posons $Q_2(res, v1, v2, v3) \Leftrightarrow comp(v1, comp(v2, v3)) \equiv res$

$Q'_2(res, v1, v2, v3) \Leftrightarrow comp(comp(v1, v2), v3) \equiv res$

On définit Q_2 et Q'_2 par rapport à P_1 :

$Q_2(res, v1, v2, v3) \Leftrightarrow Pcomp(res, v1, w) \wedge Pcomp(w, v2, v3)$

$\Leftrightarrow P_1(res, 1(v), v1, w) \wedge P_1(w, 1(v), v2, v3)$

$Q'_2(res, v1, v2, v3) \Leftrightarrow P_1(res, 1(v), w, v3) \wedge P_1(w, 1(v), v1, v2)$

On note $1(v) \equiv 1(v1) \equiv 1(v2) \equiv 1(v3)$.

Définissons maintenant :

$Q_1(res, 1, v1, v2, v3) \Leftrightarrow P_1(res, 1, v1, w) \wedge P_1(w, 1, v2, v3)$

$Q'_1(res, 1, v1, v2, v3) \Leftrightarrow P_1(res, 1, w, v3) \wedge P_1(w, 1, v1, v2)$

Le calcul par cas donne :

$Q_1(creerv, 0, v1, v2, v3) \Leftrightarrow P_1(creerv, 0, v1, creerv) \wedge P_1(creerv, 0, v2, v3)$

$\Leftrightarrow$ vrai

(note : on utilise aussi la propriété TRUNC (§3.2.3.) :

$P_1(res, 1, v1, v2) \Leftrightarrow P_1(res, 1, v1/1, v2/1)$

Q_1(res1, succ(l), v1, v2, v3)

$\Longleftrightarrow$ P_1(res1, succ(l), v1, w1) $\wedge$ P_1(w1, succ(l), v2, v3)

$\wedge$ w1 $\equiv$ ranger(w, succ(l), val(v2, succ(l)) + val(v3, succ(l)))

$\wedge$ res1 $\equiv$ ranger(res, succ(l), val(v1, succ(l)) + val(w1, succ(l)))

$\Longleftrightarrow$ P_1(res, 1, v1, w1) $\wedge$ P_1(w, 1, v2, v3)

$\Longleftrightarrow$ (TRUNC) P_1(res, 1, v1, w) $\wedge$ P_1(w, 1, v2, v3)

$\Longleftrightarrow$ Q_1(res, 1, v1, v2, v3)

Le système définissant Q_2 s'écrit :

Q_2(res, v1, v2, v3)$\Longleftrightarrow$$Q_1$(res, 1(v), v1, v2, v3)

Q_1(ranger(res, succ(l), val(v1, succ(l)) + val(v2, succ(l)) + val(v3, succ(l))),

succ(l), v1, v2, v3)$\Longleftrightarrow$$Q_1$(res, 1, v1, v2, v3)

Q_1(creerv, 0, v1, v2, v3)$\Longleftrightarrow$vrai

Il n'est pas difficile de voir que l'on trouvera le même système pour Q'_2, d'où on déduit que la relation caractérisée par Q_2 est la même que celle caractérisée par Q'_2, ce qui termine la démonstration de l'équation.
La démarche suivie est représentée par la fig. 4.

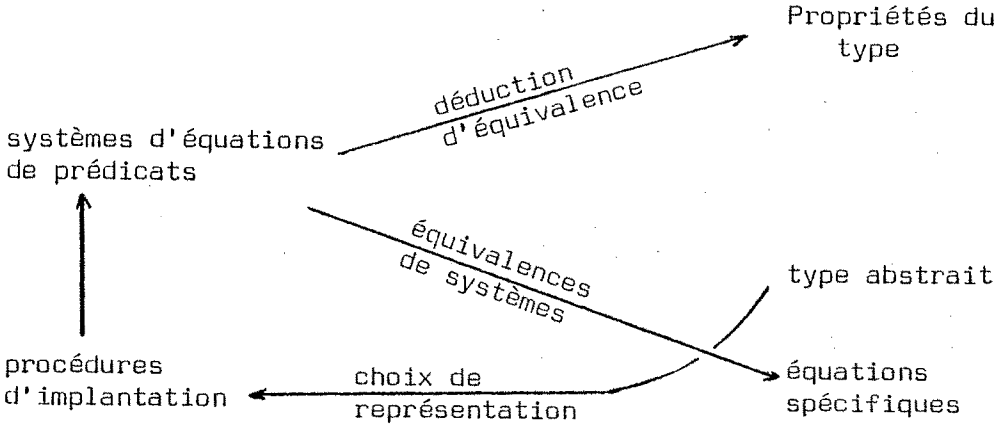


Figure 4

On ne voudrait cependant pas faire croire au lecteur que la méthode de comparaison de systèmes équationnels donne toujours de bons résultats. Il se peut en effet que les systèmes ne soient pas égaux et que la preuve de l'équivalence

des solutions exige des techniques plus complexes ou des démonstrations particulières. Donnons un exemple qui, quoique simple, ne se démontre pas facilement.

Soit le programme de la multiplication des naturels :

```
proc * (x, y : nat) → nat ;
    a, b : nat ;
début assert P1 (a, b, x, y) ;
    a := 0 ; b := y ;
    faire assert P2(a, b, x, y) ;
        si b = 0 alors sortie fsi ;
        a := a + x ;
        b := b - 1
    ffaire ;
    assert P3(a, x, y) ;
    retourner (a)
fin ;
```

Les relations entre les prédicats sont :

$P3(a, x, y) \Leftrightarrow P2(a, 0, x, y)$

$P2(a, b, x, y) \Leftrightarrow P2(a+x, b-1, x, y)$

$P2(0, y, x, y) \Leftrightarrow vrai$

Dans le type abstrait nat, on aura la spécification :

$m * 0 = 0$

$m * (n+1) = m * n + m$

Ce qui, sous forme d'équations de prédicats, est :

$M(0, m, 0) \Leftrightarrow vrai$

$M(res+n, m, succ(n)) \Leftrightarrow M(res, m, n)$

On veut démontrer que M et P3 caractérisent les mêmes domaines, c'est-à-dire : $M(res, x, y) \Leftrightarrow x * y \equiv res \Leftrightarrow P3(res, x, y)$, mais malheureusement, la comparaison des systèmes d'équations ne suffit pas ici.

3.3.3. Equivalence de programmes

Dans cette dernière partie, on évoque une dernière possibilité, qui est de vérifier l'équivalence des équations dans le domaine des programmes eux-mêmes. Comme pour les systèmes d'équations, si les structures des programmes sont équivalentes, alors les résultats seront équivalents. On voit que cette condition est évidemment suffisante pour démontrer une équation, mais qu'elle n'est pas nécessaire et qu'il est possible que des démonstrations n'aboutissent pas par cette voie. De plus, ces démonstrations reposent sur des transformations de programmes, et si les propriétés des programmes commencent à être répertoriées [ARS79][LOV77], elles ne constituent pas un système formel.
Enfin, il ne suffit pas de transformer les schémas de programmes, il faut tenir compte aussi de l'interprétation des opérations qui sont dans le schéma.
Une autre manière, qui se rapproche de § 3.3.2., est de remplacer les programmes par leur correspondant en sémantique dénotationnelle, et de montrer que les fonctions calculées sont identiques. La composition des fonctions est peut-être plus aisée dans ce cas que dans celui des programmes. Donnons deux exemples qui illustreront la méthode, mais nous ne la détaillerons pas. Les transformations que l'on utilisera sur les programmes seront toutes assez simples.

1 - Démonstration de sommet (empiler(p, x)) = x
Les programmes sont ceux du §3.2.2. Comme empiler modifie son paramètre, on doit résoudre :

{empiler (p, x) ; sommet(p)} = x

⇒ {p.s := p.s+1 ; p.tab(p.s) := x ;

si p.s = 0 alors non-défini

sinon retourner (p.tab (p.s)) fsi } = x

⇒ par la première affectation, la condition est fausse, et par la deuxième affectation, on a bien p.tab(p.s) = x.

2 - Démonstration de m* (n + 1) = m* n + m
Utilisons le programme du §3.3.2. ; le programme de la partie gauche est :

```
(ppg1) début  a := 0 ; b := n + 1 ;
       |  faire  si  b = 0  alors  sortie  fsi ;
       |           a := a + m ; b := b - 1
       |  ffaire ;
       |  retourner (a)
       fin ;
```

Le programme de la partie droite est :

```
(ppd) début  a := 0 ; b := n ;
      |  faire  si  b = 0  alors  sortie  fsi ;
      |       a := a + m ; b := b - 1
      |  ffaire ;
      |  retourner (a+m)
      fin ;
```

On développe (ppg1) par la transformation :

faire α ffaire ≡ α ; faire α ffaire

```
(ppg2) début
       |  a := 0 ; b := n + 1 ;
       |  si  b = 0  alors  retourner (0)  fsi ;
       |  a := a + m ; b := b - 1 ;
       |  faire  si  b = 0  alors  sortie  fsi ;
       |     |  a := a + m ;  b := b - 1
       |  ffaire
       |  retourner (a)
       fin ;
```

```
début  a := 0 ;
   |     a := a + m ; b := n ;
   |     faire  si  b = 0  alors  sortie  fsi ;
   |          a := a + m ; b := b - 1
   |     ffaire ;
   |     retourner (a)
fin ;
```

La dernière transformation est : si une instruction i est indépendante du test de sortie d'une boucle et peut commuter avec les autres instructions de la boucle, alors :

i ; faire ... i ... ffaire
≡ faire ... i ... ffaire ; i

Les instructions sur a et sur b sont indépendantes et peuvent commuter ; on peut appliquer la transformation :

```
(ppg3)  début  a := 0 ; b := n ;
               faire  si  b = 0  alors  sortie  fsi ;
                      a := a + m ; b := b - 1
               ffaire ;
               a := a + m ;
               retourner (a)
        fin ;
```

Par une dernière substitution, ppg3 ≡ ppd et le programme satisfait l'équation.

En conclusion de ce chapitre sur les vérifications, on peut dire que les méthodes de preuve d'équivalence ne manquent pas, mais que cette diversité nuit à la systématisation. La méthode du §3.2. est implantable à l'aide d'un démonstrateur de théorèmes pour preuve de programmes, mais on connaît bien ses inconvénients :

- la spécification doit être complète ;
- la complexité de la preuve est exponentielle.

La dernière méthode n'est guère mieux praticable, car le nombre de transformations possibles (axiomes du système), est arbitrairement grand, et beaucoup d'entre elles peuvent être appliquées à un programme donné, ce qui rend l'arbre des dérivations parfaitement incontrolable. C'est un peu la même critique que l'on peut faire à la méthode de transformation de systèmes d'équations de prédicats, à moins que celui-ci ne soit dirigé par l'utilisateur comme en [BUR77a].

De toutes façons, ces méthodes sont très liées, et on peut rappeler la remarque qui a été faite à propos des vérifications de programmes : si le programme et la spécification algébrique procèdent de la même invention, la preuve sera aisée par n'importe quelle méthode, sinon ce n'est pas la méthode qui permettra de trouver une preuve élégante pour des programmes mal faits ou des spécifications très éloignées des programmes.

4 - CONCLUSION

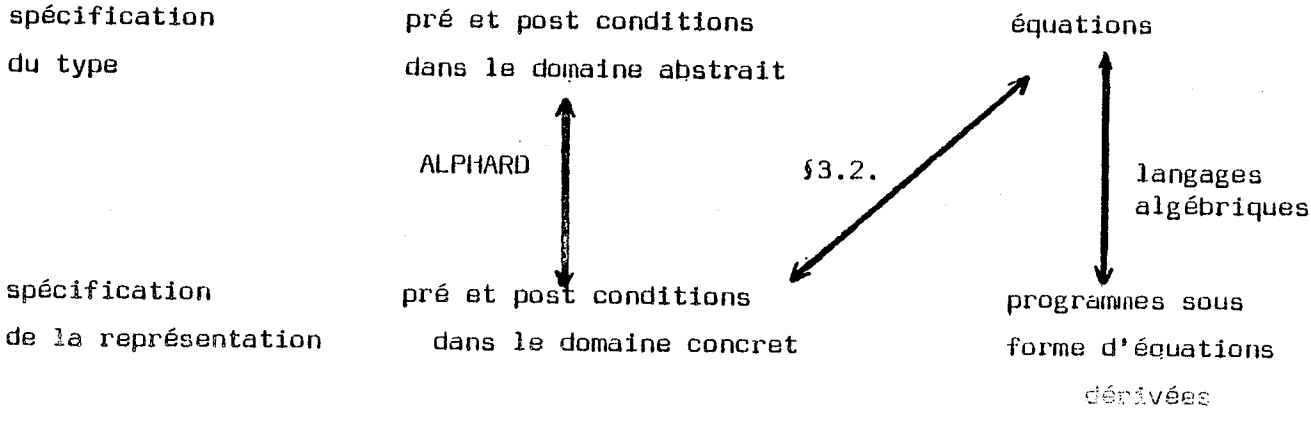
Il est bon, en terminant une thèse de revenir sur le thème de celle-ci pour faire le point. Notre but était d'apporter une contribution aux langages de types abstraits, afin d'améliorer la construction et la vérification des programmes dans le cadre générique. Pour cela, il y avait plusieurs études à entreprendre :

1 - la conception générale du langage bâti autour de la notion de types et modules génériques ; c'est ce que nous avons exposé au ch.1.

2 - les méthodes de vérification statique des définitions de types et de programmes génériques et les problèmes de compilation des fonctions génériques (sans macro-substitution), autrement dit, la mise en oeuvre. Cela a été simplement évoqué ici, mais a été présenté en [BER78][JAC78a].

3 - la spécification ; comme on l'a vu dans l'introduction, on ne peut améliorer la sécurité des gros programmes que si l'on sait en exprimer les effets par une spécification statique. On a choisi la spécification algébrique, car elle s'adapte bien aux types génériques et elle est une manière de construire les axiomes pour chaque niveau d'abstraction. Ce faisant, on n'exclut pas la spécification axiomatique par pré et post-conditions qui peut toujours être utilisée dans les procédures construites à partir d'un ensemble de types spécifiés algébriquement.

4 - la vérification des programmes ; notre travail au ch.3 a consisté à faire la vérification des axiomes d'un type, étant donné un module de représentation des fonctions d'accès du type. Par là, on a fait le lien entre les spécifications par pré et post conditions des procédures d'implantation et une spécification abstraite sous forme d'équations. Le schéma ci-dessous permet de situer notre approche par rapport aux autres :



5 - la méthodologie de la programmation ; il est bien connu que faire une preuve de programme a posteriori est une entreprise difficile et souvent vouée à l'échec. Il est préférable de construire les programmes d'après les spécifications et d'assurer la validité à chaque pas de transformation. Bien que nous n'ayons pas développé ce point, excepté pour les itérateurs, la déduction de programmes à partir des spécifications, associée à la décomposition descendante, apparaît assez naturellement dans le travail présenté.

Cela nous amène à parler des prolongements possibles de cette thèse. Ils sont à conduire dans deux directions : l'une d'elles est plutôt théorique et consiste à compléter quelques points :

- la sémantique générique des programmes ; ce point est à étudier en liaison avec la sémantique dénotationnelle "abstraite" où les transformations élémentaires ne portent plus sur un état mais sur une structure d'information.

- les vérifications des équations à l'aide des systèmes de prédicats associés aux programmes ; nous avons donné un aperçu de cette méthode qui devrait être améliorée.

L'autre direction de développement est plus pratique. Il s'agit avant tout d'une implantation d'un langage de programmation générique qui permettra :
- de faire des études sur la compilation de programmes génériques, particulièrement sur l'optimisation ;

- d'évaluer expérimentalement la méthodologie proposée et l'utilisation de la spécification algébrique.

On souhaite avoir les moyens de mener à bien cette tâche afin de fournir une alternative aux langages de spécification comme CLEAR ou aux langages qui ne supportent pas de spécification comme CLU.

BIBLIOGRAPHIE

[ADJ73] GOGUEN J.A., THATCHER J.W., WAGNER E.G., WRIGHT J.B.
"A junction between computer science and category theory: I, Basic definitions and examples", Part1, IBM Research report RC-4526 Septembre 1973.

[ADJ76] GOGUEN J.A., THATCHER J.W., WAGNER E.G.
"An initial algebra approach to the specification, correctness and implementation of abstract data types", IBM Research report RC.6487, Octobre 1976.

[ADJ77a] THATCHER J.W., WAGNER E.G., WRIGHT J.B.
"Specification of abstract data types using conditional axioms" IBM T.J. Watson Research center, Yorktown Heights, New York, 1977.

[ADJ77b] GOGUEN J.A., THATCHER J.W., WAGNER E.G., WRIGHT J.B.
"Initial algebra semantics and continuous algebras", JACM 24, 1977, pp. 68/95.

[ADJ78a] THATCHER J.W., WAGNER E.G., WRIGHT J.B.
"Data type specification: parameterization and the power of specification techniques", Proc. 10th SIGACT Symposium on theory of computing, San Diego, California, 1978.

[ADJ78b] THATCHER J.W., WAGNER E.G., WRIGHT J.B.
"Notes on algebraic fundamentals for theoretical computer science" 3rd Advanced course on Foundations of Computer Science, Amsterdam, Aout 1978.

[ADJ78c] WAGNER E.G., THATCHER J.W., WRIGHT J.B.
"Programming languages as mathematical objects", Proc. 7th Symposium on mathematical foundations of computer science 1978, Lecture notes in Computer Science n°64, Springer Verlag.

[ARS77] ARSAC J.
"La construction de programmes structurés"
Dunod, Paris, 1977.

[ARS79] ARSAC J.
"Syntactic source to source transforms and program manipulation"
C.ACM, 22,1, 1979, pp. 43/53.

[BEL79] BELLEGARDE F., FINANCE J.P., HUC B., JARAY J.
"Quelques prolongements de la méthode de programmation déductive"
GROPLAN n°6, janvier 1979, pp. 13/25.

[BER73] BERT D.
"Etude d'éléments fondamentaux des langages de programmation"
Thèse de 3ème cycle, U. Grenoble 1, 1973.

[BER77a] BERT D., JACQUET P.
"Generic abstract data types"
Proc. 5th Annual III Conference, WG 2.1. IFIP, mai 1977.

[BER77b] BERT D., JACQUET P.
"Types abstraits génériques et multireprésentation"
Bulletin n°2 GROPLAN, AFCET TTI, 1977, pp. 41/47.

[BER77c] BERT D.
"Problem specification and algorithmic programming"
ICS Liège, 1977, ed. Morlet Ribbens, North Holland Pub.Co.,
pp. 111/118.

[BER78] BERT D., JACQUET P.
"Some validation problems with parameterized types and generic functions", 3° Colloque international sur la programmation, Institut de Programmation, PARIS, 1978, DUNOD, (Recherche)

[BIR70] BIRKHOFF G., LIPSON J.D.
"Heterogeneous algebras"
J.Combinatorial theory, vol.8, n°1, 1970, pp.115/133.

[BUR77a] BURSTALL R., DARLINGTON J.
"A tranformation system for developping programs"
JACM 24, 1977, pp. 44/67.

[BUR77b] BURSTALL R.M., GOGUEN J.A.
"Putting theories together to make specifications"
Proc. Fifth int. Joint Conf. on artificial intelligence
MIT, Cambridge, Massachussetts, pp. 1045/1058.

[COH65] COHN P.M.,
"Universal algebra"
Harper and Row, New York, 1965

[COU77] COUSOT P., COUSOT R.
"Automatic synthesis of optimal invariant assertions: mathematical foundations", Proc. of the ACM Symp. on artificial intelligence and programming languages, SIGPLAN Notices 12, 8, 1977, pp. 1/12.

[COU78a] COUSOT P.
"Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes", Thèse, Univ. Grenoble 1, 1978.

[COU78b] COURCELLE B., NIVAT M.
"The algebraic semantics of recursive program schemes"
Proc. 7th symposium on mathematical foudations of computer science, 1978, Lecture notes in Comp. Sciences, n°64, Springer-Verlag.

[DAH77] DAHL O.J.
"Can program proving be made practical?", Les Fondements de la programmation, Cours de la Commission des Communautés européennes, IRIA/SEFI/Formation, 1977, pp. 57/114.

[DAR73] DARLINGTON J., BURSTALL R.
"A system which automatically improves programs"
Third IJCAI, SRI, 1973, pp. 537/542.

[DAR78] DARRINGER J.A., LAVENTHAL M.S.
"A study of the use of abstractions in program specification and verification"
IBM Research, RC 7184 , juin 1978

[DER75] DERSHOWITZ N., MANNA Z.
"On automating structured programming"
Conf. Rec. on proving and improving programs, 1975, Colloques IRIA, Arc et Senans, France, pp. 167/193.

[DIJ76] DIJKSTRA E.
"A discipline of programming"
Prentice Hall, 1976.

[DOD78] Department of Defense, USA
"Steelman - Requirements for high order computer"
Programming languages, juin 1978.

[EHR78] EHRICH H.D.
"Extensions and implementations of abstract data type specifications"
Proc. 7th symposium on mathematical foundations of computer science, 1978, Lecture notes in Computer science, n°64, Springer Verlag.

[EMD76] Van EMDEN M.H., KOWALSKI R.A.
"The semantics of predicate logic as a programming language"
J.ACM, 23, 4, octobre 1976, pp. 733/742.

[FIN78] FINANCE J.P.
"De la spécification abstraite d'une donnée à sa représentation en mémoire: les états successifs d'une information"
Théorie et techniques de l'informatique. Actes du Congrès AFCET, novembre 1978, pp. 423/433.

[FLO67] FLOYD R.W.
"Assigning meanings to programs"
Proc. Symp. in applied mathematics, vol.19, Amer. Math. Soc., 1967, pp. 19/32.

[GAU78] GAUDEL M.C., TERRINE G.
"Synthèse de la représentation d'un type abstrait par des types concrets", Congrès AFCET TTI, novembre 1978, pp. 434/445.

[GOG77a] GOGUEN J.A., TARDO J.
"OBJ-0 Preliminary users manual"
Semantics and theory of computation report n°10, UCLA Computer science dept.

[GOG77b] GOGUEN J.A.
"Abstract errors for abstract data types"
Proc. IFIP Working conference on formal description of programming concepts, MIT, 1977.

[GOG78] GOGUEN J.A., BURSTALL R.M.
"Some fundamental properties of algebraic theories: a tool for semantics of computation", DAI Research report n°53, Dept. of artificial intelligence, University of Edinburgh.

[GRE79] GREEN Reference Manual
CII HB - Louveciennes, 15 mars 1979.

[GUT76a] GUTTAG J.V., HOROWITZ E., MUSSER D.R.
"Abstract data types and software validation", University of Southern California, report ISI/RR 76 48, aout 1978.

[GUT78] GUTTAG J.V., HORNING J.J.
"The algebraic specification of abstract data types", Acta Informatica 10, pp. 27/52, 1978.

[GUT78a] GUTTAG J., STAUNSTRUP J.
"Algebraic axioms, classes, and program verification"
Computer science dpt., University of Southern California,
report n° 213 - 741 - 6288, mars 1978.

[HOA69] HOARE C.A.R.
"An axiomatic basis for computer programming"
C ACM, 12, 10, octobre 1969, pp. 576-580/583.

[HOA72] HOARE C.A.R.
"Proof of correctness of data representations"
Acta Informatica, 1, 4, 1972, pp. 271/281.

[HIG63] HIGGINS P.J.
"Algebras with a scheme of operators"
Math. Nachr 27, 1963, pp. 115/132.

[HUC77] HUC B.
"Mise en oeuvre de la méthode de programmation déductive"
Thèse, Université de Nancy I, 1977.

[JAC78a] JACQUET P.
"Les types génériques - Propositions pour un mécanisme d'abstraction dans les langages de programmation", thèse de 3ème cycle, USM Grenoble, 1978.

[JAC78b] JACQUET P.
"La généricité comme outil d'abstraction dans les langages de programmation", Congrès AFCET, Panorama de l'Informatique, novembre 1978.

[JOR72] JORRAND PH., BERT D.
"On some basic concepts for extensible programming languages"
ICS, Venise, 1972 , pp. 2/16.

[KOW74] KOWALSKI R.
"Predicate logic as programming language"
IFIP 74, North-Holland Pub. Co., pp. 569/574.

[KOT78] KOTT L.
"About a transformation system: a theoretical study"
3ème colloque international sur la programmation, mars 1978,
Transformations de programmes, Dunod, pp. 232/247.

[KRI78] KRIEG-BRUCKNER B.
"Concrete and abstract specification, modularization and program development by transformation", TUM. INFO-7805, Munchen, 1978.

[LAM77] LAMPSON B.W., HORNING J.J., LONDON R.L., MITCHELL J.G., POPEK G.L.
"Report on the programming language Euclid"
SIGPLAN Notices, vol. 12, n°2, 1977.

[LIN74] LINSEY C.H.
"Notes on Modals"
ALGOL Bulletin, 37.4.3., 1974.

[LIS74] LISKOV B., ZILLES S.
"Programming with abstract data types"
Proc. of ACM - SIGPLAN Conference on very high level languages
SIGPLAN Notices, 9.4., 1974, pp. 50/59.

[LIS75] LISKOV B.
"An introduction to CLU"
New directions in algorithmic languages, 1975
IFIP WG 2.1., ed. S.A.Schumann, IRIA.

[LIS77] LISKOV B., SNYDER A., ATKINSON R., SCHAFFERT C.
"Abstraction mechanisms in CLU"
C.ACM, vol.20, n°8, 1977, pp. 564/576.

[LOV77] LOVEMAN D.
"Program improvement by source-to-source transformation"
J.ACM, 24.1, 1977.

[MAN71] MANNA Z., WALDINGER R.
"Toward automatic program synthesis"
C.ACM 14.3., pp. 151/165, 1971.

[MIT77] MITCHELL J.G., WEGBREIT B.
"Schemes: a high level data structuring concept"
CSL 77.1, janvier 1977, Xerox, Palo Alto Research center.

[MOR73] MORRIS J.H.
"Types are not sets"
Proc. ACM Symposium on principles of programming languages,
Boston, 1973, pp. 120/124.

[NAK77] NAKAJIMA R., HONDA M., NAKAHARA M.
"Describing and verifying programs with abstract data types"
Proc. of IFIP working conference on the formal description of
programming concepts, MIT, 1977.

[NIV75] NIVAT M.
"On the interpretation of polyadic recursive schemes"
Symposia Mathematica, Vol.15, Academic Press, 1975.

[PAI74] PAIR C.
"Formalisation of the notions of data, information and information
structure in data base management systems", Klimbie-Koffman eds.,
North Holland Pub. Co., 1974, pp. 149/167.

[PAI78] PAIR C.
"La programmation : de l'énoncé au programme", Théorie et techniques
de l'Informatique, Congrès AFCET, novembre 1978, pp. 325/337.

[PAR72] PARNAS D.L.
"A technique for the specification of software modules with examples"
Com. ACM, 15, 1972, pp. 330/336.

[PEY78] PEYRIN J.P., DELAUNAY M.
"Conception d'algorithmes et langages"
RR 120, Laboratoire IMAG, 1978.

[RED79] RED Reference Manual
Intermetrics, mars 1979.

[REM74] REMY J.L.
"Structure d'information, formalisation des notions d'accès et de modification d'une donnée"
Thèse, Université Nancy I, 1974.

[ROU72] ROUSSEL P.
"Définition et traitement de l'égalité formelle en démonstration automatique", Thèse, UER de Luminy, 1972.

[SCO76] SCOTT D.
"Data types as lattices"
SIAM J. on computing, 5, n°3, pp. 522/587, 1976.

[SHA76a] SHAW M., WULF W., LONDON R.L.
"Abstraction and verification in ALPHARD: iteration and generators"
Carnegie-Mellon University, Technical report, 1976.

[SHA76b] SHAW M.
"Abstraction and verification in ALPHARD: design and verification of a tree handler", Carnegie Mellon University, 1976.

[SOL78] SOLOMON M.
"Type definitions with parameters"
Conf. Rec. on the 5th ACM symposium on principles of programming languages, 1978.

[SPI75] SPITZEN J., WEIGBREIT B.
"The verification and synthesis of data structures"
Acta Informatica, 4.2., 1975, pp. 127/144.

[TEN77] TENNENT R.D.
"On a new approach to representation independent data classes"
Acta Informatica, vol.8, fasc.4, 1977, pp. 315/324.

[VW69] Van WIJNGAARDEN E.
"Report on the algorithmic language ALGOL 68"
Num. Math. 14, 1969, pp. 79/218.

[WIR71] WIRTH N.
"Program Development by stepwise refinement"
C.ACM, 14, 4, 1971, pp. 221/227.

[WIR73] WIRTH N.
"Systematic programming, an introduction"
Prentice Hall, 1973.

[WIR76] WIRTH N.
"Algorithms + data structures = programs"
Prentice Hall, 1976.

[WUL76] WULF W.A., LONDON R.L., SHAW M.
"Abstraction and verification in ALPHARD : introduction to language and methodology"
ISI/RR.76.46, University of California, 1976.

[ZIL74] ZILLES S.N.
"Algebraic specification of data types", Project MAC. Progress, Report 11, MIT, 1974, pp. 28/52.

[ZIL79] ZILLES S.N.
"An introduction to data algebras", Draft data paper, Winter school on abstract software specifications, Copenhague, janvier 1979.