UNIVERSITE DE NANCY I

UER DE MATHEMATIQUES

SCN 79/80 B

ETUDE DE LA CONSTRUCTION DES PROGRAMMES : méthodes et langages de spécification et de résolution de problèmes

THESE

Présentée à l'UNIVERSITE DE NANCY I

Pour l'obtention du grade de

DOCTEUR ES SCIENCES MATHEMATIQUES

Par

Jean-Pierre FINANCE

Soutenue le 11 OCTOBRE 1979

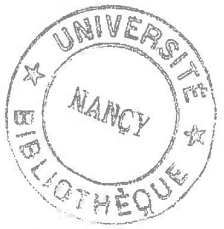
Devant la commission d'examen

RAPPORTEURS	C. PAIR	PRESIDENT
	J.P. JOUANNAUD	
	M. SINTZOFF	
EXAMINATEURS	J. ARSAC	
	J.C. DERNIAME	
	M. GRIFFITHS	
	J.P. HATON	
	M. NIVAT	

UNIVERSITE DE NANCY I

UER DE MATHEMATIQUES

ETUDE DE LA CONSTRUCTION DES PROGRAMMES : méthodes et langages de spécification et de résolution de problèmes



THESE

Présentée à l'UNIVERSITE DE NANCY I

Pour l'obtention du grade de

DOCTEUR ES SCIENCES MATHEMATIQUES

Par

Jean-Pierre FINANCE

Soutenue le 11 OCTOBRE 1979

Devant la commission d'examen

RAPPORTEURS	C. PAIR	PRESIDENT
	J.P. JOUANNAUD	
	M. SINTZOFF	
EXAMINATEURS	J. ARSAC	
	J.C. DERNIAME	
	M. GRIFFITHS	
	J.P. HATON	
	M. NIVAT	

C'est un grand plaisir pour moi de remercier ici mes collègues et amis qui ont tous, directement ou non, contribué à la réalisation de cette thèse.

C. PAIR, Président de l'INPL et Directeur du CRIN, est à la fois, l'instigateur des idées présentées dans ce travail, le guide et le conseiller de mon activité de recherche ainsi que le lecteur attentif des diverses versions de cette thèse. Sa haute valeur intellectuelle, sa lucidité et sa disponibilité malgré les nombreuses charges qu'il assure, en font un directeur de recherche hors du commun. Nous savons tous deux ce que je lui dois, aussi me contenterai-je de lui dire: merci.

M. SINTZOFF, Chercheur au Philips Research Laboratory à Bruxelles,a, lors de sa venue à Nancy il y a deux ans, apporté à notre équipe de recherche, le point de vue du chercheur professionnel, en prise directe avec les problèmes de la programmation. Sa souplesse à remettre en cause les résultats précedemment obtenus, sa recherche constante d'objectifs toujours mieux définis, sa progression rigoureuse et passionnée, m'ont montré la nécessité d'une démarche raisonnée et déductive dans l'activité de recherche. Qu'il trouve ici l'assurance de mes remerciements très sincères et amicaux.

J.P. JOUANNAUD, Professeur à Nancy II, nouvellement arrivé au sein de notre équipe, apporte à celle-ci un point de vue nouveau sur la synthèse de programmes. J'ai appris à mieux le connaître grâce aux remarques qu'il a formulées sur les versions antérieures de cette thèse: mise en évidence de concepts formels, structuration de la présentation. J'ai grand plaisir à lui adresser ici mes vifs remerciements pour la part active qu'il a eue dans ce travail.

J. ARSAC, Professeur à l'Institut de Programmation de Paris, a bien voulu s'intéresser à ce travail. Il est vrai que les idées émises dans cette thèse, notamment celles concernant la construction et la transformation d'énoncés, sont les siennes depuis longtemps déjà. Je tiens à le remercier pour les discussions que nous avons eues et pour les conseils qu'il m'a donnés, en particulier dans le cadre du

groupe "ANNAGRAMME". Je lui suis également reconnaissant d'avoir bien voulu participer à ce jury auquel il apporte, comme il se plait à le souligner, le point de vue du praticien.

J.C. DERNIAME, Professeur à Nancy I, m'a initié, il y a plus de dix ans déjà, aux joies des langages algorithmiques (!). Depuis cette époque, nous avons eu de nombreuses occasions de collaborer, tant dans le cadre de la recherche que dans celui de l'enseignement. En particulier, les récents travaux de son équipe sur les types abstraits, rapprochent grandement nos objectifs de recherche. Je le remercie sincèrement d'avoir bien voulu juger cette thèse.

M. GRIFFITHS, Professeur à l'IUT de Nancy, se préoccupe lui aussi, de problèmes de conception de logiciel. Depuis son arrivée à Nancy, maintes discussions nous ont permis de mieux nous connaître. Nous avons pu travailler ensemble, mais de manière trop partielle, dans le cadre du projet MEFIA; aussi souhaiterai-je pouvoir profiter de son expérience et de ses recherches en production de gros logiciel pour valider certaines des idées et méthodes développées dans cette thèse. Qu'il soit assuré de mes remerciements amicaux.

J.P. HATON, Professeur à Nancy I, a su, en quelques années, développer une des équipes de recherche les plus actives et efficaces du CRIN. Il apporte dans ce jury le point de vue de l'utilisateur de méthodes de programmation. Il est fondamental que nous expérimentions sur des problèmes effectifs, les différentes idées présentées dans ce travail et plus généralement au sein du groupe de recherche CASTOR. La collaboration que nous entreprenons cette année est une première étape dans cette voie. Merci à lui de participer à ce jury.

M. NIVAT, Professeur à Paris VI, s'est, depuis ma thèse de troisième cycle, intéressé à mon travail de recherche. Il m'a, à plusieurs reprises, conseillé sur certains problèmes, notamment à propos de la nécessité de dégager clairement la démarche suivie lors de la construction des programmes. L'originalité de ses travaux sur la sémantique en font un chercheur et un animateur de recherche de réputation internationale. Aussi est-ce un grand plaisir pour moi que de le voir siéger à ce jury.

Ces remerciements s'adressent également à tous mes amis du groupe CASTOR pour leur activité, leurs idées et leurs conseils sans lesquels ce travail n'aurait certainement pas vu le jour. Parmi eux, je tiens à remercier A. QUERE, P. LESCANNE et J.L. REMY avec lesquels j'ai plus particulièrement travaillé.

Ce travail de recherche m'a également permis de collaborer avec de nombreux chercheurs du CRIN que je remercie ici et en particulier, P. MARCHAND, M. QUERE et tous mes collègues de l'IUT.

Ces remerciements ne seraient pas complets si je n'évoquais pas ici les membres des groupes "GERBIER", "GROPLAN" et "ANNAGRAMME" de l'AFCET avec lesquels j'ai pu avoir d'enrichissantes discussions. Enfin je remercie M.C. GAUDEL et G. TERRINE de l'IRIA pour les nombreux échanges de vue que nous avons eus.

Je ne voudrais pas terminer sans remercier vivement M. TESOLIN qui a assuré dans des délais record la dactylographie de la plus grande partie de cette thèse ainsi que L. BAURAIN, N. GAUTIER et D. MARCHAND qui y ont participé.

RESUME

Les deux objectifs de ce travail sont :

L'étude d'une *théorie de la construction des programmes*

La description à partir de cette étude, de méthodes, langages et outils de construction de programmes.

Nous nous sommes limité à ce que nous considérons comme les deux premières étapes, les moins étudiées jusqu'à présent, d'une démarche de construction de programmes :

- Première spécification précise d'un problème. Il est indispensable qu'elle soit structurée, il est utile qu'elle soit formelle. Aussi avons-nous été conduit à définir un cadre formel de description de Problème fondé sur le calcul des prédicats du premier ordre typé. Ce cadre rend compte de l'*univers* du problème.

L'énoncé du problème est alors défini comme une formule de son univers. L'étude de cette formalisation des problèmes nous a permis de dégager une méthode de spécification "déductive" (On commence à spécifier le résultat, ce qui introduit des intermédiaires (objets, fonctions, univers) qui doivent être à leur tour spécifiés) sous forme d'un "méta-algorithme de spécification"

- Première explicitation du problème : on se restreint aux solutions calculables d'un problème, c'est-à-dire aux solutions u pour lesquelles on peut démontrer un théorème r ≡ u (r est le résultat). Ceci implique une forme particulière de l'univers et de l'énoncé du problème. L'univers doit être constructif : les types abstraits algébriques répondent à ce souhait. L'énoncé doit être explicite : récurrent ou récursif. A partir de l'étude de ces concepts, nous esquissons une méthode d'explicitation permettant de passer d'une spécification initiale à un couple (univers spécifié algébriquement, énoncé explicite). Cette méthode peut se décomposer en un certain nombre de *stratégies d'explicitation* plus locales, fondées sur des *règles de construction*. Nous montrons enfin qu'il est utile de disposer d'un langage permettant de décrire la suite de choix et de raisonnements mis en oeuvre lors d'une explicitation, c'est-à-dire de décrire les calculs d'un méta-algorithme d'explicitation.

SOMMAIRE

SOMMAIRE

INTRODUCTION ..... p. 1

1. La Construction des Programmes 1
2. Le Concept de Programme 2
3. Objectifs et Idées Générales de ce Travail 2
4. Première Analyse et Détermination de Sous Objectifs 4
   4.1. Etude de la Notion de Problème 4
   4.2. Etude de la Construction des Programmes 5
5. Contexte de ce Travail 7
6. Plan de cette Thèse 8
   6.1. Première Partie 8
   6.2. Deuxième Partie 9

PREMIERE PARTIE : Etude et Spécification des Problèmes

CHAPITRE 1.1. Problème de la spécification des problèmes 11

1. Enoncé et univers d'un problème 11
   1.1. Exemples de problèmes 11
   1.2. Classes de problèmes informatiques 16
2. Spécification des problèmes 18
   2.1. Qu'est-ce-que spécifier un problème ? 19
   2.2. Forme et niveau de détail du premier énoncé d'un problème .. 21
   2.3. Vers une méthode de spécification de problèmes 28
3. Langages d'expression d'énoncés formels 31
   3.1. Langage ensembliste et prédicats 31
   3.2. Enrichissement du langage ensembliste 39
   3.3. Utilisation de l'induction 44
4. Qualités d'un cadre de spécification de problèmes 46
   4.1. Qualités de la première spécification d'un problème 46

4.2. Qualités d'un cadre de spécification de problème 51
5. Sur les langages de spécification de problème 55
5.1. Introduction
5.2. Quelques langages de spécification généraux..... 55
5.3. Langages de spécification adaptés à un domaine particulier 70
5.4. En résumé: un essai de comparaison de quelques langages . 73

CHAPITRE 1.2. Spécification Formelle de l'univers d'un Problème: les systèmes d'information 77

1. De la formalisation de l'univers d'un problème 78
1.1. Utilité des systèmes formels..... 78
1.2. Exemples de systèmes d'information 80
1.3. Systèmes d'information et types abstraits 87
2. Définition d'un système d'information 90
2.1. Eléments d'un système d'information 90
2.2. Composantes caractéristiques d'un système d'information .. 96
2.3. Conventions de notation 97
3. Propriétés fondamentales des systèmes d'information 99
3.1. Quelques théorèmes généraux sur les systèmes d'information 99
3.2. Sémantique d'un système d'information 100
3.3. Consistance et complétude 102
4. Transformation de systèmes d'information 105
4.1. Réalisation d'un système d'information 105
4.2. Extension et restriction d'un système d'information 106
5. Description de systèmes d'information 113
5.1. Opérations de transformations simples de systèmes d'information 114
5.2. s-description d'un système d'information 123
5.3. Systèmes d'information primitifs 127
6. Systèmes d'information paramétrés 129
6.1. Utilité de la paramétrisation 129
6.2. Définition et description d'un système d'information paramétré 133
6.3. Comparaison avec d'autres approches 138

7. Quelques autres opérations de construction de systèmes d'information 139
7.1. Constructions ensemblistes 141
7.2. Construction d'ensembles de fonctions 145
7.3. Un dernier exemple 151
8. En guise de conclusion 154

CHAPITRE 1.3. Enoncé Formel, Solution et Résolution d'un Problème 156

1. Notion d'énoncé de problème 158
1.1. Solution d'un problème 158
1.2. Intermédiaires dans un énoncé 159
1.3. Donnée d'un problème 162
1.4. Nature des résultats 163
2. Formalisation de la notion de problème 164
2.1. Définition d'un problème 164
2.2. Solution d'un problème 165
2.3. s-Description d'un problème 169
3. Résolution d'un problème 172
3.1. Problème de la résolution des problèmes 172
3.2. Quelques idées sur la résolution 175
4. Démonstration d'une variante de l'énoncé 179
4.1. Fondements de la méthode de résolution de ROBINSON 179
4.2. PROLOG et les limites de la méthode de résolution 184
5. Calcul direct d'une solution 187
5.1. Présentation intuitive 187
5.2. Solution calculable d'un problème 189
6. Transformation de problèmes 192
6.1. Utilité des transformations de problèmes.....192
6.2. Le problème de la transformation d'énoncés 192
6.3. Le problème de la transformation des systèmes d'information 200
7. Conclusion 218

CHAPITRE 1.4. Une Méthode de Spécification de Problèmes.....221
1. Considérations générales sur la spécification 221
1.1. Le problème de la spécification 221
1.2. Langage de spécification 225

1.3. Méthode de spécification 225
1.4. Outils de spécification 231
2. Un méta-algorithme de spécification 234
2.1. Description informelle du méta-algorithme de spécification 235
2.2. Description formelle 237
2.3. Dernier exemple de spécification 244
3. Brève conclusion 255

DEUXIEME PARTIE: Explicitation de Problèmes

CHAPITRE 2.1. Problème de l'Explicitation des Problèmes 256

1. Premières idées sur l'explicitation d'un problème 257
1.1. Forme d'une définition explicite 257
1.2. Quelques exemples d'explicitation 259
2. Etude du problème de l'explicitation des problèmes 291
2.1. Vers un méta-algorithme d'explicitation 291
2.2. Enoncés explicites et systèmes d'information constructifs298
2.3. Articulation de la suite de cette deuxième partie 298

CHAPITRE 2.2. Définitions Récursives et Description Algébrique d'un Système d'Information 299
1. Notion de définition récursive 300
1.1. Définitions s-récursives 300
1.2. Bref rappel des propriétés des systèmes d'équations récursives 302
1.3. Construction de définitions s-récursives 303
2. Notion de système d'information constructif 308
2.1. Utilité des abstractions algébriques 308
2.2. Caractéristiques d'une spécification algébrique 312
2.3. Impact méthodologique des systèmes d'information algébriques 314
3. Etude et description des systèmes d'information algébriques..322
3.1. Définitions c-récursives 322
3.2. Systèmes d'information algébriques 328

3.3. Le concept de forme normale 335
3.4. Propriétés d'un système d'information algébrique..... ...344
4. Opérations sur les descriptions algébriques.....348
4.1. Produit cartésien 349
4.2. Ensemble des parties.....349
4.3. Ensemble des fonctions.....353
4.4. Ensemble des suites 353
5. Abstraction algébrique.....353
5.1. Définition et propriétés d'une abstraction algébrique....353
5.2. Construction d'une abstraction algébrique.....356
6. Représentation d'un système algébrique.....362
6.1. Construction de représentations généralisées.....362
6.2. Construction de représentations faibles.....365
6.3. Méthode de construction d'une représentation.....368

CHAPITRE 2.3. Les Enoncés Recurrents.....372

1. Etude des énoncés récurrents.....373
1.1. Définitions.....373
1.2. Propriétés des énoncés récurrents.....378
1.3. Sémantique d'un énoncé récurrent.....380
2. Le langage MEDEE et la construction d'énoncés récurrents.....381
2.1. Description sommaire de MEDEE.....381
2.2. Sémantique statique de MEDEE.....386
2.3. Construction d'énoncés MEDEE.....393
3. Définitions itératives de plus haut niveau.....399
3.1. Insuffisance des définitions itératives précédentes.....399
3.2. Définitions itératives de haut niveau.....402
3.3. Construction de définitions itératives de haut niveau....408
4. Quelques règles de transformation de définitions itératives..411
4.1. Passage d'une itération de haut niveau à une itération générale.....411
4.2. Opérations sur les définitions itératives.....413
5. Introduction de suites intermédiaires.....416
5.1. Le problème.....416
5.2. Mise en oeuvre de la stratégie d'introduction de suites intermédiaires.....424

6. Comparaison avec d'autres travaux.....437
6.1. Langages à assignation unique.....437
6.2. Construction d'algorithmes recurrents.....439
6.3. Introduction de structures intermédiaires.....439
6.4. Transformation d'énoncés.....440

CHAPITRE 2.4. Une Méthode d'Explicitation de Problèmes.....441
1. Quelques autres statégies d'explicitation.....443
1.1. Induction sur les données et induction sur les résultats443
1.2. Généralisation.....443
1.3. Analyse récurrente.....444
1.4. Stratégies plus ponctuelles.....445
2. Méta-algorithme d'explicitation.....446
2.1. Description du méta-algorithme.....446
2.2. Description de la construction d'un énoncé explicite....446
3. Dernier exemple d'explicitation.....451

CONCLUSION.....462

1. Bilan de ce travail.....463
2. Prolongements et approfondissements.....463
2.1. Etapes suivantes de la construction.....463
2.2. Approfondissements théoriques.....464
2.3. Approfondissements méthodologiques.....465

BIBLIOGRAPHIE.....466

INTRODUCTION

PREMIERE PARTIE

étude
et spécification
des problèmes

INTRODUCTION

1.- LA CONSTRUCTION DES PROGRAMMES

Au risque de la caricaturer, l'activité de l'informaticien peut être résumée par la formule suivante :

Construire et faire vivre des programmes, ou des ensembles de programmes, efficaces, résolvant un problème donné.

Par vie d'un programme on entend généralement les activités de vérification, de maintenance, de modification. Elles dépendent fondamentalement de la manière dont le programme a été construit.

La construction des programmes, quant à elle, s'est révélée être une activité difficile nécessitant des méthodes élaborées, des langages bien adaptés et des outils efficaces. Cette constatation a été longue à s'affirmer, comme le montre bien SCHOLL dans [SCH, 79]. Elle a eu pour conséquence l'avènement d'un certain nombre de méthodes de construction de programmes dont l'étude est une branche de l'informatique appelée *méthodologie de la programmation.*

Les principales méthodes de programmation développées jusqu'à présent [WIR,73] [WIR,76],[DAH,72],[DIJ,76], [JAC,75], reposent sur le concept de "bon programme". De manière intuitive, un bon programme est un programme *clair, structuré, modulaire, et bien documenté* .L'élaboration de ces méthodes est donc généralement le fruit de nombreuses expériences pratiques, parmi lesquelles l'expérience pédagogique a souvent joué un rôle déterminant [SCH,79].

2.- LE CONCEPT DE PROGRAMME

La conception de nouveaux langages de programmation a motivé de nombreuses études de la notion de programme :

a) études pragmatiques dans lesquelles on cherche à définir des langages de programmation répondant à certaines qualités (langages généraux, langages puissants, langages simples...) parfois contradictoires :
PL/1 [BER,71], LISP [CAR,62], ALGOL 68 [WIJ,75], PASCAL [WIR,71] etc...

b) études théoriques également, avec des objectifs divers :

- étudier la nature syntaxique des langages de programmation, l'un des objectifs étant de faciliter la construction d'analyseurs syntaxiques [GOO,74]

- étudier la sémantique des langages de programmation avec pour buts [LIV,78] :

. de faciliter la construction de compilateurs [LAU,68], [FIN,76a], [GAU,77]

. de définir des méthodes de preuves de programmes [FLO,67], [HOA, 69], [MIL,76]

. de mieux comprendre ce qu'est un programme [SCO,71], [NIV,75]

. de définir et justifier des méthodes de transformation de programme [COU,77]

- préciser le concept de programme efficace en introduisant des mesures de complexité [AHO,74]

3.- OBJECTIFS ET IDEES CENTRALES DE CE TRAVAIL

Ce travail se trouve au confluent de deux courants fondamentaux de l'informatique :

- la méthodologie de la programmation
- la théorie de la programmation

En effet, s'il est indispensable d'étudier et de valider une méthode de programmation par l'expérience et les exemples, il est aussi important d'en faire une véritable activité scientifique en la faisant reposer sur des bases théoriques solides. Ceci permet de justifier plus rigoureusement l'adéquation de la méthode à l'activité de programmation. C'est également une étape obligatoire si l'on espère élaborer des systèmes automatiques ou assistés de construction de programmes [MAN,71], [MAN,75], [BID,79].

Le premier objectif de ce travail est donc de décrire un cadre (syntaxique et sémantique) et une méthode permettant de construire des programmes.

Nous ne négligeons cependant pas complètement l'aspect "vie ultérieure des programmes" qui a de l'influence sur la forme des programmes cherchés.

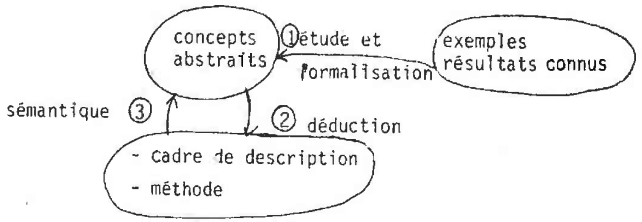
Pour atteindre cet objectif, nous étudions, en partant d'exemples et en synthétisant des résultats antérieurs, l'activité de construction de programmes.

De cette analyse nous dégageons les principaux concepts qui sont à la base de cette activité et éliminons les notions secondaires et conjoncturelles.

Ceci nous conduit à élaborer une théorie de la construction des programmes qui constitue le deuxième objectif de ce travail.

Puis, à partir de cette théorie nous déduisons un cadre et une méthode de construction de programmes.

Cette démarche consiste à partir d'exemples pour formaliser un problème afin de mieux le résoudre. Elle est fréquemment utilisée en sciences expérimentales etpeut être résumée par le schéma suivant :



La méthode de construction de programmes que nous allons dégager repose sur deux idées centrales :

- *Formaliser et structurer* l'énoncé du problème à résoudre
- Utiliser une *démarche déductive* qui consiste à partir de la spécification du résultat et à la décomposer en sous spécifications plus simples à résoudre. Cette démarche est qualifiée de *descendante* [DAH,72], [DIJ,76].

Si nous revenons à la construction de ce travail, nous constatons qu'un objectif aussi global que celui de l'étude de la construction des programmes nécessite d'être formalisé et décomposé en sous objectifs.

C'est ainsi que, suivant les conseils de C. PAIR et M. SINTZOFF. nous nous sommes efforcé d'aborder (et de résoudre en partie) le *meta problème de la construction des programmes* de la même manière que nous proposons d'aborder la résolution d'un problème (informatique) quelconque.

Il serait certainement très intéressant de comparer cette démarche à consonnance formelle, à l'approche empirique, qui est étayée par un plus grand nombre d'exemples, mais qui cependant repose sur les mêmes idées générales de décomposition et de structuration [SCH,79], [GRI,79], [DIJ,79a], [DIJ,79b], [DIJ,79c].

4.- PREMIERE ANALYSE ET DETERMINATION DE SOUS OBJECTIFS

Pour préciser la formule du paragraphe 1 il faut s'intéresser aux "mots clés" :

- *construire*
- *programme*
- *efficace*
- *problème*

Si les concepts de "programme" et de "programme efficace" ont déjà fait l'objet de nombreuses études, il n'en est pas de même de ceux de "construction" et de "problème". L'analyse de ces derniers explique la structure de cette thèse.

4.1. Etude de la notion de problème

La première étape de ce travail consiste à préciser les notions de *problème*, *solution* et *résolution*. Ceci permet ensuite de proposer un cadre de description.

Un problème est en général énoncé de manière informelle, imprécise, incomplète et ambiguë, dans une langue naturelle. Avant de chercher à le résoudre, il faut le préciser, le compléter, le rendre cohérent. Mais il est particulièrement important de le *structurer* de façon à privilégier les points fondamentaux par rapport aux détails. La *spécification précise et structurée* d'un problème constitue donc la première étape vers l'obtention d'un programme. C'est dans ce but que nous allons orienter l'étude du concept de problème et celle du langage de description qui s'en déduit. Cette première spécification doit être la plus proche possible de l'énoncé initial, à la fois pour en faciliter la construction et pour en vérifier l'adéquation. Elle sera donc plus *descriptive que constructive*, c'est-à-dire qu'elle ne permettra pas, en général, d'obtenir directement des solutions du problème. A fortiori, elle sera éloignée des programmes écrits dans un langage de programmation classique. En ce sens, nous reprenons à notre compte un certain nombre d'idées sur la nécessité de se tenir lors de la construction d'un programme, le plus longtemps possible éloigné des contraintes de la machine [BAL,76], [WIN,78]. Ainsi une première spécification est à la fois *statique* (i.e indépendante des notions dynamiques de variable informatique et d'exécution de programme) et *descriptive* (i.e. qu'elle ne fournit pas directement de solution du problème considéré). De ces propriétés nous cherchons à déduire un *langage* puis une *méthode de spécification*.

Paradoxalement, les informaticiens semblent n'avoir pris conscience que depuis peu de la nécessité de construire une "bonne" spécification : la conception de méthodes et de langages de spécification n'en est qu'à ses balbutiements [DEM,79], [LAM,79]. Il est donc très important de motiver et de convaincre les programmeurs de l'utilité de cette première étape. On retrouve, au niveau de la spécification le même contexte humain et la même nécessité de convaincre que ceux qu' ont connus (et que connaissent encore !) les pionniers de la programmation structurée [WIR,73], [COU,74], [DIJ,76], [PAI,79], [GAL,79].

4.2. - Etude de la construction des programmes

D'une première spécification, structurée et précise, il s'agit maintenant de dériver un programme efficace. Mais la distance est grande entre la spécification initiale, descriptive et statique, et le programme final, constructif, dynamique et efficace. Aussi est-il raisonnable de déterminer des étapes intermédiaires. La différence, peut être la plus importante, entre une spécification formelle initiale et un programme, réside en l'aspect constructif du second. D'où l'idée de chercher, dans un premier temps, à transformer la première spécification d'un problème en un énoncé constructif mais toujpurs statique, éventuellement peu efficace. Nous décomposons donc l'objectif "construction de programmes" en deux sous-objectifs :

- Elaboration d'unpremier énoncé constructif, nous dirons plutôt énoncé explicite, du problème
- Dérivation, à partir de cet énoncé, d'un programme efficace.

4.2.1. - Forme et construction d'un premier énoncé explicite

Cherchons tout d'abord à définir ce qu'est un énoncé explicite. Pour ce faire nous sommes guidés par les propriétés attendues d'un tel énoncé, à savoir la possibilité de *calculer* des solutions. Nous partons donc des notions de *calcul* et de *solution calculable*, ce qui permet:

- de définir la notion d'énoncé explicite et d'en étudier les propriétés
- de déduire de l'étude précédente un langage de description de tels énoncés
- de déduire de l'étude et du langage une méthode de construction d'énoncés explicites.

4.2.2. - Amélioration d'un énoncé explicite et passage au niveau dynamique

Pour diverses raisons, l'énoncé explicite obtenu à l'étape précédente n'est pas un programme efficace :

- Les "objets" apparaissant dans cet énoncé sont "abstraits" au sens où ils expriment des propriétés logiques du contexte du problème. Par exemple, on manipule à ce niveau des objets mathématiques tels que des des ensembles, des fonctions, des suites etc...,et ces objets ne sont pas toujours exprimables directement en termes de structures de données d'un langage de programmation. - L'énoncé peut avoir un caractère statique. Or, un moyen d'obtenir de l'efficacité consiste, par exemple, à remplacer une suite récurrente dont on cherche le dernier terme, par une variable informatique prenant les valeurs successives de cette suite. Un autre moyen consiste à effectuer un partage d'objets structurés.

-L'énoncé comporte souvent des constructions maladroites : présence de plusieurs itérations portant sur la même liste, intermédiaires inutiles, duplication d'objets, etc...

Il s'agit donc de *transformer* le premier énoncé explicite afin de *l'améliorer*.

a) transformation d'énoncés

La transformation d'énoncés, on dit plus souvent de programmes, est l'objet de nombreuses études complémentaires :

- Représentation des objets du problème en termes d'autres objets plus proches de ceux qui sont utilisables dans un langage de programmation.Ceci a été notamment étudié dans le cadre des types abstraits [WUL,75], [LIS,75], [LES,79]. La représentation peut être effectuée en restant à un niveau statique([GUT,78a], [REM,79, [GAU,78a] par exemple). Elle peut aussi consister à passer d'un niveau statique à un niveau dynamique (cf. [FIN,78b], [GUT,78c]).

-Transformations "à structures de données constantes", de l'énoncé du problème soit par l'utilisation de règles de transformations telles que le pliage, le dépliage, la fusion d'itérations, soit par la mise en oeuvre de stratégies plus générales telles que la suppression de la récursivité. De nombreuses études se développent actuellement sur le sujet, citons par exemple les travaux d'ARSAC [ARS,77], [ARS,78], de BAUER et al [BAU,78], de BURSTALL, DARLINGTON, FEATHER [BUR,77b], [DAR,78], [FEA,79] ou de l'équipe CASTOR à Nancy [BEL,79].

b) Critères d'amélioration

Avant de chercher à transformer un énoncé il faut se demander quelles sont les qualités recherchées du résultat. En ce sens les études sur la complexité des programmes [AHO,74], [REM,78] doivent fournir des critères caractérisant l'amélioration d'un énoncé.

En conclusion, pour aborder ce problème de l'amélioration d'un énoncé explicite, il faut :

-Chercher à nouveau à le décomposer en plusieurs étapes, en distinguant en particulier les transformations au niveau statique, du passage au niveau dynamique. Ceci va conduire à concevoir des langages et des méthodes permettant ces transformations.

- Se doter de critères objectifs et si possible quantifiables, permettant de caractériser les qualités du résultat cherché. Il serait souhaitable en ce sens d'aller plus loin que la simple description de la forme syntaxique attendue [FEA,79] en se dotant d'outils permettant de *spécifier* le résultat.

Nous n'abordons ces deux points que très partiellement, sachant qu'ils sont un complément indispensable à l'étude entreprise ici.

5.- CONTEXTE DE CE TRAVAIL

Ce travail s'est en grande partie déroulé dans le cadre de l'équipe de recherche CASTOR du CRIN (Centre de Recherche en Informatique de Nancy). Sous l'impulsion de C. PAIR, se sont développées depuis plusieurs années tant des recherches sur des aspects fondamentaux de la théorie des programmes que des recherches plus méthodologiques :

- A l'origine des travaux sur la théorie des programmes se trouve une étude, à la fois didactique et formelle, du concept de structures de données [PAI,71], [PAI,74], [REM,74]. Reposant en partie sur cette étude, ont pu être esquissées d'une part, une première approche du concept de problème ([REM,74], [PAI,75]), d'autre part une méthode de définition de la sémantique d'un langage de programmation: la sémantique calculatoire [FIN,74], [FIN,76a], [FIN, 76b]. Ces études se sont poursuivies vers le comaine des types abstraits en liaison avec l'aspect méthodologique [REM,79], [FIN,78b], [LES,79]. Une conséquence de ces travaux de recherche a été la rédaction d'un ouvrage didactique sur la théorie de la programmation [LIV,78].

- Les recherches méthodologiques ont eu pour origine essentielle des problèmes pédagogiques. L'introduction, l'évaluation et l'amélioration

d'une méthode de programmation, dite déductive, ont été les premières étapes de cette recherche [PAI,75], [PAI,77], [HUC,77], [BEL,78b], [PAI,79], [FIN,79].

Dépassant le cadre didactique, les travaux sur la méthodologie ont consisté à étudier et à systématiser certains aspects de la construction d'un programme. L'introduction d'un langage de programmation non procédural, MEDEE [BEL,78 a], la définition de sa sémantique [FIN,78c], [FIN,78d], l'introduction de règles de construction et de transformation [BEL,79], [FIN,78c] complétés par l'étude d'exemples [FIN,78a], [BRO,79a] sont en grande partie à l'origine de cette thèse.

D'autres activités notamment dans le cadre de groupes de recherche de l'AFCET, et en particulier les activités du groupe ANNA GRAMME, ont permis de confronter ces idées avec des points de vue souvent différents sinon opposés [GER,77].

6.- PLAN DE CETTE THESE

Suivant les objectifs dégagés au paragraphe 4, cette thèse comporte deux parties. La première concerne l'étude de la notion de problème et de la spécification des problèmes ; la seconde porte sur l'explicitation.

6.1. - Première partie

Le chapitre 1.1. pose le problème de la spécification. En partant d'exemples nous montrons qu'une première manière de structurer un problème consiste à distinguer son *énoncé*, qui est la relation entre les données et les résultats, de l'ensemble des informations qui permettent de le comprendre et que l'on nomme *univers*.

De cette structuration nous déduisons l'articulation de la suite de la première partie : étude des concepts d'univers et d'énoncé puis description d'une méthode de spécification. Le souci de structurer la spécification conduit à une première esquisse d'une méthode de spécification, orientée par les résultats, et déductive. Ce chapitre se termine par un essai de comparaison de divers cadres utilisés actuellement pour spécifier un programme.

Le chapitre 1.2. concerne le concept d'univers. Intuitivement un univers est formé d'ensembles et de fonctions (relations) sur ces ensembles. Mais de nombreux problèmes (bases de données par exemple) portent plus sur une *classe d'univers* que sur un univers particulier. Une spécification d'univers est donc un système formel du genre calcul des prédicats typés du premier ordre, dont un univers est un modèle. Un tel système formel est appelé *système d'information*, il est proche du concept de type abstrait. Une étude et un rappel des propriétés de ces systèmes précède l'introduction d'opérations

de composition de systèmes d'information. Une étude fondamentale de ces opérations conduit à la définition d'un langage de description de systèmes d'informations bien adapté à la spécification initiale d'un univers : le langage des sous descriptions. On se dote ainsi d'outils permettant la construction *incrémentale* de systèmes d'information. L'introduction de la paramétrisation permet de se doter d'outils de construction *génériques*. Il est alors naturel, au chapitre 1.3., de définir l'énoncé d'un problème comme étant une *formule* de son système d'information. La formalisation des concepts de résultat, donnée et solution d'un problème permet de définir un langage de description d'énoncé considéré comme un prolongement du langage des descriptions. L'étude générale de la résolution des problèmes présente dans un même cadre des techniques interprétatives telles que la méthode de résolution de ROBINSON et des méthodes fondées sur la théorie de la calculabilité. Elle conduit à un certain nombre de résultats intéressants sur la théorie de la résolution de problèmes et sur celle de la représentation des systèmes d'informations.

Utilisant les résultats obtenus aux chapitres précédents et partant des langagesde spécifications proposés, le chapitre 1.4. poursuit l'étude de la spécification commencée au chapitre 1.1. Il se termine par un essai de description formelle d'une méthode, nous dirons d'un méta-algorithme, de spécification.

En résumé, cette première partie est à la fois une étude fondamentale des concepts de problème et de résolution de problème, et une tentative pour fournir un cadre et une méthode de spécification.

6.2.- Deuxième partie

Plus encore que la première, elle est dominée par un souci méthodologique. Son objectif est de définir un langage et une méthode permettant l'explicitation des problèmes.

Le chapitre 2.1 pose, à partir de quelques exemples, le meta problème de l'explicitation. Il montre la nécessité de définir des *stratégies d'explicitations*, de disposer d'un langage de description d'énoncé explicite et d'un langage de description des choix et des raisonnements successifs conduisant à un énoncé explicite.

Le chapitre 2.2 concerne l'explicitation de systèmes d'information. Il commence par un bref rappel des propriétés des définitions récursives complété par l'étude du passage d'une sous-description d'une opération à une définition récursive. Les notions de *constructeur* et de *simplificateur* conduisent alors à la définition des *systèmes d'information algébriques*.

L'étude des proprietes de ces systèmes permet de proposer un langage de description, prolongeant celui des s-descriptions.

La fin de ce chapitre est une étude de méthodes de construction de systèmes algébriques, soit par abstraction, soit par représentation d'un autre système.

Le chapitre 2.3 concerne, quant à lui, une classe particulièrement importante d'énoncés explicites : énoncés recurrents. De leurs propriétés est déduit un langage de description (MEDEE) dont la sémantique statique permet de justifier des règles de constructions. C'est sur ces règles que repose une stratégie d'explicitation très importante : *l'affaiblissement*. Enfin ce chapitre montre l'intérêt d'introduire, dans certains cas d'explicitation, des suites intermédiaires. Il propose et étudie quelques règles de transformation d'énoncés récurrents facilitant la mise en oeuvre de cette stratégie.

Il est alors possible, au chapitre 2.4, d'esquisser un méta-algorithme d'explicitation qui apparaît comme la composition de stratégies générales.

CHAPITRE 1.1.

PROBLÈME DE LA SPÉCIFICATION DES PROBLÈMES

1.- ENONCE ET UNIVERS D'UN PROBLEME.

La première description d'un problème se fait en général dans une langue naturelle en employant un style plus ou moins précis introduisant beaucoup de sous entendus, d'imprécisions voire d'incohérences. Le travail du programmeur consiste à transformer cette description informelle en un programme d'un certain langage. Il est de plus en plus admis maintenant que cette transformation doit raisonnablement se faire de manière progressive et aussi systématiquement que possible.

La première étape de la résolution d'un problème va donc consister à en donner une première spécification précise qui doit être intuitivement la plus proche possible du problème considéré. Avant de regarder sur quelques exemples en quoi peut consister une telle spécification il est utile d'analyser un peu plus le contenu d'une description informelle d'un problème. Considérons les exemples suivants :

1.1.- Exemples de problèmes

a) Trouver deux nombres premiers entre eux et proportionnels à deux nombres donnés non nuls.

b) Quelle est la longueur maximum d'une sous suite croissante extraite d'une suite finie donnée ?

c) Calculer, par simulation, les temps moyens d'attente aux trois caisses d'un supermarché.

d) Effectuer un tri topologique.

e) Trouver les références de vols et le prix du trajet aérien le plus économique entre deux villes données

f) On considère une liste de télégrammes, chaque télégramme est séparé du suivant par le mot "zzzz", deux mots consécutifs sont séparés par des blancs. On demande d'imprimer pour chaque télégramme le nombre de mots différents de "zzzz" et "stop" et dont la longueur est d'au plus douze caractères.

g) Une entreprise de confection fabrique des articles identifiés par un code et une taille. Un article est caractérisé par son type de fabrication qui permet de connaître la succession des opérations nécessaires à sa réalisation.

Pour chaque opération on connait :

- le numéro du poste de travail
- le temps par unité d'opération

A partir d'un fichier de commandes établi pour douze semaines, on désire calculer la charge hebdomadaire des postes de travail pour le trimestre suivant.

h) On dispose d'une mémoire principale et d'une mémoire secondaire; Définir un mécanisme de mémoire virtuelle permettant le changement de pages à la demande

i) "Etat d'avancement de livraisons sur les commandes en cours" :
On donne en entrée un fichier sequentiel à mettre à jour appelé cde-en-cours et un fichier "mouvement" appelé livraison. En sortie on devra obtenir un état (sur imprimante) appelé état-des-commandes et un fichier mis à jour cde-maj séquentiel et dont la description est la même que celle de cde-en-cours. Le fichier cde-en-cours est trié sur le numéro de commande, il est formé d'enregistrements de 4 types possibles :

type = 1 : renseignements sur la commande.
type = 2 : il existe un enregistrement par article commandé, les numéros d'articles sont classés par ordre croissant.
type = 3 : un enregistrement correspond à une livraison.
type = 4 : récapitulatif.

Note : pour une commande qui vient d'être enregistrée, il n'existe aucun enregistrement de type 3.

Le fichier livraison est sur cartes, il est trié par numéro de commande, numéro d'article, date de livraison croissants.
La description détaillée de ces fichiers est donnée en annexe
On demande de créer cde-maj à partir de cde-en-cours :

- en ajoutant les livraisons d'articles (enregistrement de type = 3)
- en modifiant l'enregistrement récapitulatif (de type = 4) avec impression pour chaque commande, d'un en-tête, et par article :
  - numéro d'article
  - quantité commandée
  - total des quantités livrées
  - nombre de livraisons
  - quantité restant à livrer
  - date limite de livraison

j) "Mise à jour simple de fichier" :
On désire mettre à jour un fichier PRIX, dont chaque élément est constitué d'un numéro de produit, d'un prix unitaire et de caractéristiques diverses. Dans ce but on dispose d'un fichier MAJ dont chaque élément est formé :

- d'un code de mise à jour : - m (modifier)
  - s (supprimer)
  - a (ajouter)
- d'un numéro de produit
- d'un prix
- de caractéristiques diverses

} dans le cas où le code est m ou a

On suppose que les deux fichiers sont triés par numéro de produit croissant et que dans MAJ se trouve au plus un élément de code.s ou de code a pour un produit donné

k) Trouver, dans une base de données représentant des registres d'état civil, le nombre d'aïeuls mâles, ayant eu plus de 4 enfants, d'une personne donnée.

l) Evaluer une expression arithmétique donnée sous forme post fixée

m) Ecrire un compilateur d'un langage à structure de bloc

n) Problème simplifié de remboursements d'emprunts bancaires dû à J.R. ABRIAL [ABR,78]

Une banque prête de *l'argent* à des *clients* et on demande de construire un *système gérant les remboursements*.

Un client est dit *actif* lorsqu'il a emprunté un certain montant d'argent qu'il n'a pas encore fini de rembourser.

Chaque *mois* chaque client actif effectue un *versement* qui peut être supérieur, égal ou inférieur (éventuellement nul) à une certaine *mensualité*.

Le client peut donc prendre du retard, payer exactement sa mensualité ou anticiper le remboursement.

Pour chaque client la mensualité est calculée d'après le *montant* du prêt et sa *durée* : elle est égale au quotient du montant par la différence entre la date de *fin* du prêt et la date de *début* du prêt.

Chaque client actif possède donc, chaque mois, un certain *crédit* qui est égal à la somme des *versements* qu'il a effectués depuis le début du prêt. De même il possède un *débit* qui est égal à la somme des mensualités depuis le début du prêt ou au montant total du prêt si le mois en question est postérieur à la date de fin du prêt.

Chaque mois un certain nombre de clients actifs ont *terminé* de rembourser leur emprunt. Ils ne sont donc plus actifs dès le mois suivant.

Par ailleurs, chaque mois, apparaissent de *nouveaux* clients (non actifs) qui commenceront à effectuer leur versement dès le mois suivant et seront dès lors considérés comme des clients actifs.

Enfin, chaque mois, apparaissent de *nouveaux* clients (non actifs) qui commenceront à effectuer leur versement dès le mois suivant et seront dès lors considérés comme des clients actifs.

o) Problème du baguenaudier [GER, 77]

Un baguenaudier est formé de deux parties (figure 0)

- des anneaux imbriqués et reliés à un support par une tige coulissante
- une barre mobile : la navette

Au début du jeu la navette passe à l'intérieur de tous les anneaux ; le but est, par une succession de manipulations, de la dégager complètement. Une étude expérimentale montre rapidement que chaque anneau admet deux positions par rapport à la navette :

- l'anneau est traversé par la navette (on dit qu'il est en position 1)
- l'anneau n'est pas traversé par la navette (il est en position 0).

De plus, pour passer d'une position à l'autre, on vérifie, après quelques expériences, qu'on ne peut appliquer que deux règles :

- on peut inverser la position de l'anneau 1 (le plus à gauche)
- on peut inverser la position du pième anneau à partir de la gauche à condition que les p-2 premiers soient en position 0 et que le (p-1)ème soit en position 1.

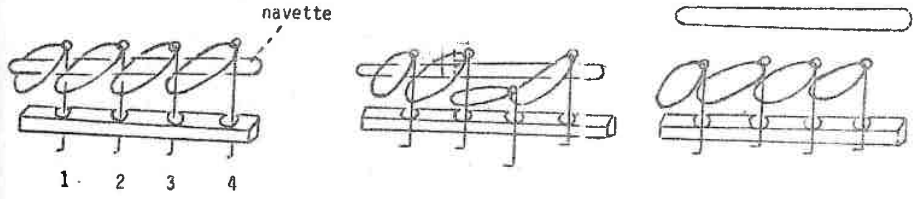


Figure 0 : état du baguenaudier

p) On cherche le nombre de mots de longueur 1, 2, 3 ... 20 d'un texte donné.

On suppose que 20 est la longueur maximum d'un mot, que deux mots sont séparés par un ou plusieurs caractères blancs et que le texte se termine par le caractère "*" précédé d'un blanc.

Ces problèmes concernent des domaines assez différents mais on remarque que leur énoncé précise presque toujours le résultat, la plupart du temps les données et qu'il introduit en outre un certain nombre de renseignements complémentaires caractérisant le *contexte* du problème. Dans le cas des problèmes scientifiques les sous entendus sont très nombreux car la plupart des mots ont un sens très précis. Dans les domaines moins formalisés il faut en revanche préciser un certain nombre de notions. Mais dans tous les cas le programmeur devra spécifier les différents concepts nécessaires à la compréhension du problème, cette spécification allant jusqu'à un certain niveau de détail dépendant des *fonctions de base* dont il dispose.
Nous appellerons *énoncé* d'un problème la relation entre les données et les résultats, *univers* du problème le contexte du problème comportant la description des ensembles, fonctions, prédicats nécessaires à la compréhension de l'énoncé. Une *solution d'un problème pour une donnée* d est un objet s tel que (d, s) vérifient l'énoncé . Par extension une *solution* d'un problème est une fonction associant à toute donnée le résultat correspondant. *Résoudre* un problème consiste donc à *expliciter* l'énoncé en une fonction. Ces différentes notions sont précisées dans la suite de cette première partie.

1.2.- Classes de problèmes informatiques

Les exemples précédents, bien que ponctuels, illustrent la diversité des problèmes que peut avoir à résoudre un informaticien. L'une des question cruciales consiste alors à se demander s'il est raisonnable de considérer tous ces problèmes de la même manière ou si au contraire on ne peut pas les regrouper en grandes classes à chacune desquelles s'appliquent des techniques particulières ?

Par exemple, en reprenant une classification proposée par J. ARSAC aux journées GENIE LOGICIEL organisées par le SESORI à Pont à Mousson en 1979, on peut distinguer deux grands axes (figure 1) :

- L'un concerne la difficulté algorithmique d'un problème : la recherche d'une solution demande beaucoup de travail. C'est le cas de nombreux problèmes de type scientifique, problèmes numériques, problèmes de graphes etc.... Les exemples a) et b) précédents sont de ce type

- L'autre concerne le nombre de détails qu'il est nécessaire de prendre en compte, c'est le cas des problèmes administratifs (paye), de gestion (facturation) etc.... Les exemples e), g), h) précédents sont de ce type.

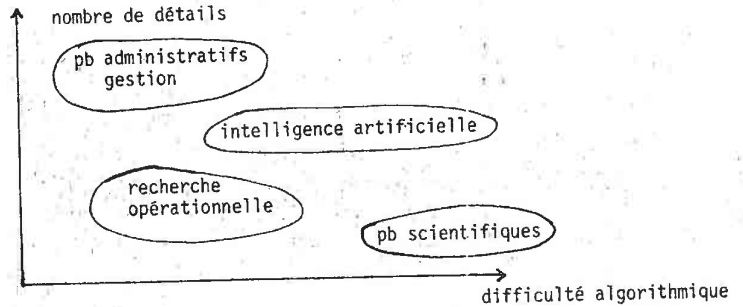


figure 1 : Une classification des problèmes.

Mais bien sûr il existe des problèmes faisant appel à de nombreuses informations et difficiles à résoudre ; c'est le cas en particulier des problèmes d'intelligence artificielle (reconnaissance de la parole, traduction automatique etc...).

On pourrait bien entendu trouver beaucoup d'autres classifications faisant intervenir des contraintes telles que le temps réel, le parallélisme, la forme des objets traités... Cependant, nous nous référerons dans la suite à ces deux axes car ilssituent :

- Des classes d'informaticiens qui trop souvent s'ignorent mutuellement
- Des types de langages de programmation différents, les uns mettant l'accent sur certaines sortes de "structures de données", les autres insistant davantage sur la puissance des "structures de contrôle".
- L'endroit où l'on rencontre le plus de difficultés lors de la résolution d'un problème : la spécification d'un problème dans lequel apparaissent de nombreux détails est difficile mais la résolution peut faire appel à des techniques quasi automatiques (problèmes de gestion) alors que c'est le contraire qui se produit pour de nombreux problèmes numériques.

Dans la suite nous nous efforcerons de mettre en évidence les idées générales apparaissant plus ou moins implicitement dans la spécification, l'analyse, la résolution algorithmique d'un problème, tout en étant conscient du fait que certaines s'appliquent surtout à certaines catégories de problèmes.

2.- SPECIFICATION DES PROBLEMES.

Il est maintenant de plus en plus admis que la première spécification d'un problème est une tâche difficile. Elle l'est d'autant plus si l'on cherche à donner conjointement une spécification et une résolution du problème (c'est-à-dire un algorithme) [PAI, 79] . Il est donc raisonnable de chercher à spécifier formellement le problème de la manière la plus convaincante intuitivement ce qui conduit à un énoncé *statique*, c'est à dire indépendant de toute idée d'exécution et même de résolution.

Nous nous attachons dans ce paragraphe à préciser le concept de *spécification* ce qui permet de dégager les principales étapes de l'étude menée dans cette première partie pour proposer une méthode et un cadre de spécification de problèmes.

2.1.- Qu'est-ce que spécifier un problème ?

De manière informelle on peut énoncer le problème de la spécification des problèmes de la manière suivante :

> Etant donné un problème énoncé de manière imprécise en donner une spécification précise à partir de laquelle on puisse en dériver une solution.

Cet énoncé est lui-même insuffisant dans la mesure où on ne sait pas ce qu'est une *spécification précise*, c'est à dire que l'on ignore la *forme* et les *propriétés* du résultat souhaité.

On veut donc étudier le concept de spécification précise ; mais cette étude a pour complément indispensable la recherche d'une méthode de construction d'une telle spécification. Commençons donc par rassembler quelques idées sur sa nature et sa construction.

a) Langage de spécification.

L'une des questions les plus délicates est celle du niveau de formalisme utilisé. Il est nécessaire d'obtenir un énoncé précis, cohérent du problème. Mais peut-on espérer le décrire dans un langage non formel (sous langage d'une langue naturelle par exemple) connaissant les ambiguïtés, l'imprécision sous jacente à un tel langage ? En particulier si on utilise un tel langage la construction automatique ou la vérification d'une solution est impossible puisqu'au départ on ne dispose pas d'un cadre formel. Inversement, l'utilisation d'un langage purement formel, dans l'hypothèse où il est bien maitrisé par l'informaticien, est bien souvent malaisée voire impossible pour la personne qui pose le problème. Il n'est alors pas possible de vérifier l'adéquation de la spécification formelle au problème donné.

b) Propriétés de la spécification obtenue.

Peut être plus que le langage d'expression, ce sont les propriétés de la spécification cherchées qui sont importantes. Parmi elles deux sont les plus marquantes :

- La spécification doit présenter une certaine *structuration*. Quel que soit le langage utilisé, fut-il le plus formel possible, si les relations entre les différents objets ne sont pas décrétés de manière claire et compréhensible, la spécification obtenue ne servira à rien

- La spécification obtenue doit être fidèle à l'énoncé initial du problème : il faut éviter de trahir la pensée de la personne qui pose le problème

c) Méthode de spécification

Si la question de la construction systématique et méthodique d'un programme à partir d'un énoncé précis a été abordée depuis plusieurs années déjà, celle de la construction d'un premier énoncé en est à ses premiers balbutiements. On peut bien entendu reprendre des idées générales comme la nécessité d'une approche progressive, descendante mais il faut essayer d'aller plus loin, l'objectif étant de fournir à l'utilisateur un moyen systématique d'énoncer son problème (si c'est possible !).

L'analogie avec la construction des programmes et les remarques faites à la fin du paragraphe 1.1 nous permettent déjà de dégager quelques idées sur lesquelles pourrait reposer une méthode de spécification :

- Puisque le *résultat* d'un problème est presque toujours décrit dans l'énoncé initial, commencer par essayer de le spécifier

- Lors d'une spécification il est utile de nommer des objets : on nomme d'abord le résultat, puis pour l'exprimer on est conduit à introduire et à nommer des *intermédiaires*.

- Essayer de *structurer modulairement* la spécification en isolant des sous problèmes indépendants : l'introduction d'un intermédiaire correspond à celle d'un nouveau sous problème.

- Utiliser la séparation entre les concepts d'*univers* et d'*énoncé* pour structurer la spécification.

Ces idées sont très voisines de celles développées, pour la construction d'un algorithme, dans le cadre de la méthode déductive de programmation [PAI, 79] , [BEL, 78 a] . Elles reprennent aussi l'idée de construction progressive, descendante et modulaire développée dans les principales méthodes de programmation structurée [DAH, 72] , [DIJ, 76] .

A ce stade de l'étude il est nécessaire de considérer quelques exemples.

2.2.- Forme et niveau de détail du premier énoncé d'un problème.

Nous appliquons dans les deux exemples qui suivent les idées dégagées au paragraphe 2.1 c) sur la construction progressive d'une spécification précise d'un problème. En même temps nous discutons du cadre de description et du niveau de détail d'une telle spécification.

Exemple 1 : Problème de la fraction irréductible

Reprenons l'exemple a) (§.1.1). Pour comprendre cet énoncé il faut connaître le concept d'entier naturel ainsi que le sens des phrases : "premiers entre eux", "proportionnels à deux nombres". Pour formaliser ce problème on commence par l'exprimer sous la forme d'un énoncé mathématique, par exemple :

E : *résultat* $x, y \in \text{ENTIER}^+$ *tq* $\text{propor}(x, y, a, b) \wedge \text{prem}(x, y)$
donnée $a, b \in \text{ENTIER}^+$

Dans cette formule x, y sont les inconnues et les variables libres a, b des données ; les prédicats $\text{propor}(x, y, a, b)$ et $\text{prem}(x, y)$ signifient respectivement que x, y sont proportionnels à a et b et que x et y sont premiers entre eux.

L'univers du problème est ici ENTIER$^+$ (ensemble des entiers naturels strictement positifs) muni des deux prédicats propor et prem .

La formalisation du problème pourrait s'arrêter là si l'on admet que propor et prem sont des *fonctions de base* et que ENTIER$^+$ est *un ensemble de base*. Précisons cette idée : l'énoncé E est une formule d'un certain problème formel $\mathcal{K}$ (ici calcul des prédicats du 1er ordre) dans lequel les éléments de ENTIER$^+$ sont des constantes et propor et prem des prédicats. On peut alors définir une *solution* de E pour une valeur p, q de a, b comme étant un couple (u, v) d'éléments de ENTIER$^+$ tel que le résultat de la substitution de u, v, p, q respectivement à x, y, a, b soit un théorème de $\mathcal{K}$.

Bien entendu ceci nécessite qu'il y ait des axiomes de $\mathcal{K}$ caractérisant propor et prem ; de manière intuitive dire que ces prédicats sont "de base" signifie qu'il existe de tels axiomes.

On pourrait chercher maintenant à construire ces axiomes, par exemple par récurrence sur ENTIER$^+$, en supposant qu'il existe une "fonction de succession" sur ENTIER$^+$, puis à introduire une version constructive de E c'est à dire un algorithme. Deux reproches majeurs peuvent être faits à une telle démarche :

- D'une part l'axiomatisation des prédicats en terme de ENTIER$^+$ uniquement (par exemple en terme de la constante 1 et de la fonction de succession) risque d'être très lourde et va reconduire à "inventer" une partie de l'arithmétique . En bref cette axiomatisation risque d'être de "trop bas niveau".

- D'autre part l'écriture d'un algorithme déduit directement de E va fournir un programme très inefficace s'il consiste simplement à parcourir ENTIER$^+$ x ENTIER$^+$ jusqu'à rencontrer une solution. L'obtention d'un algorithme non trivial (ie différent de "tant que pas trouvé chercher") nécessite de connaître des propriétés des prédicats apparaissant dans E .

En résumé le niveau atteint lors de cette première formalisation du problème n'est pas satisfaisant et on va chercher à exprimer les prédicats propor et prem en fonction d'autres opérations de plus bas niveau dont les propriétés sont connues. La figure 2 présente un développement possible de l'énoncé formel de ce problème.

Lexique	Enoncé formel
- On cherche deux entiers naturels non nuls x, y proportionnels à deux entiers naturels non nuls donnés a, b et premiers entre eux - ENTIER$^+$: *de base*, entiers naturels non nuls - propor est un prédicat exprimant que deux entiers sont proportionnels à deux autres entiers - BOOL : *de base*, booléens - * : *de base* sur ENTIER$^+$, multiplication - prem est un prédicat exprimant que deux entiers sont premiers entre eux - div est la relation de divisibilité sur les entiers	*résultat* x, y : ENTIER$^+$ *tq* propor(x,y,a,b) ∧ prem(x,y) *donnée* a, b : ENTIER$^+$ propor : *fonct* (x,y,a,b : ENTIER$^+$) b' : BOOL *tq* b' = (x * b = y * a) prem : *fonct* (x, y : ENTIER$^+$)b':BOOL *tq* b' = (∀z ∈ ENTIER$^+$, z divx ∧ zdivy ⊃ z = 1) div : *op* (x, y : ENTIER$^+$)b': BOOL *tq* b' = (∃ k *tq* k ∈ ENTIER$^+$ ∧ y = k * x)

Figure 2 : spécification du problème de la fraction irréductible

L'univers du problème est ici formé de ENTIER$^+$, BOOL et * qui peuvent être précisément axiomatisés [KLE, 52] , [SHO, 67] . Il est superflu de les expliciter ici puisque le but est d'obtenir un programme dont les opérations élémentaires sont supposées vérifier les axiomes de l'arithmétique (au caractère de finitude des représentations en machine près). Ceci serait d'ailleurs encore vrai s'il fallait résoudre le problème "à la main" : il est rare en mathématique qu'on utilise les axiomes primitifs d'un système formel. En arithmétique, par exemple, on se contente d'utiliser des propriétés telles que la commutativité, l'associativité etc... qui sont des théorèmes dérivés des axiomes primitifs. C est sur ces propriétés que va reposer la résolution ultérieure du problème (cf ch 1.3) ce qui justifie de définir l'univers comme un système formel, c'est à dire un cadre permettant d'effectuer des raisonnements et des démonstrations.

Plusieurs types de description sont envisageables ici :

- L'utilisation exclusive de la colonne de droite de la figure 2 correspond à une spécification formelle, précise et structurée. Les reproches qu'on puisse lui faire concernent sa compréhensibilité par une personne non habituée au langage des prédicats ou le trop bas niveau de description qu'elle induit.

- On peut en dériver une version en langue naturelle, tout aussi structurée mais plus "bavarde" :

"On cherche deux entiers naturels non nuls x , y proportionnels à deux entiers naturels non nuls donnés a, b et qui soient premiers entre eux. On rappelle que deux entiers x, y sont proportionnels à deux autres entiers a, b si $\frac{x}{y} = \frac{a}{b}$.

On rappelle également que deux entiers x et y sont premiers entre eux si leur seul diviseur commun est 1. Un diviseur d'un entier y est un entier x tel que y soit égale au produit de x par un autre entier".

De plus les risques d'ambiguité dans une telle description sont nombreux ; par exemple dans la première phrase de l'énoncé précédent le changement de "et qui soient" en "qui sont" peut conduire à comprendre que x et y doivent être premiers entre eux ou que a et b le sont.

- La solution que nous avons choisie ici est médiane entre ces deux points de vue : on a à la fois les qualités de l'aspect formel et la traduction ponctuelle de quelques propriétés en langue naturelle. On peut alors, si nécessaire, donner une traduction complète en langue naturelle. Mais un autre aspect important de cette présentation réside dans son application méthodologique : introduction informelle des intermédiaires lors de la spécification d'un résultat qui précède et guide leurs spécifications formelles ultérieures. □

Exemple 2 : Problème des liaisons aériennes.

La formalisation du problème e) du §.1.1nécessite de disposer d'un "catalogue" des différentes liaisons aériennes. Malheureusement les conventions utilisées pour préciser les caractéristiques d'un vol différent souvent d'une compagnie à l'autre et ceci constitue l'une des difficultés de ce problème.

Nous ne rentrerons pas ici dans le détail de ces conventions et proposons (figure 3) une première spécification formelle du problème à partir de laquelle il y aurait encore beaucoup de précisions à donner pour arriver à un énoncé complet.

Examinons cet énoncé :
Il repose sur plusieurs "ensembles" de base : VILLE et LIAISON. On a ici le choix entre considérer ces ensembles comme des données ou les considérer comme des objets "prédéfinis" existant dans l'univers. Ceci illustre la différence de niveau pouvant exister entre les données d'un problème. On retrouve cet aspect dans le cas des bases de données où l'univers contient un ensemble d'objets déjà construits.

Mais de plus LIAISON dépend de VILLE : plus que d'un univers il s'agit d'une classe d'univers (ou encore univers paramétré). Nous reviendrons sur cette idée au chapitre 1.3.

La spécification de LIAISON se fait à partir de catalogues réels (cf. figure 4) à partir desquels on isole et formalise les fonctions de base. Ainsi nous avons choisi de définir LIAISON comme un produit cartésien

VILLE x VILLE x REEL x DATE x DATE

dont les projections respectives sont départ, arrivée, priel, hdep, harr.

On cherche une suite α de liaisons aériennes entre la ville de départ vd et la ville d'arrivée va dont le prix est minimum

entre : est un prédicat exprimant qu'une suite de liaisons est cohérente pour les horaires et que l'aéroport d'arrivée de chaque liaison est celui de départ de la suivante, qu'elle commence en u et se termine en v.

prix : est une fonction définissant le prix d'une suite de liaisons

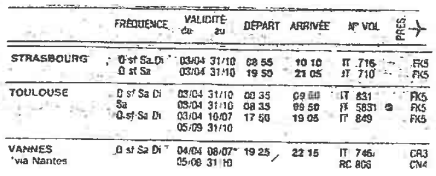
cohérent : est un prédicat exprimant que le temps de transit entre deux liaisons est convenable : l'arrivée précède de une heure le départ et le temps d'attente n'excèdepas 6 heures.

résultat α : LIAISON*
tq α entre (vd, va) ∧
(∀β ∈ LIAISON* ,β entre(vd, va) ⊃ prix(α) ≤ prix(β))
donnée vd, va : VILLE
entre : *op* (β : LIAISON* , u, v : VILLE) b : BOOL
tq b = (départ(prem(β)) = u ∧ arrivée(der(β)) = v
∧ ∀ i ∈ [1.. bs(β) -1] ,
arrivée(β_i) = départ(β_{i+1}) ∧ cohérent(β_i, β_{i+1})
prix : *fonct* (β : LIAISON*) r : REEL
tq r = Σ priel(β_i) *pour* i *dans* [1..bs(β)]
cohérent : *fonct* (ℓ1, ℓ2 : LIAISON) b : BOOL
tq b = (har(ℓ1) +1 < hdep(ℓ2) < har(ℓ1) +6)

Univers

LIAISON = < départ : VILLE, arrivée : VILLE, priel : REEL, hdep : DATE, harr : DATE >
DATE : *de base*
VILLE: *de base*
BOOL : *de base*
REEL : *de base*

Figure 3 : une spécification formelle du problème des liaisons aériennes.


LYON
Réservation ☎ (78) 52.80.45

	FRÉQUENCE	VALIDITÉ du	VALIDITÉ au	DÉPART	ARRIVÉE	N° VOL	
STRASBOURG	D sf Sa Di	03/04	31/10	08 55	10 10	IT 716	FK5
	D sf Sa	03/04	31/10	19 55	21 05	IT 710	FK5
TOULOUSE	D sf Sa Di	03/04	31/10	08 35	09 50	IT 831	FK5
	Sa	03/04	31/10	08 35	09 50	FK 5831	FK5
	D sf Sa Di	03/04	16/07	17 50	19 05	IT 849	FK5
		05/09	31/10				
VANNES *via Nantes	D sf Sa Di	04/04	08/07*	19 25	22 15	IT 745	CR3
		05/09	31/10			RC 808	CN4

A son tour DATE devrait être défini comme un ensemble d'objets muni d'une opération de comparaison ≤ et d'une opération + .

De plus l'énoncé porte sur LIAISON* : ensemble des suites sur LIAISON muni d'opérations telles que

bs : donne la borne supérieure d'une suite
prem : associe à une suite son premier élément
der : associe à une suite son dernier élément

Ainsi,parallèlement au développement de l'énoncé, on développe l'univers du problème qui, en première approximation,est formé d'univers prédéfinis (ENTIER, REEL, BOOL, etc...) et de nouveaux ensembles auxquels sont associésdes opérations (on est proche de la notion de type abstrait, cf chapitre 1.2).

On peut , comme dans l'exemple précédent, donner une version structurée en langue naturelle de cet énoncé.

"On cherche une suite α de liaisons aériennes entre la ville de départ vd et la ville d'arrivée va dont le prix soit inférieur ou égal à celui de toute autre suite de liaisons entre vd et va.

Une telle suite doit être telle que l'aéroport d'arrivée d'une liaison quelconque, sauf la dernière, soit le même que l'aéroport de départ de la suivante.

On exige de plus que l'arrivée d'une liaison précède d'au moins une heure le départ de la suivante et que le temps total d'attente de chaque escale n'excèdepas 6 heures".

La spécification de la figure 3 est structurée et semble conforme à l'énoncé initial. Mais on peut également se demander si on ne précise pas trop de propriétés, c'est à dire si le problème n'est pas ainsi *surspécifié*.

Une question connexe est celle de l'amplitude des modifications que peut subir une telle spécification : comment transformer la spécification précédente pour l'adopter aux modifications suivantes du problème.

- On accepte une solution qui soit "un peu plus" onéreuse que la précédente si cela permet de diminuer "notablement" le nombre d'escales intermédiaires.

- Donner la préférence pour le passage par tel ou tel aéroport

- Minimiser non plus le prix mais le temps total.

On voit qu'ici ces différentes modifications n'ont d'influence que sur la spécification de la relation entre α et les données vd, va.

Il est donc important de spécifier de manière structurée et élevée (i.e sans prendre en compte de détails de réalisation superflus) ; c'est une qualité nécessaire à l'adaptabilité ultérieure à des modifications. □

Remarque :

Si la distinction entre les concepts d'énoncé et d'univers apporte une aide non négligeable à une démarche méthodique permettant de décomposer la spécification d'un problème en deux parties, elle n'est pas toujours très nette d'un point de vue formel. Ainsi, on aurait très bien pu introduire un univers "suite de liaisons" dont "cohérent" aurait été une opération.

2.8.- Vers une méthode de spécification de problèmes.

La description initiale d'un problème posé par un "utilisateur" est en général incomplète, incohérente et non structurée. L'univers du problème n'est souvent pas même perçu comme une entité effective. Pour passer d'une telle description à une spécification précise de manière méthodique et aussi systématique que possible il est nécessaire de préciser les points suivants :

a) Quelles sont les propriétés attendues de la spécification ?

- que demande t-on à un énoncé ?
- que demande t-on à un univers ?

Il est nécessaire de commencer par une étude empirique à partir de laquelle les principaux concepts peuvent être isolés, formalisés et enfin étudiés formellement.

b) Dans quel cadre spécifier le problème ? C'est un complément nécessaire à l'étude des propriétés. Entre l'utilisation d'un langage uniquement formel et celle d'une langue naturelle nous choisissons une voie intermédiaire consistant à construire une spécification formelle d'un énoncé en utilisant un lexique des différents objets introduits. Le lexique est par nature incomplet et ne donne que des descriptions locales. Ainsi pour comprendre la structure profonde de l'énoncé il faut se reporter à la partie formelle. Il permet cependant de ne pas rentrer, d'un premier coup d'oeil, dans la description formelle complète des intermédiaires en se contentant d'une description peut-être plus intuitive. Cette approche présente donc les avantages suivants , sur lesquels nous reviendrons par la suite,

- intérêt de l'aspect formel pour la résolution ultérieure du problème et pour la vérification de certaines propriétés
- intérêt du lexique pour éclairer certaines spécifications
- intérêt de la dualité colonne de gauche, colonne de droite qui fournitun guide méthodologique pour une spécification descendante.

Une solution à long terme consisterait, peut-être, en suivant l'exemple des mathématiciens et des physiciens, à mêler une langue naturelle à un langage formel, le premier permettant de décrire les raisonnements, les motivations, les intentions, etc..., le second étant utilisé pour exprimer les formules ou les relations formelles entre les objets. La dernière forme du problème de la fraction irréductible (exemple 1) donne une idée de ce que l'on pourrait obtenir. Brièvement il s'agirait de donner "plus d'importance à la colonne de gauche qu'à la colonne de droite", la première contenant

la structure complète de la spécification. Malheureusement l'obtention *directe* d'une telle spécification ne semble pas très simple actuellement.

c) Quelle démarche utiliser ? C'est le point le plus délicat. Les idées issues de la méthode déductive de programmation et adaptées au cas de la spécification semblent assez prometteuses. C'est ainsi par exemple que l'introduction de fonctions intermédiaires permet d'obtenir un énoncé modulaire.

La suite de cette première partie est guidée par ces réflexions :

- la fin de ce chapitre 1.1 consiste en une étude de nouveaux exemples (§.3) à partir de laquelle on essaye de dégager les principales qualités d'une spécification précise d'un problème et d'un cadre permettant de la décrire (§.4). Le paragraphe 5 consiste en un essai de comparaison de quelques cadres de spécification.

- Le chapitre 1.2 concerne la spécification formelle des univers. L'étude des systèmes d'information (système formel du type calcul des prédicats du premier ordre) permet de définir précisément quelques unes des propriétés d'une spécification (consistance, complétude, complétude suffisante) et de se doter d'outils de construction (tels que la paramétrisation).

- Le chapitre 1.3 précise les notions d'énoncé formel et de solution d'un énoncé. On y présente les principales idées qui seront à la base de la résolution des problèmes présentée dans la partie 2.

- Le chapitre 1.4 est le point central de cette première partie puisqu'il constitue un essai de description d'une méthode de spécification. Cet essai, pour partiel et incomplet qu'il soit, constitue, du moins nous l'espérons, une première étape vers une méthode systématique de spécification se concrétisant par exemple sous la forme d'un système interactif.

3.- LANGAGES D'EXPRESSION D'ENONCES FORMELS.

3.1.- Langage ensembliste et prédicats.

Dans les exemples précédents les énoncés formels étaient exprimés dans un langage qui a fait ses preuves puisqu'il s'agit du langage mathématique ou plus exactement du langage ensembliste. Sans vouloir en donner une description précise, (cf ch. 1.2. §.5) rappelons que les principaux concepts utiles sont :

- Les ensembles, éventuellement, caractérisés par un prédicat
- Les opérations ensemblistes U, $\cap$; $\setminus$ (différence), card (cardinal pour les ensembles finis), $\subset$ (inclusion), X (produit cartésien), $^{-1}$ inverse d'une relation...
- Les ensembles de base : ENTIER, REEL, BOOL et les opérations arithmétiques et logiques
- Les fonctions de E dans F dont l'ensemble est noté $[E \rightarrow F]$ muni des fonctions dom(domaine) et codom(codomaine).

- Les suites, c'est à dire les fonctions α de $[ENTIER \rightarrow E]$ dont le domaine est un intervalle de borne inférieure $bi(\alpha)$ et de borne supérieure $bs(\alpha)$.
Nous noterons E^* leur ensemble.

- Les intervalles de ENTIER et REEL

Ce langage , avec quelques adaptations, est utilisé par plusieurs auteurs : il est à la base du langage Z (cf §. 5.2g) ainsi que de SETL (cf §.5.2 a)).

Donnons encore trois exemples de spécification ensembliste d'un problème

a) Le problème des sous suites croissantes de longueur maximum (§ 1.1 b)) peut s'exprimer sous la forme de la figure 5 [BRO, 79a]

b) Le problème du tri topologique (§ 1.1 d)) se prête particulièrement bien à la formalisation ensembliste présentée figure 6 [FIN, 78 a] .

Lexique	énoncé formel ensembliste
On cherche la longueur maximum des sous-suites croissantes de α ENTIER : *de base* V : *de base* ensemble sur lequel sont définies les suites considérées. Il est muni d'un ordre total $\sqsubseteq$: prédicat exprimant qu'une suite est une sous-suite d'une autre croiss prédicat exprimant qu'une suite est croissante || : *de base* longueur d'une suite scroiss : prédicat exprimant qu'une suite d'entiers strictement croissante bi *de base* dans V*, borne inférieure d'une suite bs *de base* dans V*, borne supérieure d'une suite	*résultat* ℓmax : ENTIER *tq* ℓmax = max {k ∈ ENTIER *tq* ∃ β : V* *tq* β ⊑ α ∧ croiss(β) ∧ |β| ≡ k} *donnée* α : V* ⊑ : *fonct*(β, γ: V*) b : BOOL *tq* b = [∃ φ : [dom(β) → dom(γ)] *tq* scroiss(φ) ∧ (∀ j ∈ dom(β), $\beta_j = \gamma_{\varphi(j)}$)] croiss : *fonct* (β : V*) b : BOOL *tq* b = (∀ i ∈ [bi(β)..bs(β)-1], $\alpha_i \leq \alpha_{i+1}$) scroiss : *fonct* (u : ENTIER*) b : BOOL *tq* b = (∀ i ∈ [bi(u)..bs(u)-1], $u_i < u_{i+1}$)

Figure 5 : le problème des sous-suites croissantes maximales

Lexique	énoncé formel
On cherche une relation d'ordre total sur E contenant la relation Γ donnée E : *de base* $\subset$, ∩, ., ∪, $^{-1}$: *de base* respectivement inclusion, intersection composition, réunion et inverse I : *de base* relation identité	*résultat* Δ : $\mathcal{P}$(ExE) *tq* I ⊂ Δ ∧ Δ ∩ Δ^{-1} ⊂ I ∧ Δ . Δ ⊂ Δ ∧ Δ ∪ Δ^{-1} = ExE ∧ Γ ⊂ Δ *donnée* Γ : $\mathcal{P}$(ExE)

Figure 6 : le problème du tri topologique

La formule obtenue exprime en effet que Δ est reflexive, antisymétrique, transitive, totale et qu'elle contient Γ. Notons qu'on aurait pu l'obtenir de manière plus progressive. Il aurait été également possible de l'exprimer non plus directement dans le treillis booléen des parties de E^2 mais en terme de prédicat portant sur des éléments de E. Il semble plus intéressant de chercher des caractérisations d'énoncés en termes de fonctions de base "de haut niveau".

c) Le problème de la gestion des remboursements d'un emprunt bancaire (§ 1.1 n)) se prête particulièrement bien à une spécification ensembliste. Il nécessite d'introduire la notion de temps (qui n'est pas le temps d'éxécution d'un programme mais une donnée particulière) ce qui le rapproche des problèmes "temps réel".

Une première spécification consiste à caractériser, pour chaque mois de la période de vie du système de gestion, les clients actifs, les clients ayant terminé, les clients rejetés et les nouveaux clients. Les résultats sont alors des fonctions définies sur l'ensemble des mois. Cette solution est exprimée par la figure 7.

Une autre possibilité consiste à définir, pour un mois donné, les mêmes ensembles (actif...) en fonction des résultats associés au mois précédent (figure 8).

Le premier point de vue correspond à une vision globale et fonctionnelle du problème, le deuxième à une vue plus locale et ensembliste plus proche d'un algorithme. On passe de l'un à l'autre essentiellement en remplaçant certaines fonctions par leurs valeurs en un point. D'un point de vue méthodologique on peut imaginer le passage successif par ces deux étapes.

Notons que, dans chaque solution, figure la description de l'univers dans un formalisme qui est détaillé au chapitre 1.2. On peut noter la grande latitude dont peut disposer le spécificateur dans l'introduction de fonctions telles que crédit, débit,... : font-elles partie de l'univers ou de l'énoncé ? La frontière entre ces deux notions apparaît, comme nous l'avons déjà remarqué, assez floue.

actif qui associe à un mois donné l'ensemble des clients actifs à la fin
e ce mois (i.e qui n'ont pas encore fini de rembourser leur emprunt)
ter qui associe à un mois donné l'ensemble des clients qui terminent
eur remboursement pendant ce mois
rej qui associe à un mois donné les clients rejetés, ce sont ceux
ont la différence entre le débit et le crédit dépasse deux mensualités
nouv qui associe à un mois donné l'ensemble des nouveaux clients qui
pparaissent ce mois .
eurs définitions utilisent de nouvelles fonctions :
pred *de base* dans MOIS (prédecesseur d'un mois)
debut *de base* dans MOIS (début d'activation du système de prêts)
MOIS *de base*
CLIENT *de base*
terminé : prédicat caractérisant les clients ayant terminé un mois donné
rejeté : prédicat caractérisant les clients rejetés un mois donné
ncl associe à un mois donné l'ensemble des noms et des caractéristiques
ontant, mfin) des nouveaux clients apparaissant ce mois. C'est une donnée
nom *de base* dans CLIENT et NCLIENT, injective
NCLIENT *de base*
montant *de base* dans CLIENT et NCLIENT
mdeb *de base* dans CLIENT, mois de début du prêt
mfin *de base* dans CLIENT et NCLIENT, mois de fin du prêt
crédit associe à un client et à un mois le crédit correspondant

```
actif : fonct (m : MOIS) c : ℘ (CLIENT)
    tq si m = début
      alors c = ∅
      sinon c = actif (pred(m)) U nouv(pred(m)) \
                (ter(m) U rej(m))
ter:fonct (m : MOIS) c : ℘ (CLIENT)
    tq c = {cl ∈ actif(pred(m)) U nouv(pred(m))
            tq terminé(cl, m) }
rej:fonct (m : MOIS) c : ℘ (CLIENT)
    tq c = {cl ∈ actif(pred(m)) U nouv(pred(m))
            tq rejet(cl, m)}
nouv:fonct (m : MOIS) c : ℘ (CLIENT)
            tq c = {cl ∈ CLIENT tq
                    ∃nc ∈ ncl(m) tq
                    nom(cl) = nom(nc) ∧
                    montant(cl) = montant(nc)∧
                    mdeb(cl) = m ∧
                    mfin(cl) = mfin(nc) }
terminé:fonct (cl :CLIENT, m : MOIS) b : BOOL
        pré cl ∈ actif(pred(m)) U nouv(pred(m))
        post b = crédit(cl, m) ⩾ montant(cl)
```


Lexique	Enoncé ensembliste formel
bit associe à un client et à un mois le débit correspondant : somme mensualités depuis le début du prêt ou montant du prêt si le mois uestion est postérieur à la date de fin de prêt nsualité *de base* dans CLIENT, quotient du montant par la durée du prêt rs associe à un client et à un mois le versement correspondant; c'est donnée c *de base* dans MOIS, successeur d'un mois donné G *de base* *de base* dans MOIS	rejeté:*fonct* (cl : CLIENT, m : MOIS) b : BOOL *pré* cl ∈ actif(pred(m)) U nouv(pred(m)) *post* b = débit(cl, m)-crédit(cl, m)> 2xmensualité(cl crédit:*fonct* (cl : CLIENT, m : MOIS) a : ARG *pré* cl ∈ actif(pred(m)) U nouv(pred(m)) *post* a =Σ vers(cl, i)*pour* i *dans* [suc(mdeb(cl)), m] débit:*fonct* (cl : CLIENT, m : MOIS) a : ARG *pré* cl ∈ actif(pred(m)) U nouv(pred(m)) *post si* m ⩾ mfin(cl) *alors* a = montant(cl) *sinon* a = Σ mensualité(cl) *pour* i *dans* [suc(mdeb(cl)), m] *donnée* vers *fonct* (cl : CLIENT, m : MOIS) a : ARG ncl *fonct* (m : MOIS) r : ℘ (NCLIENT)
est un ensemble totalement ordonné par ⩾ possédant un plus petit ent début, une fonction successeur, une fonction prédécesseur et fonction "différence"	*type* MOIS *ordonné* *avec* début, suc, pred, ⩽ - *fonct* (MOIS, MOIS) ENTIER
	type ARG = ENTIER
	type CLIENT = < nom : CHAINE, montant : ARG, mensualité : ARG, mdeb : MOIS, mfin : MOIS > *invariant* cl1, cl2 : CLIENT, nom(cl1) = nom(cl2) ⊃ cl1 = cl2 mdeb(cl1) < mfin(cl1) mensualité(cl1) = montant(cl1)/(mfin(cl1)-mdeb(cl1)).
	type NCLIENT = < nom : CHAINE, montant : ARG, mfin : MOIS> *invariant* cl1, cl2 : NCLIENT nom(cl1) = non(cl2) ⇒ cl1 = cl2

Figure 7 : première spécification du problème des emprunts bancaires

On cherche les ensembles suivants :
Actif clients actifs
Ter : clients ayant terminé
Rej : clients rejetés
Nouv : nouveaux clients
On dispose de :
AActif : clients actifs au mois précédent (donnée)
ANouv : nouveaux clients du mois précédent (donnée)
On introduit les fonctions suivantes :
maj : fonctions de "mise à jour" d'un client en fonction d'un mois donné et du versement correspondant
terminé : prédicat caractérisant les clients ayant terminé leur remboursement
rejeté : prédicat caractérisant les clients dont la différence entre le débit et le crédit dépasse deux mensualités
entrée : fonction définissant un nouveau client à l'aide de son nom et de ses caractéristiques (montant, mfin)
Ncl : ensemble des noms et caractéristiques des nouveaux clients (donnée)
crédit *de base* dans CLIENT associe à chaque client le crédit dont il dispose à la fin du mois
débit *de base* dans CLIENT associe à chaque client le débit correspondant à la fin du mois
VER : *de base* ensemble des versements
dom : *de base* domaine d'un versement
mensualité *de base* dans CLIENT

résultat
Actif : $\mathcal{P}$ (CLIENT)
tq Actif = {cl *tq* ∃ cl1 ∈ AActif U ANouv
tq cl = maj(cl1, m, ver) ∧
¬terminé(cl) ∧ ¬rejeté(cl)}
Ter : $\mathcal{P}$ (CLIENT)
tq Ter = {cl *tq* ∃ cl1 ∈ AActif U ANouv
tq cl = maj(cl1, m, ver)
∧ terminé(cl)}
Rej : $\mathcal{P}$ (CLIENT)
tq Rej ={ cl *tq* ∃ cl1 ∈ AActif U ANouv
tq cl = maj(cl1, m, ver)
∧ rejeté(cl)}
Nouv : $\mathcal{P}$ (CLIENT)
tq Nouv = {entrée(cl) *tq* cl ∈ Ncl}
terminé: *fonct* (cl : CLIENT) b : BOOL
tq b = crédit(cl) > montant(cl)
rejeté: *fonct* (cl : CLIENT) b : BOOL
tq b = (débit(cl) - crédit(cl)) > 2 x mensualité(
maj: *fonct* (cl : CLIENT, m : MOIS, v : VER) cl' : CLIENT
pré cl ∈ AActif U ANouv ∧ dom(v) = AActif U ANouv
post nom(cl') = nom(cl) ∧
montant(cl') = montant(cl) ∧
mensualité(cl') = mensualité(cl) ∧
mfin(cl') = mfin(cl) ∧
mdeb(cl') = mdeb(cl)

, montant , mensualité, mfin, mdeb ; *de base* dans CLIENT
: versements du mois considéré

début de l'activation du système, m = début et AActif =
uv = ∅

débit(cl') = *si* m ≥ mfin(cl)
alors montant(cl)
sinon débit(cl) + mensualité(cl)
crédit(cl') = crédit(cl) + v(cl)
entrée: *fonct* (nc : NCLIENT, m : MOIS) cl : CLIENT
post nom(cl) = nom(nc) ∧ montant(cl) = montant(nc) ∧
mfin(cl) = mfin(nc) ∧ mdeb(cl) = m ∧
débit(cl) = 0 ∧ crédit(cl) = 0
donnée AActif : $\mathcal{P}$ (CLIENT)
ANouv : $\mathcal{P}$ (CLIENT)
ver : VER
Ncl : $\mathcal{P}$ (CLIENT)
m : MOIS

type MOIS
type ARG : ENTIER
type CLIENT =< nom : CHAINE, montant : ARG, mensualité : ARG,
mdeb : MOIS, mfin : MOIS, débit : ARG, crédit : ARG >
invariant cl1, cl2 : CLIENT
nom(cl1) = nom(cl2) ⊃ cl1 = cl2
mdeb(cl1) < mfin(cl2)
mensualité(cl1) = montant(cl1)/(mfin(cl1) - mdeb(cl1))

type NCLIENT
type VER = *fonct* (CLIENT) ARG

Figure 8 : deuxième spécification du problème des emprunts bancaires

Remarque :

Dans la spécification formelle d'une fonction, le symbole *pré* indique que le prédicat qui suit précise les conditions que doivent vérifier les paramètres de la fonction pour que l'appel ait un sens : il caractérise donc le domaine de définition de la fonction. Le symbole *post* précède le prédicat caractérisant le résultat à l'aide des données. *post* et *tq* sont donc synonymes.

3.2.- Enrichissement du langage ensembliste.

Le langage ensembliste peut cependant être mal adapté à la spécification de certains problèmes en ce sens qu'il nécessite de descendre à un trop bas niveau de détails dans certains cas. Il est intéressant de lui adjoindre un certain nombre de concepts qui deviendront alors "de base".

a) Les multiensembles. (En anglais : bag)

Considérons le problème de la planification des charges d'une entreprise de confection (§ 1.1 g)). On spécifie l'énoncé du problème en introduisant les ensembles suivants :

ART : articles
FAB : types de fabrication
OPE : opérateurs
POS ⊂ ENTIER : numéros de postes de travail
TEM ⊂ REEL : temps
COD : codes
TAI ⊂ ENTIER : tailles
SEM : semaines

ainsi qu'un certain nombre de fonctions entre ces ensembles :

artcl ∈ [COD x TAI → ART]
type ∈ [ART ⟶ FAB]
num ∈ [OPE ⟶ POS]
temps ∈ [OPE ⟶ TEM]

Mais si l'on veut compléter cette spécification il faut décrire "l'ensemble des opérations" associées à chaque type de fabrication ainsi que le "fichier des commandes pour 12 semaines".

Pour décrire l'ensemble des opérations on peut utiliser la notion de suite ce qui revient à numéroter les opérations et en particulier à les ordonner. Une autre manière, ne faisant pas intervenir d'ordre à priori, consiste à utiliser le concept de multiensemble :

Un multiensemble m sur un ensemble E est une application définie sur E (ie dom(m) = E) à valeur dans ENTIER [KNU, 68] . Cette notion formalise l'idée intuitive de "collection" contenant plusieurs occurences d'un même objet. On peut définir des opérations multiensemblistes :

m1 U m2 est tel que $\forall x \in E$ m1 U m2 (x) = m1(x) + m2(x)

m1 ∩ m2 est tel que $\forall x \in E$ m1 ∩ m2 (x) = min (m1(x), m2(x))

Notons $\mathcal{M}(E)$ l'ensemble des multiensembles sur E(= [E → ENTIER]) alors dans le problème précédent on suppose définie

opéra ∈ [FAB → $\mathcal{M}$(OPE)] qui associe à tout type de fabrication le multiensemble de ses opérations.

De même la donnée est un multiensemble fcom ∈ $\mathcal{M}$(COM)

où COM représente l'ensemble des commandes caractérisé par l'existence de trois fonctions :

sem ∈ [COM → SEM] numéro de semaine
ident ∈ [COM → COD x TAI] article
qut ∈ [COM → ENTIER] quantité

Nous venons de préciser, assez informellement, l'univers du problème. En fait nous nous retrouvons dans la même situation que pour le problème des liaisons aériennes où il est naturel de considérer certains objets comme prédéfinis (articl, COD, ...) et d'autres comme des données (COM).

La figure 9 propose une spécification de ce problème.

b) Les listes linéaires.

Dans le problème envisagé au paragraphe précédent l'ordre des opérations ou des commandes pouvait ne pas être pris en compte dans une spécification formelle. On généralisait ainsi le problème posé ce qui pouvait conduire à des solutions très inefficaces. Pour essayer de se rapprocher davantage de l'énoncé informel en prenant en compte toutes les contraintes il est nécessaire d'introduire la notion de *fichier :*

Un fichier F sur un ensemble V est un quintuplet (ENR_F, $succ_F$, val_F, $tête_F$, $queue_F$) où ENR_F est l'ensemble des enregistrements de F.

lexique	énoncé formel
On cherche la charge hebdomadaire de chaque poste de travail pour le trimestre suivant, c'est à dire pour chaque semaine du trimestre suivant et chaque poste le temps nécessaire à la réalisation des commandes. C'est la somme, pour chaque commande de la semaine considérée, des temsp nécessités par la commande à ce poste - durée est une fonction qui associe à un poste et à une commande le temps nécessité par cette commande à ce poste - scom est une fonction qui définit l'ensemble des commandes d'une semaine donnée - sope est une fonction qui associe à un poste l'ensemble des opérations qu'il nécessite - SEM, POS, TEM, temps, opera, type, artcl, ident, qut, num : *de base*	*résultat* ch : *fonct* (s : SEM, p : POS)t : TEM *tq* t = Σ durée(p,c) * fcom(c) *pour* c *dans* scom(s) durée : *fonct*(p : POS, c : COM) t : TEM *tq* t = Σ temps(op)* opera(type(artcl(ident(c))))(op) * qut(c) *pour* op *dans* sope(p) scom : *fonct*(s : SEM) C : $\mathcal{P}$(COM) *tq* C = {c ∈ COM *tq* sem(c) = s} sope : *fonct*(p : POS) OP : $\mathcal{P}$(OPE) *tq* OP = {op ∈ OPE *tq* num(op) = p} *donnée* COM, fcom

Figure 9 : planification des charges d'une entreprise

- $tête_F$, $queue_F$ sont deux éléments particuliers de ENR_F

- $succ_F \in [ENR_F \rightarrow ENR_F]$, fonction de succession, est une bijection de $ENR_F \setminus \{queue_F\}$ sur $ENR_F \setminus \{tête_F\}$ définissant un ordre total sur ENR_F ;

elle est caractérisée par le fait que tout élément de ENR peut être atteint à partir de tête par l'application un certain nombre de fois de $succ_F$:

$$\forall e \in ENR\ ,\ \exists n \in ENTIER\ \ tq\ \ e = succ_F^n(e)$$

- $val_F \in [ENR_F \rightarrow V]$ est une fonction totale (ie partout définie)

Un tel quintuplet peut être aussi appelé liste linéaire [PAI, 74]

Ainsi pour le problème de la planification des charges il est plus naturel de considérer la donnée comme un fichier com plutôt que comme un multiensemble (qui intuitivement correspond à un fichier non ordonné) dont l'ensemble de valeurs serait un produit cartésien

$$SEM \times (COD \times TAI) \times ENTIER$$

la définition de ch devient alors :

$$ch(s, p) = \Sigma\ durée(p, val_{com}(c))\ \underline{pour}\ c\ \underline{dans}\ scom(s)$$

avec $scom(s) = \{c \in ENR_{com}\ \ tq\ sem(val_{com}(c)) = s\}$

On peut faire de même pour les opérations associées à un type de fabrication.

Notons ici que la différence entre la spécification utilisant des multiensembles et celle utilisant des fichiers est faible dans ce cas ; ceci provient du fait que la structure du résultat (fonction) est indépendante de l'ordre sur les données. Il n'en serait plus de même s'il s'agissait de planifier dans le temps la succession des opérations. De même dans le cas du problème de mise à jour simple (§ 1.1 j)) il faut prendre l'ordre en compte (figure 10).

Lexique	énoncé formel
On cherche un fichier nprix résultant de la mise à jour du fichier prix par le fichier maj NPROD : *de base* numéro des produits PUNIT : *de base* prix unitaires CAR : *de base* caractéristiques sup : entier, nombre d'enregistrements supprimés ajout : entier, nombre d'enregistrements ajoutés trié : *de base* prédicat précisant qu'un fichier est trié par ordre croissant par rapport à l'un de ses champs	*résultat* nprix *fichier sur* num : NPROD prix: PUNIT car : CAR *tq* $card(enr_{nprix}) = card(enr_{prix})$ -sup+ajout $\wedge[\forall e \in enr_{nprix}$, $(\exists e_1$ *tq* $e_1 \in enr_{prix} \wedge val(e_1) = val(e)$ $\wedge \forall p \in enr_{maj}$, $num(val(p)) \neq num(val(e)))$ $\vee(\exists p$ *tq* $p \in enr_{maj} \wedge (code(val(p))=a \vee code(val(p))=m)$ $\wedge(num, prix, car)(p)=val(e))]$ $\wedge$trié(nprix, num) *donnée* prix *fichier sur* num : NPROD prix: PUNIT car : CAR *tq* trié(prix, num) *donnée* maj *fichier sur* code : {a,m,s} num : NPROD prix : PUNIT car : CAR *tq* trié(maj, num) sup = card {$p \in enr_{maj}$ *tq* $code(val_{maj})(p)=s$ } ajout = card {$p \in enr_{maj}$ *tq* $code(val_{maj})(p)=a$ }

Figure 10 : Mise à jour simple de fichiers

Notons que pour simplifier la présentation nous avons supposé ici qu'il ne pouvait y avoir au plus qu'un enregistrement concernant un produit donné dans le fichier maj.

Il était également simple de prendre en compte les erreurs possibles en définissant le résultat comme un couple (nprix, erreur) où erreur est une suite de numéros de produits pour lesquels on a constaté une erreur (adjonction d'un élément qui existait déjà dans prix, suppression ou modification d'un élément qui n'existait pas dans prix.

c) Autres adjonctions possibles.

Le langage ensembliste évoqué au § 3.1 est suffisamment puissant pour exprimer la plupart des problèmes informatiques. Les adjonction précédentes ont donc surtout un intérêt de commodité : contrairement aux mathématiciens qui peuvent avoir intérêt à minimiser le nombre d'axiomes d'une théorie pour diminuer le nombre de propriétés à démontrer, il est intéressant en informatique d'avoir des outils simplifiant le plus possible l'expression des problèmes. On peut ainsi introduire des concept de la théorie des graphes [LAU, 76] , d'algèbre linéaire etc... .

On constate également que ce langage est d'un emploi assez lourd lorsqu'une formule contient trop de quantificateurs (cf figure 10). Il faut encore l'affiner en introduisant des constructions telles que les conditionnelles, les cas etc...

Ainsi la commodité d'utilisation est l'un des critères les plus importants de ce genre de langages.

3.3.- Utilisation de l'induction.

Pour formaliser l'énoncé de certains problèmes il peut être naturel d'introduire des fonctions définies recursivement. C'est notamment le cas lorsqu'une partie de l'univers est déjà structurée de manière inductive. Considérons par exemple le problème d'une recherche d'aïeuls (cf § 1.1

sa spécification formelle utilise une fonction ascendant qu'il est naturel et très simple de définir récursivement (figure 11)

Lexique	énoncé formel
On cherche le nombre d'ascendants d'une personne p donnée ayant eu plus de 4 enfants PERS : *de base* ensemble des personnes de la base de données nbenf : *de base* associe à toute personne son nombre d'enfants père : *de base*	*résultat* n : ENTIER *tq* n = card {a ∈ PERS *tq* ascendant(a, p) ∧ nbenf(a) > 4} *donnée* p : PERS ascendant : *fonct* (PERS, PERS) BOOL *tq* ∀a, b ascendant(père(a),a)= *vrai* ∧ ascendant(b,père(a)) = ascendant(b, a)

Figure 11 : recherche du nombre d'aïeuls ayant plus de 4 enfants.

Sur les domaines de base tels que ENTIER, REEL certaines fonctions sont définies directement de manière récursive, par exemple l'inéluctable factorielle :

fact:*fonct*(ENTIER) ENTIER
tq fact (0) = 1 ∧ ∀n fact(n+1) = n * fact(n)

Nous étudierons de telles spécifications au chapitre 2.2.

4.- QUALITES D'UN CADRE DE SPECIFICATION DE PROBLEMES.

L'étude précédente nous permet de préciser les principales qualités que l'on peut attendre d'un cadre de spécification de problèmes. On commence tout d'abord par caractériser un "bon" énoncé initial (§ 4.1) ce qui permet de dégager les propriétés que doivent posséder un langage et une méthode de spécification de problèmes (§ 4.2). Dans cette présentation nous avons repris un certain nombre d'idées de [BAL, 79a], [DEM, 79] et [WAS, 79] .

4.1.- Qualités de la première spécification d'un problème.

Elles sont pour la plupart subjectives

a) Facilité de compréhension.

Un premier énoncé doit être compréhensible, non seulement par l'informaticien qui aura à résoudre le problème, mais aussi par la personne qui pose le problème et qui doit pouvoir vérifier la validité de l'énoncé. On peut donc souhaiter disposer d'un langage de spécification utilisable par des non informaticiens, nous avons déjà évoqué ce problème au paragraphe 2.1.

b) Fidélité

C'est une qualité non formalisable puisqu'elle exprime l'adéquation de l'énoncé initial au problème donné. On ne peut que se convaincre intuitivement de cette adéquation. La formalisation de l'énoncé sera d'autant plus convaincante qu'elle sera proche de l'énoncé informel. C'est une raison supplémentaire pour s'attacher, dans un premier temps, à énoncer le problème plutôt qu'à chercher à le résoudre en en donnant un algorithme.

En revanche si le problème est en partie caractérisé par des exemples précisant pour certaines données les résultats correspondants il faut être capable de vérifier l'adéquation de la spécification formelle à ces exemples et en ce sens un algorithme semble plus approprié qu'une

description formelle non constructive. En fait la solution consiste certainement à considérer spécification informelle et exemples de manière complémentaire (cf ch 1.4 , cf § 1.2. b)).

Le problème de la fidélité d'un énoncé formel est celui de la "quantité d'information" qu'on y inclut: l'énoncé ne doit pas être *sur-spécifié* car alors il restreint les possibilités de résolution. Il ne doit pas non plus être *sous-spécifié* puisque sa résolution peut donner des solutions qui n'ont pas de sens pour l'utilisateur. Le problème du "bon niveau" de spécification est encore malheureusement complètement ouvert.

c) Cohérence

Un énoncé est *cohérent* s'il ne contient pas de spécificateurs contradictoires ou encore si on ne peut pas en déduire de définitions contradictoires d'un même objet. Il arrive fréquemment qu'un énoncé informel soit incohérent , notamment lorsqu'il comporte de nombreuses définitions conditionnelles. Il faut se doter d'outils permettant de vérifier la cohérence comme les tables de décision utilisées en informatique de gestion.

Dans le cas d'un énoncé formel on préfère le terme *consistance* à celui de cohérence. Cette notion repose alors sur des définitions formelles précises et on peut définir des techniques de vérification de consistance (ch 1.3 § 1.4).

d) Un premier énoncé doit être complètement défini

Intuitivement on souhaite que l'énoncé décrive complètement le problème. On peut préciser cette notion en reprenant les concepts introduits aux paragraphes précédents :

Un énoncé est *complètement défini* si la relation entre les données et les résultats qu'il définit s'exprime directement ou non à l'aide des fonctions de base de l'univers.

Une vérification simple consiste donc à s'assurer que chaque intermédiaire introduit dans la spécification est soit une fonction de base, soit défini à l'aide d'autres intermédiaires.

e) Un premier énoncé doit être statique

La description d'un problème ne doit pas dépendre d'une solution particulière ce qui permettra une plus grande liberté lors de la recherche de solutions. De plus la description d'un algorithme n'est pas toujours compréhensible. On préfère disposer d'un énoncé décrivant, comme on le fait habituellement en mathématiques, la relation entre les données et les résultats : on parle alors d'*énoncé statique*.

f) Facilité de passer à un niveau algorithmique

Les qualités précédentes ne sont pas toujours suffisantes. Reprenons par exemple le problème du tri topologique (§ 3.1 b)). L'énoncé formel de la figure 6 correspond bien à la définition d'une relation d'ordre total compatible avec un ordre partiel donné. Le travail ultérieur du programmeur consiste à donner un algorithme de construction de Δ (ie de passer à une définition explicite, cf ch 1.3). Partant de l'énoncé précédent on peut envisager plusieurs façons de construire Δ [FIN, 78 a)]:

- l'ensemble E étant fini, il en est de même de $\mathcal{P}(E \times E)$; on peut alors parcourir cet ensemble en recherchant un élément vérifiant la propriété caractéristique. Ce type "d'algorithme" procédant par énumérations s'applique à une grande classe de problèmes, mais de manière si inefficace qu'il est irréaliste.

- Plutôt que d'explorer $\mathcal{P}(E \times E)$ de manière effrénée, il peut être raisonnable de chercher à construire Δ par approximations successives en construisant une suite (Δ_i) de $\mathcal{P}(E \times E)$ vérifiant "de plus en plus" l'énoncé. Il serait même intéressant d'initialiser cette suite à $I \cup \Gamma$ ce qui rend vraies deux conditions de &.

En fait, la deuxième solution conduit à manipuler des suites de relations qui ne sont pas des objets mathématiques élémentaires. Nous

ne la poursuivrons pas ici mais nous préférons partir d'un autre énoncé caractérisant le problème posé.

L'énoncé du § 3.1. b) est fondé sur la définition mathématique classique d'une relation d'ordre total. On peut penser que l'objet cherché est "trop riche" : pour définir Δ, il suffit de pouvoir "numéroter" les éléments de E de façon compatible avec Γ, ce qui revient à ne définir qu'une relation dont la fermeture transitive est Δ, en laissant implicite la transitivité (c'est ce qu'illustre la représentation de α sur la figure 12. Ceci correspond à une autre vision d'une relation d'ordre total Δ sur un ensemble fini E : une application α bijective de [1 : n] sur E (où n est le nombre d'éléments de E) telle que :

$$\alpha_i \, \Delta \, \alpha_j \iff i \leqslant j$$

En revenant au problème du tri topologique, on peut caractériser la suite α en exprimant qu'elle est bijective et que :

$$\alpha_i \, \Gamma \, \alpha_j \Rightarrow i \leqslant j$$ en utilisant le fait que

$$\alpha_i \, \Delta \, \alpha_j \Rightarrow i \leqslant j$$

et que $x \, \Gamma \, y \Rightarrow x \, \Delta \, y$

On est ainsi conduit au nouvel énoncé représenté sur la figure 12.

Lexique	énoncé formel
On cherche une suite α formée des éléments de E dans un ordre compatible avec Γ $\hat{\Gamma}$: négation de la relation Γ inj : associe à k l'ensemble des injections de [1..k] dans E* V : *de base* domaine des éléments de E Γ : relation binaire sur E sans circuit	*résultat* α : E* *tq* $\alpha \in$ inj(n) $\wedge (\forall i,j \in dom(\alpha)$ $j<i \supset \alpha_i \, \hat{\Gamma} \, \alpha_j)$ $\hat{\Gamma}$: *op* (x : E, y : E) b : BOOL *tq* b = $\neg(x \, \Gamma \, y)$ inj : *bnot* (k : ENTIER)e : $\mathcal{P}$(E*) *tq* $\forall u \in E^*$, $u \in e \Leftrightarrow$ [bi(u) = 1 $\wedge$ bs(u) = k $\wedge$ $(\forall i \; j \in [1..k]$, $u_i = u_j \supset i = j)]$ *donnée* E : $\mathcal{P}$(V) Γ : *op*(E, E) BOOL *tq* $\forall x_1 .. x_p \in D$ $x_1 \, \Gamma \, x_2 \wedge .. \wedge x_{p-1} \, \Gamma \, x_p \supset x_1 \neq x_p$

Figure 12 : Une nouvelle spécification du tri topologique.

On pourrait bien sûr considérer que ce nouvel énoncé est une transformation du précédent ce qui nécessiterait des règles de transformation peu évidentes.Il est plus naturel de considérer le nouvel énoncé comme énoncé initial mais alors se pose la question du choix du cadre permettant de formaliser un problème : il ne faut plus tenir uniquement compte de la facilité de la formalisation mais également de la facilité à passer à un niveau algorithmique, en s'efforçant d'introduire, de préférence aux outils typiquement ensemblistes, des outils plus informatiques tels que les suites.

Etant donné l'état actuel de l'Art, il semble difficile de disposer à priori de critères permettant de choisir un "bon" premier énoncé initial et il est certainement plus raisonnable d'admettre des retours en arrière lorsqu'on arrive à une spécification à partir de laquelle une solution algorithmique semble difficile à obtenir. On cherche alors un nouvel énoncé et si on est capable de prouver son équivalence au premier on n'a plus à se poser à nouveau les questions a)... d) précédentes.

Malheureusement cette démarche se heurte aux difficultés suivantes :

- on ne sait pas en général passer d'un premier énoncé à un autre énoncé fondé sur d'autres idées.
- on ne dispose pas de critères permettant d'assurer que les remises en causes successives permettrons d'obtenir finalement un énoncé convenable.

g) Extensibilité.

Il est souhaitable qu' un changement minime dans le problème conduise à un changement minime dans la spécification correspondante. On peut espérer que la construction progressive d'une spécification comme nous l'avons envisagée aux paragraphes 2 et 3 permet d'obtenir cette propriété : chaque information de l'énoncé informel se traduit par un prédicat ou une relation dans l'énoncé formel. Par exemple si on transforme l'énoncé du problème de la fraction irréductible (§ 1.1 a)) en

supprimant "premier" :
"Trouver deux nombres proportionnels à deux nombres donnés non nuls"
il suffit de supprimer le prédicat "prem" dans l'énoncé de la figure 1.

Malheureusement si dans cet énoncé on remplace "deux" par "trois" on obtient un énoncé tout à fait différent.

Pour tendre vers cette extensibilité il faut chercher à modulariser et à ""paramétriser" au maximum l'énoncé initial.

4.2.- Qualités d'un cadre de spécification de problèmes.

Nous essayons maintenant de dégager les qualités que l'on peut attendre d'un langage de spécification de problèmes.

4.2.1.- Facilité d'utilisation.

a) Il doit permettre d'écrire des spécifications compréhensibles (cf 4.1 a)).

b) Existence d'une méthode : au langage doit être associée une méthode de construction d'énoncés. Dans ce but il doit faciliter les descriptions structurées et modulaires.

c) Le langage doit permettre l'écriture d'énoncés statiques (§ 4.1 e)) ; en particulier les langages de programmation algorithmiques classiques sont de trop bas niveau en ce sens. En effet ils mélangent,des notions liées à la technologie des machines et des notions liées à la résolution algorithmique,à l'énoncé proprement dit du problème.

d) L'écriture d'un énoncé sera d'autant plus compréhensible qu'elle utilisera des concepts de haut niveau bien adaptés au problème. Ainsi il est utile d'étendre le langage ensembliste en introduisant des notions telles que les multiensembles où les listes apparaissent dans de nombreux problèmes. Le langage de spécification doit posséder des *primitives de haut niveau*.

e) Domaine d'application : on souhaite que le langage utilisé soit le plus universel possible. Un certain nombre de domaines en informatique ont développé leurs propres langages de spécification mais le champ d'application est en général fort limité (problèmes d'informatique de gestion par exemple, langages de manipulation formelle, langage de simulation, etc...).

f) Facilité d'apprentissage : c'est un critère indispensable à son utilisation .

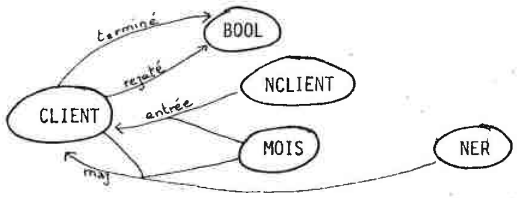
g) Existence d'un support graphique : un dessin valant mieux qu'un long discours il peut être agréable de schématiser tout ou partie de l'univers.

Ainsi on peut penser que si les organigrammes continuent à avoir une large audience auprès des programmeurs c'est certainement parce qu'ils constituent un moyen *de communication* d'un abord plus facile qu'un texte linéaire (éventuellement indenté) dans un langage quelconque.

Ainsi au niveau de spécifications on peut utiliser des schémas à différents niveaux :

- pour la description arborescente de fichiers

- pour la description plus agréable des profils des fonctions, c'est ce qui est fait dans Z [ABR, 78] ou dans [GOG, 77b] ; par exemple dans la figure 7 on pourrait schématiser les profils des différentes fonctions de la manière suivante :



- pour la description des modules :



etc.....

4.2.2.- Facilité d'effectuer des vérifications

a) Cohérence (consistance)
Prouver la cohérence d'une spécification nécessite pratiquement toujours qu'elle soit formelle. Nous verrons (ch. 1.2 §.2) qu'une technique classique consiste à exhiber un *modèle*.

b) fidélité : vérifier l'adéquation d'une spécification à un problème est un problème dual de celui qui consiste à construire une telle spécification. La construction de spécifications modulaires et structurées facilitera la vérification de la fidélité.

4.2.3.- Facilité de résolution

Le langage influe évidemment sur cette qualité des énoncés : si l'énoncé est déjà sous forme algorithmique le problème ne se pose plus.

Cependant si on souhaite pouvoir dériver le plus systématiquement possible un algorithme de l'énoncé initial (non algorithmique) il est souhaitable que ce dernier soit formel. On pourra utiliser alors des techniques de transformations (et de vérifications à posteriori) qui n'ont aucun sens pour des énoncés informels (cf ch.1.3).

4.2.4.- Vers un langage de spécification idéal ?

Comme nous l'avons déjà mentionné la conception de langages de spécification en est encore à ses balbutiements et nous verrons au paragraphe 5 qu'il n'existe pas actuellement de cadre permettant de spécifier des problèmes et possédant toutes les qualités précédentes. Mais il y a plus grave : il n'est pas évident que toutes les propriétés précédentes soient compatibles entre elles. Par exemple le formalisme peut être un obstacle à la lisibilité dans la mesure où il nécessite certaines habitudes de compréhension. De même peut-on avoir des primitives de haut niveau et être capable en même temps de spécifier n'importe quel type de

problèmes ? Nous essayons, sur le schéma de la figure 13, de mettre en évidence certaines relations entre les différentes qualités présentées aux paragraphes précédents.

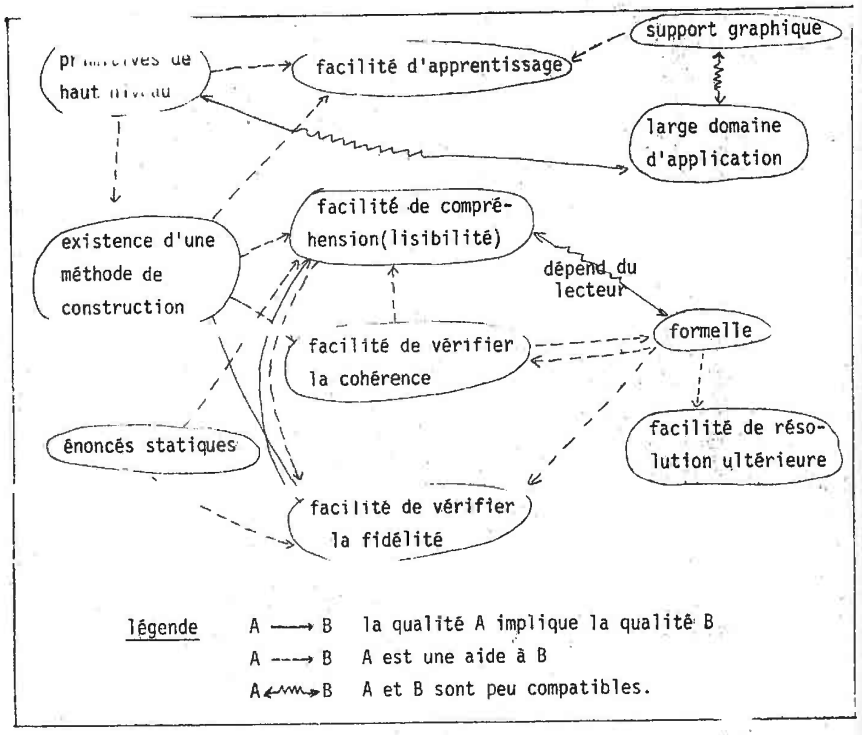


figure 13 : lien entre les différentes qualités d'un langage de spécification

Même si on ne pense pas trouver un langage universel possédant toutes ces priorités, on peut s'orienter vers un langage "noyau" possédant des primitives communes à tous les domaines (les entiers, les réels, les

booléens, etc...) et auquel on peut ajouter des extensions bien adaptées à certains domaines particuliers (fichiers, graphes ...). De plus, pour faciliter la compréhensibilité nous préconisons la coexistence d'une description informelle avec une spécification formelle.

Le paragraphe suivant est un rapide survol de quelques approches actuelles du problème de spécification .

5.- SUR LES LANGAGES DE SPECIFICATION DE PROBLEMES.

5.1.- Introduction.

De très nombreux travaux de recherche en informatique théorique ou en logiciel ont consisté à définir, à formaliser et à étudier des langages de programmation, procédureaux ou non, dont l'une des qualités essentielle est de permettre une résolution systématique, et si possible déterministe, des énoncés ou programmes qu'ils permettent d'exprimer.

Certes, au cours des années, on a élevé le niveau d'abstraction de ces langages en les rendant moins élémentaires et de plus en plus délivrés des contraintes de la machine, mais il ne semble pas que l'on soit arrivé à un niveau bien adapté à une première formalisation d'un problème. Conjointement, on a étudié des méthodes de programmation systématique, voire automatique, mais on a assez peu étudié les questions posées par la première mise en forme d'un problème. Nous essayons ici de faire un bref panorama, certainement pas exhaustif, de quelques langages (et éventuellement méthodes) de spécification d'un problème. Certains de ces langages sont (ou se disent) universels, d'autres, au contraire, sont spécifiques d'un domaine particulier.

5.2.- Quelques langages de spécification généraux.

a) SETL [KEN, 75], [SCH, 75]

SETL est un langage de *très haut niveau*, c'est à dire qu'il admet des structures de données très évoluées : ensembles, n-uplets (qui peuvent être manipulables comme des suites finies), applications (qui peuvent

être multivaluées et permettent, par exemple, de manipuler des multi-ensembles). A chacune de ces structures de données correspondent bien entendu des opérations spécifiques (opérations ensemblistes, sélection d'un élément d'un n-uplet). Notons aussi que dans sa version initiale SETL est un langage sans type et donc sans déclaration. Un programme SETL consiste en des définitions de fonctions ou d'objets composés et en des instructions au sens classique et dynamique des langages algorithmiques.

Pour illustrer ce langage, donnons une version possible en SETL de l'énoncé formel du problème des sous suites croissantes de longueur maximale (cf 3.1 a) figure 4).

Exemple 1 :

```
definef  lmax (alpha)        $ définit la longueur maximale d'une sous-suite
                               croissante de la suite alpha sur l'ensemble V
                               muni de l'ordre ≤

SUITES = { < x:  ∃y ∈ V| < x, y >∈ E >  : E in pow (UxU) and # E = # U and
                                          ∀< x, y >∈ E| ∀< u, v >∈ E | y ≠ v and
                                              U subset  V and # U le # alpha}

    $ définit l'ensemble des sous suites sur V dont la longueur est inférieure ou égale à celle de alpha.

definef  croiss(beta)        $ caractérise une suite croissante

    u = ∀i = 2.. # béta | béta (i - 1) ≤ béta(i)
  return u ;
end ;
definef  inclu(béta, alpha) $ caractérise l'inclusion entre deux suites

    BIJ = {< y : y ∈ E >: E ∈ # béta npow  i :{i = 1..# alpha }};
          $ ensemble des bijections de [1..k] dans un sous ensemble à k
            éléments de [1..n]  où k est la taille de béta et n celle
            de alpha
    u = ∃ f ∈ BIJ | ∀i = 1.. # béta| béta (i) = alpha (f(i))
    return u;
    end ;
```



```
SOUSSUITES = {béta in SUITES  |inclu(béta, alpha) and  croiss(béta)}

    m = [max : béta in  SOUSSUITES]  # béta  $ prend le maximum de longueur des suites béta de SOUSSUITES

return  m ;
end
```

L'opérateur # fournit le cardinal d'un ensemble ou la longueur d'un n-uplet, : permet de caractériser un itérateur, c'est à dire le parcours d'un ensemble. Bien qu'il manque un certain nombre de primitives telles que le produit cartésien (utilisé cependant ci-dessus pour simplifier l'écriture), l'ensemble de toutes les suites sur un ensemble fini et des propriétés telles que : "est injective", "est bijective". □

La lourdeur de l'énoncé précédent provient de l'absence de constructions primitives telles que "ensemble de suites sur V". Donnons une autre version de la formalisation de ce problème utilisant essentiellement des ensembles :

Exemple 2 :

```
    definef  lmax(alpha)
      index = {i : i = 1.. # alpha } ; $ intervalle 1..n si n est la longueur de alpha
    definef croiss(S)   $ la restriction de alpha à S est croissante
        b = ∀j in S  | ∀k  in S | j le k implies alpha(j) le alpha(k) ;
        return b ;
    end ;
    l  = [max :  S  subset index and  croiss(S) ]  # S ;
    return l ;
end
```

Ce nouvel énoncé est certainement plus clair à comprendre. Il n'est pas certain qu'il soit le plus naturel à construire car il nécessite de penser de prime abord à la manière de caractériser une sous suite croissante. □

En résumé SETL offre d'assez bons outils de formalisation d'un problème, bien qu'aucune méthodologie ne lui soit associée et que l'absence de certaines primitives restreigne la classe des problèmes aisément abordables (l'absence de concepts tels que celui de fichier complique l'expression de problèmes de gestion).

Mais il faut noter que SETL est en même temps un langage exécutable ce qui pourrait conduire le programmeur à se satisfaire du "programme" précédent et aboutirait à une exécution très inefficace. Pour améliorer ce programme on est alors tenté d'utiliser les structures de contrôle "constructives" du langage qui sont classiques (itérations, conditionnelles,..) et sont associées à la notion dynamique de variable. On se heurtait ainsi, dans les premières versions de SETL, à deux difficultés :

- Absence de règles systématiques de transformation
- Absence de structures de données de bas niveau permettant d'écrire des algorithmes efficaces.

Ainsi, l'opposition entre le niveau des structures de données (orientées spécifications) et celui des structures de contrôle (orientées exécutions) place SETL entre les langages de spécification et les langages de programmation. Conscients de ces difficultés les concepteurs de SETL, tout en continuant à chercher des techniques d'implantation fournissant des algorithmes efficaces, cherchent à étendre le langage [PAG, 79] , [DAV, 79] :

- Par l'introduction de mécanismes de définitions de types qui doivent permettre une spécification plus structurée des problèmes
- Par l'introduction de règles de transformation permettant de passer d'une spécification de haut niveau à une spécification algorithmique efficace.

b) ALICE [LAU, 76]

On pourrait placer la présentation de ce langage au paragraphe 5.3 dans la mesure où il a été essentiellement conçu pour spécifier des problèmes de nature combinatoire. Cependant, les concepts utilisés dans ce langage (ensembles, opérations ensemblistes,quantificateurs) sont suffisamment généraux pour en faire un langage permettant de spécifier des problèmes très divers. Si de nombreux problèmes mathématiques peuvent être exprimés dans ce langage (jeux, combinatoires, graphes, ...) il ne semble pas évident de l'utiliser pour des problèmes plus orientés vers la gestion.

Ici encore, l'étude d'une méthode de résolution utilisant largement des manipulations formelles, l'unification, l'indéterminisme a été le principal travail de LAURIERE. L'aspect mise en forme d'un problème a été très peu abordé (il est vrai que dans le cas de problèmes mathématiques la difficulté est souvent réduite)

c) Le projet CIP [BAU, 78 a] ,[BAU, 78 b]

CIP("computer aided, Intuition-guided Programming") développé à l'institut d'informatique de MUNICH, est un projet dont l'ambition est de définir un langage "à large spectre" permettant de spécifier, d'analyser puis de programmer efficacement un problème ainsi qu'un système, aussi automatique que possible, permettant de transformer un énoncé de ce langage en un autre.

En particulier, la conservation de différentes versions d'un énoncé, la documentation du programme, l'application de règles de transformations devraient être automatiques.

Le noyau initial de CIP est un ensemble de règles de transformation très voisines de celles développées par ARSAC [AFS, 77] ou BURSTALL et DARLINGTON [BUR, 77 b] permettant de passer d'une version récursive d'un énoncé à une version itérative puis à une représentation en machine. A ce niveau le but essentiel est l'efficacité du programme obtenu. Autour de ce noyau se sont développées des études tant méthodologiques (suppression de l'indéterminisme, représentation de structures de données ...) que théoriques (sémantique, justification des règles, etc...). Notons ici l'importance accordée à la spécification statique d'un problème qui se traduit par l'introduction d'outils de spécification de haut niveau (ensembles,quantificateurs, etc...) . Un énoncé formel apparaît comme un ensemble de déclarations de types et de fonctions exprimé dans un style Algol.

Exemple 3 :

Donnons dans le langage de CIP un énoncé du problème des sous-suites ascendantes analogue à celui de l'exemple 1

informellement :

Etant donné un tableau t à n éléments de mode m , où m est muni d'un ordre total $\leqslant$ et $n \geqslant 1$, trouver la longueur maximum d'une sous suite croissante du tableau t

spécification formelle :

mode *index* ≡ *nat* [1..n] ;
index *array* m t
nat ℓmax ≡
some *index* ℓ : ℓ = max { *index* k : ∃ *set* *index* S :
card(S) = k ∧ $crois_t$(S) }
where
funct $crois_t$(S) ≡ (*set* *index* S) *bool* :
∀ *index* j ∢ ℓ : j ∈ S ∧ ℓ ∈ S ∧ j ⩽ ℓ ⇒ t[j] ⩽ t[ℓ]

Une remarque immédiate sur l'unicité du maximum permet de transformer la définition de ℓmax en :

nat ℓmax ≡ max { *index* k : ∃ *set* *index* S : card(S) = k
∧ $crois_t$(S) } □

L'introduction explicite de l'indétermisme et de l'opérateur associé "choix de", la définition formelle de la sémantique des constructions [BRO, 78] , l'existence de règles de transformations permettant d'obtenir des algorithmes rendent ce projet séduisant. On peut cependant regretter l'absence d'outils spécifiques pour l'expression de certaines classes de problèmes (suites, par exemple) et l'inexistence actuelle d'une méthodologie de spécification de problèmes aussi bien que d'une méthodologie de mise en oeuvre des règles de transformation pour la construction de programmes.

d) MEDEE

Contrairement à SETL dont la philosophie est très orientée "structures de données", MEDEE est un langage non procédural permettant de spécifier un énoncé sous forme d'un ensemble de définitions caractérisant résultat et identificateurs intermédiaires [BEL, 78 a] . Introduit comme langage support de la méthode déductive de programmation [BEL, 78 b] [PAI, 79], il permet une expression statique (i.e indépendante de l'exécution) et explicite (puisqu'il s'agit d'un ensemble de définitions) de l'énoncé d'un problème. En ce sens on peut envisager de le considérer comme un langage de spécification.

En repoussant au chapitre 2.3 une définition plus complète de MEDEE contentons nous ici de donner un exemple

Exemple 4 :

à nouveau le problème de la fraction irréductible

lexique	définitions
Le résultat est un couple d'entiers x, y x ENTIER : proportionnel à a y ENTIER : proportionnel à b z ENTIER : pgcd de a et b u(i) ENTIER : ième terme de la suite vérifiant pgcd(u(i), v (i)) = pgcd(a, b) (algorithme d'euclide)	*res* = x, y x = a / z y = b / z z : *dernier* u(i) *pour* i *dans* l *jqa* *fin* *donnée* a, b
* u	

u ENTIER : ancienne valeur de v
v ENTIER : reste de la division de l'ancienne valeur de v par l'ancienne valeur de u
fin BOOL : condition d'arrêt, détermine l'indice du dernier terme de la suite u(i)
	u = @ v
	v = @ u *nod* @ v
	fin = (v = 0)
	$ u = a
	$ v = b
	$ fin = (b = 0)

Figure 14 : un énoncé MEDEE

Dans cet exemple la définition du résultat nécessite celle d'une suite de couples (u_i, v_i) caractérisée de manière récurrente dans le "module *u". β u précise la valeur du premier terme de la suite u . □

Si cet énoncé est statique il est cependant très constructif : il a fallu résoudre le problème pour passer de la spécification de la figure 2 à celle de la figure 14. Ainsi, sauf dans des cas simples pour lesquels l'énoncé d'un problème va de pair avec sa résolution, MEDEE sera de trop bas niveau et nécessite trop de travail pour qu'il soit considéré comme un langage de spécification satisfaisant. Nous verrons cependant au chapitre 2.3 que dans le cas de certains problèmes de gestion, une version étendue de MEDEE est d'un emploi raisonnable.

Ce que nous venons de voir pour MEDEE est bien sûr applicable dans le cas des autres langages applicatifs tels que LUCID [ASH, 76] , le langage utilisé par J. ARSAC dans [ARS, 77] , ou LISP (bien que certains "lispiens" convaincus puissent l'utiliser comme un langage de spécification [GRE, 79]) qui par la forme explicite de leurs "programmes" nécessitent de spécifier et résoudre conjointement les problèmes. C'est à fortiori vrai pour les langages de programmation classiques.

e) PROLOG

Fondé sur le calcul des prédicats du premier ordre dont les formules sont mises sous forme de clauses de Horn [WAR, 77a] le langage PROLOG devrait donc pouvoir être utilisé comme langage de spécification. Bien que nous esquissions les grandes lignes de ce langage au chapitre 1.3 paragraphe 3.2, nous pouvons dès maintenant donner un exemple de programme PROLOG

Exemple 5 :

Toujours le problème de la fraction irréductible
L'énoncé de la figure 2 peut s'exprimer en PROLOG sous la forme :


```
irréduct (x, y, a, b) ← propor(x, y, a, b) , prem(x, y)
propor (x, y, a, b)   ← mul(x, b, v), mul(y, a, v)
prem (x, y)           ← pgcd(x, y, 1)
pgcd (x, 0, x)        ←
pgcd (x, y, u)        ← pgcd(y, z, u), reste(x, y, z)
```

où une formule telle que A ← B, C signifie $B \wedge C \supset A$.

On peut aussi en donner une signification plus opératoire en considérant que A ← B, C est une déclaration de la procédure A dont le corps est B, C .

irréduct, propor, prem, pgcd sont des prédicats inconnus ; rest et mul sont des prédicats faisant partie du langage PROLOG dont le sens est immédiat : rest (x, y, z) (resp. mul (x, y, z)) s'interprète comme : z est le reste de la division entière (resp. est le produit) de x par y . □

La distance entre ce programme PROLOG et l'énoncé de la figure 2 n'est pas très grande, mais il a fallu néanmoins expliciter la définition de prem : c'est là que se trouve la difficulté de la résolution.

Cette nécessité d'obtenir des énoncés explicites, le style d'écriture très particulier induit par les clauses de Horn et l'absence de primitives ensemblistes de haut niveau empêchent PROLOG d'être un bon langage de spécification bien qu'il soit aussi universel que le calcul des prédicats. Cependant dans un certain nombre de problèmes portant sur des textes et guidés par une grammaire (compilation, traitement de langue naturelle...) l'expression directe en PROLOG apparaît assez naturelle [WAR, 77b] , [COL, 75] .

f) Langages orientés "types abstraits"

Un effort particulièrement important se développe actuellement pour faciliter la description des informations utilisées dans un problème (par exemple introduction de structures de données relationnelles [LES, 79]). Malheureusement la plupart des langages contenant le concept de "type abstrait" sont très dynamiques (!), ne permettent que des représentations concrètes et explicites de types et par là même sont très "orientés programmation", c'est le cas de CLU [LIS, 75] , d'ATM [MIN, 79] etc...

Quelques autres approches cependant permettent des spécifications statiques de types : c'est le cas des spécifications algébriques utilisées dans le système AFFIRM [MUS, 79] de MUSSER ou du langage OBJ développé par BURSTALL et GOGUEN [GOG, 79] , c'est aussi le cas des spécifications axiomatiques des langages IOTA [NAK, 79] ou ALPHARD [WUL., 75] .

Nous reviendrons au chapitre 2.1 sur la spécification algébrique de type. Signalons cependant tout de suite que, dans de nombreux cas, caractériser directement un type par un ensemble d'équations sur ses opérations n'est, à notre avis, ni simple ni naturel.

g) Le langage Z

Contrairement aux langages présentés jusqu'à présent, le langage Z créé par J.R. ABRIAL a pour but essentiel la spécification de problèmes. Plus précisément dans une version de 1977 ABRIAL a introduit deux niveaux de langages : Z_0 "orienté spécification" et Z_1 "orienté programmation" ainsi qu'un ensemble de règles de transformations (près de 400) permettant le passage de Z_0 à Z_1 et la programmation effective [ABR, 77a] , [ABR, 77b] [MEY, 78] .

Les articles plus récents [ABR, 78] sont consacrés essentiellement à l'aspect spécification que nous allons préciser maintenant.

La définition de Z repose sur quelques idées que nous défendons dans ce travail :

- Indépendance entre spécification d'un problème et implantation d'un programme qui le résout.
- Développement par raffinements successifs d'une spécification.
- Caractère statique d'une spécification
- Utilisation d'un formalisme dérivé directement du langage ensembliste

Utilisant une syntaxe de "style Algol", une spécification en Z est un ensemble de phrases composé :

- de définitions de *types* précisant des ensembles d'objets. On dispose des types primitifs (modes) booléens, numériques, caractères et de constructeurs de types (ensembles, tuples, fonctions, produit cartésien)
- de définitions d'objets de certains types
- de définitions de relations précisant les domaines et co-domaines d'une relation ainsi que sa nature (fonction partielle, totale, relation non fonctionnelle)
- d'assertions précisant les propriétés de certaines relations.

Enfin certaines extensions du langage permettent d'appréciables simplifications d'écriture.

Mais plus que sur la syntaxe nous voudrions insister sur la démarche proposée par J.R. ABRIAL et B. MEYER pour construire une telle spécification : l'informaticien lit successivement les différents éléments du cahier des charges et en donne pas à pas une traduction en Z . La première lecture met en évidence les types de base du problème, les principales relations et quelques propriétés successives.
Cette lecture est recommencée plusieurs fois de suite, chaque étape devant mettre en évidence de nouveaux objets, de nouvelles relations, de nouvelles assertions. L'arrêt de ce processus itératif correspond à l'obtention d'une spécification complète du problème. Au cours de cette démarche il est suggéré à l'informaticien de se poser des questions sur les objets, types ou relations qu'il manipule : cet ensemble est-il ordonné ? Cette relation est-elle fonctionnelle ? Cet ordre est-il partiel ou total ? etc...

Pour préciser ces notions sans trahir les idées de Z reprenons ici l'exemple décrit dans [ABR, 77c] et [ABR, 78] .

Exemple 6 :

Spécification en Z du problème des emprunts bancaires (cf.1.1, n)

```
Spécification : gestion des comptes clients
type
    CLIENT ;
définition
    ARG = NUM ;
relation
    montant : CLIENT — ARG ;
    nom     : CLIENT — CAR
# L'emprunt commence un mois donné et termine un mois donné. La mensualité des remboursements est le quotient du montant par la durée #
type
    ordered MOIS some = (début) ;  # MOIS est un ensemble ordonné contenant un élément particulier début #
assertion
    first (MOIS) = début ;
relation
    mdeb : CLIENT — MOIS
    mfin : CLIENT — MOIS
mensualité : CLIENT — ARG
assertion
    forall C in CLIENT then mfin(C) > mdeb(C) end ;
mensualité = lambda C in CLIENT ⇒
             montant (C) / (order(mfin(C)) - order(mdeb(C)))
             end
# Chaque mois il existe un certain nombre de clients actifs, ce sont ceux qui ont effectué un emprunt et qui n'ont pas fini de le rembourser #
relation
    set ACTIF : MOIS — CLIENT ;
assertion
    forall m in MOIS then
      forall C in ACTIF(m) then m > mdeb(C) end
    end ;
```



```
# Chaque mois, chaque client actif possède un crédit, un débit et effectue un versement #
définition
    MOICLIENT = subset m, C in prod (MOIS, CLIENT) where
                  C in ACTIF(m) ou m = mdeb(C)
                end ;
relation
    crédit : MOICLIENT — ARG ;
    débit  : MOICLIENT — ARG ;
    vers   : MOICLIENT — ARG ;
# Les crédit et débit sont respectivement les sommes, depuis le mois de départ jusqu'au mois considéré, des versements et des mensualités du client considéré #
assertion
    forall C in CLIENT then crédit(mdeb(C), C) = 0 and débit
                                                (mdeb(C), C) = 0 end ;
    forall m, C in MOICLIENT where m > mdeb(C) then
      crédit(m, C) = crédit(pred(m), C) + vers(m, C) and
      débit(m, C)  = débit(pred(m), C)) + rate(C)
    end ;
# Chaque mois des clients actifs ont terminé (leur crédit dépasse le montant), d'autres sont rejetés (la différence entre leur débit et leur crédit dépasse deux mensualités) #
relation
    set TER : MOIS — CLIENT
    set REJ : MOIS — CLIENT
assertion
    TER = lambda m in MOIS ⇒
            subset C in ACTIF(m) where crédit(m, C) ⩾ montant(C) end
          end ;
    REJ = lambda m in MOIS ⇒
            subset C in ACTIF(m) where
              (débit(m, C) - crédit(m, C)) > 2 x mensualité (C)
            end
          end ;
```

```
#  Chaque mois apparaissent de nouveaux clients qui seront actifs le
   mois suivant #
   relation
      set NOUV : MOIS  —  CLIENT
   assertion
      forall  m in MOIS  then
        forall  C in NOUV(m)  then  mdeb(C) = m  end
      end ;
      forall  m in  MOIS then ACTIF(m) ∩ NOUV(m) = VIDE end ;

#  Au début il n'y a pas de clients actifs. Chaque mois les clients
   actifs du mois suivant sont les clients actifs du mois considéré
   dont on exclutceux qui ont terminé ou qui sont rejetés ou qui ont
   terminé etauxquelson ajoute les nouveaux clients du mois #
   assertion
      ACTIF(début) = VIDE    ;
      forall  m in MOIS then
          ACTIF(next (m)) = (ACTIF(m) U NOUV(m)) - (TER(m) U REJ(m))
      end ;
```

L'exemple précédent montre que la syntaxe de Z est facilement compréhensible, qu'elle dispose de primitives de haut niveau (ensembles ordonnés, fonctions spécifiques, sous-ensembles ...) et que la décomposition en définition de types, d'objets, de relation et d'assertion donne une certaine structure à une spécification. On peut penser que ce style de langage est plus facilement utilisable par un informaticien que le langage ensembliste "pur" utilisé aux paragraphes précédents et aux chapitres 1.2, 1.3 et 1.4.

Si la nature formelle du résultat doit effectivement permettre des vérifications de cohérence et de complétude, la forme de la démarche s'apparente trop, à notre avis, à une *traduction pas à pas* dont se ressent la structure finale de l'énoncé : les regroupements de propriétés sont plus liés à la forme du cahier des charges qu'à leur nature logique. Par exemple il n'y a pas de distinction entre énoncé et univers : l'énoncé est l'une des relations de la spécification. Enfin demander au programmeur de réfléchir à chaque propriété que peut posséder un objet, peut conduire à l'énoncé de propriétés inutiles, redondantes voire gênantes pour la programmation ultérieure.

Au contraire, l'approche que nous proposons (cf. exemple ne consiste pas en une traduction pas à pas mais en la spécification du but du problème ce qui conduit à une *structuration logique de l'énoncé*, à la mise en évidence des fonctions de base de l'univers et à l'introduction plus systématique des propriétés des objets, ensembles ou relations. En bref notre démarche [cf. ch. 1.4] conduit à une reformulation guidée et hiérarchisée de l'énoncé du problème. Une conséquence est la nécessité d'une traduction en langue naturelle compréhensible par l'utilisateur, traduction qui n'est pas nécessaire dans la démarche proposée par ABRIAL où les commentaires sont directement issus du cahier des charges.

Pour compléter cette brève description de Z il conviendrait de préciser les règles de transformations "à sémantique constante" permettant de passer d'une première spécification à une version algorithmique. Par exemple on transforme une phrase de la forme $\forall x \in A$, $c(x)$ en

e ← *vrai* ; *pour* $x \in A$ *tant que* e *répéter* e ← $c(x)$ [MEY,78] .

Le nombre trop important de règles est un obstacle majeur à leur utilisation et il conviendrait de les hiérarchiser et de les regrouper. Signalons enfin l'utilisation de Z pour spécifier des problèmes faisant intervenir des notions de parallélisme[ABR, 79] .

h) Quelques autres approches.

Bien que la nécessité de définir des langages de spécification soit de plus en plus reconnue il semble qu'à l'heure actuelle il y ait peu de développement de tels outils, l'accent continuant à être mis sur les langages de conception et d'analyse de programme. Un bref survol des langages tels que HIPO, ISDOS ou SADT pourra être trouvé dans [DEM, 79]

Par manque d'informations sur ces langages nous ne développerons pas cette étude.

Citons également certaines études sur le rôle des langues naturelles dans la spécification qui ont, comme dans le projet SAFE [BAL,79a] pour but de construire un système aidant un utilisateur à poser son problème.

Il serait aussi intéressant de développer l'étude du système MODEL [PRY, 77] permettant à un utilisateur de spécifier pas à pas son problème et se chargeant d'ordonner les différentes "spécifications", d'en faire un contrôle de cohérence (comme en SNOOPY [HUC, 77] par une analyse de précédence). Il semble malheureusement que les "spécifications" soient déjà très explicites.

5.3.- Langages de spécification adaptés à un domaine particulier.

En faire une étude, ne serait-ce que partielle, sort largement du cadre de ce travail, aussi nous contentons-nous de présenter quelques domaines où ce genre de langage s'est bien développé.

a) L'analyse de gestion

La structure des données et des résultats dans les problèmes classiques d'analyse de gestion est presque toujours la même : fichiers en accès

direct ou séquentiel. La forme des algorithmes est elle aussi relativement constante : mise à jour (fusions) ou tri avec ou sans ruptures. Aussi ont pu être developpés depuis de nombreuses années à la fois méthodes et langages d'analyse de tels problèmes (méthode ARIANE [GAM, 77] , PROTEE [SIS, 78] etc...). La plupart de ces langages permettent des descriptions agréables des fichiers et peuvent en ce sens être considérés comme des langages de spécification. En revanche la description des algorithmes peut être de très bas niveau, voire pratiquement du langage machine :

- C'est le cas du langage GAP développé par IBM [IBM, 75] . Chaque programme GAP est constitué de trois parties :

  . une partie spécification des entrées dans laquelle on décrit la structure des fichiers en entrée, la forme de leurs enregistrements, les champs sur lesquels s'effectueront les ruptures ou les concordances, ainsi que la nature d'un fichier (maître ou secondaire).

  . une partie spécification des sorties très semblable à la précédente.

  . une partie traitement, sous forme d'une suite de conditions - actions utilisant un langage de très bas niveau avec des sauts et des formes d'instructions très élémentaires

- C'est le cas des méthodes WARNIER [WAR,75] ou JACKSON [JAC, 75] que nous détaillons un peu au chapitre 2.4.

- En revanche les tables de décisions, par nature dynamiques, peuvent permettre une spécification de l'énoncé sous forme de conditions-spécifications (spécification par cas) en remplaçant les actions par des définitions et en oubliant l'aspect séquentiel sous jacent. Elles ne permettent cependant pas de décrire directement la structure des objets.

- Citons enfin des projets plus élaborés tels que BDL : Business Definition Language [HAM, 77] dans lequel l'utilisateur peut spécifier, dans un langage proche de ses préoccupations, la forme des documents (entrée,

sortie), le flot qu'ils suivent et les transformations qu'ils subissent. Ce langage fournit des outils de spécification non procéduraux et ne nécessite pas de préciser la représentation des données : seules les contraintes logiques ont à être exprimées. La méthode de spécification associée est dirigée par les résultats ce qui permet une spécification descendante et modulaire.

Mais BDL doit fournir des programmes exécutables : on retrouve alors les principaux problèmes liés à l'inefficacité des langages de très haut niveau comme dans SETL.

b) Les bases de données.

La conception, la modification et l'interrogation de bases de données sont des cas particuliers de problèmes :

- La conception et la modification concernent la partie "univers"

- L'interrogation consiste à spécifier des énoncés.

Les principales idées que nous avons mentionnées jusqu'ici concernant l'indépendance entre la spécification et la résolution, l'utilité de se placer à un niveau statique, la nécessité de disposer d'un langage de spécification ont déjà été évoquées dans le cadre des bases de données :
- Recherche de l'indépendance entre la structure logique des objets et leur représentation physique
- Recherche de langages de "description logique" et d'interrogation de bases de données.

De nombreuses approches de description de la structure logique de bases de données sont voisines du concept d'univers tel que nous l'avons présenté : ensembles et fonctions sur ces ensembles : c'est le cas des modèles relationnels [COD, 74] ou des modèles en réseau [DAT, 75] .

Les langages de spécification quant à eux reposent directement sur la structure logique associée. On trouve dans [PIR, 76] et [PIR, 77] une intéressante comparaison de différents langages montrant qu'ils peuvent se ramener à un "langage noyau" dérivé du calcul des prédicats du premier ordre.

Mais si, dans l'étude que nous menons ici, nous envisageons de définir conjointement l'énoncé et l'univers du problème, dans le cas des bases de données les activités de conception et modification d'une part et d'interrogation d'autre part sont dissociées. Ainsi, la spécification d'un problème sur une base de données (interrogation) apparait comme un cas particulier de spécification de problèmes dont l'univers est déjà défini.

Néanmoins, il serait certainement très intéressant, pour la conception de langages de spécification, de s'inspirer des travaux effectués dans le domaine des langages d'interrogation de bases de données. En particulier, des efforts ont été faits pour se rapprocher d'une langue naturelle structurée [LAC, 77] , [CHA, 74] .

5.4.- En résumé : un essai de comparaison de quelques langages.

S'il résulte de la brève et incomplète étude précédente qu'il existe actuellement très peu de cadres ou de méthodes adaptés à la spécification d'un problème, un important travail de synthèse consisterait à extraire des différents travaux de "génie logiciel" les formalismes, techniques, outils et méthodes utilisables dans ce but : description et utilisation d'ensembles comme dans SETL ou Z, langage de haut niveau orienté utilisateur comme dans les bases de données etc....

On trouvera le début d'un tel travail dans [LAN, 79] où l'auteur pose le problème de la "programmation automatique" en comparant différents "systèmes", certains déjà d'un emploi classique comme les "packages", la plupart encore à l'état de prototypes ou de projets de recherche (programmation par exemple , "Knowledge-based systems").

Concluons ce paragraphe en proposant, à la manière de [DEM, 79] , les principales approches évoquées jusqu'ici (figure 15).

langages / qualités	langages de programmation	S E T L	A L I C E	C I P	M E D E E	P R O L O G
facilité de compréhension	SOUVENT (informaticien) NON (utilisateur)	OUI si lge ensembliste familier	oui	SOUVENT (informaticien mathématicien) NON sinon	oui	SOUVENT (mathématici NON sinon
il existe une méthode de spécification	non	non	non	non	oui	non
énoncés statiques	non	non	non	oui à un certain niveau	oui	oui
primitives de haut niveau	non	oui	oui	oui	non	non
large domaine d'application	dépend du langage	oui	non	oui	dépend des structures de données	oui
facilité d'apprentissage	dépend du langage	Oui si lge ensembliste familier	oui	oui (informaticien mathématicien)	oui	oui (mathématicie non sinon
support graphique	non	non	non	non	non	non
facilité de vérifier la cohérence	non	non	non	oui si énoncé structuré	oui si énoncé structuré	non
facilité de vérifier la fidélité	non	dépend de la structure de l'énoncé	dépend du problème	oui si énoncé structuré	oui	dépend du problème
facilité de résolution ultérieure	oui	Pb. efficacité	oui	règles de transf.	oui	oui Pb. efficacité
formelle	oui	oui	oui	oui	oui	oui
orienté univers ou énoncé	les deux	univers	énoncé	les deux	énoncé	univers

A F F I R M O B J I O T A	Z	G A P	Tables de décisions	WARNIER JACKSON	interrogation bases de données	Qualités
SOUVENT (informaticien mathématicien) NON sinon	oui si lge ensembliste familier	oui pour les données (informaticien)	oui	oui pour les données	oui	facilité de compréhension
non	non	oui	oui	pas explicite	non	il existe une méthode de spécification
oui	oui	non	possible	non	oui	énoncés statiques
non	oui	oui pour les fichiers	non	oui pour les fichiers	très lié à la base	primitives d haut niveau
oui	oui	non	non	non	non : dépend de la base	large domain d'applicatio
oui (informaticien mathématicien)	oui si lge ensembliste familier	oui (informaticien)	oui	oui	oui	facilité d' apprentissag
pour le domaine des opérations	non	cadre d'expression	cadre d'expression	oui	non	support graphique
oui	oui si énoncé structuré	oui en partie	oui en partie	oui en partie	dépend de la base	facilité de vérifier la cohérence
oui	oui	en partie	en partie	en partie	en partie	facilité de vérifier la fidélité
oui	en cours de développement	oui	oui	oui	oui	facilité de résolution ultérieure
oui	oui	oui	pas complètement	pas complètement	oui	formelle
univers	les deux	univers	énoncé	univers	énoncé	orienté uni ou énonc

CHAPITRE 1.2

SPÉCIFICATION FORMELLE DE L'UNIVERS D'UN PROBLÈME : LES SYSTÈMES D'INFORMATION

La spécification de l'univers d'un problème doit à la fois être *simple à construire, précise* et *bien adaptée à l'énoncé* puis à la *résolution du problème*. La précision et l'adéquation à la résolution peuvent être obtenues en utilisant la notion de système formel. La facilité de construction nécessite quant à elle de disposer d'un langage agréable, possédant des primitives de haut niveau et reposant sur des concepts intuitivement clairs.

Avec ces objectifs nous introduisons ici, d'une part le concept de *système d'information* qui apparaît comme un système formel permettant de spécifier l'univers d'un problème, d'autre part le langage *des s-descriptions* adapté à une construction progressive et structurée des systèmes d'information. Cette dualité entre l'aspect formel et l'aspect méthodologique constitue la ligne directrice de ce chapitre.

Ainsi, au paragraphe 1 nous discutons de l'intérêt des systèmes formels dans la spécification d'un univers. Puis nous définissons la notion de système d'information (§ 2) avant d'en étudier les propriétés fondamentales (§ 3). L'introduction, au paragraphe 4, des notions de *réalisation* et *d'extension* de systèmes d'information conduit (§ 5) à présenter le langage des s-descriptions et à définir des *opérations* de *transformation* de tels systèmes. La *paramétrisation* apparaît alors (§ 6) comme un autre moyen d'engendrer des systèmes d'information et permet de décrire (§ 7) de nouvelles opérations de construction.

1.- DE LA FORMALISATION DE L'UNIVERS D'UN PROBLEME.

1.1.- Utilité des systèmes formels.

Dans les exemples considérés au chapitre précédent la plupart des *propriétés des objets du contexte* étaient sous entendues (on ne précisait pas les propriétés de LIAISON, NPRIX, PERS, etc...). Mais il est évident que la résolution du problème va utiliser un certain nombre de ces propriétés. Il convient donc d'être capable de les mettre en évidence le moment venu : c'est l'une des qualités d'un cadre de spécification d'univers.

De plus, nous avons vu la nécessité de prouver la cohérence d'un énoncé : ceci nécessite qu'il soit exprimé dans un cadre permettant *d'effectuer des démonstrations*.

Plus que sur un univers, de nombreux problèmes portent sur une *classe d'univers*, c'est par exemple le cas du problème des liaisons aériennes (ch. 1.1. exemple 1). Il faut donc être capable de décrire de telles classes.

La nécessité d'introduire ou de prouver des propriétés et le fait qu'un problème ne porte pas toujours sur un seul univers nous incite à donner une vue plus formelle de l'univers d'un problème : *Une spécification d'un univers U est un système formel dont l'univers U est un modèle*. Ainsi l'énoncé d'un problème porte sur un système formel, les fonctions de base étant en fait des symboles fonctionnels, les différents modèles du système formel sont des univers du problème.

En examinant les quelques exemples qui précèdent on constate que les systèmes formels utilisés pour formaliser un problème peuvent être de types très variés :

- Les *théories du premier ordre*, c'est à dire les systèmes formels contenant les axiomes du calcul du prédicat du premier ordre et un certain nombre d'autres axiomes "propres" [SHO, 67] peuvent être utilisées pour formaliser certains problèmes simples dans l'énoncé desquels seules les variables-objets peuvent être quantifiées (cf. problème de la fraction irréductible, ch. 1.1 exemple 1).

- Mais bien souvent le premier ordre semble insuffisant pour énoncer précisément un problème : il est nécessaire de quantifier des symboles de fonctions ou de relations (cf. problème des sous suites croissantes maximales, ch. 1.1 § 3.1 fig. 5).
Malheureusement on dispose d'assez peu de résultats intéressants dans le cas de calcul des prédicats d'ordre supérieur à un. Aussi essaye-t-on de s'y ramener, éventuellement par le biais d'artifices consistant à considérer certaines opérations comme des symboles 0-aires. Dans la suite nous introduisons le concept de système d'information (en abrégé S.I.) :

> Un système d'information est un calcul des prédicats typé du premier ordre spécifiant l'univers d'un problème.

Un système d'information est un cadre formel ; pour spécifier un problème s'y rapportant il faut disposer d'un langage le décrivant. Comme nous l'avons déjà indiqué au chapitre précédent, nous pensons que le problème de la spécification est déjà suffisamment difficile en soi sans qu'on y ajoute des soucis liés à la résolution. C'est la raison pour laquelle nous cherchons un langage permettant de donner une définition *descriptive* des éléments (symboles fonctionnels) d'un S.I. plutôt qu'une définition *constructive* : la description à l'aide de support , ou *a-description*, d'un S.I. joue ce rôle.

Notons qu'il faudra, dans une étape ultérieure, disposer d'une définition *constructive* de ces symboles fonctionnels : il sera donc nécessaire de disposer à la fois d'un langage exprimant de telles définitions et de méthodes permettant de passer d'une description initiale à une description constructive. On aborde le problème de la *représentation* d'un S.I. par un autre, problème qui est abordé au chapitre 1.3 (§6.3) et dont la résolution est en partie facilitée par le fait que l'on utilise des systèmes formels.

Avant de définir et d'étudier plus précisément la notion de système d'information nous la présentons sur deux exemples.

1.2.- Exemples de systèmes d'information.

Exemple 1 : File d'attente

La spécification du problème de simulation (ch. 1.1 §.1.1 c)) fait apparaître une suite de files d'attente. Le S.I. "FILE" est décrit par la figure 1.

Cette description repose sur la notion "pré définie" de *suite* :

Convenons de noter $< x_k, \ldots, x_p >$ une suite quelconque à p-k+1 éléments, en admettant que $p < k$ correspond à la suite vide encore notée $< >$, alors

$$< x_k, \ldots, x_p > \sim y = < x_k, \ldots, x_p, y >$$

$$reste(< x_k, \ldots, x_p >) = < x_{k+1}, \ldots, x_p >$$

$$prem(< x_k, \ldots, x_p >) = x_k$$

$$| < x_k, \ldots, x_p > | = p-k+1$$

Une description complète du S.I. "SUITE" est donnée au paragraphe 7.2.2. La description de FILE exprime :

- qu'une file est une suite d'entiers. En ce sens on dit que le S.I. ENTIER* est le *système support* de FILE. Nous dirons alors que la figure 1 est une *s-description* de FILE.

- que la longueur d'une file est comprise, au sens large, entre 0 et 100 : la partie *invariant* de la description est donc un prédicat définissant un sous ensemble du système support.

description formelle

Profil

description informelle		
SI : FILE une file est une suite d'entiers ; sa longueur est positive et toujours inférieure ou égale à 100	f : FILE	FILE = ENTIER* *invariant* 0 ≤ |f| ≤ 100 *opérations*
ajtf est l'opération d'adjonction en queue de file	ajtf : *fonct* (FILE, ENTIER)FILE	ajtf *tq* ajtf(f, x) = f ~ x
retf est l'opération de suppression en tête de file	retf : *fonct* (FILE)FILE	retf *tq* reste(f) = reste(f)
têtef est l'accès en tête de file ; la file est, bien entendu, non vide	têtef : *fonct* (FILE)ENTIER	têtef *tq* têtef(f) = prem(f) *si* ¬vide(f)
videf est un prédicat indiquant si une file est vide	videf : *fonct* (FILE)BOOL	videf *tq* videf(f) = (|f| = 0)

Figure 1 : s-description du S.I. FILE

- que les différents symboles fonctionnels de FILE sont "ajtf", "retf", "tetef" et "videf". Leurs "domaines" et "codomaines", ce que nous appelons leurs *profils*, sont décrits dans la colonne centrale. La formule qui est précédée de *tq* (resp. de *si*) spécifie la relation entre les "données" et les "résultats" (resp. la condition que doit vérifier la "donnée" pour que l'appel soit licite) associée au symbole fonctionnel considéré, c'est la *post-condition* (resp. *pré-condition*) de ce symbole.

Notons un léger changement de notation par rapport au chapitre 1.1 : il est agréable pour la suite, à la fois d'un point de vue formel et d'un point de vue méthodologique, de dissocier la description des profils de celle des propriétés des opérations.

De manière plus formelle maintenant, la colonne de droite de la figure 1 décrit les *axiomes propres* du système formel FILE. Ils sont formés à partir

- De l'axiome associé à l'invariant :

$$0 \leqslant |f| \leqslant 100$$

- Des axiomes caractérisant chaque symbole ($\equiv$ est l'égalité formelle du système FILE) :

pour ajtf :		ajtf(f, x)	$\equiv$ f ~ x
pour retf :		retf (f)	$\equiv$ reste(f)
pour têtef :	$\neg$vide(f) $\supset$	têtef (f)	$\equiv$ prem(f)
pour videf :		videf(f)	$\equiv$ (|f| = 0)

La définition complète du S.I. FILE est donnée dans l'exemple 9 (§ 5.2)

Remarque 1 :

L'absence de la partie *si* d'une description d'un symbole fonctionnel est une convention d'écriture lorsqu'elle prend la valeur *vrai*. Notons qu'ici on aurait pu penser à adjoindre, par exemple à ajtf, la condition :

si $0 \leqslant |f| < 100$

En fait cette précondition se déduit de la post condition de ajtf et de l'invariant :

de $0 \leqslant |f'| \leqslant 100$, $f' \equiv f \sim x$ et $|f \sim x| \equiv |f| + 1$

on tire $0 \leqslant |f| + 1 < 100$, donc $0 \leqslant |f| < 100$.

Ce mécanisme est étudié au paragraphe 5.

Ceci montre que le S.I. FILE doit contenir les axiomes propres de son système support ENTIER* et précise ainsi le rôle d'un S.I. support :

- Les axiomes et symboles d'un système support font partie du S.I. qu'il permet de définir

- Mais dans un énoncé portant sur le S.I. FILE, seules les opérations de FILE peuvent être utilisées, à l'exclusion de celles de ENTIER* si ce dernier n'est pas défini comme un S.I. de base du problème considéré. Ceci donne un caractère structuré à la description des systèmes d'information.

Remarque 2 :

Dans la définition du S.I. FILE, le fait que les éléments "stockés" soient des entiers est tout à fait anecdotique et la description d'un tel S.I. sur un ensemble T quelconque serait identique. De manière analogue la valeur de la longueur maximale d'une file peut être un entier quelconque.

Ainsi à côté de S.I. "primitifs" tels que ENTIER, BOOL, etc... qui apparaissent dans la définition d'un S.I. S on trouve des S.I. *paramètres* qui font de S un constructeur de S.I. (ou encore une classe de S.I.). □

Exemple 2 : Multiensembles.

Une description informelle des multiensembles a été donnée au chapitre 1.1 (§ 3.2 a)) et nous en avons montré l'utilisation dans la spécification du problème des charges d'une entreprise.

La figure 2 présente une s-description du S.I. MULTENS1 des multi-ensembles dépendant d'un S.I. paramètre. On suppose ici que le S.I. paramètre VAL est muni d'un préordre total noté $\leqslant$.

Dans cette description on utilise encore un système support "suite" avec comme symboles fonctionnels :

dom(domaine d'une suite) : $dom(< x_k, \ldots, x_p >) = [k..p]$

bi(borne inférieure) : $bi(< x_k, \ldots, x_p >) = k$

bs(forne supérieure) : $bs(< x_k, \ldots, x_p >) = p$

tr(tranche) : $tr(< x_k, \ldots, x_p >, i, j) = < y_1 \ldots y_{\ell-i+1} >$

et $y_1 = x_i, y_2 = x_{i+1} \ldots, y_{\ell-i+1} = x_j$

De plus m_i représente l'élément d'indice i de la suite m, cette notation n'est définie que si $i \in dom(m)$.

On note que les paramètres peuvent être décrits par des prédicats plus généraux que ceux utilisés dans la description des symboles fonctionnels du S.I. considéré qui justifie une extension de la syntaxe des post conditions (cf "supm").

Il faut remarquer que cette définition de MULTENS1 ne correspond pas à celle envisagée au chapitre 1.1 dans laquelle un multiensemble sur VAL est une fonction de VAL dans ENTIER. Cette remarque conduit à une autre s-description du S.I. des multiensembles, noté maintenant MULTENS2, avec cette fois-ci comme système support

[VAL ⟶ ENTIER]

qui est le S.I. des fonctions de VAL dans ENTIER muni des opérations :

- appl (f, x) qui associe à une fonction f sa valeur en x
- ajout(f, x, y) adjonction d'un nouvel élément au domaine d'une fonction f avec pour valeur associée y
- chval(f, x, y) change la valeur d'une fonction f en x , la nouvelle valeur est y

lexique	profil	description formelle
Δ : MULTENS1		
VAL est le S.I. paramètre de MULTENS, il est muni d'un préordre total noté $\leqslant$.	VAL : *S.I.* $\leqslant$: *op* (VAL, VAL) BOOL	*param* VAL *muni de* $\leqslant$ *tq* $\forall$ v1, v2, v3, (v1 $\leqslant$ v2 $\vee$ v2 $\leqslant$ v1) $\wedge$ (v1 $\leqslant$ v2 $\wedge$ v2 $\leqslant$ v3 $\supset$ v1 $\leqslant$ v3)
Un multiensemble est une suite d'éléments de VAL		*fparam* MULTENS = VAL* *opération* apt *tq* vapt m = $\exists$ i $\in$ dom(m) *tq* m_i = v
apt teste l'appartenance d'un élément v au multiensemble m : il faut qu'il existe un indice i pour lequel la valeur de la suite en i est m.i	apt : *op* (VAL, MULTENS) BOOL	
ajtm est l'opération d'adjonction d'un élément à un multiensemble	ajtm : *fonct* (MULTENS, VAL) MULTENS	ajtm *tq* ajtm(m, v) = m ~ v
supm est l'opération de suppression d'un element dans un multiensemble.	supm : *fonct* (MULTENS, VAL) MULTENS	supm(m, v) = m' *tq* $\exists$ i $\in$ dom(m) *tq* m_i = v $\wedge$ tr(m', bi(m'), i-1) = tr(m, bi(m), i-1) $\wedge$ tr(m', i, bs(m')) = tr(m, i+1, bs(m)) *si* v apt m
Elle n'est définie que pour des éléments appartenant au multiensemble.		

Figure 2 : s-description du S.I. Multiensemble.

S.I. MULTENS2	VAL : SI ≤ : *op* (VAL, VAL)BOOL m : MULTENS apt : *op* (VAL, MULTENS) BOOL ajtm : *fonct* (MULTENS, VAL) MULTENS supm : *fonct* (MULTENS, VAL) MULTENS	*param* VAL *muni de* ≤ *tq* ∀ v1, v2, v3, (v1 ≤ v2 ∨ v2 ≤ v1) ∧ (v1 ≤ v2 ∧ v2 ≤ v3 ⊃ v1 ≤ v3) *fpcnm* MULTENS = [VAL ⟶ ENTIER] *invariant* ∀ v ∈ dom(m), appl(m, v) > 0 *opérations* apt *tq* v apt m = ∃ i ∈ dom(m) *tq* m_i = v ajtm *tq* ajtm(m, v) = *si* v apt m *alors* chval(m, v, appl(m,v)+1) *sinon* ajout(m, v, 1) supm *tq* supm(m, v) = *si* appl(m, v) = 1 *alors* supm(m, v) *sinon* chval(m, v, appl(m,v)-1) *si* v apt m
VAL est le S.I. paramètre de MULTENS, il est muni d'un préordre total noté ≤ . Un multiensemble est une fonction de VAL dans ENTIER dont l'image d'un élément quelconque est strictement positive. apt teste l'appartenance d'un élément à un multiensemble : il suffit que l'élément appartienne au domaine du multiensemble. ajtm est l'opération d'adjonction d'un élément à un multiensemble. supm est l'opération de suppression d'un élément dans un multiensemble Elle n'est définie que pour des éléments appartenant au multiensemble.		

Figure 3 : une autre s-description du S.I. Multiensemble

- dom(f) associe à une fonction son domaine
- supr(f, x) supprime x du domaine d'une fonction

On obtient alors la s-description de la figure 3.

Nous sommes donc en présence de deux spécifications formelles d'un même concept intuitif. Peut-on dire qu'elles sont équivalentes en un certain sens ? Ce qui nous intéresse ici c'est que des opérations telles que l'appartenance aient le même résultat dans les deux formalisations.

Cette notion de conservation des propriétés vis à vis des opérations "d'auscultation" permet de "changer" de S.I tout en spécifiant le même problème (notion de représentation d'un S.I. par un autre). □

1.3.- Systèmes d'information et types abstraits.

La forme des spécifications présentées dans les exemples précédents est inspirée de celle des définitions de *types abstraits* du langage ALPHARD [WUL, 75] .

1.3.1.- Notion de type abstrait

Lors de la spécification d'un problème on souhaite ne donner qu'un minimum de propriétés sur les objets traités. En particulier la plupart des propriétés sur ces objets sont des propriétés de fonctions portant sur ces objets : *on décrit un type d'objets en précisant les propriétés des fonctions associées*. Cette idée fondamentale que nous avons mise en évidence dans les exemples précédents est tout aussi importante lors de la résolution d'un problème : il s'agit de conserver le plus longtemps possible un maximum d'abstraction en évitant de préciser trop tôt la forme (la représentation) des objets. C'est d'ailleurs ce deuxième point de vue qui est le plus développé dans les travaux actuels : la notion de type abstrait est apparue historiquement [LIS, 75] , [WUL, 75] comme l'analogue au niveau des structures de données de la notion de module au niveau des structures de contrôle [DER, 74] .

1.3.2.- Les types abstraits comme outil de spécification.

Trois caractéristiques de la notion de type abstrait en font un bon outil de spécification :

- L'indépendance des objets vis à vis de toute représentation concrète
- L'introduction d'un cadre permettant de spécifier les propriétés des opérations
- Le regroupement "géographique" des propriétés d'un type donne une présentation propice à une spécification descendante : un type peut dépendre de "types primitifs" qui sont spécifiés à leur tour etc.... Notons cependant qu'il peut être utile de définir conjointement plusieurs ensembles d'objets, ce qui ne semble pas être le cas dans les différentes approches actuelles (cf. exemple de la pagination de la mémoire virtuelle où la mémoire physique est composée d'une mémoire principale et d'une mémoire secondaire).
En ce sens les S.I. sont des types abstraits.

1.3.3.- Les types abstraits comme outil de programmation.

L'introduction de types abstraits dans les langages de programmation présente de nombreux avantages :

- Elle évite, par l'introduction d'une distinction entre la structure logique et la structure physique d'un objet, qu'un utilisateur ne manipule directement la représentation d'objets. On précise ainsi clairement la distinction entre l'utilisateur d'un type et son réalisateur [MIN, 79] , [WUL, 75]
- Elle permet de définir des mécanismes de protection et de confidentialité interdisant certains accès à certains utilisateurs [DER, 79] ,

- Elle est adaptée à la conservation d'objets (au même titre que la compilation séparée classique permet la conservation de modules) et le partage d'objets entre plusieurs utilisateurs [BAN, 78] .

Les deux derniers aspects prennenttout leur intérêt dans le cadre des bases de données et des systèmes d'information (au sens logiciel habituel du terme et non pas au sens logique utilisé dans ce travail).

Cet aspect "ingenerie du logiciel" des types abstraits remonte aux classes de SIMULA [DAH, 72] et a notamment été développé par LISKOV [LIS, 75] WULF [WUL, 75] et apparaît dans des langages algorithmiques modernes tels qu'EUCLID [LAM, 76] , ADA [IGH ,79] , MEFIA [CUN, 79] ou ATM [MIN, 79] . Bien que ce point de vue soit actuellement en plein développement dans le cadre de langages algorithmiques classiques (fondés sur le concept dynamique de variable) nous ne nous y attarderons pas ici dans la mesure où nous nous attachons surtout aux problèmes de conception plus qu'aux problèmes de vie du logiciel.

1.3.4.- s-description de système d'information et description prê-post d'un type abstrait.

Nous avons déjà signalé l'analogie entre les s-descriptions des S.I. et les descriptions des types abstraits du langage ALPHARD (ce que l'on peut appeler description pré-post d'un type abstrait [DER, 79]). Dans l'un et l'autre cas on décrit des objets en termes d'objets prédéfinis ce qui permet de préciser les pré et post conditions des opérations. Mais dans les premiers exemples donnés en ALPHARD [WUL, 75] , [LON, 76] la notion dynamique de variable était prépondérante ce qui nécessitait des définitions constructives des opérations déjà trop éloignées d'une spécification initiale. De plus l'utilisation de variables implique l'utilisation d'une sémantique axiomatique ([LIV, 78]) élaborée. Il semble qu'actuellement les

concepteurs d'ALPHARD cherchent à se détacher du niveau dynamique en abordant, comme nous essayons de le faire dans ce travail, la résolution d'un problème par étapes progressives partant d'une spécification statique initiale et aboutissant à un programme efficace [BEN, 79] .

En résumé on doit pouvoir, sans beaucoup de difficultés, définir ou énoncer en ALPHARD les mêmes notions et propriétés que nous introduisons dans le cas des s-descriptions de S.I..

2.- DEFINITION D'UN SYSTEME D'INFORMATION.

2.1.- Eléments d'un système d'information.

Définition 1 :

Un système d'information S est un calcul des prédicats du premier ordre, égalitaire et typé (on dit encore théorie du premier ordre typée) [SHO, 67] ,[KLE, 52] . □

Pour détailler cette définition, précisons, en reprenant les notations de [REM, 74] , les éléments du quadruplet ($\mathcal{A}$lph , $\mathcal{F}$or , $\mathcal{A}$x , $\mathcal{R}$i) qui constitue le système formel S ($\mathcal{A}$lph est l'alphabet, $\mathcal{F}$or l'ensemble des formules, $\mathcal{A}$x celui des axiomes, $\mathcal{R}$i celui des règles d'inférences).

2.1.1.- Alphabet et sortes de S

a) Alphabet

$\mathcal{A}lph = \mathcal{F}ym \cup \mathcal{V}ar \cup \{ \equiv, \neg, \supset, \exists, (,) \}$ où :

- $\mathcal{F}ym$ est un ensemble de *symboles fonctionnels* ou *fonctions*
- $\mathcal{V}ar$ un ensemble de *variables*
- $\equiv$ désigne l'égalité formelle

b) Sortes

L'ensemble $\mathcal{S}or$ des *sortes* de S est un ensemble de symboles qui s'interpréteront comme les noms des ensembles apparaissant dans les domaines et codomaines des interprétations des symboles fonctionnels de

c) Profils

On associe à chaque symbole fonctionnel f de $\mathcal{F}ym$:

- un q -uplet $\overline{S_1}, \ldots, \overline{S_q}$ de sortes appelé la *source* de f ;

source(f) = $\overline{S_1} \ldots \overline{S_q}$

- Une sorte $\overline{S_b}$ appelée le *but* de f : but(f) = $\overline{S_b}$

Le q + 1 uplet $\overline{S_1}, \ldots, \overline{S_q}, \overline{S_b}$ est appelé le *profil* de f que l'on note de la manière suivante :

profil(f) = $\overline{S_1}, \ldots, S_q \rightarrow \overline{S_b}$

Avec cette notation, q est appelé *l'arité* de f .

Les symboles 0-aires de $\mathcal{F}ym$ sont les *constantes* de S .

Les variables de S (éléments de $\mathcal{V}ar$) sont des symboles 0-aires n'appartenant pas à $\mathcal{F}ym$ et chacune est d'une sorte unique. Ainsi la fonction profil est définie sur $\mathcal{V}ar$ et induit une partition de $\mathcal{V}ar$ en autant de sous ensembles qu'il y a de sortes dans S ; on note $\mathcal{V}ar(\overline{S_i})$ les variables de profil $\rightarrow \overline{S_i}$.

L'existence de $\mathcal{S}or$ et de profil fait de S un *système typé*.

2.1.2.- Formules de S

a) Schémas fonctionnels

L'ensemble $\mathcal{S}ch$ des *schémas fonctionnels* ou *termes* de S représente toutes les compositions de symboles fonctionnels ou de variables de S compatibles avec les profils.

De manière plus formelle :

L'ensemble $\mathcal{S}ch$ des schémas fonctionnels sur S est défini par :

$$\mathcal{S}ch = \bigcup_{\overline{S_i} \in \mathcal{S}} \mathcal{S}ch(\overline{S_i}) \quad \text{où}$$

$\{ \mathcal{S}ch(\overline{S_i}) \text{ tq } \overline{S_i} \in \mathcal{S}ch \}$ est l'unique solution du système à point fixe suivant :

$$\begin{cases} \mathcal{T}ch(\bar{S}) = \bigcup_{q \geq 0} \{ f \mid \mathcal{T}ch(\bar{S}_1) \ldots \mathcal{T}ch(\bar{S}_q) \text{ et } f \in \mathcal{F}ym \text{ , profil}(f) = \bar{S}_1 \ldots \bar{S}_q \rightarrow \bar{S} \} \\ \quad \cup \mathcal{V}ar(\bar{S}) \\ \text{pour } \bar{S} \in \mathcal{S}or \end{cases}$$

On étend la fonction profil sur $\mathcal{T}ch$ en posant profil(u) = $\bar{S}_i$ si $u \in \mathcal{T}ch(\bar{S}_i)$

b) Formules atomiques

L'ensemble $\mathcal{A}tom$ des formules atomiques de S est défini par :

$$\mathcal{A}tom = \bigcup_{\bar{S} \in \mathcal{S}or} \mathcal{T}ch(\bar{S}) \equiv \mathcal{T}ch(S)$$

Intuitivement, on ne peut égaler formellement deux termes que s'ils ont même profil.

c) Formules

L'ensemble $\mathcal{F}or$ des formules de S est l'unique solution du système à point fixe :

$$\mathcal{F}or = \mathcal{A}tom \cup \neg \mathcal{F}or \cup (\mathcal{F}or \supset \mathcal{F}or) \cup \bigcup_{x \in \mathcal{V}} \exists x \mathcal{F}or$$

2.1.3.- Axiomes de S

L'ensemble $\mathcal{A}x$ des axiomes de S est la réunion des ensembles suivants :

a) $\mathcal{T}rop$ est l'ensemble des *axiomes propositionnels* sur $\mathcal{E}$ = $\mathcal{E} = \mathcal{A}tom \cup \bigcup_{x \in \mathcal{V}ar} \exists x \mathcal{F}or$; c'est la réunion des trois ensembles suivants :

$\mathcal{T}rop_1 = \{ \varphi \supset (\Psi \supset \varphi) \text{ tq } \varphi, \Psi \in \mathcal{E} \}$

$\mathcal{T}rop_2 = \{(\varphi \supset (\psi \supset \eta)) \supset ((\varphi \supset \psi) \supset (\varphi \supset \eta)) \text{ tq } \varphi, \psi, \eta \in \mathcal{E}\}$

$\mathcal{T}rop_3 = \{((\neg\varphi \supset \neg\psi) \supset (\psi \supset \varphi) \text{ tq } \varphi, \psi \in \mathcal{E}\}$

b) $\mathcal{E}g$ est l'ensemble des *axiomes caractérisant l'égalité formelle* ; il consiste à exprimer que $\equiv$ est reflexive, symétrique, transitive et qu'elle est stable pour les symboles fonctionnels. $\mathcal{E}g$ est la réunion des trois ensembles d'axiomes suivants :

$\mathcal{E}g_1 = \{ x \equiv x \text{ tq } x \in \mathcal{V}ar(\bar{S}) \text{ et } \bar{S} \in \mathcal{S}or \}$

$\mathcal{E}g_2 = \{ (x_1 \equiv x_2 \supset x_1 \equiv x_3 \supset x_2 \equiv x_3 \text{ tq } x_1, x_2, x_3 \in \mathcal{V}ar(\bar{S}) \text{ et } \bar{S} \in \mathcal{S}or \}$

$\mathcal{E}g_3 = \{ x \equiv y \supset f\, u_1 \ldots u_{p-1} x\, u_{p+1} \ldots u_q \equiv f\, u_1 \ldots u_{p-1} y\, u_{p+1} \ldots u_q \text{ tq } x \equiv y \text{ et } f\, u_1 \ldots u_{p-1} x\, u_{p+1} \ldots u_q \equiv f\, u_1 \ldots u_{p-1} y\, u_{p+1} \ldots u_q \in \mathcal{A}tom \}$

c) $\mathcal{E}gs$ est l'ensemble des axiomes caractérisant l'égalité dans les sortes. De manière intuitive, une sorte formalise un ensemble, et on souhaite pouvoir comparer deux termes de même profil. Or l'égalité formelle $\equiv$ *n'est pas un symbole fonctionnel*, donc ne peut apparaître dans un terme. On associe à chaque sorte $\bar{S}$ de S un symbole $eg_{\bar{S}}$ représentant l'égalité.

Plus précisément tout S.I. S contient :

- La sorte BOOL de booléens et les opérations booléennes habituelles dont les symboles o-aires *vrai* et *faux*
- Pour chaque sorte $\bar{S}_i$ un symbole $eg_{\bar{S}_i}$ de profil $\bar{S}_i, \bar{S}_i \rightarrow$ BOOL
- Les axiomes $\mathcal{E}gs$ associant $eg_{\bar{S}_i}$ à l'égalité formelle

$$\mathcal{E}gs = \{ (eg_{\bar{S}_i} x\, y \equiv \textit{vrai} \supset x \equiv y) \wedge (x \equiv y \supset eg_{\bar{S}_i} x\, y \equiv \textit{vrai}) \text{ tq } \bar{S}_i \in \mathcal{S}or \quad x \text{ et } y \in \mathcal{V}ar(\bar{S}_i) \}$$

d) $\mathcal{S}ub$ est l'ensemble des *axiomes de substitution* :

Rappelons tout d'abord les définitions suivantes :

Définition 2 :

- Une occurrence d'une variable x est *liée* dans une formule φ si elle apparaît dans une sous formule ψ de φ de la forme $\exists x \psi$.
- Une occurrence d'une variable x est *libre* dans une formule φ si elle n'est pas liée dans φ .
- Une variable x est *liée* (resp. *libre*) dans φ si une occurrence de x est liée (resp. libre) dans φ - □

Définition 3 :

Soit φ une formule et x une variable. On dit qu'un terme u *est substituable* à x *dans* φ si x et u ont même profil et si, pour toute variable libre y de u , aucune formule φ de la forme $\exists y \psi$ ne contient d'occurrence de x libre dans φ . On note alors $\varphi[u/x]$ la formule obtenue en remplaçant chaque occurrence de x libre dans φ par u (l'utilisation de cette notation sous entend que les conditions précédentes soient vérifiées). □

La deuxième partie de la condition pour qu'un terme u soit substituable à une variable x exprime qu'aucune occurrence libre d'une variable y de u ne devient liée dans $\varphi[u/x]$.

Par exemple $u = y + 1$ n'est pas substituable à x dans $\varphi = \exists y\ x \equiv y + 4$
mais l'est dans $\psi = \exists z\ y \equiv 8 * z * x$

Alors l'axiome de substitution précise une propriété des formules quantifiées par $\exists$:

$\mathcal{Sub} = \{ \varphi[u/x] \supset \exists x \varphi \ \text{tq} \ \varphi \in \mathcal{For}$, $u \in \mathcal{Ch}$ et u *substituable* à x dans $\varphi \}$

e) L'ensemble $\mathcal{Axp}$ est formé des *axiomes propres* de S . Ce sont les axiomes qui permettent de distinguer un S.I. d'un autre. Par opposition les autres axiomes décrits ci-dessus sont appelés *axiomes logiques*.
Une forme particulièrement intéressante de ces axiomes propres correspond à la s-description des symboles fonctionnels (§ 5.1.1). Une autre forme est celle des définitions récursives utilisées dans les spécifications algébriques (ch 2.2).

2.1.4.- Règles d'inférence

Deux règles d'inférence caractérisent les S.I. :

a) Le modus ponens :

$$\varphi,\ \varphi \supset \psi \vdash \psi$$

b) L'introduction de $\exists$:

$$\varphi \supset \psi \vdash \exists x \varphi \supset \psi \qquad \text{si } x \text{ n'est pas libre dans } \psi$$

Rappelons alors

Définition 4 :

Une formule φ de S est un théorème de S , ce que l'on note $\vdash_S \varphi$ ou simplement $\vdash \varphi$ s'il n'y a pas d'ambiguité, si φ est un axiome de S ou si φ est déduit de théorèmes de S par application d'une règle d'inférence □

On note $\mathcal{Th}$ l'ensemble des théorèmes de S

Définition 5 :

Une formule φ de S est *fermée* si elle ne contient aucune variable libre. □

Alors, du modus ponens on peut déduire le (meta) théorème suivant :

Théorème 1 : méta-théorème de la déduction.

Soit $S[\varphi_1, \ldots, \varphi_p]$ le S.I. obtenu à partir du S.I. S en ajoutant les formules *fermées* $\varphi_1, \ldots, \varphi_p$ aux axiomes propres de S . Une formule φ de S est un théorème de $S[\varphi_1, \ldots, \varphi_p]$ si et seulement si $(\ldots(\varphi_1 \supset \varphi_2) \supset \ldots. \supset \varphi_p) \supset \varphi$ est un théorème de S □ .

Remarque : Pourquoi le premier ordre ?

Malgré les remarques que nous avons faites au paragraphe 1.1 sur l'insuffisance du premier ordre nous restreignons les systèmes formels considérés dans un tel cadre. Ceci essentiellement parce que les théories du premier ordre ont été bien étudiées et que l'on dispose sur elles de résultats in-

teressantsconcernant en particulier la résolution des énoncés (chapitre I.3 § .) ce qui n'est pas le cas pour les théories d'ordre supérieur. Mais il n'en reste pas moins nécessaire de manipuler des fonctions et des relations : dans ce but nous introduisons de nouveaux symboles de sortes :

$\mathcal{P}(\bar{S})$ (ensemble des parties)

$[\bar{S}_1 \rightarrow \bar{S}_2]$ (ensemble des fonctions)

etc...

avec une axiomatique fournissant les propriétés attendues (§ 7). En particulier à $[\bar{S}_1 \longrightarrow \bar{S}_2]$ est associé un symbole appl de profil

$$[\bar{S}_1 \longrightarrow \bar{S}_2] \ , \ \bar{S}_1 \longrightarrow \bar{S}_2$$

tel que appl(f , x) représente ce qu'habituellement on nomme f(x).

On reste ainsi dans un système formel pour lequel les variables ne peuvent être que o-aires.

2.2.- Composantes caractéristiques d'un système d'information.

Il ressort évidemment de la définition précédente que seuls certains éléments du quadruplet $(\mathcal{A}lph, \mathcal{F}or, \mathcal{A}x, \mathcal{R}i)$ permettent de distinguer deux systèmes d'information, les autres étant des constituants logiques communs à tous les systèmes.

a) Signature

On appelle *signature* d'un système d'information S le triplet

$\Sigma(S) = (\mathcal{F}or, \mathcal{F}ym, \text{profil})$

définissant les symboles fonctionnels de S , leur source et leur but.

b) Axiomes propres

L'ensemble $\mathcal{A}xp(S)$ est la deuxième caractéristique de S
Pour définir S il suffit d'en préciser la signature et les axiomes propres.

2.3.- Conventions de notation

a) En cas d'ambiguité on indice les éléments $\mathcal{F}ch$, $\mathcal{F}ym$, $\mathcal{F}or$ etc... d'un S.I. S par S

b) De manière tout à fait habituelle nous introduisons d'autres symboles logiques considérés comme des moyens d'abréger les formules :

$\varphi \vee \psi$	représente	$\neg\varphi \supset \psi$
$\varphi \wedge \psi$	représente	$\neg(\neg\varphi \vee \neg\psi)$
$\varphi \leftrightarrow \psi$	représente	$(\varphi \supset \psi) \wedge (\psi \supset \varphi)$
$u \neq v$	représente	$\neg(u \equiv v)$
$\forall x \varphi$	représente	$\neg\exists x \neg\varphi$

c) Dans l'utilisation de certains symboles fonctionnels nous préférons une notation infixée à une notation préfixée. Par exemple $a \leqslant b$ est utilisé à la place de $\leqslant (a, b)$

d) Pour simplifier les formules nous adoptons des contractions d'écriture :

d1) Il arrive que l'on ait à utiliser des symboles de prédicats [SHO, 67] , ou encore des symboles fonctionnels à valeur dans les booléens. On évite alors la manipulation explicite des symboles *vrai* et *faux* en remplaçant $u \equiv \textit{vrai}$ (resp. $u \equiv \textit{faux}$) par u (resp. $\neg u$).

Par exemple $a \leqslant b \equiv \textit{vrai}$ est remplacé par $a \leqslant b$
De manière analogue $b \equiv \textit{vrai} \leftrightarrow \exists\, a \ll \text{ident, bloc(ts), a} \gg \in \text{rel(ts)}$ est remplacé par $b \equiv \exists\, a \ll \text{ident, bloc(ts), a} \gg \in \text{rel(ts)}$

De même puisque d'après l'axiome $\mathcal{A}gs$
$\vdash eg_{\bar{S}}\, x\, y \equiv \textit{vrai} \leftrightarrow x \equiv y$

On convient de remplacer $eg_{\bar{S}}\, x\, y$ par $x \equiv y$.

d2) On suppose que chaque S.I. S contient le symbole *si alors sinon* de profil

$$\text{profil}(\textit{si alors sinon}) = \text{BOOL} \ , \ \bar{S}_i \longrightarrow \bar{S}_i$$

(En fait il faudrait introduire autant de tels symboles qu'il y a de sortes dans S . Par convention nous les notons tous de la même manière).

Ce symbole vérifie l'axiome *si vrai alors* x *sinon* y ≡ x ∧
si faux alors x *sinon* y ≡ y

d1) et d2) permettent, par exemple, d'écrire :

m' ≡ *si* appl(m, v) - 1 *alors* supr(m, v)
sinon chval(m, v, appl(m, v)-1) (figure 3)

d3) Si ∈ est un symbole du S.I. S (§ 7.1.2), comme à l'habitude :

∀x ∈ A φ (resp. ∃ x ∈ A φ) sera utilisé à la place de

∀x x ∈ A∧φ (resp. ∃x x ∈ A ∧ φ)

d4) Enfin, à la notation a ≤ b ≤ c on préfère a ≤ b ∧ b ≤ c

e) La lisibilité est améliorée par

-L'introduction de redondance dans certaines formules, par exemple à

∃a ≪ ident, bloc(ts), a≫ ∈ rel(ts)

On préfère ∃a : at ≪ ident, bloc(ts) , a≫ ∈ rel(ts)

ce qui explicite le profil de a

- Des signes syntaxiques supplémentaires et non significatifs :

. *tq* (tel que) dans ∃x : S *tq* φ à la place de ∃x : S φ

. des virgules ∃x, ∃y, ∀z φ à la place de ∃x ∃y ∀z φ

- Dans de nombreux cas, la liste des variables libres d'une formule φ, ainsi φ(x_1, ..., x_p), est préférée à φ

f) Enfin selon l'habitude nous autorisons la suppression de parenthèses en utilisant les règles de priorité courante.

3.- PROPRIETES FONDAMENTALES DES SYSTEMES D'INFORMATION

3.1.- Quelques théorèmes généraux sur les systèmes d'information.

Nous rappelons ici , sans les démontrer quelques propriétés du calcul des prédicats égalitaire et typé du premier ordre qui résultent des axiomes et des règles d'inférences précédentes [SHO, 67] .

Théorème 2 : introduction de ∀

si ⊢ φ ⊃ ψ et x n'est pas libre dans φ alors ⊢ φ ⊃ ∀x ψ

Théorème 3 : règle de généralisation

si ⊢ φ alors ⊢ ∀ x φ

Théorème 4 : règle de substitution

si ⊢ φ alors ⊢ φ [u_1/x_1 , u_2/x_2, ..., u_p/x_p]

Théorème 5 : méta-théorème de substitution

supposons pour $1 \leq j \leq p$, profil (x_j) = profil(u_j)

a) ⊢ φ[u_1/x_1, u_2/x_2, ..., u_p/x_p] ⊃ ∃x_1 ∃x_2..... ∃x_p φ

b) ⊢ ∀x_1 ∀x_2 ..... ∀x_p φ ⊃ φ [u_1/x_1, ..., u_p/x_p]

Théorème 6 : règle de distribution

si ⊢ φ ⊃ ψ alors ⊢ ∃ x φ ⊃ ∃ x ψ
et ⊢ ∀ x φ ⊃ ∀ x ψ

Théorème 7 : théorème de fermeture

Soit φ une formule dont les variables libres sont $x_1 \ldots x_p$. La formule ∀$x_1 \ldots$ ∀x_p φ est la *fermeture* de φ . Alors sous ces hypothèses

⊢ ∀$x_1 \ldots$ ∀x_p φ si est seulement si ⊢ φ

Définition 6 :

Une *fonction de vérité* sur un S.I. S est une application ν de l'ensemble

$$\mathcal{E} = \mathcal{A}\text{tom} \cup \bigcup_{x \in \mathcal{V}ar} \exists x \mathcal{F}or$$

des formules élémentaires de S dans {VRAI, FAUX} □

Définition 7 :

Une formule φ de S est une *tautologie* si, pour toute fonction de vérité ν , $\nu(\varphi)$ = VRAI

une formule φ est une *conséquence tautologique* de $\varphi_1 \ldots \varphi_p$ si, pour toute fonction de vérité ν , $\nu(\varphi_1) = \nu(\varphi_2) = \ldots = \nu(\varphi_p)$ = VRAI ⇒

$\nu(\varphi)$ = VRAI □

Théorème 8 : théorème des tautologies

Si φ est une conséquence tautologique de $\varphi_1, \ldots, \varphi_p$ et si

$\vdash\varphi_1, \ldots, \vdash\varphi_p$ alors $\vdash\varphi$

En particulier toute tautologie est un théorème. □

3.2.- Sémantique d'un système d'information

Un système d'information étant un calcul des prédicats du 1er ordre, la définition de sa sémantique repose sur les notions d'*interprétation* et de *modèle*. Nous nous contentons encore ici d'en rappeler les principales idées et les résultats les plus caractéristiques.

3.2.1.- Interprétation d'un système d'information.

Soit S un S.I. ayant pour sortes $\overline{S_1}, \ldots, \overline{S_n}$. une *interprétation* de S est un n+1 uplet IN = $(\hat{S}_1, \ldots, \hat{S}_n, in)$ où les S_i sont des ensembles non vides et in est une application associant à tout symbole

$F \in SYM(S)$ de profil $\overline{S_{i_1}}, \ldots, \overline{S_{i_p}} \longrightarrow \overline{S_j}$ une application in(f) de $\hat{S}_{i_1} \times \ldots \times \hat{S}_{i_p}$ dans $\hat{S}_j$. IN se prolonge aux schémas fonctionnels de S : si u est un schéma de profil $\overline{S_j}$, à k variables $x_1 \ldots x_k$ de profils respectifs $\overline{S_{i_1}} \ldots \overline{S_{i_k}}$, in(u) est une application de $\hat{S}_{i_1} \times \ldots \times \hat{S}_{i_k}$ dans $\hat{S}_j$.

3.2.2.- Modèle d'un système d'information.

L'application d'une interprétation IN d'un système d'information S se prolonge naturellement sur les formules *fermées* (définition 5) de S :

si $\varphi = u \equiv v$ alors in(φ) = VRAI si et seulement si in(u) = in(v)
si $\varphi = \neg\psi$ alors in(φ) = NON in(ψ)
si $\varphi = \psi \supset \rho$ alors in(φ) = in(ψ) ⇒ in(ρ)
si $\varphi = \exists x\, \psi$ avec profil(x) = $\overline{S}$, alors in(φ) = VRAI si et seulement si il existe $u \in \mathcal{S}ch(\overline{S})$ tel que in(ψ [u/x]) = VRAI

Définition 8 :

Une *instantiation* d'une formule φ à k variables libres $x_1, \ldots, x_k$ est une formule de la forme $\varphi[u_1/x_1, \ldots, u_k/x_k]$ où les $u_1 \ldots u_k$ sont respectivement substituables (déf. 3) à $x_1 \ldots x_k$ dans φ □

Définition 9 :

Une formule φ d'un S.I. S est *valide* pour une interprétation IN de S si in(φ') = VRAI pour toute instantiation φ' de φ □

On peut vérifier en particulier que les axiomes logiques d'un S.I. sont des formules valides pour toute interprétation. □

Définition 10 :

Un *modèle* ou *univers* d'un système d'information S est une interprétation R de S pour laquelle tous les axiomes propres sont valides . □

Cette définition permet donc de relier la notion formelle de S.I. avec celle d'univers d'un problème. □

Définition 11 :

Une formule φ de S est valide (dans S) si et seulement si elle est valide dans tout modèle de S . □

L'intérêt de la notion de modèle réside dans le résultat suivant qui donne une caractérisation des théorèmes d'un système d'information.

Théorème 9 : première forme du théorème de complétude de GÖDEL.

Une formule φ d'un système d'information S est un théorème de S si et seulement si φ est valide dans S □

3.3.- Consistance et complétude.

3.3.1.- Consistance.

C'est la formalisation de la notion intuitive de cohérence : un même objet ne peut pas être caractérisé de deux façons différentes et incompatibles.

Définition 12 :

Un système d'information S est *consistant* s'il existe une formule φ de S qui n'est pas un théorème ; ce qui signifie que $\mathfrak{Th} \not\subseteq \mathfrak{Sor}$ □

Autrement formulée, cette définition signifie qu'on ne peut prouver que toute formule de S est un théorème. □

La démonstration directe de la consistance est en général difficile et on peut utiliser plutôt la deuxième forme du théorème de GÖDEL ;

Théorème 10 : deuxième forme du théorème de complétude de GÖDEL

Un S.I. S est consistant si et seulement si il admet un modèle □

Plutôt que de chercher à construire directement et globalement un modèle d'un S.I., ce qui peut être compliqué et difficile (surtout si le système n'est pas consistant !) il est plus raisonnable d'adopter une (ou plusieurs) des attitudes suivantes :

- Utiliser la construction progressive d'une s-description (cf. § 5.2) en commençant par construire des modèles des S.I. primitifs et, en partant d'eux, construire un modèle d'un S.I. considéré.

- Utiliser les opérations globales de construction de S.I. puisqu'elles conservent la consistance (cf § 5, § 7)

- Utiliser la notion de *représentation* d'un système d'information par un autre : une représentation de S est un "modèle formel" de S (cf ch. 1.3) donc si cette représentation est consistante il en est de même de S .

3.3.2.- Complétude

Savoir qu'un système d'information est consistant est insuffisant en général, on souhaite pouvoir démontrer si un objet possède ou non une propriété ou, plus formellement, si une formule est, ou non, un théorème.

Définition 13 :

Un système d'information est *complet* s'il est consistant et si, pour toute formule φ , φ ou $\neg\varphi$ est un théorème. □

Cette propriété s'exprime également en termes de modèles :

Théorème 11 :

Un S.I. S est complet si et seulement si il admet un modèle et si tous ses modèles sont isomorphes. □

Démontrer la complétude n'est pas non plus une tâche aisée et on peut utiliser la description progressive d'un système pour y parvenir.

Exemple 3 :

Pour prouver la complétude du système MULTENS1 (exemple 2, figure 2), il suffit de montrer que :

- pour tout m de profil $\overline{MULTENS}$ et tout v de profil $\overline{VAL}$, v apt m ≡ *vrai* est un théorème ou ¬v apt m = *vrai* est un théorème, ce qui résulte d'un raisonnement par cas et de la complétude du système VAL*
- pour tout m, m_1 de profil $\overline{MULTENS}$ et tout v de profil $\overline{VAL}$, ajtm (m, v) ≡ m_1 ∧ ¬jatm (m, v) ≡ m_1 est un théorème ou encore m ~ v ≡ m_1 ∧ ¬m ~ v ≡ m_1 est un théorème ce qui résulte encore de la complétude de VAL*.
- le résultat est encore valable pour supm. □

Mais la complétude est souvent une propriété trop forte. Ainsi dans le cas de MULTENS1 on peut démontrer que

ajtm (ajtm (m, v1), v2) et ajtm (ajtm (m, v2), v1)

sont différents si v1 et v2 le sont. Mais intuitivement on souhaite qu'il s'agisse des mêmes multiensembles. C'est à dire que dans la description du système MULTENS on donne trop de propriétés (et en particulier on décrit la structure interne des objets de la sorte d'intérêt) ce qui va contraindre abusivement les possibilités de représentation. En fait, il suffit de décrire le comportement d'un multiensemble vis à vis de "l'opération externe" apt : deux multiensembles sont identiques s'ils donnent les mêmes valeurs à apt . GUTTAG [GUT, 78 b] introduit ainsi une propriété de complétude plus faible que la précédente : la *complétude suffisante*. Cette notion est liée au fait qu'un S.I. est défini à partir d'autres S.I. plus élémentaires : on parle d'*extension* d'un système par un autre. L'étude de ce concept permet de définir formellement la complétude suffisante. (§ 4.2.1.)

4.- TRANSFORMATION DE SYSTEMES D'INFORMATION

4.1.- Réalisation d'un système d'information.

Réaliser un S.I. S par un S.I. S' consiste à associer à chaque formule de S une formule de S' de telle manière que tout théorème de S soit associé à un théorème de S'.

Cette notion de réalisation est l'une des plus fondamentales puisqu'elle permet de transporter la "structure" d'un S.I..

Définition 14 :

Soit S un système d'information. On appelle *opération généralisée* associée à un schéma fonctionnel u à q variables $x_1 \ldots x_q$ le symbole $\tilde{u}$ de profil

$\overline{S_1}, \ldots, \overline{S_q} \longrightarrow \overline{S}$ où $\overline{S_1} \ldots \overline{S_q}$ sont les profils respectifs de $x_1 \ldots x_q$ et $\overline{S}$ est le profil de u

tel que $\tilde{u}\, u_1 \ldots u_q \equiv u\,[u_1/x_1 \ldots, u_q/x_q]$ pour tout q-uplet $u_1 \ldots u_q$ de profils respectifs $\overline{S_1}, \ldots, \overline{S_q}$

Définition 15 :

Soit S un système d'information. Une *préréalisation* de S est un couple (S', r) où S' est un S.I. et r une fonction associant :

- à chaque sorte $\overline{S}$ de S une sorte $r(\overline{S})$ de S'
- à chaque symbole fonctionnel f de S, une opération généralisée r(f) de S', de même arité que f.

Alors r se prolonge de manière canonique aux schémas fonctionnels puis aux formules de S ; on note encore r ce prolongement.

Une *réalisation* (S', r) de S est une préréalisation de S telle que les images par r des axiomes propres de S soient des théorèmes de S' □

Par abus de langage, on appellera encore réalisation l'application r .

Remarque :

Lorsqu'un S.I. S' est complet, il n'admet qu'un seul modèle à un isomorphisme près. On peut alors assimiler S' à son unique modele ce qui qui justifie l'autre définition suivante d'un modèle:

Un modèle d'un S.I. S est une réalisation (S', r) de S telle que S' soit complet.

Théorème 12 :

Tout modèle d'une réalisation (S', r) d'un S.I. S est un modèle de S. Ce théorème résulte immédiatement de la définition 15, de la définition 10 d'un modèle et de la définition d'une interprétation (§ 3.2.1.).

Corollaire :

Un S.I. S est consistant si et seulement si il admet une réalisation (S', r) telle que S' soit consistant. □

Définition 16 :

Une réalisation (S', r) du S.I. S est *conservative* si pour toute formule φ de S telle que $r(\varphi)$ soit un théorème de S' , φ est un théorème de S □

Ainsi, une réalisation conservative préserve exactement l'ensemble des théorèmes de S :

$\vdash_{S'} r(\varphi)$ si et seulement si $\vdash_S \varphi$

4.2.- Extension et restriction d'un système d'information.

Un cas particulièrement important de réalisation est celui où S' est un *sur-système* de S c'est à dire une *extension* de S .
Ce concept permet de définir un système d'information par enrichissements successifs d'un S.I. initial.

De manière analogue on peut définir un S.I. S' comme un *sous-système* ,

on dit une *restriction* d'un système S .

Extension et restriction sont deux techniques à la base de la construction de S.I. par transformations successives à partir d'un S.I. initial. On parle encore de construction par *enrichissement* ou *appauvrissement incrémental*.

Nous donnons au paragraphe 5.1 un cadre syntaxique permettant de mettre en oeuvre cette technique.

Définissons maintenant ces différents concepts.

4.2.1.- Extension d'un système d'information.

Définition 17 :

Une extension d'un système d'information S est une réalisation (S', r) de S où r est l'application identique □

Dans ce cas on dit encore que S' est une extension de S.

Ainsi, dire que S' est une extension de S équivaut à :

$$\begin{cases} \Sigma(S) \subset \Sigma(S') \\ \mathcal{A}xp(S) \subset \mathcal{T}h(S') \end{cases}$$

(Σ définit la signature, § 2.2. a)) les axiomes propres de S sont des théorèmes de S' .

(cf [REM, 74] ou [SHO, 67]).

Il résulte immédiatement de cette définition et de la définition 16 d'une réalisation conservative le théorème suivant :

Théorème 13 :

Soit S' une extension d'un S.I. S . S' est conservative si et seulement si

$$(\forall \varphi \in \mathcal{F}or(S) , \vdash_S \varphi \iff \vdash_{S'} \varphi) \qquad \square$$

Corollaire :

Toute extension conservative d'un S.I. consistant est consistante.

Notation :

- On note $\mathcal{E}xt(S, \Sigma", A")$ l'extension S' de la S.I. S telle que
  $\Sigma(S') = \Sigma(S) \cup \Sigma"$
  $\mathcal{A}xp(S') = \mathcal{A}xp(S) \cup A"$
- Par abus de langage, on préfère la notation $S[A"]$ à $\mathcal{E}xt(S, \Sigma", A")$ lorsque $\Sigma"$ ne contient pas de nouvelles sortes et que tous les symboles nouveaux de $\Sigma"$ occurrent dans $A"$.

Exemple 4 :

L'adjonction du symbole nbocc de profil $\overline{MULTENS}, \overline{VAL} \rightarrow \overline{ENTIER}$ et de l'axiome : nbocc(m, v) ≡ *si* apt(m, v) *alors* appl(m, v)
sinon 0
au S.I. MULTENS2 décrit par la figure 3 définit une extension conservative de MULTENS2. □

Définition 18 :

La *réunion* de deux systèmes d'information S_1 et S_2 est l'extension S' de S_1 définie par

$$S' = \mathcal{E}xt(S_1, \Sigma(S_2), \mathcal{A}xp(S_2)) \quad \square$$

Il est alors immédiat que

$$S' = \mathcal{E}xt(S_2, \Sigma(S_1), \mathcal{A}xp(S_1))$$

Et on note $S' = S_1 \cup S_2$ la réunion de S_1 et S_2

Il est immédiat que cette opération est associative, commutative et admet pour élément neutre le système vide.

Cette opération sur les systèmes d'information est appellée "Combine" dans le langage CLLAR [BUR, 77 a] .

Théorème 14 :

Soit S' une extension de S . S' est conservative si et seulement si tout modèle de S peut être étendu en un modèle de S' .

Démonstration :

a) Supposons que tout modèle de S puisse être étendu en un modèle de S' . Il faut montrer que toute formule φ de S, qui est un théorème de S', est un théorème de S :
si φ est un théorème de S' alors φ est valide dans tout modèle de S' (théorème 8, de complétude de GODEL), donc φ est valide dans tout modèle de S , donc, toujours avec le théorème 8 φ est un théorème de S .

b) Supposons maintenant que S' est une extension conservative de S

Lemme 1 :

avec cette hypothèse, pour toute formule φ de S , $S'[\varphi]$ est une extension conservative de $S[\varphi]$

Démonstration :

On utilise le meta-théorème de la déduction (théorème 1) qui porte sur des formules fermées.

Soit donc $\bar{\varphi}$ la fermeture de φ (rappelons, théorème 7, que $\bar{\varphi}$ est déduite de φ en quantifiant universellement chaque variable libre de φ). Alors avec le théorème 7 :

$$S[\varphi] = S[\bar{\varphi}] \quad \text{et} \quad S'[\varphi] = S'[\bar{\varphi}]$$

Ainsi, pour toute formule ψ de S :
si ψ est un théorème de $S'[\bar{\varphi}]$, alors $\bar{\varphi} \supset \psi$ est un théorème de S' (théorème 1) donc, par hypothèse sur S' et S , $\bar{\varphi} \supset \psi$ est un théorème de S , d'où on tire, avec le modus ponens , que ψ est un théorème de $S[\bar{\varphi}]$.
cqfd

Ce lemme se généralise immédiatement par récurrence sur p, au cas de $S[\varphi_1, \ldots, \varphi_p]$ pour $p \geq 1$ quelconque.

Lemme 2 :

On suppose toujours que S' est une extension conservative de S. Soit S_1 une extension de S ayant même ensemble de formules que S. Il existe une extension S'_1 de S' qui est une extension conservative de S_1.

Démonstration :

Posons $S'_1 = S' \cup S_1$ (cf définition 18).

Soit φ une formule de S qui est un théorème de S'_1. Dans la démonstration de φ interviennent des formules $\varphi_1 \ldots \varphi_p$ qui n'appartiennent pas à S' (sinon, puisque S' est une extension conservative de S, on a immédiatement φ théorème de S).

Ainsi φ est un théorème de $S'[\varphi_1 \ldots \varphi_p]$, donc de $S[\varphi_1 \ldots \varphi_p]$ d'après la généralisation du lemme 1, donc de S_1.

cqfd

On peut alors démontrer la deuxième partie du théorème :

- Si S n'est pas consistante alors c'est immédiat
- Si S est consistante, un modèle de S est un modèle d'une extension complète S_1 de S avec même ensemble de formules (théorème 18).

Donc, avec le lemme 2, il existe une extension S'_1 de S' qui est une extension conservative de S_1.

Avec le corollaire du théorème 13, S'_1 est consistante, alors tout modèle de S'_1 est un modèle de S'' et c'est une extension de l'unique modèle de S_1 qui est le modèle considéré de S.

cqfd

Théorème 15 :

Si S est un S.I. complet, toute extension consistante S' de S est conservative □

Démonstration :

Soit φ une formule de S qui est un théorème de S'. Alors $S[\varphi]$ est contenu dans S', donc consistante. On en tire donc, avec la complétude de S, que φ est un théorème de S.

cqfd

Définition 19 : équivalence de deux S.I.

Deux systèmes d'information sont *équivalents* si chacun est une extension de l'autre □

Cette définition signifie qu'ils ont même signature et même ensemble de théorèmes (et non pas même ensemble d'axiomes).

Définition 20 :

Une extension S' d'un S.I. S est *suffisamment complète vis à vis de* S (ou simplement *suffisamment complète* lorsqu'il n'y a pas d'ambiguités) si, pour tout terme u de S' dont le profil est une sorte de S, il existe un terme v de S tel que $u \equiv v$ soit un théorème de S' □

D'un point de vue méthodologique les extensions conservatives sont très importantes : elles n'ajoutent pas de théorèmes dans le S.I. de départ S. On peut ainsi hiérarchiser la construction d'un S.I. sachant qu'à chaque enrichissement on préserve les propriétés des systèmes déjà construits.

De manière complémentaire, la propriété de complétude suffisante précise qu'on introduit suffisamment d'axiomes concernant les termes dont le profil appartient au S.I. de départ.

4.2.2.- Information d'un système d'information.

Parmi les extensions d'un S.I. la classe de celles qui sont obtenues par simple adjonction de nouveaux axiomes présente un intérêt particulier pour la résolution des problèmes (ch 1.3).

Définition 21 :

Une information d'un S.I. S est une extension de S obtenue par adjonction de nouveaux axiomes □

Ce concept permet en particulier de rendre compte de l'association de valeurs à des données d'un problème : cette association est un ensemble d'axiomes que l'on ajoute à la spécification de l'univers du problème.

L'opération d'extension induit sur l'ensemble $\mathcal{I}(S)$ des informations d'un S.I. S un ordre partiel, parfois noté comme l'inclusion $\subset$, qui en fait un treillis complet, l'élément minimum étant S, le maximum étant le système inconsistant. Une étude de ce treillis, dans un cadre formel très voisin, a été développée dans [FIN, 74] et [REM, 74] .

4.2.3.- Restriction d'un système d'information.

Définition 22 :

On dit qu'un S.I. S est une restriction d'un S.I. S' si S' est une extension de S. □

On peut montrer que l'opération d'extension induit également sur l'ensemble $\mathcal{R}(S)$ des restrictions d'un S.I. S une structure de treillis complet dans lequel l'élément minimum est le S.I. vide, le maximum S.

5.- DESCRIPTION DE SYSTEMES D'INFORMATION.

Les concepts de réalisation et d'extension permettent de définir des S.I. à partir d'autres. Nous décrivons dans ce paragraphe des outils de construction de S.I. à partir de S.I. plus simples. Cette conception *incrementale* de la construction d'un S.I. permet une définition progressive des S.I. qui peut aussi bien être:

- descendante : à une étape donnée on construit un S.I. à l'aide de S.I. qui seront décrits ultérieurement. Ceci permet en particulier de s'abstraire, à un certain niveau de spécification, de certains détails.
- qu'ascendante.

Mais de manière complémentaire à cette approche liée à l'aspect hiérarchisé de l'univers d'un problème il est important de pouvoir spécifier "localement" chaque symbole fonctionnel que l'on introduit à une étape donnée. De façon plus précise, on souhaite considérer la spécification d'une opération comme la spécification d'un nouveau problème. La s-description des symboles fonctionnels répond à cet objectif puisqu'elle consiste à caractériser un symbole à l'aide de deux formules qui en sont ses pré-et post conditions.

Ce paragraphe fournit donc des outils de description de S.I. à deux niveaux :

- Un niveau "macroscopique" consistant en des opérations de transformation de S.I.. Certaines sont locales (extensions, restrictions, etc...) d'autres sont plus globales (s-description d'un S.I.)
- Un niveau "microscopique" consistant, par l'intermédiaire des pré-et post conditions, à nous ramener au problème général de la spécification, chaque post condition apparaissant comme la spécification d'un nouveau problème.

5.1.- Opérations de transformations simples de systèmes d'information.

5.1.1.- s-description d'un symbole fonctionnel

a) Sortes d'intérêt d'un système d'information

La construction incrémentale des S.I. conduit à *stratifier* l'ensemble $\mathcal{S}or$ des sortes d'intérêt d'un S.I. S en regroupant les sortes introduites à une même étape. En particulier on associe à tout S.I. S un sous ensemble $\mathcal{S}ori$ de $\mathcal{S}or$ que l'on appelle *ensemble des sortes d'intérêt* de S . $\mathcal{S}ori$ va être précisé pour chaque type d'opération qui permet de construire un S.I. . Le complémentaire de $\mathcal{S}ori$ dans $\mathcal{S}or$ est l'ensemble $\mathcal{S}ore$ des *sortes externes* de S .

Par exemple MULTENS est la sorte d'intérêt de MULTENS2 ; $\overline{\text{VAL}}$, $\overline{\text{BOOL}}$, $\overline{\text{ENTIER}}$ en sont les sortes externes (cf. figure 3) □

Bien souvent un S.I. n'a qu'une seule sorte d'intérêt, on dit alors qu'il est *uni-sorte*. Dans ce cas on note canoniquement $\overline{S}$ la sorte d'intérêt unique du S.I. S. Dans le cas contraire les systèmes d'information sont dits *multi-sortes*.

b) Définition d'une s-description d'une opération

Une s-description, ou description pré-post, d'un symbole fonctionnel f est un couple formé :

i) d'une *description du profil* de f , de la forme

$$f : \left\{ \begin{array}{c} op \\ fonct \end{array} \right\} (u_1, \ldots, u_k)u$$

- *op* précise que f sera utilisé de manière infixée, *fonct* qu'il sera utilisé de manière préfixée
- Les u_i et u sont soit des sortes, soit des noms de S.I. uni-sorte et dans ce dernier cas ils représentent la sorte d'intérêt du S.I. considéré

Par exemple

apt : *op* (VAL, MULTENS) BOOL

ajtm : *fonct* (MULTENS, VAL) MULTENS

(figure 3) sont des descriptions de profils

ii) d'une *s-description des axiomes* de f , de la forme

$f(x_1, \ldots, x_p) = y$

tq φ

si ψ

- φ est une formule caractérisant la relation entre y et $x_1, \ldots, x_p$
- ψ est une formule précisant une précondition sur $x_1 \ldots x_p$ (intuitivement, ψ permet de caractériser le domaine de définition de f) . Par convention l'absence de ψ signifie qu'elle est égale à *vrai* .

Par exemple

supm(m, v) ≡ m'

tq $\exists i \in dom(m)$ *tq* $m_i \equiv v \wedge tr(m', bi(m'), i-1) = tr(m, bi(m), i-1) \wedge$
$tr(m', i, bs(m')) = tr(m, i+1, bs(m))$

si v apt m

Définition 23 : Avec les notations ci-dessus,

- Le profil associé à la s-description de f est profil(f) = $u_1, \ldots, u_k \rightarrow u$
- L'axiome associé à la s-description de f est
  $\psi \wedge f\, x_1 \ldots x_p \equiv y \supset \varphi$

Notation :

- Dans une s-description on utilise le symbole = à la place de ≡
- Lorsque φ est de la forme y = u on admet la forme simplifiée suivante

$$\left\{ \begin{array}{l} f \ \ tq \ \ f(x_1, \ldots, x_p) = u \\ \quad si \ \psi \end{array} \right. \quad \text{à la place} \quad \left\{ \begin{array}{l} f(x_1 \ldots x_p) = y \\ \quad tq \ \varphi \\ \quad si \ \psi \end{array} \right.$$

5.1.2. s-description d'une extension d'un système d'information

On s'intéresse ici aux extensions des S.I. obtenus par adjonction de nouveaux symboles caractérisés par des s-descriptions. C'est un des moyens les plus simples de construire un S.I. par enrichissements successifs. Cette oépration correspond à l'opérateur "Enrich" du langage CLEAR [BUR, 77a] .

Une s-description d'une extension, ou plus brièvement une *s-extension* d'un S.I. S se présente sous la forme suivante :

profil	énoncé formel
S.I. S' = S *étendu par*	
P_1 ⋮ P_k	s-A_1 ⋮ s-A_k

où chaque couple $(P_i, \text{s-}A_i)$ est une s-description d'un symbole f_i n'appartenant pas à S .

Exemple 5 :

S.I. MULTVAL = MULTENS1 *étendu par*	
nbocc : *fonct*(MULTENS, VAL) ENTIER	nbocc *tg* nbocc(m, v) = *si* v apt m *alors* card{i ∈ dom(m) *tg* m_i = v } *sinon* 0

MULTVAL est une s-extension de MULTENS1 . card et { } sont définis au paragraphe 7.1.2. avec leur sens habituel. □

Définition 24 : Avec les notations précédentes :

Le S.I. S' défini par la s-extension précédente de S est l'extension suivante de S

$$S' = (S \cup S_1 \cup \ldots \cup S_p)\ [A] \quad \text{où}$$

- $S_1 \cup \ldots \cup S_p$ est la réunion des plus petits S.I. dans lesquels apparaît une sorte utilisée dans la description des profils et qui n'appartient pas à S , ou un symbole fonctionnel utilisé dans la s-description des axiomes et qui n'appartient pas à S .

- A est la réunion des axiomes associés aux s-descriptions des nouveaux symboles .

Les sortes d'intérêt de S' sont celles de S

On note S' = s-$\mathcal{E}$xt (S, {P_1, s-A_1, P_2, s-A_2, ... }) □

Par exemple, la s-description de l'exemple 5 définit MULTVAL par :

MULTVAL = (MULTENS ∪ ENTIER ∪ $\mathcal{P}$(ENTIER)) [nbocc(m, v) ≡ n ⊃
n ≡ *si* v apt m
alors card {i ∈ dom(m) *tg* m_i ≡ v}
sinon 0]

L'introduction de ENTIER est nécessitéepar le profil de nbocc ; celle de $\mathcal{P}$(ENTIER) , qui représente le S.I. "ensemble des parties de ENTIER" (cf. § 7.1.2.), par l'utilisation de "card" et de " {.... }" .

Ainsi une s-extension peut agrandir considérablement un S.I.. Se posent alors les questions cruciales de savoir si une telle extension est conservative ou suffisamment complète. Il faudrait chercher des critères sur la forme des s-descriptionsdes axiomes pour caractériser de telles extensions.

Remarque :

Lorsque S est uni-sorte, le fait que la sorte d'intérêt de S' soit celle de S contredit la notation canonique identifiant la sorte d'intérêt au nom du S.I..

Dans ce cas nous convenons de substituer $\overline{S'}$ à $\overline{S}$ dans tous les profils de S' et dans $\mathcal{S}or(S')$. Ainsi, S' est une réalisation de S (avec $r(\overline{S}) = \overline{S'}$) plus qu'une extension.

5.1.3.- s-description d'une restriction d'un système d'information.

C'est le cas où on ne conserve qu'un sous ensemble des symboles fonctionnels d'un S.I. et les axiomes les caractérisant. Méthodologiquement, ceci correspond à une *abstraction* de certaines propriétés d'un S.I.. On retrouve sous cette forme l'opérateur "Derive" du langage CLEAR [BUR, 77a] .

Soit S un S.I. . Une s-description d'une restriction de S , ou s-restriction de S, se présente sous la forme

S.I. S' = S *restreint à* $f_1, \ldots, f_p$

Exemple 6 :

S.I. COMPTEUR SIM = ENTIER *restreint à* +, -, ≤, ≥, <, > □

Définition 25 : Avec les notations précédentes :

S' est la restriction de S définie par

i) $\mathcal{S}ym(S')$ est l'ensemble des symboles de $\mathcal{S}ym(S)$ apparaissant dans l'un au moins des axiomes permettant de caractériser l'un des f_i .
De manière plus favorable, notons UTIL l'application associant à tout sous ensemble $\mathcal{S}$ de $\mathcal{S}ym(S)$ les symboles apparaissant dans les axiomes caractérisant les éléments de $\mathcal{S}$. Alors $\mathcal{S}ym(S')$ est le plus petit point fixe de l'équation :

$$\mathcal{S}ym(S') = \text{UTIL } \{f_1, \ldots, f_p\} \cup \text{UTIL } (\mathcal{S}ym(S'))$$

ii) $\mathcal{S}or(S') = \{ \overline{S} \in \mathcal{S}or(S) \text{ tq } \exists f \in \mathcal{S}ym\ S') \text{ tq } \overline{S} \text{ dans profil } (f)\}$

$= \{ \overline{S} \in \mathcal{S}or(S) \text{ tq } \overline{S} \text{ dans profil}(\mathcal{S}ym(S')) \}$

où $\overline{S}$ *dans* profil(f) $= \exists \overline{S}_1, \ldots \overline{S}_{i-1}, \overline{S}_{i+1}, \ldots, \overline{S}_q, \overline{S}_{q+1}$

$\ast$. *tq* profil(f) $= \overline{S} \ldots \overline{S}_{i-1}\ \overline{S}\ \overline{S}_{i+1} - \overline{S}_q \rightarrow \overline{S}_{q+1}$

Par convention, et avec la même remarque qu'au paragraphe 5.1.2.

$$\mathcal{S}ori(S') = \mathcal{S}ori(S)$$

iii) $profil_{S'}$ est la restriction à $\mathcal{S}ym(S')$ de $profil_S$

iiii) $\mathcal{E}xp(S')$ est le plus petit sous ensemble de $\mathcal{E}xp(S)$ contenant les symboles de $\mathcal{S}ym(S')$.

on notera parfois S' = s-$\mathcal{R}es(S, \{ f_1 \ldots, f_n \})$ □

En fait ici sont réduits à la fois l'ensemble des symboles et celui des sortes. On pourrait ne pas modifier ce dernier ou alors le modifier de manière "contrôlée" en précisant, comme dans l'opération "Derive" de CLEAR, les sortes que l'on veut trouver dans S' .

Notons également, qu'en général, S' contient plus de symboles que ceux que l'on souhaite conserver : ce sont des symboles "auxiliaires" (on parle encore de fonction cachée [GUT, 78a]) permettant de définir les autres symboles.

Se pose également ici le problème de la caractérisation de la restriction : si S est consistante, S' l'est nécessairement mais dans quels cas S est-elle une extension conservative ou suffisamment complète de S' ? Ici encore il faudrait rechercher des critères caractérisant une "bonne" restriction.

5.1.4.- s-dérivation d'un système d'information

Bien souvent un S.I. est déduit d'un autre par suppression de certains symboles et adjonction d'autres : on dit qu'il est *dérivé* du premier. La *s-dérivation* peut ainsi être définie comme la composition d'une s-restriction et d'une s-extension ; c'est l'équivalent des extensions de type ("form extensions") en ALPHARD [WUL, 75]

Soit S un S.I. . Une s-dérivation S' de S se présente sous la forme

S.I. S' = S *restreint à* $f_1 \ldots f_n$ *étendu par*	
P_1 | P_k	s- A_1 | s-A_k

Chaque couple $(P_i, s\text{-}A_i)$ est une s-description d'un symbole g_i n'appartenant pas à $\mathcal{Sym}(S)$.

Définition 26 : Avec les notations précédentes :

La s-dérivation S' de S est définie par :

$$S' = s\text{-}\mathcal{R}es\ (s\text{-}\mathcal{E}xt\ (S, \{P_1, s\text{-}A_1, \ldots, P_n, s\text{-}A_n\}), \{f_1, \ldots, f_n\})$$ □

Exemple 7 :

Le S.I. COMPTEUR peut être défini simplement à partir du S.I. ENTIER [WUL, 75] .

S.I. COMPTEUR = ENTIER *restreint à* <, >, ≤, ≥ *étendu par*	
incr : *op* (COMPTEUR) COMPTEUR	incr(x) = x' *tq* x' = x + 1
decr : *op* (COMPTEUR) COMPTEUR	decr(x) = x' *tq* x' = x - 1 *si* x ≥ 1
nul : *op* COMPTEUR	nul = x' *tq* x' = 0

On retrouve évidemment ici encore les problèmes de consistance du résultat. □

5.1.5.- s-introduction de sous sortes

Dans l'exemple 7 on exige qu'un compteur soit un entier positif ou nul à l'aide de la pré condition de "decr", ce qui est un moyen peu naturel. Il est donc utile de munir l'ensemble des S.I. d'une opération consistant à "restreindre" certaines sortes. En termes de modèle ceci va correspondre à définir des sous ensembles.

De manière intuitive, l'introduction d'une sous-sorte va consister à "restreindre" les symboles fonctionnels dont l'un au moins des constituants du profil est la sorte concernée.

Définition 27 :

Soit S un S.I. et φ une formule de $\mathcal{L}$ à une seule variable libre x de profil $\overline{S_i}$.

Le S.I. S' obtenu à partir de S en restreignant la sorte $\overline{S_i}$ à φ est défini par :

$$S' = \check{S}\ [\bigcup_{f \in \mathcal{Sym}(S)} \mathcal{R}(f)] \quad \text{où}$$

- $\overset{\vee}{S}$ est déduit de S en indiquant chaque symbole : chaque $f \in \mathcal{Sym}(S)$ est remplacé par f'

- $\mathcal{A}\mathcal{C}(f) = \bigwedge_{y_j \in PS_i} \varphi\,[y_j/x] \supset f\,x_1 \ldots x_p \equiv f' x_1 \ldots x_p$

avec $PS_i = \{x_j \ tq\ 1 \leq j \leq p$ et $profil(x_j) = \overline{S_i}\}$ U si

but(f) = $\overline{S_i}$ alors $\{f\,x_1 \ldots x_j\}$ sinon $\emptyset$ □

On commence donc par changer les différents symboles fonctionnels, ce qui permet ensuite de les redéfinir par $\mathcal{A}\mathcal{C}(f)$ en introduisant des préconditions

Notation

On note $S\,@\,(\overline{S_i}|\varphi)$ le S.I. S' ainsi obtenu.

Nous avons ainsi choisi de formaliser ce concept en "dupliquant" les symboles fonctionnels ce qui n'est pas gênant, si, en pratique, on ne s'autorise plus à considérer les anciens symboles. C'est en particulier le cas lorsque S est une sorte et que $\overline{S_i}$ est la sorte d'intérêt $\overline{S}$. On note alors $S|\varphi$ à la place de $S\,@\,(\overline{S}|\varphi)$. Il est donc naturel de considérer dans $S|\varphi$ les nouvelles opérations sur S, c'est à dire les opérations sur la restriction de la sorte d'intérêt par φ, plutôt que les anciennes. Les symboles privés (f'...) sont conservés car c'est sur eux que repose la spécification des "nouveaux" symboles. Mais les symboles privés ne sont plus utilisés dans les formules ou énoncés portant sur le S.I. restreint : il s'agit ici de symboles "cachés".

Exemple 8 :

Pour définir COMPTEUR on utilise le S.I. ENTIER|$(x \geqslant 0)$. Dans ce dernier on a en particulier l'opération + définie par :

$$x_1 \geqslant 0 \ \wedge \ x_2 \geqslant 0 \quad \supset \quad x_1 + x_2 \equiv x_1 +' x_2$$

C'est la notion classique de restriction d'une fonction. Et classiquement aussi on la note encore + sur ENTIER|$(x \geqslant 0)$ □

5.2.- s-description d'un système d'information

Les opérations de construction présentées au paragraphe précédent sont locales. Mais dans la construction d'une spécification d'un univers on est souvent conduit à préciser en même temps les sortes d'intérêt de la spécification et les nouvelles opérations. Dans ce but on introduit le concept de s-description qui peut être défini comme composition d'un certain nombre d'opérations de transformation de S.I..

5.2.1.- Syntaxe d'une s-description

En se reportant aux exemples des figures 1, 2 ou 3, une s-description d'un S.I. S se présente sous la forme de la figure 4 (nous introduirons seulement au paragraphe 6 les S.I. paramétrés).

lexique	profil	énoncé formel
S.I. S		
Description informelle des sortes d'intérêt		$D(\overline{S_1})$ ⋮ $D(\overline{S_k})$
des invariants	$s_1 : \overline{S_1}$ ⋮ $s_k : \overline{S_k}$	*invariant* n_1 ⋮ n_k
des opérations	P_1 ⋮ P'_n	*opérations* $s\text{-}A_1$ ⋮ $s\text{-}A_n$

Figure 4 : forme générale d'une s-description d'un S.I.

Commentaires :

a) $\overline{S_1}$, ..., $\overline{S_k}$ sont les sortes d'intérêt de S

b) $D(\overline{S_i})$ est la déclaration de la sorte $\overline{S_i}$. Ces déclarations reposent sur l'utilisation de "constructeurs de sortes". En fait ici sont utilisées conjointement :

. Des opérations de construction de S.I. comprenant celles du paragraphe précédent et celles du paragraphe 7.

. Des opérations permettant de passer d'un S.I. à l'une de ses sortes : "sélection d'une sorte" $D(\overline{S_i})$ est de l'une des formes suivantes :

. $\overline{S_i} = \overline{So_i}$ où $\overline{So_i}$ est une sorte d'un S.I. SUP_i

. $\overline{S_i}$ = SUP si SUP est uni-sorte, cette définition équivaut alors à la précédente avec $\overline{So_i} = \overline{SUP}$

Dans le cas fréquent où $k = 1$, $\overline{S_i}$ est remplacé dans les définitions précédentes par S .
Les différents S.I. SUP permettant de définir les sortes d'intérêt de S sont appelés *systèmes supports* de S .

c) La partie "*invariant*" est un ensemble de formules définissant des sous sortes des sortes déclarées au-dessus. Lorsque η_i = *vrai* on convient évidemment de ne pas le mentionner explicitement.

d) La partie "*opérations*" est un ensemble de s-descriptions des symboles fonctionnels nouveaux de S (cf § 5.1.1).

5.2.2.- Système d'information défini par une s-description.

Les invariants d'une s-description ayant pour rôle de restreindre les sortes utilisées dans les déclarations des sortes d'intérêt de S , ils correspondent donc aux s-introductions de sous-sortes. On définit S de la manière suivante :

pas 1 Pour tout i , $1 \leqslant i \leqslant k$, on pose

$$SUP'_i = SUP_i \ @ \ (\overline{So_i} \mid \eta_i) \qquad \text{(cf. § 5.1.5.)}$$

En particulier lorsque $k = 1$ et que SUP est uni-sorte :

$$SUP' = SUP \mid \eta$$

Intuitivement ce pas consiste à introduire les axiomes caractérisant les restrictions des opérations déterminées par les invariants.

Exemple 9 :

Reprenons le S.I. FILE défini sur la figure 1.
On a ici SUP = ENTIER* , donc à l'issue du pas 1 :

SUP' = ENTIER* $\quad lo \leqslant |f| \leqslant 100$

SUP' contient en particulier les axiomes (en posant $\eta = o \leqslant |f| \leqslant 100$

$o \leqslant |f| \leqslant 100 \ \wedge \ o \leqslant |f \sim x| \leqslant 100 \supset f \sim x \equiv f \sim' x$

et $o \leqslant |f| \leqslant 100 \ \wedge \ o \leqslant |\text{reste } f| \leqslant 100 \supset \text{reste } f \equiv \text{reste' } f$

soit $o \leqslant |f| < 100 \supset f \sim x \equiv f \sim' x$

$o < |f| < 100 \supset \text{reste } f \equiv \text{reste' } f$ □

pas 2

Soit $SP' = SUP'_1 \cup ... \cup SUP'_k$ réunion des transformés par le pas 1 des S.I. supports

On appelle SP le système d'information déduit de SP' en substituant à chaque $\overline{So_i}$ la sorte d'intérêt $\overline{S_i}$:

$$SP = Sub\ (SP', \overline{S_1}/\overline{So_1}, ..., \overline{S_k}/\overline{So_k})$$

Exemple 9 (suite) :

Il n'y a qu'un seul système support ENTIER*
Ainsi SP = Sub (SUP', $\overline{FILE}$/ENTIER*)

En particulier on a :

$profil_{SP}(\sim') = profil_{SP}(\sim) = \overline{FILE}, \overline{ENTIER} \rightarrow \overline{FILE}$

$profil_{SP}(reste') = profil_{SP}(reste) = \overline{FILE} \rightarrow \overline{FILE}$ □

pas 3

$S = s\text{-}\mathcal{C}xt\ (SP, \{P_1, s\text{-}A_1, \ldots, P_n, s\text{-}A_n\})$

c'est à dire (§ 5.1.2) que S est une s-extension de SP.

Intuitivement, on étend la réunion des systèmes supports, convenablement restreints pas les invariants, à l'aide des s-descriptions des nouvelles opérations.

Complétons cette définition en posant :

$\mathcal{S}ori(SP) = \{\overline{S_1}, \ldots, \overline{S_k}\}$

ce qui permet, avec la définition 24, d'obtenir

$\mathcal{S}ori(S) = \{\overline{S_1}, \ldots, \overline{S_k}\}$

Exemple 9 (suite et fin) :

L'extension $FILE = s\text{-}\mathcal{C}xt\ (SP, \{P_1, s\text{-}A_1, \ldots, P_4, s\text{-}A_4\})$

est obtenue en réunissant SP et les S.I. utilisés dans les s-descriptions des nouvelles opérations : ENTIER et BOOL. Finalement FILE est défini ainsi :

i) $\mathcal{S}ori(FILE) = \overline{FILE}$ (sorte d'intérêt)

$\mathcal{S}ore(FILE) = \overline{ENTIER}, \overline{BOOL}$ (sortes externes)

- $\mathcal{S}ym(FILE) = \mathcal{S}ym(ENTIER) \cup \mathcal{S}ym(BOOL) \cup \mathcal{S}ym(SUP') \cup \{ajtf, retf, têtef, videf\}$

- profil est inchangée sur ENTIER, BOOL. Il résulte du pas 2 pour les symboles des systèmes supports

- $\mathcal{A}xp(FILE) = \mathcal{A}xp(ENTIER) \cup \mathcal{A}xp(BOOL) \cup \mathcal{A}xp(SUP') \cup A$

avec A : $ajt\ f\ x \equiv f' \supset f' \equiv f \sim x$

$retf\ f \equiv f' \supset f' \equiv restef$

$\neg vide(f) \wedge têtef\ f \equiv x \supset x \equiv premf$

$vide\ f \equiv b \supset b \equiv (|f| \equiv 0)$

Alors en utilisant les axiomes résultants du pas 1 on obtient les théorèmes :

$0 \leqslant |f| < 100 \wedge ajtf\ f\ x \equiv f' \supset f' \equiv f \sim' x$

$0 < |f| \leqslant 100 \wedge retf\ f \equiv f' \supset f' \equiv reste'\ f$

$0 < |f| \leqslant 100 \wedge têtef\ f \equiv x \supset x \equiv prem'f$

où $\sim'$, reste', prem' sont les anciens symboles $\sim$, reste, prem correctement axiomatisés.

Remarque :

Comme nous l'avons déjà noté au paragraphe 3.3.2 , une s-description d'un S.I. est en quelque sorte "trop complète" puisqu'elle nécessite de décrire chaque nouvelle opération à l'aide des opérations des S.I. externes. Ceci peut limiter de façon abusive les possibilités de résolution et en particulier de représentation. Les descriptions algébriques de S.I. introduites au chapitre 2.2 sont un moyen de n'exprimer *que les propriétés d'un univers* utiles pour la résolution du problème considéré.

5.3.- Systèmes d'information primitifs.

La mise en oeuvre d'opérations de construction de S.I. à partir de S.I. plus élémentaires nécessite de disposer de S.I. de bases "canoniques et prédéfinis".

Dans la suite, nous supposons disposer des S.I. suivants :

a) ENTIER
b) REEL
c) BOOL

chacun d'eux étant muni des opérations habituelles (lois de composition interne, opérateurs de comparaison, etc...) dont nous ne précisons pas l'axiomatisation.

d) CHAINE que l'on peut définir comme le S.I. des suites de caractères (§ 7.2.2). Nous le considérons plutôt comme un S.I. primitif avec pour opérations la concaténation, la sélection d'une sous-chaîne, la longueur etc...

e) ENTIERC qui est l'extension de ENTIER par deux symboles o-aires $+\infty$ et $-\infty$. On ajoute alors différents axiomes permettant de "prolonger" les opérations sur ENTIER :

$$x + (+\infty) = +\infty$$
$$x + (-\infty) = -\infty$$
$$\forall x \in \text{ENTIER}, \quad -\infty < x < +\infty$$

etc...

Ce S.I. permet en particulier de tenir compte des suites infinies

f) - $\{a_1, \ldots, a_p\}$ S.I. "ensemble fini" formé des symboles o-aires $a_1, \ldots, a_p$ et muni des opérations ensemblistes habituelles.

- $\emptyset$ S.I. "ensemble vide"

6.- SYSTEMES D'INFORMATION PARAMETRES

6.1.- Utilité de la paramétrisation.

Le point de vue incrémental de la construction des S.I., décrit au paragraphe 5, permet une conception progressive descendante et modulaire mais peu adaptée aux modifications des problèmes. Ainsi dans la spécification du problème de la simulation, la description du S.I. FILE doit être changée si par exemple on change la longueur maximum d'une file ou encore si on désire faire diverses statistiques sur les personnes qui sont en attente et donc stocker dans la file d'autres objets que de simples entiers.

De manière complémentaire à l'approche incrémentale, il est intéressant d'avoir un point de vue *générique* [BER, 75] des S.I., c'est à dire de pouvoir définir des S.I. paramétrés :

- Il est intéressant, évidemment, de pouvoir paramétrer un S.I. par des "objets" : symboles fonctionnels q-aires ($q \geq 0$) . On retrouve ainsi le concept de paramètre formel de procédure développé dans les langages de programmation. Par exemple on peut paramétrer FILE par sa longueur maximum n .

- Mais il est encore plus intéressant d'introduire des S.I. "formels" comme paramètres. C'est par exemple le cas de VAL dans MULTENS1 (figure 2). On augmente ainsi largement l'utilité méthodologique des S.I. puisqu'un système d'information paramétré par d'autres systèmes d'information apparaît comme une *opération de construction* de S.I..

6.1.1.- Implications méthodologiques de la paramétrisation.

L'introduction de paramètres dans la description d'un S.I. induit une économie de pensée dans la mesure où elle permet l'utilisation d'une même description dans différents contextes. Ceci correspond à un effort d'abstraction lors de la spécification. On peut ainsi :

- Utiliser des systèmes d'information prédéfinis (PILE, FILE, etc...) lors de la spécification d'un problème. La rencontre des fonctions de bases de S.I. prédéfinis permet d'arrêter le processus descendant de la spécification.

- Rendre modulaire la spécification d'un problème en permettant à des personnes différentes de spécifier des parties différentes de l'univers. Mais savoir ce qui peut être paramètre d'un S.I. et ce qui doit être complètement défini est un cas particulier de la question du juste niveau de spécification d'un problème : il ne doit être ni sur-spécifié, ni sous-spécifié. Nous avons déjà noté qu'il s'agissait d'un des aspects les plus délicats du problème de la spécification. Le guide méthodologique qu'il semble raisonnable de choisir est de "paramétrer au maximum". Ainsi dans FILE, il est raisonnable de paramétrer la longueur et les "objets stockés" et on ne peut rien paramétrer d'autre puisque les autres S.I. externes sont des intermédiaires techniques (suites, opérations sur les suites, booléens.).

Souvent heureusement, les S.I. paramétrés apparaissent naturellement comme des constructeurs de S.I..

Exemple 9 .

Spécification d'un S.I. CUMUL permettant de définir des bilans.

Il est fréquent que l'on ait à appliquer répétitivement une loi de composition interne sur un ensemble d'objets (par exemple somme, produit, minimum, maximum.... d'un ensemble de nombres). Corrélativement on veut déterminer le cardinal de cet ensemble. On peut introduire un système d'information CUMUL facilitant ces opérations (figure 5).

Dans une telle définition on est conduit à préciser, comme dans MULTENS, le S.I. paramètre en donnant des propriétés de certaines de ses opérations.

Le S.I. COMPTEUR est décrit dans l'exemple 7 (§ 5.1.4).

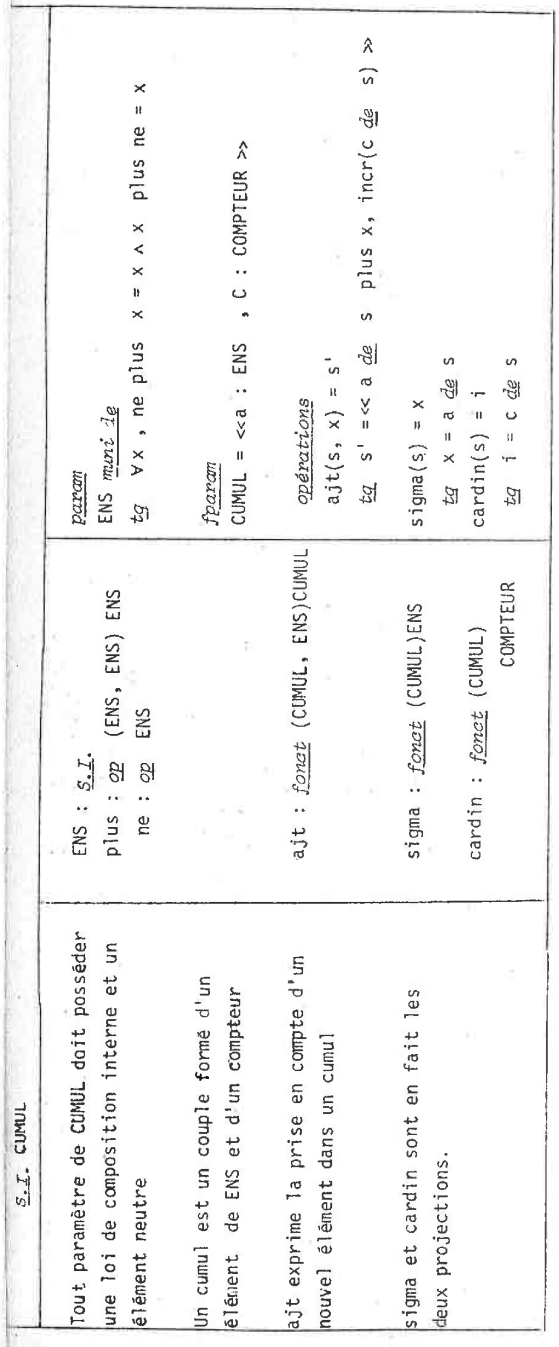


Figure 5 : le S.I. paramétré CUMUL.

6.1.2.- Problèmes posés par la paramétrisation.

L'analogie entre les S.I. paramétrés et les procédures des langages de programmation s'arrête bien vite :

- D'une part dans les langages de programmation classiques le type de paramètre ne peut pas être lui-même un paramètre (l'union de modes en ALGOL 68 permet, en partie, de dépasser cette contrainte).

- D'autre part, il est nécessaire, si on veut que le concept de paramétrisation soit suffisamment puissant, de pouvoir exprimer des propriétés que doivent vérifier les paramètres (comme dans MULTENS ou CUMUL). Ces propriétés concernent certaines opérations des S.I. paramètres. C'est en particulier le cas si, comme nous faisons au chapitre 1.3, la paramétrisation permet de rendre compte des *données* d'un problème. Il est alors nécessaire d'exprimer les conditions que doivent vérifier ces données (ce qui, en termes de preuves de programmes s'appelle parfois les pré-conditions du problème [LIV, 78]).

Dans la suite nous définissons et étudions le concept de S.I. paramétré ainsi que celui de p-réalisation d'un tel système (qui intuitivement consiste à rendre "effectifs" les S.I. paramètres) ; puis nous proposons une syntaxe permettant de décrire ces concepts (§ 6.2).

Notons dès maintenant que, si les notions relatives aux systèmes paramétrés sont assez simples à définir formellement et à mettre en oeuvre au niveau d'un langage de spécification de problèmes, elles posent des difficultés non négligeables si on veut les introduire dans un langage de programmation. En effet, il faut alors introduire des mécanismes non classiques de définition de procédures admettant des types en paramètres : la vérification des propriétés des paramètres, lors de l'identification entre paramètres effectifs et paramètres formels, en rendent l'implantation délicate. Ces problèmes sont étudiés dans [JAC, 78] , [BER, 79] , [MIL, 78] .

6.2.- Définition et description d'un système d'information paramétré

6.2.1.- Définitions.

Définition 28 :

Un système d'information paramétré est un triplet $(S, \mathcal{S}op, \mathcal{S}yp)$ où
- S est un S.I.
- $\mathcal{S}op$ est un sous ensemble $\overline{T}_1, \ldots, \overline{T}_n$ des sortes de S : sortes paramètres
- $\mathcal{S}yp$ est un sous ensemble $g_1, \ldots, g_p$ des symboles de S : symboles paramètres.

En particulier $\mathcal{S}yp$ contient tous les symboles de S dont le profil contient l'un des $\overline{T}_i$ □

Par exemple, le S.I. paramètre CUMUL a pour sorte paramètre $\overline{ENS}$ et pour symboles paramètres "plus" et "ne".

Ainsi, avec cette définition un S.I. paramétré apparaît comme un S.I. dans lequel on distingue certaines sortes et opérations. On ne fait pas, de cette manière, de différence entre les "paramètres formels" et les autres éléments du S.I.. Ce point de vue repose pleinement sur le fait que les éléments d'un S.I. sont précisément des éléments formels. Remplacer les "paramètres formels" par des "paramètres effectifs" , on dit p-réaliser le système paramétré, revient ici à "substituer" certaines sortes et opérations à d'autres. La notion de réalisation (§ 4.1) d'un système d'information permet de décrire précisément cette transformation.

Définition 29 :

Une *p-réalisation* d'un S.I. paramétré $(S, \mathcal{S}op, \mathcal{S}yp)$ est une réalisation (S', p) de S qui est l'identité sur $\mathcal{S}or(S) \setminus \mathcal{S}op$ et $\mathcal{S}ym(S) \setminus \mathcal{S}yp$, et qui associe à chaque $\overline{T}_i$ une sorte $\overline{E}_i$ et à chaque g_i un symbole f_i de même arité. $\overline{E}_i$ (resp. f_i) est le *paramètre effectif* associé au *paramètre formel* $\overline{T}_i$ (resp. g_i) . □

Ainsi dans une p-réalisation d'un système paramétré, les images des axiomes caractérisant les paramètres formels sont des théorèmes.

Mais il faut que l'on puisse préciser les paramètres effectifs, c'est ce dont rend compte la définition suivante.

Définition 30 :

Soit E un S.I., $(S, \mathcal{F}op, \mathcal{F}yp)$ un S.I. paramétré et ρ une application de S dans $S \cup E$ telle que l'image de chaque $\overline{T_i}$ soit un $\overline{E_i}$ de E, celle de chaque g_i soit un f_i de E. Avec ces notations, on appelle *p-réalisation de* $(S, \mathfrak{D}op, \mathfrak{D}yp)$ *par* (E, ρ) la p-réalisation $(S \cup E, \rho)$

Notation :

On note $S\{E, \rho\}$

ou plus explicitement $S\{E, \overline{E_1}, \ldots, \overline{E_n}, f_1, \ldots, f_p\}$ la p-réalisation de E par ρ.

Par exemple CUMUL {ENTIER, $\overline{\text{ENTIER}}$, *, 1} est une p-réalisation valide de CUMUL (figure 5) puisque * et 1 vérifient dans ENTIER :

$$\forall x \quad 1 * x \equiv x \wedge x * 1 \equiv x$$

Lorsque, comme dans l'exemple précédent, les sortes effectives sont les sortes d'intérêt de S.I. uni-sortes, et lorsque E est la réunion de ces différents S.I. on convient de remplacer chaque $\overline{E_i}$ par E_i et de négliger E. Ce qui donne pour l'exemple ci-dessus :

CUMUL {ENTIER, *, 1}

Remarque :

En général une p-réalisation consistante n'est pas une réalisation conservative, ainsi dans l'exemple ci-dessus on a :

ρ (x plus y ≡ y plus x) qui est un théorème de S' sans que son antécédent en soit un de S.

6.2.2.- s-description d'un système d'information paramétré.

a) Une s-description d'un S.I. paramétré se présente sous la forme de la figure 6.

S.I. S		
Description informelle des S. I. paramètres et de leurs opérations	ST_1 : *S.I.* P^1_1 ⋮ $P^1_{k_1}$	*param* ST_1 *muni de* *tg* A^1_1 ⋮ *tg* $A^1_{k_1}$
	ST_2 : *S.I.* ⋮ ST_n : *S.I.*	ST_2 *muni de* ⋮ ST_n *muni de*
descrition informelle de symboles paramètres dont les sortes du profil ne sont pas des sortes paramètres.	Q_1 ⋮ Q_ℓ	s-B_1 ⋮ s-B_ℓ
		fparam
Description informelle des sortes d'intérêt		$D(\overline{S_1})$ ⋮ $D(\overline{S_k})$
Des invariants		*invariant*
	$s_1 : \overline{S_1}$ ⋮ $s_k : \overline{S_k}$	η_1 ⋮ η_k
Des opérations		*opérations*
	P_1 ⋮ P_n	s-A_1 ⋮ s-A_n

Figure 6 : forme générale d'une s-description paramétrée

Commentaires :

- $ST_1, \ldots, ST_2$ sont des noms de S.I. paramètres. Pour simplifier les notations nous supposerons qu'ils sont uni-sortes. Si on admet des paramètres multi-sortes il faut en préciser les différentes sortes d'intérêt.
- P_1^1 ... sont des descriptions de symboles fonctionnels
- A_1^1 .... sont des formules portant sur les symboles fonctionnels décrits par les P_1^1 ... . Ces formules *ne contiennent pas d'autres symboles fonctionnels.*
- Q_i, $s\text{-}B_i$ sont des s- descriptions de symboles fonctionnels s_i qui ne sont pas liés à un S.I. paramètre.
- φ_2 ... sont des formules caractérisant les s_1 .... Ce sont des formules respectivement des S.I. S_1 ...

b) Le S.I. paramétré défini par une telle s-description est formé

i) Par le système S défini de la manière suivante :

pas 1 Soit $\tilde{S}$ le S.I. défini par la s-description obtenue à partir de la s-description suivante en supprimant la partie *param ... fparam*

pas 2 S est l'extension de $\tilde{S} \cup S_1 \cup \ldots \cup S_k$ obtenue par adjonction

- Des sortes $\overline{ST_1}$...
- Des symboles g_1^1.... décrits par les P_1^1 ...
- Des symboles s_1 ... décrits par les Q_1...
- Des profils de ces différents symboles
- Des axiomes $A_1^1, \ldots, s\text{-}B_1 \ldots, s\text{-}B_\ell$

ii) par les sortes $\overline{ST_1} \ldots\ldots \overline{ST_n}$

iii) par les symboles $g_1^1, \ldots, s_1, \ldots, s_k$

6.2.3.- s-description d'une p-réalisation

a) Une sp-réalisation se présente, avec les notations du paragraphe précédent, sous la forme :

$$S \ (SE_1, f_1^1, \ldots, f_n^1 \ , \ SE_2, f_1^2 \ldots, SE_n, \ldots, e_1, \ldots, e_k)$$

où $SE_1, \ldots$ sont des S.I. dont les $f_1^1 \ldots$ sont des opérations et $e_1, \ldots, e_k$ sont des symboles o-aires de $S_1, \ldots, S_k$. L'ordre des paramètres effectifs dans une telle s-description correspond à celui des paramètres formels dans la partie *param ... fparam*

Par exemple MULTENS1 { ENTIER, $\leqslant$ }, MULTENS2 { $\mathcal{F}$ (ENTIER), $\in$ } ,

CUMUL { REEL, +, 0} sont des sp-réalisations respectivement de MULTENS1, MULTENS2 et CUMUL.

b) Le couple (S', ρ) défini par une sp-réalisation est formée

- du S.I. $S' = S \cup SE_1 \cup \ldots \cup SE_n$,
- de l'application ρ vérifiant

$$\forall 1 \leqslant i \leqslant n \quad \rho(\overline{ST_i}) = \overline{SE_i}$$

$$\forall 1 \leqslant j \leqslant n_i \ , \ \rho(g_j^i) = f_j^i$$

$$\forall 1 \leqslant i \leqslant k \quad \rho(s_i) = e_i$$

et qui est l'identité sur les autres sortes et symboles de S .

Théorème 15 : Avec les notations précédentes

(S', ρ) est une p-réalisation de (S, $\mathcal{S}$op, $\mathcal{S}$yp) si et seulement si l'image par ρ de toute formule φ de la partie *param*...*fparam* est un théorème de S' .

□

Ce théorème résulte immédiatement du fait que les seuls axiomes de S qui ne sont pas des axiomes de S' sont ces formules φ .

6.3.- Comparaison avec d'autres approches.

Nous ne développerons pas ici les problèmes liés à la paramétrisation d'un type lorsqu'on se place dans un langage de programmation (cf [BER, 79] par exemple).

Plutôt que d'admettre n'importe quelle propriété caractérisant les types paramètres, certains auteurs ont cherché à en restreindre la forme. C'est ainsi qu'en ALPHARD [WUL, 75] , les propriétés d'un type paramètre s'expriment dans les cas simples par la liste des opérations qui doivent appartenir à ce type.

Par exemple *form* VECT (T : *form* < +, 0 >) précise que VECT est un type admettant pour paramètre un type T quelconque muni des opérations + et 0

Mais en général on souhaite préciser certaines propriétés des opérations des types paramétres. Dans ce but les noms d'opérations des paramètres sont suivis de pré et post conditions analogues à celles utilisées en ALPHARD pour spécifier les opérations d'un type

Par exemple le type CUMUL pourrait être déclaré de la façon suivante :

form CUMUL(ENS : *form* < ne : ENS,
≤ : (x : ENS, y : ENS) *return* (z : ENS)
post y = ne ⊃ x plus y = x ∧ y plus x = x >)

De manière analogue, dans le cadre algébrique que nous développons au chapitre 2.2, il est naturel de préciser les propriétés sous forme d'équations.

On peut aussi introduire un certain nombre de primitives de haut niveau telles que : le type paramètre T est un groupe, est un corps, admet une opération commutative et associative etc... . De telles primitives facilitent le travail du spécificateur ainsi que la vérification de la validité du passage formel → effectif (que nous avons appelé p-réalisation) [JAC,78], [BER,79].

La recherche de telles primitives doit être essentiellement empirique.

Exemple 10 :

Une légère extension de la syntaxe proposée précédemment permet de définir de manière assez élégante le S. I. VECTEUR muni d'une opération d'addition, d'un prédicat "trié" et d'une opération interclassement (figure 7). Dans cette description BIJ (A, B) est la sorte des bijections de A sur B. Elle peut être définie précisément à l'aide des opérations introduites au paragraphe 7. □

7.- QUELQUES AUTRES OPERATIONS DE CONSTRUCTION DE SYSTEMES D'INFORMATION

Nous avons introduit au paragraphe 5 des outils de construction de S.I. à partir de S.I. plus simples qui reposent essentiellement sur la notion d'extension.

Chacun de ces outils (s-extension, s-restriction etc...) peut être considéré comme une *opération primitive mise à la disposition du spécifieur*. A l'opposé la paramétrisation permet au *spécificur de définir de nouvelles opérations* de construction de S.I.. Mais certaines de ces opérations sont d'un usage si général qu'il est pertinent d'en donner une définition standard permettant de les considérer comme d'autres "outils de base" et évitant ainsi à chaque spécifieur de les redéfinir. Nous nous attachons, dans ce paragraphe, à définir quelques unes de ces opérations.

Le S.I. paramètre est un monoïde pour les opérations plus et nul. Il est muni d'un préordre total ≤ .	ELEM : *S.I.* plus : *gp* (ELEM, ELEM) ELEM nul : ELEM ≤ : *gp* (ELEM, ELEM) BOOL n : ENTIER	*param* ELEM *muni de* (plus, nul) *monoïde* ≤ *préordre total* n *tq* n ⩾ 1 *fparam* VECTEUR = ELEM* *invariant* bi(v) = 1 ∧ bs(v) = n *opérations*
vnul est le vecteur nul vplus est "l'addition" de deux vecteurs	vnul : VECTEUR vplus : *gp* (VECTEUR,VECTEUR) VECTEUR	vnul *tq* ∀i ∈ [1..n] $vnul_i = 0$ vplus(v1, v2) = v3 *tq* ∀i ∈ [1..n] $v3_i = v2_i$ plus $v1_i$
trié indique si un vecteur est une suite croissante	trié : *fonct*(VECTEUR)BOOL	trié (v) = b *tq* b = (∀i ∈ [1..n-1], $v_i \leqslant v_{i+1}$
inter définit l'interclassement de deux vecteurs triés	inter : *fonct* (VECTEUR, VECTEUR)VECTEUR(ELEM,2*n)	inter (v1, v2) = v3 *tq* trié (v3) ∧ [∃J ⊂ [1..2*n] *tq* |J| = n ∧ ∃f ∈ BIJ(J, [1..n]) ∧ ∃g ∈ BIJ([1..2*n]∼ J, [1..n]) *tq* (∀i ∈ J, $v3_i = v1_{f(i)}$) ∧ (∀j ∈ [1..2n] ∼ J, $v3_j$ = v2 g(j))]
val associe à un vecteur et un indice la valeur de l'élément correspondant.	val : *fonct*(VECTEUR,ENTIER) ELEM	*et* trié(v1) ∧ trié(v2) val(v, i) = x *tq* x = v_i *et* 1 ≤ i ⩽ n

Figure 7.1. le S.I. VECTEUR

7.1.- Constructions ensemblistes.

On nomme ainsi deux opérations de construction de S.I. issues directement de la théorie des ensembles : le produit cartésien et la définition de l'ensemble des parties d'un ensemble. Notons que l'on dispose déjà de l'opération primitive qu'est la réunion (§ 4.2.1). On pourrait définir d'autres opérations comme l'intersection, la différence ensembliste etc...

7.1.1.- Produit cartésien

Soient S_1 et S_2 deux S.I. uni-sorte et s_1 , s_2 deux symboles o-aires.

Le *produit cartésien* de S_1 et S_2 relativement aux *sélecteurs* (ou *projections*) s_1 et s_2 est le S.I. $\mathcal{C}xt(S_1 \cup S_2, \Sigma, A)$ où

i) $\Sigma = (\{\overline{S_1} \times \overline{S_2}\}, \{<< >>, s_1, s_2\}, \text{profil})$

. $\overline{S_1} \times \overline{S_2}$ est une nouvelle sorte : la sorte produit de $\overline{S_1}$ et $\overline{S_2}$

. $profil(s_1) = \overline{S_1} \times \overline{S_2} \rightarrow \overline{S_1}$ première projection

$profil(s_2) = \overline{S_1} \times \overline{S_2} \rightarrow \overline{S_2}$ deuxième projection

$profil(<<\gg) = \overline{S_1}, \overline{S_2} \rightarrow \overline{S_1} \times \overline{S_2}$ formation de couples, on note $<< x_1, x_2 >>$ à la place de $<< >>{}^{x_1}{}_{x_2}$

ii) $A = \{ s_1 << x_1, x_2 >> \equiv x_1 \quad ,$

$s_2 << x_1, x_2 >> \equiv x_2 \quad ,$

$<< x_1, x_2 >> \equiv << y_1, y_2 >> \leftrightarrow x_1 \equiv x_2 \wedge y_1 \equiv y_2 \ \}$

Notations :

- Le produit cartésien de S_1 et S_2 relativement à s_1 et s_2 est noté

$<< s_1 : S_1 , s_2 : S_2 >>$

- s_i *de* x peut être utilisé à la place de $s_i(x)$ si x est de profil $\overline{S_1} \times \overline{S_2}$

Par définition nous posons $\mathcal{P}ori(<< s_1 : S_1, s_2 : S_2 >>) = \overline{S_1} \times \overline{S_2}$

A cette définition directe de l'opération produit cartésien on peut en substituer d'autres utilisant la paramétrisation :

- Soit en utilisant pour S.I. support l'ensemble des "paires ordonnées" $\{ x, \{ x, y\}\}$ comme en théorie axiomatique des ensembles [KRI, 72] , [SHO, 67]

- Soit en ne précisant pas de S.I. support ce qui conduit à une description algébrique donnée au paragraphe 4.1 du chapitre 2.2 .

Théorème 16 :

$<< s_2 : S_1, s_2 : S_2 >>$ est une extension conservative de $S_1 \cup S_2$

Résulte immédiatement du fait que tout produit cartésien $U_1 \times U_2$ de modèles de S_1 et S_2 est un modèle de $<< s_1 : S_1, s_2 : S_2 >>$

L'opération produit cartésien se généralise immédiatement par récurrence au produit de n S.I. .

Remarque :

Il est possible, à l'aide des diverses opérations de construction de S.I. de définir des équations sur les S.I.. Par exemple on peut poser :

SUITE = << début : SUITE, der : VAL >> $\cup$ {< >} .

L'étude de telles équations repose sur la théorie du point fixe de SCOTT : on caractérise ainsi ce que l'on peut appeler des types récursifs et les spécifications algébriques de S.I. (ch. 2.2) en sont un cas particulier [LEH, 77].

7.1.2.- Ensemble des parties

Soit S un S.I. uni-sorte. On appelle système d'information *ensemble des parties* de S l'extension $\mathcal{E}xt(S \cup BOOL, \Sigma , A)$ de S définie de la façon suivante :

i) $\Sigma = (\{ \mathcal{P}(S)\} , \mathcal{S}ym, profil)$ où $\mathcal{S}ym$ contient les symboles suivants :

$\emptyset$	avec profil $(\emptyset) = \rightarrow \mathcal{P}(S)$	(vide)
$\in$	avec profil $(\in) = \overline{S}, \mathcal{P}(\overline{S}) \rightarrow \overline{BOOL}$	(appartenance)
$\subset$	avec profil $(\subset) = \mathcal{P}(\overline{S}), \mathcal{P}(\overline{S}) \rightarrow \overline{BOOL}$	(inclusion)
$\cap$	avec profil $(\cap) = \mathcal{P}(\overline{S}), \mathcal{P}(\overline{S}) \rightarrow \mathcal{P}(\overline{S})$	(intersection)
$\cup$	avec profil $(\cup) = \mathcal{P}(\overline{S}), \mathcal{P}(\overline{S}) \rightarrow \mathcal{P}(\overline{S})$	(réunion)
$\setminus$	avec profil $(\setminus) = \mathcal{P}(\overline{S}), \mathcal{P}(\overline{S}) \rightarrow \mathcal{P}(\overline{S})$	(différence)
$\{ \}$	avec profil $(\{\}) = [\overline{S} \rightarrow \overline{BOOL}]^{(+)} \rightarrow \mathcal{P}(\overline{S})$	(définition "en compréhension" d'un sous ensemble)
$\{ \}^n$	avec profil $(\{ \}^n) = \underbrace{\overline{S}, \ldots, \overline{S}}_{\text{n fois}} \rightarrow \mathcal{P}(\overline{S})$	(définition "en extension" d'un sous ensemble)
card	avec profil $(card) = \mathcal{P}(\overline{S}) \rightarrow$ ENTIER	

(On peut définir d'autres symboles : différence symétrique , complémentaire etc... nous nous limitons ici aux opérations précédentes).

ii) L'ensemble A des axiomes caractérisant ces opérations peut se déduire, par exemple, des axiomes de ZERMELO-FRAENKEL [KRI, 72] , [SHO, 67] définissant axiomatiquement la théorie des ensembles. Donnons en quelques exemples :

(+) $[\overline{A} \rightarrow \overline{B}]$ est la sorte "fonctions de A dans B" définie au paragraphe 7.2.1.

. $x \in \emptyset \equiv$ *faux*

. $a \subset b \equiv \forall x\ x \in a \supset x \in b$

. $(\forall x\ x \in a \leftrightarrow x \in b) \leftrightarrow a \equiv b$

. $x \in \{\}\varphi \equiv$ appl(φ, x) ici on précise que le sous-ensemble caractérisé par le prédicat φ est formé des x pour lesquels $\varphi(x)$, plus précisément appl(φ, x) (cf. § 7.2.1) est *vrai*

. $x \in \{\}^n\ x_1 \ldots x_p \equiv (x \equiv x_1 \vee x \equiv x_2 \vee \ldots \vee x \equiv x_p)$

. $x \in a \cap b \equiv x \in a \wedge x \in b$

Notations :

- L'ensemble des parties du S.I. S est noté $\mathcal{P}(S)$
- $\{\}\varphi$ est noté habituellement $\{ x$ *tq* appl(φ , x) $\equiv$ *vrai* $\}$
  ou plus simplement $\{ x$ *tq* $\varphi(x)\ \}$
- $\{\}\ x_1 \ldots x_n$ est noté habituellement $\{ x_1, x_2, \ldots, x_n \}$

Par définition, $\mathcal{S}ori(\mathcal{P}(S)) = \mathcal{P}(S)$

En complétant de façon classique les axiomes précédents, on pourrait montrer que l'ensemble des parties d'un modèle d'un S.I. S est un modèle de $\mathcal{P}(S)$.

Théorème 17 :

$\mathcal{P}(S)$ est une extension conservative de S □

7.1.3.- Intervalles d'entiers.

Le S.I. INTENT des intervalles d'entiers peut être défini à partir de $\mathcal{P}$(ENTIER) en précisant que dans un intervalle les éléments compris entre deux éléments quelconques appartiennent encore à l'intervalle. On introduit de plus un certain nombre d'opérations spécifiques (figure 8).

Notations :

On préfère la notation [a...b] à [ ](a, b) pour l'opération de construction d'un intervalle

On définit de manière analogue INTENTC le S.I. des intervalles sur ENTIERC. On peut avoir dans ce cas bs(it) = $+\infty$ ou bi(it) = $-\infty$.

S.I. INTENT		
	it : INTENT	INTENT = $\mathcal{P}$ (ENTIER) *invariant* $\forall i, j, k \in$ ENTIER $i \leqslant k \leqslant j \wedge i \in it \wedge j \in it \supset k \in it$ *opérations*
bi est la borne inférieure de l'intervalle	bi : *fonct* (INTENT) ENTIER	bi(it) = b *tq* $(\forall x\ x \in it \supset x \geqslant b)$
bs est la borne supérieure	bs : *fonct* (INTENT) ENTIER	bs(it) = b *tq* $(\forall x\ x \in it \supset x \leqslant b)$
[ ] permet de construire un intervalle	[ ] : *fonct* (ENTIER, ENTIER) INTENT	[ ] (b, s) = it *tq* $x \in it \leftrightarrow b \leqslant x \leqslant s$

Figure 8 : les intervalles d'entiers

7.2.- Construction d'ensembles de fonctions

On peut définir le S.I. "fonctions de S_1 dans S_2" à partir du produit cartésien. Nous préférons ici en donner une caractérisation directe.

7.2.1.- Systèmes d'information "ensemble des fonctions"

S_1 et S_2 étant deux S.I. uni-sorte, le S.I. ensemble des fonctions de S_1 dans S_2 est l'extension $\mathcal{E}xt\ (\mathcal{P}(S_1 \cup S_2), \Sigma, A)$ avec :

i) $\Sigma = (\{[\overline{S_1} \to \overline{S_2}]\}, \mathcal{S}ym, profil)$ où $\mathcal{S}ym$ contient les opérations suivantes :

appl avec profil (appl) = $[\overline{S_1} \to \overline{S_2}, \overline{S_1}] \to \overline{S_2}$

dom avec profil (dom) = $[\overline{S_1} \to \overline{S_2}] \to \mathcal{P}(\overline{S_1})$

imag avec profil (imag) = $[\overline{S_1} \to \overline{S_2}] \to \mathcal{P}(\overline{S_2})$

ajout avec profil (ajout) = $[\overline{S_1} \to \overline{S_2}], \overline{S_1}, \overline{S_2} \to [\overline{S_1} \to \overline{S_2}]$

supr avec profil (supr) = $[\overline{S_1} \to \overline{S_2}], \overline{S_1} \to [\overline{S_1} \to \overline{S_2}]$

chval avec profil (chval) = $[\overline{S_1} \to \overline{S_2}], \overline{S_1}, \overline{S_2} \to [\overline{S_1} \to \overline{S_2}]$

fvide avec profil (fvide) = $\to [\overline{S_1} \longrightarrow \overline{S_2}]$

ii) A est formé des axiomes :

- $\exists y$ *tq* appl(f, x) $\equiv y \longleftrightarrow x \in dom(f)$
- $x \in dom(f) \longleftrightarrow appl(f, x) \in imag(f)$
- $dom(fvide) \equiv \emptyset$, $imag(fvide) = \emptyset$
- $\neg x \in dom(f) \supset dom(ajout(f, x, y)) \equiv dom(f) \cup \{x\}$
- $\forall x' \in dom(appl(f, x'))$ $appl(f, x') \equiv$ *si* $x \equiv x'$ *alors* y *sinon* appl(f, x))
- $x \in dom(f) \supset dom(supr(f, x)) \equiv dom(f) \setminus \{x\}$
- $\forall x' \in dom(supr(f, x))$ $appl(f, x') \equiv appl(f, x)$
- $x \in dom(f) \supset dom(chval(f, x, y)) \equiv dom(f)$
- $\forall x' \in dom(chval(f, x, y))$ $appl(chval(f, x, y), x') \equiv$ *si* $x \equiv x'$ *alors* y *sinon* appl(f, x')
- $f \equiv g \longleftrightarrow dom(f) \equiv dom(g) \wedge (\forall x \in dom(f)\ appl(f, x) \equiv appl(g, x))$

Notations :

. On note $[S_1 \to S_2]$ le S.I. des fonctions de S_1 dans S_2

. On convient de noter parfois $f(x)$ à la place de appl(f, x)

. On note $[S_1 \times S_2 \times \ldots \times S_q \to S_{q+1}]$ à la place de $[S_1 \to [S_2 \to [\ldots [S_q \to S_{q+1}]]..]$.

. et on préfère $appl(f, x_1 \ldots x_q)$ à $appl(appl\ldots(appl(f, x_1), x_2)\ldots$

Par définition, $\mathcal{S}ori([S_1 \to S_2]) = [\overline{S_1} \to \overline{S_2}]$

Comme pour le produit cartésien ou l'ensemble des parties, on obtient un modèle de $[S_1 \to S_2]$ en considérant l'ensemble des fonctions d'un modèle de S_1 dans un modèle de S_2 .

Théorème 18 :

$[S_1 \to S_2]$ est une extension conservative de $S_1 \cup S_2$ □

7.2.2.- Suites sur un système d'information.

Le S.I. des suites sur un S.I. V est une extension de [ENTIERC → V] . La figure 9 en donne une s-description.

Notons que nous considérons aussi bien les suites finies que les suites infinies.

Dans cette description, pour alléger les notations, bi et bs désignent à la fois les bornes inférieures et supérieures d'intervalles entier et de suite.

Notations :

- La notation V* est préférée à SUITE(V) , $\overline{V*}$ en est la sorte d'intérêt
- $< x >$ représente la suite vérifiant $bi(< x >) = 1$, $bs(< x >) = 1$, $appl(< x >, 1) = x$

	V : *S.I.*	*param* V *fparam* SUITE = [ENTIERC → V] *invariant* dom(u) ∈ INTENTC *opérations*
	u : SUITE	
- la suite vide < > a pour domaine ∅ - l'adjonction en queue augmente de 1 la borne supérieure ; elle n'est définie que sur les suites finies - début diminue de 1 la borne supérieure, si elle n'est pas + ∞ .	< > : *op* SUITE ~ : *op* (SUITE, V)SUITE	< > *tq* dom < > = ∅ ~ *tq* u ~ x = ajout (u, bs(u) +1, x) *si* bs(u) < + ∞
	début : *fonct* (SUITE)SUITE	début(u) = v *tq* bi(v) = bi(u) ∧ bs(v) = bs(u) -1 ∧ ∀i ∈ [bi(v)..bs(v)] appl(v, i) = appl(u, i) *si* bs(u)< + ∞ ∧|u| ⩾ 1
- der fournit le dernier élément d'une suite finie non vide - reste augmente de 1 la borne inférieure, si la suite n'est pas vide.	der : *fonct* (SUITE)V	der *tq* der(u) = appl(u, bs(u)) *si* bs(u) < +∞ ∧ |u| ⩾ 1
	reste : *fonct*(SUITE)SUITE	reste(u) = v *tq* bi(v) = bi(u) +1 ∧ bs(v) = bs(u) ∧ ∀i ∈ [bi(v)..bs(v)] appl(v, i) = appl(u, i) *si* |u| ⩾ 1
- prem fournit le premier élément d'une suite non vide - la borne inférieure d'une suite est la borne inférieure de son domaine.	prem : *fonct* (SUITE)V	prem *tq* prem(u) = appl(u, bi(u)) *si* |u| ⩾ 1
	bi : *fonct* (SUITE)ENTIERC	bi *tq* bi(u) = bi(dom(u))
- la borne supérieure d'une suite est la borne supérieure de son domaine.	bs : *fonct* (SUITE)ENTIERC	bs *tq* bs(u) = bs(dom(u))

S.I. SUITE (2)

| | fournit la longueur d'une suite	| | : *op* (SUITE)ENTIERC	| | *tq* |u| = card(dom(u))
* représente la concaténation de 2 suites	* : *fonct* (SUITE,SUITE)SUITE	u * v = w *tq* |u| < +∞ ⊃ [bi(w) = bi(u) ∧ bs(w) = bs(u) + |v| ∧ ∀i ∈ [bi(u)..bs(u)] appl(w, i) = appl(u, i) ∧ ∀i ∈ [bs(u)+1.. bs(w)][appl(w, i) = appl(v, i)] ∧ |u| = + ∞ ⊃ w = u
tr fournit une tranche d'une suite : c'est une restriction de la suite à un intervalle de son domaine.	tr : *fonct*(SUITE, ENTIERC, ENTIERC) SUITE	tr(u, i, j) = v *tq* bi(v) = 1 ∧ bs(v) = j-i+1 ∧ ∀k ∈ [1..bs(v)] appl(v, k) = appl(u, i+k-1) *si* bi(u) ⩽ i ∧ j ⩽ bs(u)
trsp transpose deux éléments dans une suite	trsp : *fonct* (SUITE, ENTIER, ENTIER) SUITE	trsp(u, i, j) = v *tq* bi(v) = bi(u) ∧ bs(v) = bs(u) ∧ (∀k ∈ [bi(v)..i-1] U [i+1..j-1] U [j+1..bs(v)] appl(v, k) = appl(u, k)) ∧ appl(v, i) = appl(u, j) ∧ appl(v, j) = appl(u,i) *si* bi(u) ⩽ i ⩽ j ⩽ bs(u)

Figure 9 : le S.I. des suites sur un S.I. V

- On préfère la notation u_i à $u(i)$ ou $appl(u, i)$

- Si V est à son tour un S.I. SUITE on convient que :

W^{n*} représente $(\ldots((\underbrace{W*)*)* \cdots)*}_{n \text{ fois}}$

et W^{o*} représente W

- Si $V = W^{n*}$ on préfère la notation $appl(u, i_1, \ldots, i_n)$ ou encore parfois plus simplement $u_{i_1 \ldots i_n}$ à $appl(appl\ldots(appl(u, i_1), i_2)\ldots i_n)$

Ici encore :

Théorème 19 :

V* est une extension conservative de V □

Remarque 1 :

Les divers constructeurs de S.I. présentés ci-dessus sont des S.I. paramétrés.

Cependant, hormis SUITE, leur définition par des s-descriptions serait compliquée et peu intéressante. Elle consisterait en effet à définir les sortes d'intérêt en termes de sortes support plus primitives encore. On retrouve la demarche utilisée , par exemple, dans l'axiomatisation de la théorie des ensembles où l'on cherche à tout définir à l'aide d'un nombre minimal de concepts. Ici la minimalité n'est d'aucun intérêt , il faut au contraire introduire des opérations de construction de suffisamment haut niveau pour que le spécifieur puisse ultérieurement rester au niveau des s-descriptions.

Remarque 2 :

Il faudrait introduire d'autres opérations de construction telles que : l'ensemble des relations sur deux S.I., les listes sur un S.I. etc...

7.3.- Un dernier exemple.

Exemple 11 : La table des symboles [GUT, 78 a] , [WUL, 75] (cf. figure 10)

La spécification d'un univers peut apparaître à différents stades de l'analyse d'un problème : lors de la spécification évidemment, mais aussi lors de la phase d'explicitation (cf. partie 2) où il est parfois nécessaire de spécifier et de résoudre de nouveaux sous problèmes, donc d'introduire de nouveaux univers.

C'est ainsi que certaines résolutions du problème de la compilation d'un programme (cf. exemple : § 1.1 ch. 1.1) introduisent des objets intermédiaires (i.e n'apparaissant pas dans la spécification initiale du problème) appelés tables des symboles. Une telle table est construite et utilisée au fur et à mesure de la traduction du texte source et associe à chaque identificateur un ensemble de caractéristiques (type, dimension, etc...).

Nous ne cherchons pas ici, et pour cause !, à spécifier et à résoudre le problème de la compilation mais nous nous contentons d'analyser la partie relative aux identificateurs :

- Toute "génération de code" faisant intervenir un identificateur nécessite la connaissance des attributs qui lui sont associés. La table des symboles doit donc permettre l'accès à l'attribut défini par une déclaration d'identificateur ; une fonction valattr réalise cet accès.

- La rencontre d'une déclaration doit se traduire par le rangement d'un attribut dans la table. La fonction insert effectue ce rangement.

- Si le langage possède la structure de bloc, il est nécessaire de prendre en compte les règles de portée des identificateurs. Une fonction debbloc (resp. finbloc) rend compte du début (resp. de la fin) d'un bloc.

Notons que nous ne nous préoccupons pas du problème des références en avant : si on effectue plusieurs passes il est évidemment résolu, sinon il faut compliquer le modèle, par exemple en stockant dans la table des symboles

AT est le S.I. des attributs associé aux identificateurs n est la "capacité de stockage maximum " d'une table des symboles Une table des symboles est un couple formé : - d'un entier qui est le numéro de bloc courant, c'est la profondeur de ce bloc dans l'arborescence (pour l'inclusion) des blocs - d'une relation fonctionnelle rel associant à un identificateur id et à un numéro de bloc bl l'attribut correspondant à la déclaration de id dans bl. L'invariant exprime que le nombre de triplets de rel est au plus égal à n ; que bloc est supérieur ou égal à tous les numéros de blocs conservés ; que rel est fonctionnelle.	AT : *S.I.* n : ENTIER	*param* n *tq* n ⩾ 1 *fparam* TABSYM = << bloc : ENTIER, rel : $\mathcal{P}$ (<< id : CHAINE, bl : ENTIER, att : AT >>)
	ts : TABSYM	*invariant* (card(rel *de* ts) ⩽ n) ∧ (∀r ∈ rel *de* ts, 0 ⩽ bl *de* r ⩽ bloc *de* ts) ∧ (∀r_1, r_2 ∈ rel *de* ts , id *de* r_1 = id *de* r_2 ∧ bl *de* r_1 = bl *de* r_2 ⊃ att *de* r_1 = att *de* r_2
presom : fonction booléenne qui indique si un identificateur est présent au sommetde la table des symboles, c'est à dire s'il vient d'être déclaré dans le bloc courant.	presom : *fonct*(TABSYM, CHAINE)BOOL	*opérations* presom(ts, ident) = b *tq* b = ∃a : AT *tq* << ident, bloc *de* ts,a >>∈ rel *de*

S.I.TABSYM (2)

insert : définit une nouvelle table des symboles par adjonction, à la relation d'une table donnée, d'un triplet < ident, bloc *de* ts, a >	insert : *fonct* (TABSYM, CHAINE, AT) TABSYM	insert(ts, ident, a) = ts' *tq* ts' = << bloc *de* ts, rel *de* ts ∪ { << ident, bloc *de* ts, a >> } >> *si* ¬presom(ts, ident)∧ ¬vide(ts)
valattr associe à une table des symboles et à un identificateur ident l'attribut correspondant à la déclaration de ident dans le bloc le plus immédiatement englobant.	valattr : *fonct*(TABSYM, CHAINE) AT ∪ {ω}	valattr(ts, ident)=a *tq* a = *si* ∃y ∈ rel *de* ts *tq* id *de* y = ident ∧ (∀z ∈ rel *de* ts, id *de* z = ident ⊃ bl *de* z ⩽ bl *de* y) *alors* att *de* y *sinon* ω
debbloc incrémente le numéro de bloc courant	debbloc : *fonct*(TABSYM)TABSYM	debbloc *tq* debloc(ts) = << bloc *de* ts +1, rel *de* ts >>
finbloc décrémente le numéro de bloc courant et supprime tous les triplets correspondant aux déclarations dans ce bloc	finbloc : *fonct*(TABSYM)TABSYM	finbloc(ts) ts' *tq* ts' = << bloc *de* ts -1, rel *de* ts ⧵ { << ident, x, a >> *tq* x ⩾ bloc *de* ts } >> *si* ¬ vide(ts)
pleine fonction booléenne qui indique si une table des symboles est pleine	pleine : *fonct*(TABSYM)BOOL	pleine *tq* pleine(ts) = (card(rel *de* ts) = n)
vide fonction booléenne qui indique si une table des symboles est vide.	vide : *fonct*(TABSYM)BOOL	vide *tq* vide(ts) = (rel *de* ts = ∅ ∧ bloc *de* ts = 0)

Figure 10 : table des symboles.

les différents endroits où un identificateur est utilisé. En revanche, on souhaite que la spécification du système d'information formalisant les tables des symboles soit le plus possible indépendante du langage à compiler ; ce que l'on obtient, en particulier, en paramétrisant le type des attributs. Cette recherche du plus haut niveau possible d'abstraction est un pas vers la construction de générateurs de compilateurs.

Dans cette description sont utilisées quelques unes des opérations de construction introduites précédemment ($\mathcal{P}$, << >>). Cependant, s'il est naturel de définir une table des symboles comme une relation entre identificateurs et attributs, la codification des blocs par des entiers est déjà un choix de représentation : on pourrait avoir une vue plus abstraite de la notion de bloc avec des opérations propres telles que l'inclusion d'un bloc dans l'autre (cf. par exemple [WIN, 75]).

8.- EN GUISE DE CONCLUSION.

Le concept de système d'information qui vient d'être développé est un pivot dans la spécification et la résolution des problèmes :

a) Un S.I. constitue un cadre formel, reposant sur des bases théoriques classiques et solides, à partir duquel nous allons définir les concepts d'énoncé et de solution d'un problème (ch. 1.3). De plus, il est possible, à partir de ce cadre, d'étudier des méthodes générales de résolution dont deux des éléments fondamentaux sont les techniques *d'abstraction* (ch 2.2) et de *représentation* (ch. 1.3).

b) Le langage des s-descriptions associé aux opérations de construction de S.I. constitue à la fois un moyen d'expression et un guide méthodologique pour une spécification descendante et structurée de l'univers d'un problème. En particulier l'existence d'un double niveau hiérarchique (niveau des S.I., niveau des symboles fonctionnels) permet un découpage des difficultés en permettant de distinguer la spécification d'un nouveau symbole, qui est celle

d'un sous univers. Nous reviendrons sur ce point au chapitre 1.4.

L'étude d'opérations globales de constructions de types abstraits a été assez peu développée jusqu'à présent. Nous avons déjà cité BURSTALL et GOGUEN dans [BUR, 77 a] , qui utilisent le cadre des spécifications algébriques pour la définition de leur langage OBJ, et WULF, LONDON et SHAW dans [WUL, 75] qui introduisent le concept d'extension en ALPHARD.

De manière assez analogue, NAKAJIMA et *al* [NAK, 78] , [NAK, 79] étudient dans le cadre du langage IOTA, les relations entre un type et les types primitifs à partir desquels il est défini. Les systèmes formels sous jacents à IOTA sont comme ici des théories du premier ordre mais la forme des axiomes est tout à fait quelconque. Nous pensons que, sans restreindre les possibilités du spécificateur, l'introduction de S.I. supports et d'axiomes pré-post est à la fois suffisante et naturelle. Dans OBI, comme dans IOTA, l'accent est mis sur l'utilité de la hiérarchisation des types dans la construction et la vérification de représentations. Nous n'abordons ce problème qu'au chapitre 2.2 et n'insistons ici que sur l'aspect spécification progressive de l'univers d'un problème.

CHAPITRE 1.3

ENONCÉ FORMEL, SOLUTION ET RÉSOLUTION D'UN PROBLÈME

Le chapitre précédent décrit un cadre formel et un langage de spécification pour l'univers d'un problème. Pour terminer la description du concept de problème il reste à définir ce qu'est un énoncé et à se doter d'un langage de spécification.

Le concept d'énoncé est étroitement lié à ceux de *solution* et de *résolution* d'un problème. De façon informelle le *méta problème de la résolution* des problèmes peut s'énoncer de la façon suivante :

"On cherche la solution d'un problème pour une certaine valeur des données du problème".

Dans cette spécification informelle, le résultat est "la solution", les données sont "le problème",et "la valeur des données".

Conformément à l'esprit de ce travail il faut spécifier précisément ce méta problème avant d'envisager de le résoudre. Ce qui nécessite de définir les concepts de solution, donnée, valeur des données et de problème, donc le concept d'énoncé. Ainsi, pour spécifier le *méta problème de la spécification*, nous sommes conduits à étudier ce que l'on attend des problèmes une fois spécifiés, c'est-à-dire que l'on doit envisager ce qu'est la résolution d'un problème.

Ce chapitre apparaît donc comme un pivot dans ce travail. Etudiant les notions de solution et résolution d'un problème, il permet :

- d'une part de formaliser le concept de problème, puis de préciser le méta problème de la spécification.

- d'autre part de décrire les principales idées à la base de la résolution des problèmes et ainsi d'introduire le méta problème de l'explicitation des problèmes dont l'étude est l'objet de la partie 2.

C'est ainsi qu'après une étude empirique des notions de solution et de donnée (§1) nous proposons d'une part une définition formelle de ce qu'est un problème et d'autre part un langage de spécification de problème (§2). Quelques réflexions générales sur la résolution d'un problème (§3) conduisent à distinguer deux grandes classes de techniques de résolution :

- la démonstration d'une variante de l'énoncé (§4)
- le calcul d'une solution lorsque l'énoncé est sous forme explicite (§5).

La mise en oeuvre du deuxième type de techniques nécessite en général de transformer le problème initial en un problème mieux adapté aux calculs, les idées générales sur ces transformations sont présentées au paragraphe 6.

1.- NOTION D'ENONCE DE PROBLEME.

Plusieurs points de vue peuvent être adoptés pour caractériser la notion d'énoncé, étudions les sur quelques exemples.

1.1.- Solution d'un problème.

Dans le problème P_1 de la fraction irréductible (figure 2, ch. 1.1) on peut admettre dans un premier temps que propor et prem sont deux symboles de base. L'énoncé se réduit alors à :

$$\varepsilon_1 = \text{propor}(x,y,a,b) \wedge \text{prem}(x,y)$$

Cet énoncé est donc une *formule* d'un certain système d'information S_1 contenant les symboles propor et prem admettant x, y, a, b pour variables libres. x,y est le *résultat* et a,b la *donnée* de P_1. Le *problème* P_1, quant à lui, est le couple (ε_1, S_1).

Nous verrons (exemple 9, § 5.1.1) qu'un autre énoncé de ce problème peut être :

$$\varepsilon_2 = x \equiv a/z \wedge y \equiv b/z \wedge z \equiv \text{pgcd}(a,b)$$

dans un système d'information S_2 contenant les symboles / et pgcd. Dans ce deuxième énoncé, z est un *résultat intermédiaire*, c'est-à-dire que c'est le résultat d'un sous-énoncé de l'énoncé initial.

Puisque, intuitivement, un énoncé représente une relation entre des données et des résultats on peut utiliser la notion de théorème pour préciser la notion de solution :

Un couple de termes sans variable, on dit encore *termes clos*, u,v du S.I. S_1 est une *solution* de P_1 pour une valeur α, β (termes sans variable) des données si la formule $\varepsilon_1[u/x, v/y, \alpha/a, \beta/b]$ est un *théorème* de S_1.

Mais il est bien évident que si deux autres termes clos u',v' sont tels que $\varepsilon_1[u'/x, v'/y, \alpha/a, \beta/b]$ est encore un théorème de S_1, avec $\vdash_{S_1} u \equiv u'$ et $\vdash_{S_1} v \equiv v'$, (u',v') n'est pas une autre solution de P_1.

Ainsi, plus qu'un couple (u,v) de termes sans variables, une solution de P_1 est un couple $(\tilde{u}, \tilde{v})$ de classes d'équivalence pour la relation $\sim$ sur l'en-

semble des termes : $u \sim u' \Leftrightarrow \vdash_{S_1} u \equiv u'$

Par exemple $(\hat{4},\hat{3})$ est une solution de P_1 pour la valeur (24,18) des données.

La définition précédente n'implique évidemment pas l'unicité de la solution, par exemple l'énoncé :

x div 8

admet plusieurs solutions *distinctes* dans un système d'information contenant les entiers naturels et le symbole div.

Il apparaît donc nécessaire, non seulement d'être capable de caractériser les problèmes admettant des solutions, mais aussi de disposer de techniques de *résolution* de problèmes, c'est-à-dire de méthodes permettant de déterminer la ou éventuellement les solutions d'un problème.

1.2.- Intermédiaires dans un énoncé.

Dans le cas d'un énoncé comportant des résultats intermédiaires comme ε_2 la définition précédente d'une solution est encore utilisable à condition d'y inclure les résultats intermédiaires : $(\hat{4},\hat{3},\hat{6})$ est une solution de P_2 pour la valeur (24,18) des données ; $(\hat{4},\hat{3})$ en est la *solution principale*.

Une autre façon de prendre en compte les résultats intermédiaires consiste à les quantifier existentiellement. Par exemple à ε_2 on préfère :

$\varepsilon_3 = \exists\ z\ \underline{tq}\ x \equiv a/z \wedge y \equiv b/z \wedge z \equiv pgcd(a,b)$

et alors $(\hat{4},\hat{3})$ est solution du problème correspondant.

Dans les deux cas x, y et z sont des variables. Mais ceci est un cas très particulier d'énoncé et il est fréquent que les intermédiaires soient des fonctions. C'est par exemple le cas de l'énoncé complet de la figure 2, chapitre 1.1, dans lequel propor et prem sont des intermédiaires. Pour définir le concept de solution pour cet énoncé plusieurs possibilités sont envisageables :

a) Généraliser les systèmes d'information en ne les limitant plus au premier ordre. On doit ainsi introduire un calcul des prédicats d'ordre 2 (mais aussi d'ordre supérieur pour certains problèmes). On peut alors, comme précédemment, soit considérer que propor et prem sont des variables représentant des résultats intermédiaires, soit les quantifier.

b) Rester dans le cadre des S.I. introduits au chapitre précédent et imposer que les résultats et les intermédiaires soient des variables. Ceci

nécessite que le système d'information dans lequel on se place contienne des sous-systèmes "fonctionnels" (cf. ch. 1.2, § 7.2.1) tels que :

$[<<p_1:ENTIER,p_2:ENTIER,p_3:ENTIER,p_4:ENTIER>> \rightarrow BOOL]$ pour propor

$[<<p_1:ENTIER,p_2:ENTIER>> \rightarrow BOOL]$ pour prem

On se ramène ainsi au premier ordre.

c) Remplacer propor(x,y,a,b) et prem(x,y) par leur définition ; l'énoncé devient alors :

$(x * b \equiv y * a) \wedge (\forall\ z \in ENTIER^+,\ z\ div\ x \wedge z\ div\ y \supset z \equiv 1)$

Malheureusement cette méthode, qui intuitivement revient à remplacer un appel de fonction par son corps (dépliage en terme de [BUR,77b]) ne s'applique pas en général. Considérons par exemple l'énoncé ε_2 (§ 1.1) et la définition suivante de pgcd :

pgcd(x,y) ≡ *si* y ≡ 0 *alors* x *sinon* pgcd(y,reste(x,y))

Un simple remplacement ne conduira pas à un énorcé sans pgcd à moins de compliquer le système formel de manière à considérer des termes infinis.

d) Privilégier les résultats x,y en imposant qu'ils soient des variables, ce qui est une hypothèse raisonnable pour la plupart des problèmes, mais considérer que les sous-énoncés caractérisant les intermédiaires définissent une *extension* du S.I. initial :

Le S.I. initial S_1 est étendu en S_1' par l'adjonction des symboles fonctionnels propor(profil(propor) = $\overline{ENTIER\ ENTIER\ ENTIER\ ENTIER} \rightarrow \overline{BOOL}$) et prem(profil(prem) = $\overline{ENTIER\ ENTIER} \rightarrow \overline{BOOL}$), ainsi que des axiomes :

$b \equiv propor(x,y,a,b) \supset b \equiv (x * b \equiv y * a)$

$b \equiv prem(x,y) \supset b \equiv (\forall\ z \in ENTIER^+\ z\ div\ x \wedge z\ div\ y \supset z \equiv 1)$

Le concept de solution introduit au paragraphe précédent s'applique évidemment à S_1'.

On peut d'ailleurs développer cette idée d'extension pour caractériser les énoncés dont les résultats ne sont pas 0-aires. Nous reviendrons sur cette question au paragraphe 2.2.3.

Sortir du cadre du premier ordre (a)) pose un certain nombre de problèmes pour la recherche d'une solution comme nous l'évoquerons au paragraphe 3.1.2. Remplacer des appels de fonctions par leur corps (c)) ne s'applique pas dès que la spécification est récursive. Choisir entre b) et d) est un peu plus délicat :

i) Le point de vue b) fait jouer le même rôle à la formule caractérisant

le résultat et à celles concernant les intermédiaires. Ce qui rend bien compte de l'analyse descendante d'un problème. Le point de vue d) à l'inverse introduit un déséquilibre entre le résultat et les résultats intermédiaires, sauf si on considère que l'énoncé tout entier définit une extension du S.I. de départ.

ii) A l'inverse b) ressemble à un artifice technique dans la mesure où l'on "transforme" des symboles q-aires ($q \geq 1$) en symboles 0-aires. Mais il faut en mesurer les difficultés de mise en oeuvre. Ainsi considérer les intermédiaires fonctionnels comme faisant partie de la solution risque d'être difficile dans le cas de fonctions définies sur des domaines finis et impossible dans les autres cas. Il apparaît alors nécessaire de quantifier ces intermédiaires, ce qui pose de sérieuses difficultés théoriques puisqu'effectuer des démonstrations de la forme $\exists\, f\, \varphi$ nécessite en général de disposer d'axiomes sur les fonctions (axiomes du deuxième ordre). Il serait donc intéressant de développer la comparaison entre les solutions a) et b) et préciser dans quels cas b) est applicable.

Remarque 1

Bien que b) et d) ne relèvent pas du même point de vue théorique, il est intéressant de noter que les *techniques d'explicitation* qui sont développées dans la deuxième partie, dans le but de résoudre des problèmes, sont les mêmes (mutatis mutandis) qu'il s'agisse de symbole fonctionnel q-aire ($q \geq 1$) ou d'un symbole 0-aire de profil $[\overline{S}_1 \times \ldots \times \overline{S}_q \rightarrow \overline{S}]$. □

Remarque 2

Il n'est pas équivalent de quantifier les intermédiaires ou d'étendre le S.I. du problème en ajoutant les axiomes qui les caractérisent (ce qui, notons le, renforce la distinction entre b) et d) développée en ii) ci-dessus) Considérons par exemple l'énoncé :

$$y \equiv a \wedge a^2 \equiv 4$$

sur le S.I. ENTIER.

On peut démontrer que $\exists\, a(y \equiv a \wedge a^2 \equiv 4)$ admet 2 et -2 pour solutions dans ENTIER : $\exists\, a(2 \equiv a \wedge a^2 \equiv 4)$ et $\exists\, a(-2 \equiv a \wedge a^2 \equiv 4)$ sont des théorèmes de ENTIER.

Mais $y \equiv a[2 \,/\, y]$ et $y \equiv a[-2 \,/\, y]$ ne sont pas des théorèmes de l'extension ENTIER$[a^2 \equiv 4]$. En effet de $a^2 \equiv 4$ on ne peut ni déduire $2 \equiv a$, ni déduire $-2 \equiv a$.

En fait, résoudre l'énoncé dans lequel les intermédiaires ont été quantifiés revient à résoudre la partie de l'énoncé concernant le résultat dans l'extension définie par les intermédiaires si ces intermédiaires sont *uniques* (i.e. si les axiomes les caractérisant définissent des "objets uniques"). □

Il résulte de la discussion précédente que le point de vue d) semble à la fois général, ne posant pas trop de difficultés théoriques, et assez simple à mettre en oeuvre. C'est lui que nous adopterons par la suite. Le point de vue b) mériterait lui aussi une étude approfondie.

1.3.- Donnée d'un problème.

Dans des exemples de problèmes numériques comme celui des fractions irréductibles, le résultat ne dépend que de données 0-aires. En revanche dans le cas du problème des liaisons aériennes (figure 3, ch. 1.1, § 2.2), et plus généralement dans les problèmes portant sur des bases de données, le résultat dépend à la fois des données (villes de départ, vd , et d'arrivée, va) mais aussi de LIAISON qui est un univers. Il en est de même dans le problème de tri topologique où la donnée est formée d'un ensemble et d'une relation sur cet ensemble vérifiant la propriété de non circularité.

Pour définir le concept de "valeur d'une donnée" une première idée peut encore consister à utiliser la notion d'extension : si f est un symbole de donnée 0-aire, d sa valeur,on peut ajouter l'axiome $f \equiv d$ au système d'information du problème et travailler dans l'extension ainsi obtenue. Cette solution est encore acceptable dans le cas de symboles q-aires ($q \geq 1$) à condition de définir leurs valeurs en chaque point du domaine de définition (cas des fichiers par exemple). En reprenant les concepts du chapitre précédent (§ 4.2.2) une solution d'un problème serait ainsi relative à une *information* du S.I. du problème dans laquelle chaque donnée est caractérisée par un axiome précisant sa valeur.

Mais ce point de vue se généralise mal au cas où les données sont des univers ; par exemple dans le problème des liaisons aériennes il faudrait introduire le S.I. de "tous les ensembles de liaisons possibles" et une valeur de la donnée "LIAISON" apparaîtrait comme un élément de ce système. De même pour le tri topologique il faudrait définir la donnée (E,Γ) comme un élément d'un certain univers.

Ainsi, plutôt que de chercher à "coder" une donnée qui est un univers, nous sommes conduits à la considérer comme un sous-système d'information du système S du problème considéré. S devient ainsi un S.I. *paramétré*. Une

valeur d'une telle donnée SD est alors une *réalisation* D de cette donnée et pour définir la solution correspondante on se place dans la p-réalisation du système S définie par D.

1.4.- Nature des résultats.

Les réflexions précédentes ont conduit à distinguer la partie de l'énoncé caractérisant les intermédiaires de celle concernant le résultat. Mais s'il semble raisonnable de considérer qu'un résultat est 0-aire, on s'aperçoit que le seul intérêt d'avoir des variables est de permettre des substitutions. En fait il semble plus naturel de considérer qu'un énoncé définit de "nouveaux objets" c'est-à-dire des *constantes*. Ceci nécessite d'étendre l'opération de substitution (Définition 3, ch. 1.2) aux symboles 0-aires, ce qui ne pose aucune difficulté. Dans la suite nous admettrons qu'il en est ainsi, ce qui permet d'éviter une difficulté liée aux variables : le fait que le résultat soit une variable libre de l'énoncé incite à le quantifier universellement alors que c'est une quantification existentielle qui est nécessaire :

$\vdash \varepsilon[u / r] \Leftrightarrow \exists\, r(r \equiv u \wedge \varepsilon)$

2.- FORMALISATION DE LA NOTION DE PROBLEME.

2.1.- Définition d'un problème.

Définition 1

Un problème P est un couple $(E,(S,\mathcal{D}op,\mathcal{D}yp))$ où

i) $(S,\mathcal{D}op,\mathcal{D}yp)$ est un S.I. paramétré (ch. 1.2, § 6), $\mathcal{D}op$ et $\mathcal{D}yp$ constituent les *données formelles* du problème.

ii) E est l'*énoncé* du problème P.

$E = (\Sigma,\varepsilon_s,\varepsilon_i)$

- $\Sigma = (\emptyset,\{r_1,\ldots,r_q,h_1,\ldots,h_n\},\text{profil})$ est une signature caractérisant les symboles 0-aires $r_1,\ldots,r_q$ qui sont les *résultats* de P et les symboles $h_1,\ldots,h_n$ qui en sont les *intermédiaires*. Σ ne contient pas de nouvelle sorte.

- ε_s est une formule sur $\Sigma(S) \cup \Sigma$ appelée *énoncé strict* de P, elle caractérise les résultats $r_1,\ldots,r_p$ de P.

- ε_i est une formule sur $\Sigma(S) \cup (\emptyset,\{h_1,\ldots,h_n\},\text{profil})$ appelée *énoncé intermédiaire* de P. Elle caractérise les intermédiaires $h_1,\ldots,h_q$ de P.

Lorsqu'il n'y a pas d'ambiguité on appelle encore $\varepsilon_s \wedge \varepsilon_i$ l'*énoncé* de P.

Exemple 1

Dans le problème des liaisons aériennes (cf. ch. 1.1, § 2.2, figure 3) :

i) $S = \text{REEL} \cup \text{LIAISON}^* \cup \text{BOOL} \cup \text{VILLE}$

$\mathcal{D}op = \{\text{LIAISON},\text{VILLE}\}$

$\mathcal{D}yp = \{\text{départ},\text{arrivée},\text{priel},\text{har},\text{hdep},\text{vd},\text{va}\}$

ii) $\Sigma = (\emptyset,\{\text{entre},\text{prix},\text{cohérent}\},\text{profil})$ avec

$\text{profil}(\text{entre}) = \text{VILLE},\text{VILLE},\text{VILLE}^* \to \text{BOOL}$

$\text{profil}(\text{prix}) = \text{LIAISON}^* \to \text{REEL}$

$\text{profil}(\text{cohérent}) = \text{LIAISON},\text{LIAISON} \to \text{BOOL}$

- $\varepsilon_s = \text{entre}(\text{vd},\text{va},\alpha) \wedge \forall\, \beta\ \text{entre}(\text{vd},\text{va},\beta) \supset \text{prix}(\alpha) \leq \text{prix}(\beta)$

- $\varepsilon_i = b \equiv \text{entre}(u,v,\beta) \supset b \equiv \text{départ}(\text{prem}(\beta)) \equiv u \wedge \text{arrivée}(\text{der}(\beta)) \equiv v \wedge$
$\forall\, i \in [1 \ldots b_s(\beta)-1],$
$\text{arrivée}(\beta_i) \equiv \text{départ}(\beta_{i+1}) \wedge \text{cohérent}(\beta_i,\beta_{i+1})$
$\wedge\ r \equiv \text{prix}(\beta) \supset r \equiv \Sigma\ \text{priel}(\alpha)$ *pour* i *dans* $\text{dom}(\beta)$
$\wedge\ b \equiv \text{cohérent}(\ell_1,\ell_2) \supset b \equiv (\text{har}(\ell_1)+1 \leq \text{hdep}(\ell_2) \leq \text{har}(\ell_1)+6)$

□

2.2.- Solution d'un problème.

2.2.1.- Définitions

Définition 2

Avec les notations du paragraphe 2.1, une *donnée* du problème P est un couple (D,ρ) tel que (S ∪ D , ρ) soit une p-réalisation de (S,$\mathcal{D}$op,$\mathcal{D}$yp). □

(Cf. définition 29, ch. 1.2).

Dans la suite on préfère la notation SD à S ∪ D.

On exprime ainsi, d'une part que l'association d'une donnée à une donnée formelle est une p-réalisation, d'autre part qu'une donnée doit être "complète" pour les symboles fonctionnels de la donnée formelle.

Exemple 2

VILLEE = {nancy,paris,marseille,lyon} est un S.I. "ensemble fini" (cf. ch. 1.2, § 5.3).

LIAISONE = REEL ∪ VILLEE ∪ DATE ∪ (($\mathcal{S}$ore,$\mathcal{S}$yme,profile),Ae) avec

- $\mathcal{S}$ore = {$\overline{\text{LIAISONE}}$}
- $\mathcal{S}$yme = {départe,arrivéee,priele,hare,hdepe,ij200,it202,ij204,af727}
- profil(départe) = profil(arrivéee) = $\overline{\text{LIAISONE}} \rightarrow \overline{\text{VILLEE}}$
  profil(priele) = $\overline{\text{LIAISONE}} \rightarrow \overline{\text{REEL}}$
  profil(hare) = profil(hdepe) = $\overline{\text{LIAISONE}} \rightarrow \overline{\text{DATE}}$
- Ae =

  départ ij200 ≡ nancy ∧ arrivée ij200 ≡ paris ∧ priel ij200 ≡ 270.00
  ∧ hdep ij200 = 6.45 ∧ harr ij200 = 7.40

  départ it202 ≡ paris ∧ arrivée it202 ≡ marseille ∧ priel it202 ≡ 340.50
  ∧ hdep it202 ≡ 8.50 ∧ harr it202 = 9.55

  départ ij204 ≡ nancy ∧ arrivée ij204 ≡ lyon ∧ priel ij204 ≡ 200.00
  ∧ hdep ij204 ≡ 7.00 ∧ harr ij204 ≡ 7.40

  départ af727 ≡ lyon ∧ arrivée af727 ≡ marseille ∧ priel af727 ≡ 250.00
  ∧ hdep af727 ≡ 9.40 ∧ harr af727 ≡ 10.20

Alors ρ($\overline{\text{VILLE}}$) = $\overline{\text{VILLEE}}$,ρ($\overline{\text{LIAISON}}$) = $\overline{\text{LIAISONE}}$,ρ(départ) = départe, ρ(arrivée) = arrivéee,ρ(priel) = priele,ρ(har) = hare,ρ(hdep) = hdepe, ρ(vd) = nancy,ρ(va) = marseille est une donnée du problème des liaisons aériennes spécifié dans l'exemple 1. □

Par souci de simplification nous supposons ici que le problème P n'a qu'un seul résultat r, cas auquel on peut toujours se ramener en considérant des p-uplets de variables.

Soit S un S.I., la relation :

$$u \sim v \Leftrightarrow \vdash_S u \equiv v$$

est une relation d'équivalence sur l'ensemble des termes clos de S.

Notation

- $\mathcal{C}l(S)$ est l'ensemble des termes clos de S, $\mathcal{C}l(\overline{S}_i)$ le sous-ensemble de $\mathcal{C}l(S)$ formé des termes de profil $\overline{S}_i$.
- $\mathcal{C}l(S) / \sim$ est l'ensemble des classes d'équivalences pour la relation précédente.
- $\tilde{u}$ est la classe d'équivalence de u pour cette relation.

Définition 3 . Avec les notations précédentes :

Une *solution* du problème P pour la donnée (D,ρ) de P est une classe d'équivalence $\tilde{u}$ de termes clos de SD telle que profil(u) = profil(r) et

$$\rho(\varepsilon_s)[u / r] \text{ est un théorème de } SD[\rho(\varepsilon_i)]$$ □

Ainsi pour définir une solution d'un problème, on commence par préciser les données par une p-réalisation, puis on étend cette p-réalisation en introduisant les axiomes caractérisant les symboles intermédiaires et c'est dans ce S.I. que l'on cherche à démontrer un théorème $\rho(\varepsilon_s)[u / r]$.

Notation

On préfère ε'_s(resp ε'_i) à $\rho(\varepsilon_s)$ (resp. $\rho(\varepsilon_i)$)

Exemple 3

Si (D,ρ) est la donnée de l'exemple 2, on montre dans $SD[\varepsilon'_i]$:

entre(nancy,marseille,<ij204,af727>) ∧
entre(nancy,marseille,<ij200,it202>) ⊃ 450.00 ≤ 610.00

donc $\varepsilon'_s[<ij204,af727> / \alpha]$ est un théorème de $SD[\varepsilon'_i]$.

C'est-à-dire que <ij204,af727> est solution du problème. □

Ainsi, montrer que $\tilde{u}$ est une solution revient à démontrer un théorème. Ce point de vue est à la base des techniques de résolution dans lesquelles on cherche à prouver l'énoncé du problème (cf. §3).

Définition 4

Un problème P est *cohérent* (cf. ch. 1.1, § 4.1 c)) si $SD[\varepsilon'_i]$ est consistante et si $\exists r\ \varepsilon'_s$ en est un théorème pour toute donnée (D,ρ). □

Malheureusement cette caractérisation de la cohérence est peu constructive puisque sa mise en oeuvre revient en fait à résoudre P.

Définition 5

Un problème P est *déterminé* si pour toute valeur (SD,ρ) des données pour laquelle il admet une solution, cette solution est unique. □

2.2.2.-Caractérisations des solutions

Dans ce paragraphe nous cherchons d'autres caractérisations de la notion de solution dans un double but :

- découvrir des propriétés des solutions d'un problème sur lesquelles fonder des méthodes de résolution.
- étendre le concept de solution au cas où les résultats sont des symboles fonctionnels q-aires (q ≥ 1). Ceci permettrait une vue plus unifiée de l'énoncé d'un problème dans laquelle les résultats et les résultats intermédiaires seraient des objets de même nature.

Dans la suite nous posons $SD_i = SD[\varepsilon'_i]$.

Théorème 1

$\tilde{u}$ est une solution du problème P pour la donnée (D,ρ) si et seulement si $SD_i[\varepsilon'_S \wedge r \equiv u]$ est une extension conservative de SD_i.

Démonstration

a) Si $\tilde{u}$ est une solution de P, $\varepsilon'_S\ [u / r]$ est un théorème de SD_i
donc ε'_S est un théorème de $SD_i[r \equiv u]$
et donc $SD_i[r \equiv u] = SD_i[\varepsilon_S \wedge r \equiv u]$
Mais $SD_i[r \equiv u]$ est une extension conservative de SD_i :
- c'est évident si SD_i n'est pas consistante
- si SD_i est consistante, dans tout modèle (U,it) on a
$it(u) = it(r)$ sinon l'image $it(\varepsilon'_S[u / r])$ ne serait pas VRAI.

b) $SD_i[\varepsilon'_S \wedge r \equiv u]$ contient le théorème $\varepsilon'_S[u / r]$, qui est encore un théorème de SD_i puisque l'extension est conservative.

cqfd

Corollaire 1

Si $\tilde{u}$ est une solution de P, $SD_i[\varepsilon'_S \wedge r \equiv u]$ est une extension suffisamment complète de SD_i. □

Résulte du a) de la démonstration précédente ; $SD_i[\varepsilon'_S \wedge r \equiv u] = SD_i[r \equiv u]$.

Corollaire 2

$\tilde{u}$ est une solution de P pour la donnée (D,ρ), avec SD_i complet, si et seulement si $SD_i[\varepsilon'_S \wedge r \equiv u]$ est une extension consistante de SD_i. □

Résulte du théorème précédent et du théorème 15 du chapitre 1.2.

Corollaire 3

a) Si $SD_i[\varepsilon'_S]$ est non conservative, le problème P n'a pas de solution.
b) Si SD_i est consistante et $SD_i[\varepsilon'_S]$ est inconsistante, le problème P n'a pas de solution. □

Par hypothèse s'il existe une solution $\hat{u}$, $SD_i[\varepsilon'_S \wedge r \equiv u]$ est conservative, donc $SD_i[\varepsilon'_S]$ l'est. Le raisonnement est similaire pour b).

Ce corollaire exprime donc qu'une condition nécessaire pour qu'un problème soit cohérent (définition 4) est que l'extension définie par ε'_S conserve la consistance ou qu'elle soit conservative. Mais la réciproque est fausse : cette extension peut préserver la consistance sans qu'il y ait nécessairement de solution. Par exemple considérons

$S = (\{\overline{ENTIER}\},\{0,suc\},\{profil(0) = \overline{ENTIER}$
$profil(suc) = \overline{ENTIER} \to \overline{ENTIER}\},$
$\{\forall x\ \ suc\ x \neq 0 \wedge x \neq y \supset suc\ x \neq suc\ y\}).$

(entiers engendrés par 0 et la fonction de succession suc, sans axiome de récurrence).

$\varepsilon_S = suc(r) \equiv r.$

Ici il n'y a pas de données ni d'intermédiaires, donc $SD_i = S$ et ρ = identité. Alors $S[\varepsilon_S]$ a un modèle : $\mathbb{N} \cup \{+\infty\}$, l'interprétation de 0 (resp. suc, r) étant 0 (resp. +1, $+\infty$) et cependant il n'existe pas de u tel que suc(u) ≡ u soit un théorème de S.

2.2.3.-Généralisation

Le théorème 1 permet de généraliser le concept de solution au cas de symboles q-aires (q ≥ 1) sans quitter le cadre des S.I. c'est-à-dire sans introduire de variables non 0-aires.

Reprenons une définition, similaire à la définition 1, de la notion de problème en modifiant la caractérisation de ε_S :

ε_S contient p symboles fonctionnels $r_1 \ldots r_p$, non nécessairement 0-aires, qui sont les résultats du problème.

On peut alors définir une *solution* d'un tel problème de la façon suivante :

Définition 6

Avec les notations précédentes (en se ramenant à un seul résultat dans P).

Une solution du problème P pour la donnée (D,ρ) est un ensemble $\tilde{F}$ de couples $(\widetilde{g u_1 \ldots u_q}, \tilde{u})$ de classes d'équivalences tel que, si F est un ensemble de formules $g\ u_1 \ldots u_q \equiv u$ constituées à partir de représentants de ces classes, $SD_i[\varepsilon'_S, F]$ est une extension conservative de SD_i. □

Intuitivement, dans cette définition on définit la valeur du résultat en chaque point de son domaine de définition. Nous ne poursuivrons pas dans cette voie.

2.3.- s-description d'un problème.

2.3.1.-Forme syntaxique d'une s-description

Nour pouvons maintenant préciser la forme d'une s-description, en introduisant ici encore la même modification par rapport aux exemples du chapitre 1.1 que celle présentée pour les S.I. et concernant la description des profils. La figure 1 schématise une telle description.

La première partie de cette description concerne l'énoncé, les résultats et les données du problème. Les parties suivantes caractérisent les différents sous-univers du problème.

lexique	profil	description formelle
description formelle du résultat	$r_1 : p_1$ ⋮ $r_q : p_q$	*résultat* $r_1,\ldots,r_q$ *tq* ε_s
description informelle des intermédiaires	P_1 ⋮ P_n	$s - A_1$ $s - A_n$
description informelle des données		*donnée* ⋮ *fdonnée*
S.I. S_1		
⋮		
S.I. S_2		
⋮		

figure 1 : forme générale d'une s-description d'un problème

- ε_s est une formule caractérisant le résultat.
- Les différents $(P_i, s\text{-}A_i)$ sont des s-descriptions (ch. 1.2, § 5.1.1 b)) des symboles fonctionnels intermédiaires.
- La partie *donnée* ... *fdonnée* a une syntaxe similaire à la partie *param* ... *fparam* des s-descriptions des S.I. paramétres (ch. 1.2, § 6.2.2). Bien souvent, lorsque les données ne sont que des symboles fonctionnels, on les place à la suite de *donnée* et on supprime *fdonnée*.
- La partie *donnée* ... *fdonnée* peut être décomposée en plusieurs parties qui peuvent être disséminées à différents endroits entre certains $s\text{-}A_i$.

Cette forme de description est un support bien adapté à la mise en oeuvre d'une méthode de spécification descendante et structurée que nous précisons au chapitre 1.4.

On peut cependant regretter que cette syntaxe ne permette pas d'exprimer complètement que la spécification de la post condition d'un symbole fonctionnel est celle d'un nouveau sous-problème. Pour ce faire on peut imaginer d'associer à chaque s-description d'un symbole un nouveau module sous la forme suivante :

	f : *fonct*...	f x_1 ... x_p = y *tq* φ f *si* ψ f
φ f		
comme pour la description d'un problème complet	P_1 ⋮	*résultat* y *tq* ... $s - A_1$ ⋮

chaque module étant identifié par un nom de formule (ou mieux par celui du résultat de cette formule). On obtient ainsi une double structuration de la description d'un problème, l'une étant liée à l'imbrication des univers, l'autre

aux post conditions des symboles définis à un niveau donné (cf. introduction du paragraphe 5, ch. 1.2). Par souci de simplicité nous ne développerons cette syntaxe que partiellement dans le cas des énoncés MEDEE (cf. ch. 2.3).

Notons enfin que la syntaxe suggérée par la figure 1 fait ressortir les relations qui existent entre description d'énoncé et description d'univers, donc entre énoncé et univers : à la spécification du résultat près, la s-description d'un énoncé est analogue à celle d'une "extension paramétrée" d'un S.I., le S.I. étendu étant la réunion de S_i. On justifie ainsi intuitivement la définition précédente d'une solution.

2.3.2.- Problème défini par une s-description

a) Le S.I. paramétré $(S,\mathcal{D}op,\mathcal{D}yp)$ du problème P défini par une s-description de la forme de la figure 1 est formé :

- de S qui est l'extension de $S_1 \cup S_2 \cup \ldots$ par l'adjonction
  - . des sortes apparaissant dans la partie *donnée* ... *fdonnée*
  - . des symboles fonctionnels de la partie *donnée* ... *fdonnée*
  - . des profils de ces différents symboles décrits dans la partie "profil"
  - . des axiomes de la partie *donnée* ... *fdonnée*
- des sortes de la partie *donnée* ... *fdonnée*
- des symboles de la partie *donnée* ... *fdonnée*

b) . La signature $\Sigma = (\emptyset,\{r_1,\ldots,r_q,h_1,\ldots,h_n\},\text{profil})$ est obtenue à partir de $P_1 \ldots P_n$.

. L'énoncé strict ε_s est la formule caractérisant le résultat.

. L'énoncé ε_i des intermédiaires est la conjonction des axiomes associés aux s-descriptions $s\text{-}A_1 \ldots s\text{-}A_n$.

c) La p-réalisation du problème P par la donnée (D,ρ) peut être décrite par une sp-réalisation (ch. 1.2., § 6.2.3) de $(S,\mathcal{D}op,\mathcal{D}yp)$ définie par la donnée.

3.- RESOLUTION D'UN PROBLEME.

3.1.- Problème de la résolution des problèmes.

3.1.1.- Spécification du méta-problème de la résolution

En reprenant les notations du paragraphe précédent, le méta-problème de la résolution peut s'exprimer de la manière suivante (figure 2).

Le résultat $\tilde{u}$ est une classe d'équivalence pour $\sim$ de termes clos tel que la substitution d'un terme u quelconque de la classe à r dans $\rho(\varepsilon_s)$ en fasse un théorème de SD_i. SD_i est l'extension par les intermédiaires de la valeur des données. (D,ρ) est une valeur des données du problème P. valdon : *de base* associe à un problème l'ensemble de toutes ses données. Il est associé à la définition 2. clos : *de base* prédicat qui indique qu'un terme est un terme clos d'un certain S.I.	$\tilde{u}$: EQU u : TERME SD_i : *S.I.* (SD,ρ) : REALISATION P : PROBLEME	*résultat* $\tilde{u}$ *tq* $\vdash_{SD_i} \rho(\varepsilon_s)[u/r]) \wedge clos(u,SD)$ SD_1 *tq* $SD_j = SD[\rho(\varepsilon_j)]$ *donnée* (D,ρ) *tq* $(D,\rho) \in valdon(P)$ P *tq* $P = ((\Sigma,\varepsilon_i \wedge \varepsilon_s),$ $(S,\mathcal{D}op,\mathcal{D}yp))$ *fdonnée*

figure 2 : spécification du méta-problème de la résolution

On ne précise pas davantage les différents méta- S.I. apparaissant dans cette spécification, ceci reviendrait en fait à réécrire ici une partie du chapitre 1.2 et du paragraphe 2.

3.1.2.- Résolution du méta-problème

Résoudre ce méta-problème pour certaines données $(P,(D,\rho))$ n'est d'aucun intérêt et on veut le résoudre pour des classes de problèmes les plus vastes possibles. De même, au niveau d'un problème particulier on veut disposer d'une technique permettant de fournir une solution pour une donnée quelconque.

Etudions, de façon empirique, quelques idées sur lesquelles pourraient être fondées des techniques de résolution.

Résoudre P pour la donnée (D,ρ) consiste à vérifier la formule de la figure 2 caractérisant $\tilde{u}$, c'est-à-dire :

i) à trouver un terme clos u de SD.

ii) à démontrer que $\tilde{u}$ est solution de P.

Une première idée, naturelle et naïve, consiste à parcourir les différents termes clos u de SD et à essayer pour chacun d'eux de démontrer que la formule $\rho(\varepsilon_s)[u/r]$ est un théorème de SD_i. C'est appliquer ici ce que nous appellerons au chapitre 2.3 une stratégie d'affaiblissement de l'énoncé strict consistant à introduire une partie invariante (clos(u,SD)) et une partie variante ($\vdash_{SD_i} \rho(\varepsilon_s)[u/r]$) caractérisant l'arrêt de l'itération.

Exemple 4

Dans le problème de la fraction irréductible on peut explorer l'ensemble $ENTIER^+ \times ENTIER^+$ des solutions éventuelles par un procédé diagonal :

(1,1),(2,1),(2,2),(3,1),(3,2),(3,3) etc...

Pour chaque couple (u,v) on essaye de démontrer $\varepsilon_s[u/x,v/y] \equiv propor(u,v,a,b) \wedge prem(u,v)$.

Ainsi, dans le cas d'une valeur (D,ρ) telle que $\rho(a) = 24$, $\rho(b) = 18$

$\varepsilon_1(1,1)$ n'est pas un théorème de SD_i car $1 * 18 \not\equiv 1 * 24$

⋮

$\varepsilon_1(4,3)$ est un théorème car $4 * 18 \equiv 3 * 24$

et $\forall z, \exists k_1 \ \underline{tq}\ 4 \equiv k_1 * z \wedge \exists k_2 \ \underline{tq}\ 3 \equiv k_2 * z$

$\supset (z \equiv 4 \vee z \equiv 2 \vee z \equiv 1) \wedge$

$(z \equiv 3 \vee z \equiv 1)$

$\supset z \equiv 1$ □

Il est bien évident que cette manière de "résoudre" un problème, parce qu'elle nécessite un parcours du domaine des solutions, est en général inapplicable dans le cas de domaines infinis et au mieux très inefficace dans le cas de domaines finis (et inutilisable si l'un des domaines auxiliaires est infini). En bref, ce caractère exploratoire en rend son application quasi impossible.

Mais l'exploration du domaine (i.e. la recherche) des solutions n'est pas le seul problème à résoudre : il faut aussi être capable de *démontrer* qu'une formule d'un certain S.I. SD_i est un théorème. Ceci suppose que l'on dispose, pour le système formel spécifiant l'univers du problème, d'une *méthode de décision* [SHO,67] c'est-à-dire un algorithme permettant de décider, en un nombre fini d'étapes, si une formule est un théorème. Rechercher une telle méthode pour un système formel donné consiste à résoudre le *problème de décision* pour ce système.

Pour préciser cette notion de méthode de décision les logiciens ont formalisé le concept d'algorithme en créant la théorie de la calculabilité, soit à partir des fonctions récursives (CHURCH,KLEENE), soit à partir de machines formelles (TURING), soit encore en introduisant des systèmes de réécriture (THUE, POST, MARKOV). Rappelons brièvement [SHO,67] qu'un système formel $\mathcal{U}$ est *décidable* s'il existe une méthode de décision, ce qui équivaut à dire que l'ensemble des *nombres d'expression* des théorèmes de $\mathcal{U}$ est récursif (ou calculable) (le nombre d'expression d'une formule est un entier défini par récurrence sur la longueur de la formule, on dit encore que la formule est "gödelisée").

Un autre point de vue consiste à étudier l'ensemble des formules valides (c'est-à-dire, chapitre 1.2, § 3.2, des formules qui sont valides dans tout modèle). Le *problème de validité* pour un système formel consiste à étudier l'existence d'un algorithme décidant de la validité ou la non validité d'une formule quelconque.

Ce concept de formule valide aurait pu être utilisé pour définir celui de solution d'un problème : avec les notations du paragraphe précédent, $\tilde{u}$ est une solution de P si $\varepsilon_s'[u/r]$ est valide.

Cette définition peut être considérée comme plus naturelle que celle énoncée au paragraphe précédent dans la mesure où on cherche plus une solution au niveau d'un univers du problème qu'au niveau formel. En fait elle lui est équivalente d'après le théorème de complétude de GÖDEL qui affirme qu'une formule du calcul des prédicats du premier ordre est un théorème si et seulement si elle est valide.

Malheureusement, on démontre que ni le problème de décision, ni donc celui de validité, n'a de solution dans le cas du calcul du prédicat du premier ordre (ni donc dans le cas des systèmes d'information) : il n'existe pas d'algorithme permettant de décider si une formule est ou non un théorème. On dit que le calcul des prédicats du premier ordre est *indécidable*. On a cependant la *semi-décidabi-*

lité : il existe un algorithme permettant d'affirmer qu'une formule est un théorème, lorsqu'elle l'est effectivement, et pouvant ne pas donner de résultat si la formule n'est pas un théorème.

Dans le cas du calcul des prédicats du deuxième ordre la situation est encore moins favorable : l'ensemble des théorèmes est un sous-ensemble strict de celui des formules valides et si on a toujours la semi-décidabilité pour les théorèmes, on ne l'a même plus pour les formules valides [MAN,74]. C'est l'une des raisons pour lesquelles le second ordre est mal adapté à la définition du concept de solution, donc à la spécification des problèmes.

Signalons enfin que l'existence d'une méthode de résolution n'est satisfaisante que si elle est efficace, c'est-à-dire que sa mise en oeuvre conduit à des résultats dans des délais raisonnables.

3.2.- Quelques idées sur la résolution.

Nous nous préoccupons maintenant de dégager les concepts sur lesquels peut être fondée une technique de résolution de problèmes. Il ne s'agit pas, dans la suite de ce paragraphe et au paragraphe 4, de trouver de nouvelles techniques de résolution mais plutôt de montrer comment les considérations théoriques précédentes sur les problèmes permettent de dégager les fondements des différentes techniques de résolution et d'en faire une présentation synthétique.

3.2.1.- Choix du système formel

Pour résoudre un problème il apparaît donc naturel de se placer dans un système formel pour lequel existent à la fois une méthode de résolution (éventuellement partielle) et une méthode de construction de la solution. C'est la raison pour laquelle nous nous sommes restreints au premier ordre dans la définition des systèmes d'information : nous allons voir que de telles méthodes (en général partielles) peuvent y être développées. Il faudrait aussi chercher à définir d'autres systèmes formels pour lesquels existe une méthode de décision efficace.

3.2.2.- Choix de la forme de l'énoncé

Même s'il n'existe pas de méthode de résolution totale dans l'univers d'un problème on peut arriver à résoudre certains énoncés ayant une forme particulière Par exemple, l'énoncé suivant :

x = f(3)

f *tq* f(x) = *si* x = 0 *alors* 1 *sinon* f(x-1)*x

peut être résolu par une suite de substitutions. Donc tout ce qui est demandé au système formel sous-jacent est de disposer d'un axiome de substitution. Il est donc intéressant de chercher à définir des méthodes de résolution pour certaines formes d'énoncés :

De manière intuitive, un *énoncé explicite* d'un système d'information S est une conjonction de formules de la forme

$$u \equiv t$$

où u est une variable ou un terme de la forme $f\ u_1 \ldots u_q$, f étant un symbole q-aire, $u_1 \ldots u_q$ des termes pouvant être réduits à des variables, t un terme quelconque de même profil que u.

Un énoncé qui n'est pas explicite est dit *implicite*.

Exemple 5

- propor(x,y,a,b) ∧ prem(x,y) ∧
  b' ≡ propor(x',y',a',b') ⊃ b' ≡ (x' * b' ≡ y' * a') ∧
  b" ≡ prem(x',y') ⊃ b" ≡ (∀ z ∈ $\mathbb{N}^+$, z div x' ∧ z div y' ⊃ z ≡ 1)
  est un énoncé implicite du problème de la fraction irréductible.

- x ≡ a / z ∧ y ≡ b / z ∧ z ≡ pgcd(a,b) ∧
  pgcd(u,v) ≡ *si* v ≡ 0 *alors* u *sinon* pgcd(v,reste(u,v))
  en est un énoncé explicite. □

Ainsi la recherche d'une méthode de décision va de pair à la fois avec la forme du système formel et celle de l'énoncé. Nous étudierons dans la deuxième partie les deux classes principales d'énoncés explicites : les définitions récursives et les énoncés récurrents.

3.2.3.- Transformation d'énoncés

Soit P un problème de S.I. S. On peut chercher à résoudre directement P en utilisant une méthode de décision, éventuellement partielle, dans S.

On peut aussi chercher à *transformer* S ou l'énoncé E de P de manière à faciliter la résolution. Il faut donc étudier des mécanismes de transformation permettant de traduire un énoncé portant sur un S.I. S_1 en un énoncé "équivalent" portant sur un S.I. S_2. Plus précisément il faut être capable de définir le passage (on dit la *représentation*) de S_1 à (par) S_2 et de préciser le lien entre l'énoncé de départ et l'énoncé obtenu (ce qui nécessite d'introduire des règles de preuve de correction des transformations).

De manière complémentaire, il faut étudier des techniques permettant, à S.I. constant, de transformer un énoncé en un autre plus facilement résoluble : c'est le problème de l'*explicitation* d'un énoncé.

Explicitation de formules et représentation de S.I. sont les deux idées qui sont à la base de la construction d'algorithmes efficaces à partir d'un énoncé initial. Il faut donc préciser et étudier ces concepts, puis en déduire une méthodologie de mise en oeuvre : c'est l'objet de la deuxième partie de ce travail qui est guidée par l'étude théorique que nous développons au paragraphe 6 de ce chapitre.

La résolution directe du problème donné correspond à un point de vue *interprétatif*, la transformation de ce problème en un autre correspond à un point de vue *compilatoire*.

3.2.4.- Techniques classiques de résolution de problème

Qu'il soit donné directement ou qu'il soit le résultat d'un certain nombre de transformations il faut finalement résoudre l'énoncé du problème. Les deux grandes familles de techniques de résolution sont très étroitement liées aux deux points de vue sur la définition d'une solution (cf. § 2.2) :

a) La première idée (fondée sur la définition 3 d'une solution) consiste à chercher à démontrer directement dans un S.I. SD_i, qui est l'extension par les axiomes des intermédiaires d'une valeur des données du problème, un théorème de la forme $\varepsilon'_s[u/r]$. Cette technique a été développée pour le langage PROLOG [WAR,77a]. Nous y faisons brièvement allusion au paragraphe 3.

b) La deuxième, reposant sur le théorème 1, consiste à ajouter l'énoncé strict ε'_s à SD_i et à chercher à démontrer dans l'extension $SD_i[\varepsilon'_s]$ ainsi obtenue un théorème de la forme $r \equiv u$. Une telle démonstration s'appelle en général un *calcul* et ne peut être mise en oeuvre que si les énoncés sont explicites. C'est en particulier le cas des problèmes énoncés dans les langages de programmation classiques.

La mise en oeuvre de la première idée repose sur les résultats obtenus en intelligence artificielle concernant la démonstration automatique de théorèmes. La mise en oeuvre de la deuxième idée relève de la théorie de la calculabilité.

Remarque

Le concept de solution d'un problème admet deux sens différents en informatique :

- une solution est la valeur du résultat correspondant à certaines valeurs des données. C'est le sens commun du mot "solution". C'est aussi celui que nous avons utilisé ici.

- par extension, en informatique, on appelle souvent solution d'un problème la fonction qui associe à une valeur des données le résultat correspondant, cette fonction étant définie de manière explicite (récursivement dans le cas général).

Préciser ce deuxième point de vue nécessite d'avoir déjà caractérisé le premier, c'est la démarche que nous avons suivie ici, en reportant à la deuxième partie l'étude des définitions explicites et des méthodes d'explicitation ; ce que l'on peut appeler des *solutions générales* de problèmes (i.e. indépendantes des données).

4.- DEMONSTRATION D'UNE VARIANTE DE L'ENONCE.

Nous nous plaçons désormais dans le cadre du calcul des prédicats du premier ordre. La technique développée ici est une mise en oeuvre de la semi-décidabilité de ce système formel : on précise une méthode (peut-on parler d'algorithme ?) permettant de décider qu'une formule donnée est un théorème s'il en est bien ainsi. Nous commençons par mettre en évidence les concepts sur lesquels est fondée cette méthode (§ 4.1) puis nous décrivons sa mise en oeuvre en PROLOG (§ 4.2).

4.1.- Fondements de la méthode de résolution de ROBINSON. [CHA,73]

4.1.1.- Skolémisation

Théorème 2 (SKOLEM)

Toute formule (bien formée) du calcul des prédicats du premier ordre peut s'écrire sous forme *clausale*. □

Une *clause* est une formule de la forme

$$B_1 \vee B_2 \vee \ldots \vee B_m \subset A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

notée plus simplement

$$B_1, \ldots, B_m \leftarrow A_1, \ldots, A_n$$

où les B_i, A_j sont des formules atomiques.

Une formule φ est dite sous forme clausale si elle s'écrit

$\forall x_1 \ldots \forall x_j \quad C_1 \wedge \ldots \wedge C_k$ où les C_i sont des clauses

ou simplement $C_1 \wedge \ldots \wedge C_k$ puisque $\vdash C$ si et seulement si $\vdash \forall x\, C$

qui se note encore :

C_1
C_2
$\vdots$
C_k

Le théorème de SKOLEM précise que l'on peut associer à toute formule du calcul des prédicats une formule sous forme clausale qui lui est équivalente (i.e. qu'elles sont valides toutes les deux ou qu'aucune ne l'est). La démonstration de ce théorème est constructive; le point central consiste à supprimer les quantificateurs existentiels par l'introduction de fonctions supplémentaires (c'est la skolémisation).

Exemple 6

La formule définissant la factorielle s'écrit sous forme clausale :

(1) fact(0,s(0)) ←

(2) fact(s(x),z) ← fact(x,t),prod(s(x),t,z)

fact est un nouveau prédicat, prod caractérise la multiplication, il est lui-même défini à partir de plus caractérisant l'addition.

(3) prod(s(x),y,z) ← prod(x,y,t),plus(t,y,z)

(4) prod(0,y,0) ←

(5) plus(s(x),y,s(z)) ← plus(x,y,z)

(6) plus(0,x,x) ←

0 est un symbole 0-aire et s un symbole unaire interprétable comme la succession dans les entiers. □

Dans l'exemple précédent, en partie gauche de ← on ne trouve qu'un seul atome : il s'agit de *clauses de HORN*.

4.1.2.- Algorithme d'unification

Rappelons qu'une substitution θ est un ensemble $\{t_1/x_1,\ldots,t_n/x_n\}$ où chaque x_i est une variable et chaque t_i un terme ; on suppose que tous les x_i sont différents et que t_i est différent de x_i. Pour toute formule φ on note $\varphi\theta$ l'expression obtenue en remplaçant simultanément chaque occurrence de x_i ($1 \leq i \leq n$) par t_i.

Soit u et v deux termes, on nomme *unificateur* de u et v toute substitution θ telle que $u\theta = v\theta$.

Par exemple $\theta_1 = \{s(0)/x,z/t\}$ et $\theta_2 = \{s(0)/x,t/z\}$

sont deux unificateurs de fact(s(s(0)),z) et fact(s(x),t).

Un *plus grand unificateur* de deux formules φ et ψ est un unificateur σ tel que pour tout autre unificateur θ de φ et ψ il existe une substitution α telle que $\theta = \sigma \circ \alpha$(†). Ainsi θ_1 et θ_2 précédents sont des plus grands unificateurs des deux termes considérés.

(†) Si $\sigma = \{t_1/x_1,\ldots,t_p/x_p\}$ et $\alpha = \{z_1/y_1,\ldots,z_k/y_k\}$, $\sigma \circ \alpha$ est la substitution déduite de $\{t_1\alpha/x_1,\ldots,t_p\alpha/x_p,z_1/y_1,\ldots z_k/y_k\}$ en supprimant tous les $t_i\alpha/x_i$ lorsque $t_i\alpha = x_i$ et chaque z_i/y_i lorsque y_i est l'un des x_j.

On trouve par exemple dans [CHA,73] ou [MAN,74] la description d'un algorithme d'unification pour deux termes u et v donnés. Cet algorithme fournit un plus grand unificateur s'il en existe et signale, si ce n'est pas le cas, que u et v ne sont pas unifiables (ainsi le problème de l'unification de deux termes est décidable dans les théories du premier ordre).

4.1.3.- Description de la méthode de résolution [CHA,73],[MAN,74]

Une formule d'un S.I. est dite *inconsistante* ou encore *non satisfaisable* si elle prend la valeur FAUX pour toute interprétation. En se rappelant qu'une formule est valide si et seulement si elle prend la valeur VRAI pour toute interprétation, on obtient une nouvelle caractérisation de la validité :

Une formule φ d'un système d'information est valide si et seulement si $\neg\varphi$ est inconsistante.

Cette remarque n'a d'intérêt que dans la limite où l'on dispose d'une méthode permettant de vérifier l'inconsistance d'une formule : c'est le rôle de la *méthode de résolution* de ROBINSON. Avant de la détailler, commençons par énoncer le théorème de HERBRAND qui donne une première caractérisation des formules inconsistantes.

Rappelons tout d'abord que l'*univers de HERBRAND* associé à un S.I. S, noté H_S, est l'ensemble de tous les termes clos de S. Une *interprétation de HERBRAND*, on dit encore *interprétation libre*, de S est obtenue en associant à chaque symbole 0-aire a le terme a et à chaque symbole fonctionnel f de profil $\overline{S}i_1,\overline{S}i_2,\ldots,\overline{S}i_q \to \overline{S}i$ la fonction associant à chaque q uplet de termes $(u_{i_1},\ldots,u_{i_q})$ de H_S de profils respectifs $\overline{S}_{i_1} \ldots \overline{S}_{i_q}$, le terme $f\, u_{i_1} \ldots u_{i_q}$ de H_S ayant pour profil $\overline{S}_i$. Alors il est immédiat de démontrer qu'une formule est inconsistante si et seulement si elle prend la valeur FAUX pour toute interprétation de HERBRAND. Le théorème de HERBRAND renforce ce résultat en ne demandant de vérifier cette propriété que dans un nombre fini de cas :

Théorème 3 (HERBRAND)

Une formule sous forme clausale est inconsistante si et seulement si il existe une conjonction finie d'instances de base de ses clauses qui est inconsistante. □

On appelle *instance de base* d'une clause C apparaissant dans une formule sous forme clausale φ toute clause déduite de C en substituant à chaque variable de C (en respectant les profils) un terme de la restriction H_φ à φ de l'univers de HERBRAND H_S.

Exemple 7

plus(s(s(s(0))),s(0),s(0)) ← plus(s(s(0)),s(0),0)

est une instance de base de la clause

plus(s(x),y,s(z)) ← plus(x,y,z)

de la formule de l'exemple 6. □

Ce théorème fournit une méthode de vérification de l'inconsistance d'une formule φ : il suffit d'engendrer successivement les instances de base des différentes clauses de φ et de tester si leur conjonction est inconsistante (méthode de DAVID et PUTNAM). On dispose ainsi d'un "algorithme" semi-décidable: il s'arrête pour une formule φ inconsistante et donne le résultat, en revanche il peut boucler si φ n'est pas inconsistante.

ROBINSON [ROB,65] a amélioré cette idée en remarquant qu'une formule sous forme clausale $C_1 \wedge \ldots \wedge C_p$ est inconsistante si et seulement si on peut trouver une formule équivalente à l'une de ses interprétations de HERBRAND qui contient la clause vide (dont toute interprétation est FAUX) :

← (intuitivement *vrai* ⊃ *faux*)

que l'on note encore □.

La "formule équivalente" peut aussi être considérée comme une *démonstration* de □ à partir de la formule $C_1 \wedge \ldots \wedge C_p$, en utilisant la règle d'inférence suivante, appelée *méthode de résolution de ROBINSON* :

De $B \leftarrow A_1,\ldots,A_i,\ldots,A_k$

et $A \leftarrow D_1,\ldots,D_p$

on déduit $B\,\theta \leftarrow (A_1,\ldots,A_{i-1},D_1,\ldots,D_p,A_{i+1},\ldots,A_k)\,\theta$

si θ est un unificateur de A_i et A.

(Cette règle est simplement une variante de la transitivité de l'implication).

Le théorème de ROBINSON permet de démontrer la complétude de cette méthode.

Théorème 4 (ROBINSON)

Une formule φ sous forme clausale est inconsistante si et seulement si la clause □ peut en être déduite par application de la méthode de résolution (on dit alors que l'on *réfute* φ). □

La conjonction de la méthode de résolution et de l'algorithme d'unification donne un outil puissant de démonstration dans le calcul des prédicats du premier ordre.

Décrivons sa mise en oeuvre sur un exemple.

Exemple 8

On veut prouver fact(s(s(0)),s(s(0))) en utilisant la formule de l'exemple 6. Pour cela on ajoute (cf. § 2.2) cette formule aux axiomes du système d'information considéré (en fait ici ce seront les seuls axiomes que l'on considèrera et on ajoute la négation du résultat demandé. Il faut alors chercher à démontrer dans ce système la clause □ .

On ajoute donc : (7) ← fact(s(s(0)),s(s(0))).

Une démonstration possible est alors :

(7) . ← fact(s(s(0)),s(s(0)))

(8) . ← fact(s(0),t),prod(s(s(0)),t,s(s(0)))

unification (2) et (7), θ = {s(0)/x,s(s(0))/z}

(9) . ← fact(0,t'),prod(s(0),t',t),prod(s(s(0)),t,s(s(0)))

unification du premier terme de (8) avec (2) θ = {0/x,t/z,t'/t}

(10) . ← prod(s(0),s(0),t),prod(s(s(0)),t,s(s(0)))

unification du premier terme de (9) avec (1) θ = {s(o)/t'}

(11) . ← prod(0,s(0),t"),plus(t",s(0),t),prod(s(s(0)),t,s(s(0)))

unification du permier terme de (10) avec (3) θ = {0/x,s(0)/y,t/z,t"/t}

(12) . ← plus(0,s(0),t),prod(s(s(0)),t,s(s(0)))

unification du premier terme de (11) avec (4) θ = {s(0)/y,0/t"}

(13) . ← prod(s(s(0)),s(0),s(s(0)))

unification du premier terme de (12) avec (6) θ = {s(0)/x,s(0)/t}

(14) . ← prod(s(0),s(0),t),plus(t,s(0),s(s(0)))

unification de (13) avec (3) θ = {s(0)/x,s(0)/y,s(s(0))/z}

(15) . ← prod(0,s(0),t'),plus(t",s(0),t),plus(t,s(0),s(s(0)))

unification du premier terme de (14) avec (3) θ = {0/x,s(0)/y,t'/t,t/z}

(16) . ← plus(0,s(0),t),plus(t,s(0),s(s(0)))

unification du premier terme de (15) avec (4) θ = {s(0)/y,t'/0}

(17) . ← plus(s(0),s(0),s(s(0)))

unification du premier terme de (16) avec (6) θ = {s(0)/x,s(0)/t}

(18) ← plus(0,s(0),s(0))

unification de (17) avec (5) θ = {0/x,s(0)/y,s(0)/z}

(19) □

unification de (18) avec (6) θ = {s(0)/x}

Cet exemple consiste simplement à démontrer que la valeur de 2! est 2. Mais il est bien évident qu'en général on ne connaît pas la valeur du résultat et qu'il s'agit, en même temps que se fait la démonstration, de déterminer cette valeur. Ici par exemple on chercherait à réfuter le prédicat fact(s(s(0)),a) et les unifications successives conduiraient à déterminer a :

(7) . ← fact(s(s(0)),a)

la suite de la déduction serait la même que précédemment en remplaçant dans chaque clause intermédiaire, jusqu'à la clause (17) incluse, s(s(0)) "le plus à droite" par a.

Alors (17) ← plus(s(0),s(0),a)

et (18) ← plus(0,s(0),b) où a = s(b)

puis (19) □ si b = s(0)

Conjointement on démontre la formule initiale et on détermine la valeur de a = s(s(0)). □

4.1.4.- Mise en oeuvre de la méthode de résolution

Pour réfuter une clause on peut, comme dans l'exemple précédent, partir de cette clause et chercher à l'unifier aux termes gauches des axiomes : c'est une méthode de résolution descendante (top-down). Or peut aussi partir des termes de droite des axiomes et les remplacer par les termes de gauche jusqu'à introduire la négation de la clause à prouver : méthode ascendante (bottom up). En général, dans l'une et l'autre méthode,des choix sont nécessaires quant aux axiomes et aux unificateurs à utiliser ; cette technique de résolution est *indéterministe* et sa mise en oeuvre nécessite des retours en arrière dans l'arbre des choix. Mais l'utilisation exclusive de clauses de HORN facilite en partie le travail dans la mesure où elle permet une *résolution linéaire*, donc ne nécessitant pas la mémorisation d'intermédiaires contrairement à ce qui se passerait dans le cas de clauses générales.

Comme pour tout algorithme indéterministe, des améliorations sont possibles par l'introduction d'heuristiques et par la recherche de formes d'énoncés mieux adaptées à la mise en oeuvre de la méthode (en particulier en ordonnant les clauses). Quelques unes de ces techniques sont discutées dans [KOW,74],[CHA,73], [WAR,77b] etc...

4.2.- PROLOG et les limites de la méthode de résolution.

4.2.1.- Description sommaire du langage PROLOG

Le langage PROLOG [ROU,75],[WAR,77a] est une mise en oeuvre concrète des idées précédentes. Les phrases de PROLOG sont les clauses de HORN et l'interprète travaille de manière descendante. Un programme PROLOG peut être considéré de deux manières :

- c'est un ensemble d'assertions qui représente un énoncé

- c'est un ensemble de définitions et d'appels de procédures puisque toute clause $B \leftarrow A_1,\ldots,A_n$ peut être considérée comme la définition de la procédure B dont le corps est formé des appels $A_1,\ldots,A_n$.

Le premier point de vue correspond à la spécification d'un problème, le deuxième à sa résolution ; à chacun d'eux correspond une sémantique particulière : sémantique déclarative qui définit l'ensemble des formules valides et une sémantique opérationnelle qui décrit les règles de calcul;dans ce dernier c il est nécessaire de définir un ordre sur les clauses. On retrouve cette dualité entre le point de vue expression d'un problème et celui concernant sa résolution dans la plupart des langages "déclaratifs" (ou encore non procéduraux) :

- Les langages fondés sur la notion de suite tels que LUCID [ASH,76], [ASH,77], MEDEE [BEL,78a] ou celui introduit dans [ARS,77] peuvent être à la fois considérés comme un système formel permettant d'énoncer un problème ou com un langage de programmation (cf. ch. 2.3).

- Il en est de même de LISP (pur) [WEI,67] qui est fondé sur le système formel du λ-calcul de CHURCH et qui permet d'exprimer des énoncés récursifs.

La différence fondamentale entre PROLOG et ces autres langages "statiques est donc la forme de la méthode de résolution qui est liée à la forme des énoncés (clausale en PROLOG, explicite dans les autres langages).

4.2.2.- Problèmes posés par la mise en oeuvre de la méthode de résolution

Le fait, pour un langage tel que PROLOG, d'être à la fois un langage d'assertions et un langage de programmation présente des avantages, mais aussi inconvénients sérieux :

- l'analyse et la programmation d'un problème peuvent se faire dans un m langage et le programmeur dispose ainsi d'outils de très haut niveau.

- en revanche une mauvaise spécification (au sens mal adaptée à l'algorithme de résolution) va conduire à une exécution très inefficace : ainsi il fau quand même que le programmeur prenne en compte l'exécution ultérieure lors de l'écriture du programme. De plus l'écriture sous forme clausale n'est pas toujours très intuitive (!) et une démarche rationnelle consiste à exprimer son problème dans la logique du calcul des prédicats puis à le transformer sous forme clausale, transformation qui peut être automatisée [MAN,74]. Ceci devier beaucoup plus difficile si l'énoncé initial utilise un autre langage de spécification que le calcul des prédicats du premier ordre.

D'autre part la semi-décidabilité du calcul des prédicats implique qu'un programme PROLOG peut ne pas se terminer et on ne dispose pas actuellement, comme dans les langages de programmation classiques, de techniques telles que la recherche d'ensembles bien fondés [LIV,78] pour prouver la terminaison.

De plus, l'indéterminisme inhérent à la méthode de résolution peut rendre l'exécution d'un programme PROLOG très inefficace. Ainsi dès que la taille de l'énoncé est grande ou que la structure des objets est complexe on a peu de chances d'obtenir un résultat dans un délai raisonnable. Cependant les programmeurs experts en PROLOG peuvent arriver à une efficacité raisonnable à la suite d'un certain nombre de transformations, codification, ordonnancement de l'énoncé de départ. Il serait intéressant d'étudier plus formellement ce genre de techniques. Malgré cela l'explosion combinatoire liée aux choix à faire fait reculer de nombreux chercheurs qui espéraient beaucoup de ce formalisme.

Signalons également que l'utilisation du calcul des prédicats conduit à exprimer de manière analogue énoncé et objets. Ce peut être considéré comme un avantage puisqu'on arrive ainsi directement à une expression uniforme. C'est aussi un inconvénient dans la mesure où la structuration du programme, et par là même sa facilité de compréhension et de construction, ne sont pas immédiates (et nécessiteraient la définition de primitives de haut niveau, inexistantes dans le langage de base).

Terminons en indiquant que si PROLOG a été utilisé pour résoudre des problèmes d'analyse syntaxique [COL,75], il y a, de manière plus générale, des rapports étroits entre les clauses de HORN et les grammaires à attribut [LOR,74] ou les doubles grammaires [SIM,79] (ainsi, on peut comparer la méthode de résolution à la dérivation dans une telle grammaire)

5.- CALCUL DIRECT D'UNE SOLUTION.

5.1.- Présentation intuitive.

L'autre technique classique de résolution d'un problème P (cf. § 3.24 b)) consiste à ajouter le résultat r et l'énoncé strict ε'_s de P à l'extension $SD_i = SD[\varepsilon'_i]$ correspondant à la donnée (D,ρ) de P, puis à déterminer un terme clos u de SD tel que $r \equiv u$ soit un théorème de $SD_i[\varepsilon'_s]$.

Sa mise en oeuvre nécessite que l'énoncé strict ε_s soit explicite.

5.1.1.- Quelques exemples

Exemple 9

Soit P = (E,(ENTIER,∅,{a})) un problème où E contient
$\varepsilon = r \equiv fact(a) \wedge fact(x) \equiv$ *si* $x \equiv 0$ *alors* 1 *sinon* $fact(x-1) * x$.

Pour définir la solution correspondant à la donnée ({2},ρ) avec ρ(a) = 2, on étend SD par l'énoncé $\varepsilon'(=\varepsilon'_s \wedge \varepsilon'_i)$. Dans SD[ε] on a la démonstration suivante:

$r \equiv fact(2)$
$fact(2) \equiv$ *si* $2 \equiv 0$ *alors* 1 *sinon* $fact(1) * 2$
$fact(2) \equiv fact(1) * 2$
$fact(1) \equiv$ *si* $1 \equiv 0$ *alors* 1 *sinon* $fact(0) * 1$
$fact(1) \equiv fact(0) * 1$
$fact(0) \equiv$ *si* $0 \equiv 0$ *alors* 1 *sinon* $fact(-1) * 0$
$fact(0) \equiv 1$
$1 * 1 \equiv 1$
$fact(1) \equiv 1$
$1 * 2 \equiv 2$
$fact(2) \equiv 2$
$r \equiv 2$

Notons tout de suite qu'une telle démonstration nécessite que l'application des différents axiomes et règles d'inférence se fasse dans un certain ordre : il est nécessaire de définir des *règles de calcul*. □

Exemple 10

Soit P = (E,(S,∅,{a,b})) où
- S contient ENTIER, a, b, pgcd et les axiomes caractérisant pgcd.
- E contient l'énoncé $\varepsilon = x \equiv a \,/\, z \wedge y \equiv b \,/\, z \wedge z \equiv pgcd(a,b)$, le résultat étant (x,y).

Dans l'extension de SD (avec ρ(a) = 24, ρ(b) = 18) définie par ε' on peut effectuer la démonstration suivante :

$z \equiv pgcd(24,18)$
$pgcd(24,18) \equiv 6$
$z \equiv 6$
$x \equiv 24 \,/\, z$
$x \equiv 24 \,/\, 6$
$24 \,/\, 6 \equiv 4$
$x \equiv 4$
$y \equiv 18 \,/\, z$
$y \equiv 18 \,/\, 6$
$18 \,/\, 6 \equiv 3$
$y \equiv 3$ □

5.1.2.- Conditions d'applicabilité de cette technique

La technique de résolution esquissée sur les deux exemples précédents nécessite :

- d'une part que l'extension SD[ε'] soit consistante, sinon la démonstration de $r \equiv u$ n'a aucun sens. Une condition suffisante pour qu'il en soit ainsi est que le problème P admette une solution pour (D,ρ) (corollaire 3 du théorème 1).

- d'autre part il est nécessaire que la solution du problème P soit unique. En effet s'il existe deux termes clos u, v distincts dans SD (i.e. tels que $\vdash_{SD} u \not\equiv v$) et si $r \equiv u$ est un théorème de $SD_i[\varepsilon'_s]$, c'est encore un théorème de $SD_i[\varepsilon'_s \wedge r \equiv v]$, donc dans ce S.I. on a le théorème $u \equiv v$, ce qui avec $u \not\equiv v$ implique que $SD_i[\varepsilon'_s \wedge r \equiv v]$ est inconsistant. On obtient une contradiction avec le fait que $\tilde{v}$ soit une solution.

Mais ces deux conditions ne sont pas suffisantes pour assurer la complétude de la méthode. Ainsi, en reprenant un exemple de [PAI,75], considérons le S.I. S "des entiers modulo 2" avec pour seuls symboles fonctionnels les constantes 0 et 1 et le symbole binaire +, pour seuls axiomes

$0 + 0 \equiv 0$, $0 + 1 \equiv 1$, $1 + 0 \equiv 1$, $1 + 1 \equiv 0$

Alors le problème P sur ce S.I. dont l'énoncé est $x + x \equiv x$ admet une solution unique 0 qui ne peut être obtenue par un calcul, c'est-à-dire que $x \equiv 0$ n'est pas un théorème de S [$x + x \equiv x$]. Ceci est dû au fait que l'énoncé considéré n'est pas explicite.

5.2.- Solution calculable d'un problème.

Nous formalisons maintenant les concepts de calcul d'une solution et de solution calculable, ce qui permet de préciser des conditions d'existence d'une solution calculable.

5.2.1.- Définition et propriétés des solutions calculables

On reprend ici les notations du paragraphe 2.

Définition 7

Soit P = (E,S) un problème. Si P admet une solution et si, u étant un terme clos de SD, r ≡ u est un théorème de $SD_i[\varepsilon'_S]$, la classe d'équivalence $\tilde{u}$ de u est appelée *solution calculable* de P. □

Ainsi $\tilde{2}$ est une solution calculable du problème de l'exemple 9.

Théorème 5

Toute solution calculable d'un problème P est une solution de P. □

Démonstration

Par hypothèse P admet une solution $\tilde{v}$, ainsi $SD_i[\varepsilon'_S \wedge r \equiv v]$ est une extension conservative de SD_i, il en est donc de même de $SD_i[\varepsilon'_S]$.

Mais $\vdash_{SD_i[\varepsilon'_S]} r \equiv u \Rightarrow SD_i[\varepsilon'_S] = SD_i[\varepsilon'_S \wedge r \equiv u]$ qui est conservative. On en tire, avec le théorème 1, que $\tilde{u}$ est une solution de P. □

Remarque

Il ressort de cette démonstration la nécessité de l'existence d'une solution pour que puisse exister une solution calculable. Cependant si SD_i est complet on peut affaiblir cette hypothèse (corollaire 2 du théorème 1) en demandant seulement que $SD_i[\varepsilon'_S]$ soit consistant.

Théorème 6

Si SD_i (=$SD[\varphi(\varepsilon_i)]$) est une extension conservative de SD et si $\tilde{u}$ est une solution calculable de P, $\tilde{u}$ est l'unique solution de P. □

Démonstration

Si $\tilde{v}$ est une solution de P, r ≡ u est un théorème de $SD_i[\varepsilon'_S \wedge r \equiv v]$ puisque c'est un théorème de $SD_i[\varepsilon'_S]$.

Ainsi $\vdash_{SD_i[\varepsilon'_S \wedge r \equiv v]} u \equiv v$, donc $\vdash_{SD_i} u \equiv v$ puisque $SD_i[\varepsilon'_S \wedge r \equiv v]$ est conservative, et par hypothèse $\vdash_{SD} u \equiv v$.

cqfd

La notion de solution calculable permet de caractériser les problèmes qui n'ont qu'une seule solution, ce que nous avons appelé les problèmes déterminés (définition 5).

Théorème 7

Si $SD_i[\varepsilon'_S]$ est complet alors P admet au plus une solution pour la donnée (D,ρ). De plus si cette solution existe elle est calculable. □

Démonstration

Soit u solution de P pour (D,ρ). $SD_i[\varepsilon'_S \wedge r \equiv u]$ est une extension consistante de SD_i, cette extension contient $SD_i[\varepsilon'_S]$ qui est complète, elle lui est donc égale par maximalité des extensions complètes dans le treillis des extensions consistantes de SD_i : $SD_i[\varepsilon'_S] = SD_i[\varepsilon'_S \wedge r \equiv u]$.

Ainsi donc r ≡ u est un théorème de $SD_i[\varepsilon'_S]$, u est ainsi calculable et donc unique. cqfd

Théorème 8

Si SD_i est complet et si P admet une solution calculable alors $SD_i[\varepsilon'_S]$ est complet. □

Démonstration

Puisque r est le seul symbole de $SD_i[\varepsilon'_S]$ qui n'est pas dans SD_i, il suffit de montrer que pour tout terme v de SD_i de même profil que r, on a soit $\vdash_{SD_i[\varepsilon'_S]} r \equiv v$ soit $\vdash_{SD_i[\varepsilon'_S]} r \not\equiv v$.

Mais on a $\vdash_{SD_i} u \equiv v$ ou $\vdash_{SD_i} u \not\equiv v$ puisque SD_i est complet.

Comme $\vdash_{SD_i[\varepsilon'_S]} r \equiv u$ on a donc bien le résultat cherché.

cqfd

Ces deux théorèmes peuvent se regrouper sous la forme suivante :

Corollaire

S'il existe une solution $\tilde{u}$ de P pour (D,ρ) et si SD_i est complet alors u est une solution calculable de P si et seulement si $SD_i[\varepsilon'_S]$ est complet.

5.2.2.- Calcul d'une solution

a) Calcul et démonstration

La recherche d'une solution calculable est donc aussi une démonstration. Il faut alors chercher des formes d'énoncés (énoncés explicites) et des "restrictions" de S.I. pour lesquelles les choix de démonstration soient les plus réduits possible. Une "restriction" de S.I. consistant ici à restreindre les règles d'inférence (essentiellement la substitution) en des *règles de calcul* à partir desquelles on peut définir une stratégie de démonstration *indépen-*

dante de la valeur des données et *déterministe*.

"Déterministe" doit être compris avec le même sens que lorsqu'on affirme qu'un programme FORTRAN est déterministe, c'est-à-dire qu'en partant des axiomes il n'y a qu'un (ou qu'un nombre réduit) de chemins conduisant à r ≡ u. En reprenant le vocabulaire précédent, une stratégie de démonstration est un méta-problème dont la donnée est un problème P et le résultat, une solution calculable de P. Pour résoudre cette stratégie on doit indiquer la forme des données (énoncé explicites) et en donner un méta-énoncé explicite formé de règles de calcul.

Nous précisons dans la deuxième partie deux classes particulièrement importantes d'énoncés (et les règles de calculs associées) : les systèmes d'équations (schémas) récursifs et les énoncés récurrents qui peuvent être ou non déterministes.

b) Retour sur la notion de solution

Dans les définitions précédentes nous partions d'une valeur (SD,ρ) des données et nous effectuions une démonstration dans $SD_i[\rho(\varepsilon_s)]$. Ceci correspond à un point de vue *interprétatif*.

Nous aurions également pu partir du S.I. S du problème en ne précisant pas la valeur des données et associer à P *toutes les démonstrations* ou *tous les calculs* possibles : on traduit (compile) P en un *ensemble de calculs*.

Ces deux points de vue correspondent à deux manières de définir la sémantique d'un langage de programmation [LIV,78]:

- sémantique opérationnelle
- sémantique algébrique [COU,78] ou calculatoire [FIN,76]

De plus, il y a au moins deux façons d'envisager le concept de démonstration :

- c'est une suite de théorèmes ou d'axiomes telle que l'on passe de l'un à l'autre en appliquant les règles d'inférence (définition des logiciens)
- c'est une suite de *modifications* (i.e. transformations d'informations, cf. ch. 1.2., § 4.2.2), partant d'un S.I. SD et lui adjoignant successivement des théorèmes de $SD[\varepsilon_s']$ jusqu'à obtenir une information contenant le théorème r ≡ u. Ce deuxième aspect permet de passer à une vue plus "dynamique" de la résolution d'un problème en exprimant qu'une information formalise l'état d'une machine, d'état initial SD, pendant l'exécution d'un programme [FIN,76]. Il permet de rendre compte de la notion de *variable informatique* et des problèmes connexes d'efficacité (partage d'objet, représentation d'une suite par une variable ou par un tableau selon que la définition d'un élément n'utilise que l'élément précédent ou non, etc...).

6.- TRANSFORMATION DE PROBLEMES.

6.1.- Utilité des transformations de problèmes.

Lorsque la forme de l'énoncé ou la forme des axiomes propres du S.I. ne permet pas d'effectuer de calculs pour résoudre un problème, on peut chercher à transformer l'énoncé ou le S.I..

La transformation d'un énoncé implicite en un programme s'appelle habituellement la *synthèse de programme*. La transformation d'un S.I. en un autre repose sur le concept de réalisation (ch. 1.2, § 4), ce peut être soit une *abstraction* soit une *représentation*.

En général le passage d'un énoncé à un programme ne s'effectue pas en une seule étape. Etudier le problème de transformation d'énoncé consiste, comme pour tout (méta-)problème :

- à le spécifier (§ 5.2.1)
- à l'expliciter en définissant des techniques (méta-algorithmes) de transformation. Elles dépendraient en général de la forme des énoncés de départ et d'arrivée et certaines d'entre elles pourront être automatisées : c'est la *synthèse automatique* de programmes [MAN,71],[MAN,75],[BID,79]. On peut aussi, de façon moins irréaliste, rechercher un méta-algorithme d'explicitation fondé sur des règles de transformation (cf. § 6.2.2) et une méthode descendante et progressive (cf. deuxième partie).

De façon analogue, étudier le problème de la transformation d'un S.I. par un autre nécessite :

- de le spécifier (cf.§6.3)
- de l'expliciter, c'est-à-dire de définir des techniques de transformation (cf. ch.2.2).

6.2.- Le problème de la transformation d'énoncés.

6.2.1.- Spécification du problème

De manière intuitive, si $P_2 = (E_2,S)$ est un transformé d'un problème donné $P_1 = (E_1,S)$, on souhaite que toute solution de P_2 soit une solution de P_1. Ainsi pour résoudre P_1 il suffit de résoudre P_2.

Comme, dans la définition d'une solution, nous distinguions l'énoncé strict ε_s des spécifications ε_i des intermédiaires, ces dernières étant ajoutées à la p-réalisation SD, nous nous intéressons ici aux transformations d'énoncés stricts.

la transformation des spécifications ε_i relevant alors des transformations de S.I. .

Définition 8

Soit $P_1 = ((\Sigma, \varepsilon_{s_1} \wedge \varepsilon_i), (S, \mathcal{D}op, \mathcal{D}yp))$

et $P_2 = ((\Sigma, \varepsilon_{s_2} \wedge \varepsilon_i), (S, \mathcal{D}op, \mathcal{D}yp))$

deux problèmes de mêmes S.I. et de mêmes intermédiaires.

On dit que P_2 *vérifie* P_1 si $\varepsilon_{s_2} \supset \varepsilon_{s_1}$ est un théorème de $S[\varepsilon_i]$. □

Théorème 9

Si P_2 vérifie P_1, pour toute donnée (D,ρ), l'ensemble des solutions de P_2 est inclus dans l'ensemble des solutions de P_1. □

Ce théorème résulte immédiatement de la définition d'une solution (définition 3) et du fait que la p-réalisation par (D,ρ) de $(S[\varepsilon_i], \mathcal{D}op, \mathcal{D}yp)$ est $(SD[\rho(\varepsilon_i)], \rho)$.

C'est le point de vue classiquement utilisé dans les preuves de programme [LIV,78] : P_2 est un "programme" et P_1 en est une "spécification". Prouver la correction partielle du programme consiste à prouver que tout résultat de P_2 vérifie P_1.

Exemple 11

S est le S.I. de l'exemple du problème de la fraction irréductible (ch. 1.1 fig.2). ε_{s_2} est l'énoncé strict de la figure 2. ε_{s_1} est l'énoncé strict :

résultat x, y *tq* propor(x,y,a,b)
donnée a,b

Il est immédiat que P_2 vérifie P_1. □

Exemple 12

Si on suppose que le S.I. de l'exemple précédent contient la division et la fonction pgcd correctement axiomatisés, le problème P_3 d'énoncé strict :

$$\varepsilon_{s_3} = \exists z \ tq \ x \equiv a \ / \ z \wedge y \equiv b \ / \ z \wedge z \equiv pgcd(a,b)$$

et n'ayant pas d' intermédiaires, vérifie P_2.

Il est en effet assez simple ici de prouver que $\varepsilon_{s_3} \supset \varepsilon_{s_2}$ dans $S[\varepsilon_i]$. Mais on peut aussi essayer d'utiliser la construction de ε_{s_3} à partir de ε_{s_2} pour trouver une relation entre ces deux énoncés :

Indiquons brièvement les différentes étapes de la transformation de ε_{s_2} en ε_{s_3} :

① $x * b \equiv y * a \supset x \equiv a \ / \ (b \ / \ y)$

$x * b \equiv y * a \supset \exists z \ tq \ x \equiv a \ / \ z \wedge z \equiv b \ / \ y$

$x * b \equiv y * a \supset \exists z \ tq \ x \equiv a \ / \ z \wedge y \equiv b \ / \ z$

② $prem(x,y) = \forall t (t \ div \ x \wedge t \ div \ y \supset t = 1)$

①,② $x * b \equiv y * a \wedge prem(x,y) \supset$

$\exists z \ tq \ (x \equiv a \ / \ z \wedge y \equiv b \ / \ z) \wedge \forall t \ (t \ div \ a/z \wedge t \ div \ b/z \supset t \equiv 1)$

$x * b \equiv y * a \wedge prem(x,y) \supset$

$\exists z \ tq \ (x \equiv a/z \wedge y \equiv b/z) \wedge \forall t (i * z \ div \ a \wedge t * z \ div \ b \supset t \equiv 1)$

$x * b \equiv y * a \wedge prem(x,y) \supset$

$\exists z \ tq \ (x \equiv a/z \wedge y \equiv b/z) \wedge \forall u (u \ div \ a \wedge u \ div \ b \supset u \ div \ t)$

$x * b \equiv y * a \wedge prem(x,y) \supset$

$\exists z \ tq \ x \equiv a/z \wedge y \equiv b/z \wedge z \equiv pgcd(a,b)$

On construit donc ε_{s_3} à partir de ε_{s_2}, mais en même temps on prouve que $\varepsilon_{s_2} \supset \varepsilon_{s_3}$; cette implication suffit-elle à assurer la correction de la transformation ou alors faut-il prouver, en plus, que $\varepsilon_{s_3} \supset \varepsilon_{s_2}$? Précisons cette notion : □

Définition 9

Avec les notations de la définition 8.

On dit que P_2 *vérifie* P_1 pour les solutions calculables, ou encore P_2 *c-vérifie* P_1, si $\varepsilon_{s_1} \supset \varepsilon_{s_2}$ est un théorème de $S[\varepsilon_i]$. □

L'intérêt de cette définition résulte du théorème suivant :

Théorème 8

Si P_1 admet une solution, si P_2 c-vérifie P_1 et si P_2 admet une solution calculable $\tilde{u}$ pour la donnée (D,ρ), alors $\tilde{u}$ est la solution calculable de P_1 pour (D,ρ). □

Démonstration

Par hypothèse, $r \equiv u$ est un théorème de $SD_i[\varepsilon_{s_2}]$. Mais

$\varepsilon'_{s_1} \supset \varepsilon'_{s_2}$ est un théorème de $SD_i[\varepsilon'_{s_1}]$

donc ε'_{s_2} est un théorème de $SD_i[\varepsilon'_{s_1}]$

et donc $SD_i[\varepsilon'_{s_1}]$ est une extension de $SD_i[\varepsilon'_{s_2}]$

d'où $r = u$ est un théorème de $SD_i[\varepsilon'_{s_1}]$. L'existence d'une solution pour P_1 termine la preuve. cqfd

Ainsi, lorsque P_2 c-vérifie P_1, on peut affirmer que l'existence d'une solution calculable $\hat{u}$ pour P_2, entraîne que $\hat{u}$ est la solution calculable de P_1. Il est cependant possible qu'il n'existe pas de solution calculable pour P_2. Ce concept n'est donc pratiquement utilisable que si l'on peut affirmer, par un autre moyen et de manière simple, l'existence d'une solution calculable pour le transformé P_2 de P_1.

Nous pouvons maintenant donner une première idée de ce que peut être la spécification du méta-problème des transformations (figure 3). La s-description qui est proposée est évidemment incomplète : en compléter l'énoncé reviendrait essentiellement à introduire des symboles correspondants aux différentes définitions de ce chapitre. Mais il faudrait également en spécifier l'univers appelé ici PROBLEME, par exemple, en se restreignant aux problèmes définis par des s-descriptions. La spécification de cet univers est faite en partie au chapitre 1.4 à l'occasion de la description du méta-algorithme de spécification.

Mais ce que l'on peut surtout reprocher à cette s-description est l'absence de précision sur la nature du résultat par rapport à la donnée : il faut exprimer qu'on souhaite, en un certain sens, *améliorer* la donnée. Ainsi avant de chercher un méta-algorithme de transformation, il est utile d'étudier la forme des problèmes recherchés, c'est-à-dire la forme des énoncés explicites, c'est ce qui est entrepris dans la deuxième partie.

Il faut ensuite expliciter cette spécification. On peut mettre en évidence un certain nombre de règles de transformation qui peuvent être considérées comme des méta-procédures, solutions du problème de la figure 2. Mais ces règles sont insuffisantes pour expliciter un problème et il faut en général développer des *stratégies d'explicitations* (utilisant en général certaines règles de transformation) qui peuvent être considérées comme la restriction du méta-problème de l'explicitation à certaines formes de problèmes. Nous reportons également à la deuxième partie une étude plus précise de ce problème de l'explicitation et nous nous contentons maintenant de décrire sommairement et intuitivement les principales idées à la base des techniques générales de transformation d'énoncés.

P_2 est le problème résultant de la transformation. L'univers et les intermédiaires ne sont pas changés. vérifie est le prédicat associé à la définition 8. c-vérifie est le prédicat associé à la définition 9. calcul est le prédicat exprimant qu'un problème admet une solution calculable, il est associé à la définition 7. exsol est le prédicat exprimant qu'un problème admet une solution. Il est associé à la définition 3. valdon est la fonction associant à un problème l'ensemble de toutes ses données. Il est associé à la définition 2. On ne l'explicite pas ici.	P_2 : PROBLEME P_1 : PROBLEME vérifie : *op*(PROBLEME,PROBLEME) BOOL c-vérifie : *op*(PROBLEME,PROBLEME) BOOL calcul: *fonct*(PROBLEME) BOOL exsol : *fonct*(PROBLEME) BOOL	*résultat* P_2 *tq* [P_2 vérifie P_1 ∨ (P_2 c-vérifie P_1∧calcul(P2)∧exsol(P_1)) ∧ $P_2 = (\langle\Sigma, \varepsilon_{s_2} \wedge \varepsilon_i\rangle, (S, \mathcal{D}op, \mathcal{D}yp))$ *donnée* P_1 *tq* $P_1 = ((\Sigma, \varepsilon_{s_1} \wedge \varepsilon_i), (S, \mathcal{D}op, \mathcal{D}yp))$ vérifie *tq* P_2 vérifie $P_1 = (\vdash_{S[\varepsilon_i]} \varepsilon_{s_2} \supset \varepsilon_{s_1})$ c-vérifie *tq* P_2 c-vérifie $P_1 = (\vdash_{S[\varepsilon_i]} \varepsilon_{s_1} \supset \varepsilon_{s_2})$ calcul *tq* calcul(r) = ∀(D,o) ∈ valdon(P) (∃ $\hat{v}$ *tq* $\vdash_{SD[\varepsilon_i]} \varepsilon'_s[v/r]$) ∧ (∃ $\hat{u}$ *tq* $\vdash_{SD[\varepsilon_i \wedge \varepsilon_s]} r=u$) exsol *tq* exsol(P) = ∀(D,o) ∈ valdon(P) ∃ $\hat{v}$ *tq* $\vdash_{SD[\varepsilon_i]} \varepsilon'_s[v/r]$

figure 2 : spécification du problème des transformations

6.2.2.- Aperçu sur les techniques de transformation d'énoncés

a) Synthèse fondée sur les preuves de théorèmes

L'objectif de la *synthèse d'énoncés* est la construction systématique, voire automatique, d'un énoncé explicite vérifiant un énoncé donné.

Dans ce but, plutôt que de chercher un terme clos u tel que $\varepsilon'_S[u/r]$ soit un théorème de SD_i (cf. §2), on cherche à démontrer $\exists r\ \varepsilon'_S$ (r étant ici considéré comme une variable). En fait les principaux travaux considèrent le cas où la donnée d est 0-aire, elle est alors considérée comme une variable et on cherche à démontrer le théorème $\forall d\ \exists r\ \varepsilon$ dans le S.I. du problème. C'est le cas en particulier lorsque ε décrit la relation qui existe entre la donnée d d'une fonction et le résultat r. On espère qu'au cours de la démonstration on fera apparaître une définition explicite de la fonction f telle que r = f(d).

En fait il est souvent impossible de démontrer directement $\forall d\ \exists r\ \varepsilon$ et on doit faire des choix de *stratégies de démonstration* qui sont donc des choix de *stratégies d'explicitation*. L'une des stratégies les plus utilisées est fondée sur l'*induction*, c'est-à-dire sur la règle d'inférence.

$$\forall d\ (\ \forall d'\ (d' < d \supset \exists r'\ \varepsilon[r'/r,d'/d]) \supset \exists r\ \varepsilon) \vdash \forall d\ \exists r\ \varepsilon$$

où < est l'ordre associé à l'induction.

Stratégie d'induction

Pour prouver $\forall d\ \exists r\ \varepsilon$, chercher à prouver :

$$\forall d\ \exists d'\ \forall r'\ \exists r\ (d' < d \supset \varepsilon\ [r'/r,d'/d]) \supset \varepsilon$$

Au cours de la preuve, chercher à faire apparaître deux fonctions h et g telles que

$$r \equiv h(r',d) \wedge d' \equiv g(d) \vdash\!\!\!\!-_S\ (d' < d \supset \varepsilon[r'/r,d'/d]) \supset \varepsilon$$

Les preuves de théorèmes que l'on met ainsi en oeuvre utilisent les techniques développées pour la démonstration automatique, notamment l'unification et l'induction.

Remarque

Dans la définition du concept de S.I. nous n'avons pas inclus la règle d'inférence "induction". Dans ce cadre la stratégie d'induction repose alors sur le méta-raisonnement suivant :

Si, pour tout d et d', $(d' < d \supset \exists r'\ \varepsilon[r'/r,d'/d]) \supset \exists r\ \varepsilon$ est un théorème de S, alors pour tout d, $\exists r\ \varepsilon$ est un théorème de S.

Ceci est suffisant pour une application "à la main" de la stratégie d'induction puisqu'on cherche r pour une valeur quelconque de d, mais insuffisant si l'on souhaite construire un système automatique puisque dans ce cas on peut souhaiter prouver $\forall d\ \exists r\ \varepsilon$. Ainsi BURSTALL et GOGUEN introduisent une opération "induce" qui munit un type abstrait de la règle d'induction [BUR,77a], de même

les types du langage IOATA [NAK,79] possèdent cette règle d'inférence.

Nous étudierons de manière plus précise les stratégies d'explicitation dans la deuxième partie, nous ne faisons ici qu'illustrer ces idées sur un exemple.

Exemple 13

On suppose que ENTIER est muni de la règle d'induction.

Pour synthétiser le prédicat a div b = ($\exists k \in \mathbb{N}$, $a \equiv b * k$) on cherche à démontrer le théorème $\forall a, b \in \text{ENTIER} \times \text{ENTIER}^+\ \exists r \in \text{BOOL}$ *tq* $r \equiv (\exists k \in \text{ENTIER}, a \equiv b * k)$. Il est naturel de chercher une preuve par induction ; pour éviter une induction sur ENTIER × ENTIER$^+$ nous supposons que b est un entier positif fixé, l'induction ne porte alors que sur a.

Posons $\varepsilon = r \equiv (\exists k\ \ a \equiv b * k)$

On cherche donc à prouver (stratégie d'induction) :

$\forall a\ \exists a'\ \forall r'\ \exists r\ \ (a' < a \supset \varepsilon[r'/r,a'/a]) \supset \varepsilon$

Soit (en utilisant la définition de ε : dépliage de ε) :

$\forall a\ \exists a'\ \forall r'\ \exists r\ \forall k'\ \exists k\ \ (a' < a \supset r' \equiv (a' \equiv b * k')) \supset r \equiv (a \equiv b * k)$

Pour continuer à déplier cette formule il faut utiliser l'axiome caractérisant "*" :

$u * v \equiv$ *si* $v \equiv 0$ *alors* 0 *sinon* $u * (v-1) + u$

Donc en instantiant cet axiome :

$b * k \equiv$ *si* $k \equiv 0$ *alors* 0 *sinon* $b * (k-1) + b$

En dépliant la définition de r :

$r \equiv$ *si* $k \equiv 0$ *alors* $a \equiv 0$ *sinon* $a \equiv b * (k-1) + b$

En utilisant l'axiome de la conditionnelle, les propriétés de ENTIER$^+$ et de +

$(k \equiv 0 \supset r \equiv (a \equiv 0)) \wedge (k > 0 \supset r \equiv (a-b \equiv b * (k-1)))$

Analyse par cas :

i) $k > 0$ on cherche à unifier certains sous-termes de la définition de r avec la définition de r', ce qui conduit à $a - b < a \supset r' \equiv (a-b \equiv b * (k-1))$ en substituant a-b à a' et k-1 à k'.

La condition a-b < a est toujours vérifiée puisque b > 0

ainsi $k > 0 \supset r \equiv r'$ (1)

ii) $k \equiv 0$ alors $r \equiv (a \equiv 0)$ (2)

On cherche à exprimer k en fonction de a et b :

$r' \equiv (a-b \equiv b * (k-1)) \equiv$ *vrai* par hypothèse

or b > 0, ainsi $a-b \geq 0$ si et seulement si $k-1 \geq 0$

donc $k > 0$ équivaut à $b \leq a$ (3)

Finalement, avec (1), (2) et (3) :

$$\begin{cases} r \equiv \underline{si}\ a < b\ \underline{alors}\ a \equiv 0\ \underline{sinon}\ r' & (=h(r',d)) \\ a'' \equiv \underline{si}\ a < b\ \underline{alors}\ a\ \underline{sinon}\ a-b & (=g(d)) \end{cases}$$

Soit a div b ≡ *si* a < b *alors* a ≡ 0 *sinon* a-b div b □

MANNA et WALDINGER [WAL,69],[MAN,71],[MAN,75] ont été parmi les premiers auteurs à explorer la voie de la synthèse automatique permettant de transformer un énoncé exprimé dans calcul des prédicats en un algorithme itératif ou récursif. Plutôt qu'un énoncé explicite, certaines recherches visent à obtenir un énonc sous forme clausale [HOG,78]. Dans l'un et l'autre cas les mêmes techniques ont été utilisées pour obtenir un énoncé plus efficace qu'un énoncé donné (suppression de la récursivité, diminution de l'indéterminisme dans un énoncé clausal).

Bien que l'idée de synthétiser automatiquement un programme soit séduisante il faut bien constater que la pauvreté des techniques de démonstration automatique conduisent à des transformations très peu efficaces (ainsi la synthèse du programme pgcd par la technique de MANNA et WALDINGER coûte, paraît-il, une nuit de calculs).

b) Règles de transformation d'énoncés

Les premières transformations de programmes qui ont été introduites ont eu pour rôle d'améliorer l'efficacité d'un programme, par exemple en supprimant la récursivité [BUR,77b],[ARS,79]. Mais de plus en plus on cherche à définir des règles de transformation générales qui doivent permettre de passer progressivement d'un énoncé implicite à un algorithme efficace : des projets tels que CIP [BAU,78],[BUR,77b],[ARS,78] ont pour ambition de fournir au programmeur un jeu de transformations élémentaires dont l'emploi répété fournira un programme efficace. Brièvement ceci revient à introduire des règles d'inférence supplémentaires dans l'univers du problème et des conditions d'utilisation de ces règles.

Ces règles de transformation sont des outils précieux pour la construction d'un algorithme à partir d'un énoncé :

- on peut les utiliser "manuellement" dans le cadre d'une méthode de construction de programme [WEG,75] en introduisant ainsi des *règles de construction* (qui étaient jusqu'à présent plus couramment appelées *règles de preuves*).
- on peut aussi les utiliser dans un système de synthèse automatique (cf. b)). Dans ce cas on leur associe des heuristiques d'utilisation. C'est ainsi que les règles de pliage (folding), dépliage (unfolding), généralisation (embeding) sont utilisées dans des systèmes automatiques [BUR,77b].

Ainsi, les règles de transformation apparaissent comme des règles de calcul (ou des opérations) dans le méta-système d'information dans lequel se fait l'explicitation ou la transformation de l'énoncé. Elles n'ont donc d'intérêt que dans la mesure où elles permettent de définir des stratégies d'explicitation qui apparaissent ainsi comme des énoncés sur ce méta-système. Il n'est pas raisonnable de penser expliciter ces diverses stratégies mais il faut définir à la fois des heuristiques en facilitant la mise en oeuvre et des conditions d'utilisation. Nous essayons, dans la deuxième partie, de définir de telles stratégies.

Ce point de vue sur l'explicitation est un premier pas vers l'écriture d'un système interactif [MAN,78] libérant le programmeur de certains calculs correspondants à l'application de certaines règles de construction mais lui réservant les choix inhérents à la mise en oeuvre de stratégies. Nous reviendrons sur ce point au chapitre 2.4.

6.3.- Le problème de la transformation de systèmes d'information.

6.3.1.- Nécessité des transformations de systèmes d'information

Le seul fait de savoir expliciter des énoncés stricts ne permet pas de transformer un problème en un autre problème résoluble (i.e. pour lequel on sait trouver une solution), encore faut-il que l'on puisse exprimer les axiomes du S.I. sous-jacent sous une forme permettant la résolution. Nous retrouvons tout naturellement la symétrie entre énoncé et S.I. et la nécessité, parallèlement à l'explicitation d'énoncé, d'expliciter le S.I.

Exemple 14 . Encore la fraction irréductible

Dans l'exemple 12 nous avions supposé que l'opération pgcd correctement axiomatisée faisait partie du S.I. S du problème. Si on ne le suppose pas, c'est-à-dire si on pose S = ENTIER, il faut compléter l'énoncé du problème P_3 de la façon suivante :

ε_{s_3} = ∃z *tq* x ≡ a/z ∧ y ≡ b/z ∧ z ≡ pgcd(a,b) (énoncé strict)

ε_{i_3} = pgcd(a,b) ≡ p *tq* p div a ∧ p div b ∧ ∀u (u div a ∧ u div b ⊃ u div p) (énoncé intermédiaire)

Chercher directement une solution de P_3 pour une donnée (D,ρ) nécessite de se placer dans l'extension $SD[\varepsilon_{i_3} \wedge \varepsilon_{s_3}]$ dans laquelle la spécification de pgcd n'est pas explicite. Pour appliquer une technique de résolution portant sur les définitions explicites, il faut donc expliciter pgcd donc changer de S.I. : ce n'est plus dans $S[\varepsilon_{i_3}]$ que l'on résoudra le problème mais dans un autre

système $S[\varepsilon_{i_4}]$. Se posent alors les questions suivantes :

- sous quelle forme chercher ε_{i_4} ?
- quand peut-on "remplacer" le problème P_3 par un nouveau problème P_4 ?
- comment construire ε_{i_4} ?

Ici encore on va chercher les axiomes caractérisant les intermédiaires sous forme de définitions explicites. Il faut alors que ce changement d'axiomes ne change "pas trop" les solutions du problème et, à tout le moins, n'en ajoute pas de nouvelles. Les techniques de construction quant à elles vont être très voisines.

Notons que les mêmes questions se poseraient si on supposait que ε_{i_3} est un axiome de S : les intermédiaires d'un problème P et les opérations du S.I. S de P ont le même rôle vis-à-vis de la résolution. Les réponses aux questions précédentes seront donc les mêmes.

La mise en oeuvre d'une stratégie d'induction analogue à celle développée dans l'exemple 13 pour l'explicitation du prédicat div conduit à :

ε_{i_4} = pgcd(a,b) ≡ *si* b ≡ 0 *alors* a *sinon* pgcd(b,a-b * (a÷b))

La construction de cette formule permet en même temps de montrer que c'est un théorème du S.I. $\text{ENTIER}[\varepsilon_{i_3}]$, (ici S = ENTIER). En d'autres termes ($\text{ENTIER}[\varepsilon_{i_3}]$ id) , où id est l'application identique, est une réalisation de $\text{ENTIER}[\varepsilon_{i_4}]$. Il semble clair alors que si $\hat{u}$ est une solution de $P_4 = (\{\emptyset, ix, y, pgcd\}, \varepsilon_{i_3} \wedge \varepsilon_{i_4})$, (ENTIER,∅,{a,b}) pour la donnée (D,ρ) , c'est encore une solution de P_3 pour (D,ρ). □

La résolution de problèmes nécessite donc

i) de découvrir une forme des axiomes propres des S.I. à partir de laquelle pourront être développées des techniques de résolutions. Ici encore on s'intéressera aux axiomes explicites (définitions récursives ou récurrentes). Nous étudierons au chapitre 2.2 les S.I. algébriques caractérisés par le fait que leurs axiomes sont explicites.

ii) de caractériser les transformations de S.I. préservant en un certain sens les solutions d'un problème. Le concept de réalisation en donne une première idée. Deux types de transformations sont intéressants :

- les *abstractions* qui, très intuitivement, consistent à remplacer un axiome caractérisant une opération par une définition constructive (en général récursive de cette opération. Elles sont essentiellement utilisées lors du passage d'une première spécification d'un univers à une version constructive de cet univers (cf. ch. 2.2).

- Les *représentations* qui consistent à remplacer certaines opérations par la composition d'autres plus élémentaires ou plus propices à la résolution. Elles sont surtout utilisées pour transformer un problème déjà explicite dans un but d'efficacité.

iii) De proposer des méthodes de transformation (i.e. de résoudre le méta-problème de la transformation de S.I.).

Nous repoussons, comme nous l'avons déjà fait au paragraphe précédent pour les énoncés stricts, l'étude des S.I. "explicites" (i.e. algébriques) ainsi que celle de méthodes de transformations de S.I. à la deuxième partie en nous limitant ici à une étude plus fondamentale des concepts d'abstraction et de représentation.

6.3.2.-Abstractions et représentations simples

Elles reposent sur le concept de réalisation.

a) Définitions

Définition 9

Soit S un S.I.

a) Une représentation simple de S par un S.I. S' est une application μ telle que (S',μ) soit une réalisation de S. (S',μ) s'appelle une représentation de S.

b) Une abstraction simple de S par un S.I. A est une application α telle que (S,α) soit une réalisation de A. (A,α) s'appelle une abstraction de S.

Exemple 15

Reprenons le S.I. FILE décrit par la figure 1 du chapitre 1.2.

Soit AFILE le S.I. obtenu en ajoutant à FILE le symbole 0-aire fvide de profil FILE, le symbole unaire long de profil FILE → ENTIER et en remplaçant les différents axiomes caractérisant les opérations par :

- retf(ajtf(f,x)) ≡ *si* vide(f) *alors* fvide *sinon* ajtf(retf(f),x)
- têtef(ajtf(f,x)) ≡ *si* vide(f) *alors* x *sinon* têtef(f)
- videf(fvide) ≡ *vrai*
- videf(ajtf(f,x)) ≡ *faux*
- long(fvide) ≡ 0
- long(ajtf(f,x)) ≡ long(f) + 1
- l'invariant devient : 0 ≤ long(f) ≤ 100

Alors (AFILE,α) est une abstraction simple de FILE si

α(long) = | | , α(fvide) = < >

et α est l'identité ailleurs.

Il suffit de prouver que l'image par α de chaque nouvel axiome est un théorème de FILE. Montrons-le par exemple pour le premier :

① f1 = retf(ajtf(f,x)) = retf(f ∿ x) = reste(f ∿ x)

avec les axiomes sur reste et ∿ (ch. 1.2 § 7.2,2 figure 9) :

bi(f1) ≡ bi(f)+1 ; bs(f1) ≡ bs(f)+1 ;

∀i ∈ [bi(f1)...bs(f)] $f1_i \equiv f_i$; appl(f1,bs(f)+1) ≡ x

cas 1 : vide(f) ≡ *vrai*

alors bs(f1) < bi(f1) donc f1 ≡ < > ≡ α (fvide)

cas 2 : vide(f) ≡ *faux*

alors soit f2 = ajtf(retf(f),x) = retf(f) ∿ x

bi(f2) ≡ bi(f) + 1 ; bs(f2) ≡ bs(f) + 1

∀i ∈ [bi(f2)...bs(f)] $f2_i \equiv f_i$; appl(f2,bs(f) + 1) ≡ x

donc f2 = f1

cqfd

De manière duale, (FILE,α) est une représentation de AFILE. □

Remarque

Il arrive fréquemment que l'application α d'une abstraction (A,α) d'un S.I. S soit l'identité, c'est-à-dire que S est une extension de A. Plus précisément dans le cas d'un S.I. défini par une s-description c'est l'identité sur les opérations introduites par la s-description. Il est parfois nécessaire d'introduire des fonctions supplémentaires permettant d'éviter toute référence à des systèmes support (appelées fonctions auxiliaires ou fonctions cachées). C'est le cas de long et fvide dans l'exemple ci-dessus. Nous reviendrons sur ce point au chapitre 2.2.

Puisque par définition tout théorème d'une abstraction (A,α) de S est un théorème de S il semble clair qu'une solution d'un problème sur A soit solution du problème "correspondant" sur S. Pour préciser cette idée il faut prendre en compte la notion de donnée ; donc définir l'abstraction et la représentation d'un S.I. paramétré.

Définition 10. Soit (S, $\mathfrak{Sop}$, $\mathfrak{Syp}$) un S.I. paramétré

a) Une représentation simple de (S, $\mathfrak{Sop}$, $\mathfrak{Syp}$) par un S.I. paramétré de mêmes paramètres (S', $\mathfrak{Sop}$, $\mathfrak{Syp}$) est une application μ qui est l'identité sur $\mathfrak{Sop}$, $\mathfrak{Syp}$ telle que (S',μ) soit une réalisation de S.

b) Une abstraction simple de (S, $\mathfrak{Sop}$, $\mathfrak{Syp}$) par un S.I. paramétré de mêmes paramètres (A, $\mathfrak{Sop}$, $\mathfrak{Syp}$) est une représentation simple α de (A, $\mathfrak{Sop}$, $\mathfrak{Syp}$) par (S, $\mathfrak{Sop}$, $\mathfrak{Syp}$). □

b) Abstractions, représentations et solutions des problèmes

Définition 11.

Soit P = ((Σ,ε_s,ε_i),(S,$\mathfrak{Dop}$,$\mathfrak{Dyp}$)) un problème sur S. Soit ((S',$\mathfrak{Dop}$,$\mathfrak{Dyp}$),μ) une représentation simple de (S,$\mathfrak{Sop}$,$\mathfrak{Syp}$). On appelle représentation du problème P par μ le problème μ(P) = ((Σ,μ(ε_s),μ(ε_i)),(S',$\mathfrak{Dop}$,$\mathfrak{Dyp}$)). □

Théorème 10. En reprenant les notations ci-dessus.

Soit P un problème sur S et (S',μ) une représentation de S. Si (D,ρ) est une donnée commune à P et à μ(P), toute solution $\hat{\mathcal{V}}$ de P pour (D,ρ) a pour image par μ une solution de μ(P) pour (D,ρ). □

On suppose ici que μ se prolonge en l'identité sur le résultat et les intermédiaires de P.

Démonstration

Par hypothèse $\rho(\varepsilon_s)[u/r]$ est un théorème de $SD[\rho(\varepsilon_i)]$. Mais on a le lemme suivant :

lemme : Si (S',μ) est une représentation simple de S, φ une formule de S alors (S'[μ(φ)],μ) est une représentation simple de S[φ]. □

Ainsi $\mu(\rho(\varepsilon_s))[\mu(u)/r]$ est un théorème de $S'D[\mu(\rho(\varepsilon_i))]$. La commutativité du diagramme :

$$\begin{array}{ccc} S & \xrightarrow{\rho} & S \cup D \\ \downarrow{\scriptstyle \mu} & & \downarrow{\scriptstyle \mu} \\ S' & \xrightarrow{\rho} & S' \cup D \end{array}$$

(μ est prolongé en l'identité sur D)

fournit le résultat. cqfd

Remarque 1

Très souvent μ est l'identité sur les termes dont le profil est le domaine du résultat. C'est en particulier le cas lorsque S et S' sont deux extensions d'un même S.I. R, contenant le domaine des résultats, et invariant par μ.

Remarque 2

Dans le théorème précédent nous nous sommes placés dans le cas où (D,ρ) est une donnée commune aux deux problèmes. Mais il faut généraliser ce résultat au cas où les données sont transformées par une représentation. Pour ce faire nous commençons par énoncer un théorème dans le cas où seules les données sont représentées.

Théorème 11

Soit (D,ρ) une donnée d'un problème P sur un S.I. S et (D',σ) une représentation simple de D. Alors $(D',\sigma\circ\rho)$ est une donnée de P et, pour toute solution $\hat{u}$ de P pour (D,ρ), l'image par μ de $\hat{u}$ est une solution de P pour $(D',\sigma\circ\rho)$. □

On suppose ici que σ se prolonge en l'identité sur S.

Démonstration

$(D',\sigma\circ\rho)$ est une donnée car la composée de deux réalisations est une réalisation.

$\varepsilon'_s[u/r]$ est un théorème de $SD[\varepsilon'_i]$ donc $\sigma(\varepsilon'_s)[\sigma(u)/r]$ est un théorème de $SD'[\sigma(\varepsilon'_i)]$.

cqfd

Corollaire des théorèmes 10 et 11

Soit P un problème sur un S.I. S et (D,ρ) une donnée de P. Soit d'autre part (S',μ) une représentation simple de S et (D',σ) une représentation simple de D. Alors $(D',\sigma\circ\rho)$ est une donnée de $\mu(P)$ et pour toute solution $\hat{u}$ de P pour (D,ρ), l'image de $\hat{u}$ par $\sigma\circ\mu$ est une solution de $\mu(P)$ pour $(D',\sigma\circ\rho)$. □

Théorème 10'

Soit $P_a = ((\Sigma,\varepsilon_s,\varepsilon_i),(A,\mathcal{D}op,\mathcal{D}yp))$ un problème sur l'abstraction simple $((A,\mathcal{D}op,\mathcal{D}yp),\alpha)$ du S.I. paramétré $(S,\mathcal{D}op,\mathcal{D}yp)$. Notons P le problème qui est l'image $((\Sigma,\alpha(\varepsilon_s),\alpha(\varepsilon_i)),(S,\mathcal{D}op,\mathcal{D}yp))$ de P_a par α. Soit enfin (D,ρ) une donnée commune à P_a et P.

Alors toute solution $\hat{u}$ de P_a pour (D,ρ) a pour image par α une solution de P pour (D,ρ). □

C'est le théorème 10 exprimé en termes d'abstraction. Dans la suite nous nous contentons d'énoncer les théorèmes portant sur les abstractions, ceux concernant les représentations s'en déduisent immédiatement.

Théorème 12

Avec les notations du théorème 10', si $\hat{u}$ est une solution calculable de P_a, l'image par α de $\hat{u}$ est une solution calculable de P. □

Démonstration

Par hypothèse, $r \equiv u$ est un théorème de $AD[\rho(\varepsilon_i) \wedge \rho(\varepsilon_s)]$

alors $\alpha(r \equiv u)$ est un théorème de $SD[\alpha(\rho(\varepsilon_i)) \wedge \alpha(\rho(\varepsilon_s))]$

donc $r \equiv \alpha(u)$ est un théorème de $SD[\rho(\alpha(\varepsilon_i)) \wedge \rho(\alpha(\varepsilon_s))]$

cqfd

Les théorèmes précédents, s'ils sont *nécessaires* pour caractériser une "bonne" abstraction ne sont certainement pas *suffisants* : ils n'assurent aucunement l'existence de solution au niveau de l'abstraction, même s'il en existe au niveau "inférieur". En d'autres termes, on souhaite caractériser une "bonne" abstraction par le fait qu'*elle ne supprime pas de solution*. Nous cherchons ainsi des conditions sous lesquelles les "réciproques" des théorèmes précédents seraient vérifiées. C'est le but des trois sous-paragraphes suivants.

c) Abstraction complète

Une manière brutale de ne pas supprimer de solution est d'imposer que α soit une "bijection pour les théorèmes", soit de façon plus précise :

Définition 12

Une abstraction simple $((A,\mathcal{S}op,\mathcal{S}yp,\alpha)$ de $(S,\mathcal{S}op,\mathcal{S}yp)$ est *complète* si la réalisation (S,α) de A est conservative.

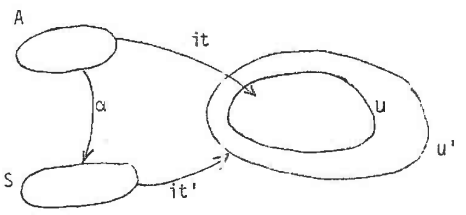
Autrement dit, une abstraction est complète si toute formule φ de A telle que $\alpha(\varphi)$ est un théorème de S, est elle-même un théorème de A.

Théorème 13

Soit (A,α) une abstraction simple de S. C'est une abstraction simple complète si et seulement si tout modèle de A peut être étendu, modulo α, en un modèle de S. □

Le deuxième membre de l'équivalence signifie que, si $(\mathcal{U},it)$ est un modèle de A, il existe un modèle $(\mathcal{U}',it')$ de S tel que

$it = it' \circ \alpha$



La démonstration de ce théorème est la même que celle du théorème similaire sur les extensions (théorème 14, ch. 1.2) ; il suffit d'y prendre en compte α ; nous la développons dans un cas un peu plus général (théorème 19), au paragraphe 6.3.3.

d) Abstraction complète pour un ensemble de formules

Exiger qu'une abstraction soit complète est souvent trop contraignant. Ce qui nous intéresse en fait c'est que si $\mathcal{U}$ est une solution d'un problème alors c'est encore une solution de "l'abstraction" de ce problème. On ne s'intéresse donc qu'à la "classe des formules associées à l'énoncé d'un problème". De façon plus précise, si $P = ((\Sigma,\varepsilon_s,\varepsilon_f),(S,\mathcal{D}op,\mathcal{D}yp))$, la classe des formules associée à P est formée des formules $\varepsilon_s[u/r]$. On souhaite que tout théorème de cette forme soit conservé par abstraction. D'où les définitions suivantes :

Définition 13

Une abstraction simple (A,α) de S est *complète pour un ensemble F* de formules de A si, toute formule φ de F telle que $\alpha(\varphi)$ est un théorème de S est elle-même un théorème de A. □

On a une définition analogue dans le cas des S.I. paramétrés.

Il est intéressant de comparer cette définition au fait que la restriction de A à F est complète.

Définition 14

Un S.I. A est *complet pour un ensemble* F de ses formules si, pour toute formule φ de F qui n'est pas un théorème de A on a $\neg\varphi$ qui est un théorème de A. □

Le théorème suivant relie la complétude de (A,α) pour F à celle de A pour F.

Théorème 14

Soit (A,α) une abstraction simple de S. Si S est consistant et A complet pour un ensemble F de ses formules alors (A,α) est une abstraction simple complète de S pour F. □

Démonstration

Soit φ une formule de F. Si $\alpha(\varphi)$ est un théorème de S, $\neg\alpha(\varphi)$ n'est pas un théorème de S (hypothèse de consistance), donc $\alpha(\neg\varphi)$ n'est pas un théorème de S, ainsi $\neg\varphi$ n'est pas un théorème de A, d'où φ est un théorème de A. cqfd

La plupart du temps, le résultat d'un problème a un profil qui est d'une sorte externe du S.I. du problème. Ceci correspond intuitivement au fait que les S.I. intermédiaires sont des "auxiliaires techniques" alors que le S.I. associé au résultat est fondamental (par exemple dans un problème de compilation, le choix de la définition du S.I. table des symboles ne doit avoir aucune influence sur les solutions).

Il est donc intéressant de chercher à caractériser des abstractions d'un S.I. S qui "préservent" un S.I. commun. Nous nous plaçons donc maintenant dans le cas où A et S sont deux extensions d'un même S.I. R complet (qui intuitivement contient le profil du résultat d'un problème). Il est alors intéressant de s'intéresser aux ensembles de formules formées d'égalités dont le profil est R (puisque l'énoncé du problème est formé de formules atomiques de cette forme).

Théorème 15

Soit (A,α) une abstraction simple de S. On suppose que S et A sont deux extensions d'un S.I. R complet et que S est consistant. Soit F un ensemble de formules de A de la forme $u \equiv v$ où u et v ont pour profil une sorte de R.

Alors si A est une extension suffisammert complète de R, (A,α) est une abstraction simple de S complète pour F. □

Démonstration

Lemme

Soit A une extension consistante du S.I. R complet et F comme dans le théorème. Si A est une extension suffisammert complète de R alors A est complète pour F. □

Démonstration

Soit $u \equiv v$ formule de F qui n'est pas un théorème de A. Puisque A est une extension suffisamment complète de R, il existe u_0 et v_0 termes de R tels que $\vdash_A u \equiv u_0$ et $\vdash_A v \equiv v_0$. Alors $u_0 \equiv v_0$ n'est pas un théorème de A, ce n'est donc pas un théorème de R. Par hypothèse de complétude de R, $u_0 \not\equiv v_0$ est un théorème de R, donc de A, donc $\neg u \equiv v$ est un théorème de A.
cqfd

La démonstration du théorème 15 résulte immédiatement de ce lemme, du fait que si S est consistante A l'est aussi, et du théorème 14. cqfd

Ce théorème se généralise immédiatement aux formules non atomiques.

Théorème 15'

Soit (A,α) une abstraction simple de S. On suppose que S et A sont deux extensions d'un S.I. R complet et que S est consistant. Soit F un ensemble de formules de A dont les constituants atomiques ont pour profil une sorte de R. Alors (A,α) est une abstraction simple de S complète pour F. □

Démonstration

Soit φ une formule de F telle que $\vdash_S \alpha(\varphi)$. Si u, v ... sont les termes des formules atomiques de φ, il existe u_0, v_0 ... termes de R tels que $\vdash_A u \equiv u_0$, $\vdash_A v \equiv v_0$ ... . Alors $\alpha(\varphi[u_0/u,v_0/v\ldots]) = \varphi[\alpha(u_0)(\alpha(u),\alpha(v_0)/\alpha(v)\ldots] = \varphi[u_0/u,v_0/v\ldots]$.

Ainsi $\vdash_S \alpha(\varphi) \Leftrightarrow \vdash_S \varphi[u_0/u, v_0/v \ldots] \Leftrightarrow \vdash_R \varphi[u_0/u, v_0/v \ldots] \Leftrightarrow$

cqfd $\vdash_A \varphi[u_0/u, v_0/v \ldots] \Leftrightarrow \vdash_A \varphi$ par hypothèse sur u_0, v_0 ... .

Remarque

Ces deux théorèmes se généralisent immédiatement au cas où R n'est plus complet mais où S est une extension conservative de R.

En effet si $\alpha(u) \equiv \alpha(v)$ est un théorème de S, $\vdash_A u \equiv u_0$ et $\vdash_A v \equiv v_0$ implique $\vdash_S \alpha(u) \equiv u_0$ et $\vdash_S \alpha(v) \equiv v_0$. D'où $u_0 \equiv v_0$ théorème de S, donc de R, donc de A, d'où $u \equiv v$ est un théorème de A. □

Ces résultats seront étendus au chapitre 2.2 dans le cas de spécifications algébriques.

e) Abstractions complètes et solutions de problèmes

Nous pouvons maintenant préciser les "réciproques" des théorèmes 10' et 12 ; ce qui, intuitivement, caractérise les "bonnes" abstractions.

Théorème 16

Avec les notations du théorème 10'. On suppose que (A,α) est une abstraction complète de S. Si $\alpha(\tilde{u})$ est solution de P pour la donnée (D,ρ) commune à P_a et P, $\tilde{u}$ est solution de P_a pour cette donnée. □

Autrement dit, pour chaque donnée commune, les ensembles de solutions de P_a et P coïncident.

Démonstration

Par hypothèse $\rho(\alpha(\varepsilon_s))[\alpha(u)/r]$ est un théorème de $SD[\rho(\alpha(\varepsilon_i))]$ donc par commutativité de ρ et α et par définition d'une réalisation conservative

cqfd $\rho(\varepsilon_s)[u/r]$ est un théorème de $AD[\varepsilon_i]$.

Ce théorème se généralise au cas où (A,α) est complète pour la classe des formules associée au problème P.

Théorème 17

Toujours avec les notations du théorème 10'. Supposons que (A,α) soit une abstraction simple de S complète pour $F = \{\varepsilon_s[u/r]$ $\exists q$ u terme clos de A de même profil que $r\}$.

Si $\alpha(\tilde{u})$ est solution de P pour la donnée (D,ρ) commune à P_a et P, $\tilde{u}$ est solution de P_a pour cette donnée. □

La démonstration est analogue à celle du théorème 16, en utilisant la définition 13 d'une extension conservative.

Ce théorème est d'autant plus intéressant en pratique que l'on dispose de critères assurant la complétude par rapport à un ensemble de formules. Par exemple de la remarque suivant le théorème 15' et du théorème 17 on tire :

Corollaire

Soit (A,α) une abstraction simple de S, P_a un problème sur A dont la représentation par α est P. Si S est une extension conservative de R (consistant), si A est une extension suffisamment complète de R et si les formules atomiques de l'énoncé strict ε_s de P_a ont leur profil dans R alors si $\alpha(\tilde{u})$ est une solution de P pour la donnée commune (D,ρ), $\tilde{u}$ est une solution de P_a pour cette donnée. □

L'utilisation du théorème 15' fournirait un résultat analogue lorsque R est complet.

Nous nous intéressons maintenant au cas des solutions calculables. Le théorème suivant est une sorte de "réciproque" du théorème 12.

Théorème 18

Soit (A,α) une abstraction simple de S et P_a un problème sur A dont la représentation par α est notée P. On suppose que S est une extension conservative d'un S.I. consistant R et que A est aussi une extension de R ; R contient le domaine des résultats de P_a.

Si P_a admet une solution $\tilde{u}$ pour la donnée (D,ρ) commune à lui et à P et si P admet une solution calculable $\tilde{v}$ pour (D,ρ) alors $\tilde{u}$ est l'unique solution de P_a et $\alpha(\tilde{u}) = \tilde{v}$. □

Notons que $\tilde{u}$ n'est pas nécessairement une solution calculable de P_a.

Démonstration

$\rho(\varepsilon_s)[u/r]$ est un théorème de $AD[\rho(\varepsilon_i)]$, donc $\alpha(\rho(\varepsilon_s))[\alpha(u)/r]$ est un théorème de $SD[\alpha(\rho(\varepsilon_i))]$. C'est-à-dire que $\alpha(\tilde{u})$ est une solution de P pour (D,ρ). Par unicité d'une solution calculable on en tire que $\alpha(u) \equiv v$ est un théorème de SD. C'est donc un théorème de R puisque S est une extension conservative de R et que u et v ont leur profil dans R. Si $\tilde{u}'$ est une autre solution de P_a un raisonnement analogue conduit à $\alpha(u') \equiv v$ est un théorème de R, donc $\alpha(u) \equiv \alpha(u')$ est un théorème de R. Alors d'après la remarque suivant le théorème 15' (A,α) est complète pour la classe des formules associées à P, donc $u \equiv u'$ est un théorème de AD.

cqfd

Les théorèmes 16 et 17 fournissent des résultats importants d'un point de vue méthodologique : on ne change pas les solutions d'un problème en le remplaçant par une de ses abstractions pourvu qu'elle soit complète pour la classe des formules qui lui sont associées.

Le théorème 18 précise quant à lui que, sous certaines hypothèses assez naturelles, on ne change pas les solutions calculables en effectuant une abstraction.

Il va de soi, d'après la dualité entre abstraction simple et représentation simple, que les théorèmes précédents sont encore applicables dans le cas des représentations simples. Ce que l'on peut résumer schématiquement sous la forme :

- une représentation simple "transporte" les solutions et les solutions calculables (théorème 10 et 12).
- une représentation simple conservative (S',μ) de S est telle que toute solution d'une représentation P' d'un problème P soit l'image par μ d'une solution de P (théorème 16). On peut énoncer l'analogue du théorème 17, et de son corollaire, en introduisant le concept de représentation conservative pour un ensemble F de formules de S.
- on a aussi l'analogue du théorème 18 : l'existence d'une solution d'un problème et celle d'une solution calculable pour la représentation assure, sous certaines hypothèses de conservativité, que la solution dans S est unique. Ceci justifie la plupart des techniques classiques de représentation.

6.3.3.- Abstractions et représentations généralisées

Les concepts d'abstraction et de représentation simples ne sont pas suffisants pour exprimer complètement les concepts intuitifs d'abstraction et de représentation :

Par définition d'une réalisation, à chaque symbole de S est associé, dans la représentation (S',μ) une opération généralisée (cf. ch. 1.2 définition 14) de S' (c'est-à-dire encore un schéma fonctionnel). Mais on peut envisager des représentations pour lesquelles l'image d'un symbole ne peut pas s'écrire comme composée de symboles du S.I. "cible" mais n'est défini que par un énoncé (explicite bien souvent, c'est-à-dire par une définition récursive). C'est par exemple le cas de l'image de valattr opération de S.I. TABSYM (figure 10, chapitre 1.2) lorsqu'on représente une table des symboles par une pile de tables (cf. § 6 chapitre 2.2) :

μ(valattr)(p,ident) ≡ *si* p ≡ pileyide
alors ω
sinon *si* exist(sommet(p),ident)
alors eval(sommet(p),ident)
sinon μ(valattr)(depiler(p),ident)

On ne peut exprimer ρ(valattr)(p,ident) sous forme d'un schéma fonctionnel *fini* du S.I. cible.

De manière analogue, bien que le problème semble moins fréquent, une abstraction simple ne permet de rendre compte que des *fonctions cachées* (cf. remarque suivant l'exemple 15) qui se réalisent sous forme d'opérations généralisées du S.I. initial. C'est par exemple le cas de long = || et fvide = <> dans le cas des files. Mais ici encore il se peut que la réalisation d'une fonction cachée ne puisse pas s'exprimer comme la composée d'opérations du S.I. initial. C'est par exemple le cas de decr dans l exemple 13 du chapitre 2.2.

Pour ces raisons nous généralisons les notions d'abstraction et de représentation.

Définition 15

Soit S un système d'information.

a) Une *représentation* (généralisée) de S par un S.I. S' est une application μ telle qu'il existe une *extension conservative* S" de S' avec (S'',μ) réalisation de S. (S'',μ) s'appelle encore représentation de S par S'.

b) Une *abstraction* (généralisée) de S par un S.I. A est une application α telle qu'il existe une extension conservative S" de S avec (S'',α) réalisation de A. (S'', α) s'appelle encore abstraction de S par A. □

Comme au paragraphe précédent nous considérons le point de vue "abstraction" sachant que toutes les définitions et résultats se traduisent immédiatement en termes de représentation.

Théorème 19. Avec les notations précédentes.

Si φ est un théorème de A dont l'image par μ est une formule $\alpha(\varphi)$ de S. Alors $\alpha(\varphi)$ est un théorème de S. □

Résulte immédiatement du fait que $\alpha(\varphi)$ est un théorème de S" et que S" est conservative.

Corollaire

Pour tout modèle $(\mathcal{U},\mathrm{id})$ de S, $(\mathcal{U}, \mathrm{id}_{\circ}\alpha)$ est un modèle de A restreint aux opérations et sortes dont l'image est dans S. □

Les différents résultats du paragraphe précédent sont encore valables ici.

Définition 16

Une abstraction $(A,S",\alpha)$ de S est *complète* si toute formule φ de A qui est telle que $\alpha(\varphi)$ soit un théorème de S est elle-même un théorème de A. □

Théorème 20

$(S",\alpha)$ est une abstraction complète de S par A si et seulement si la restriction, modulo α, à S de tout modèle de A peut se prolonger en un modèle de S. □

Le deuxième membre de l'équivalence signifie que si $(\mathcal{U},it')$ est la restriction à S du modèle $(\mathcal{U},it)$ de A, c'est-à-dire que $it = it'_{\circ}\alpha$, $(\mathcal{U},it')$ peut se prolonger en un modèle $(\mathcal{U}",it")$ de S.

Démonstration

Elle est similaire à celle du théorème 14 du chapitre 1.2.

a) La condition est suffisante :

Soit φ une formule de A, telle que $\alpha(\varphi)$ soit un théorème de S. Alors

$i(\alpha(\varphi)) = VRAI$ pour tout modèle $(\mathcal{U},i)$ de S.

Si $(\mathcal{U},it')$ est la restriction à S du modèle $(\mathcal{U},it)$, il existe it" prolongement de it'

$it"(\alpha(\varphi)) = VRAI.$

Or $it"(\alpha(\varphi)) = it(\alpha)$

ainsi pour tout modèle $(\mathcal{U},it)$ de A, $it(\alpha) = VRAI$. D'après le théorème de complétude de GÖDEL, α est un théorème de A.

b) La condition est nécessaire :

Il faut donc construire un prolongement de $(\mathcal{U},it')$ restriction de $(\mathcal{U},it)$ à S, en un modèle $(\mathcal{U}",it")$ de S.

Lemme 0

Soit F un ensemble de formules de A. Si $(S",\alpha)$ est une abstraction de S par A alors $(S"[\alpha(F)],\alpha)$ est une abstraction de $S[\alpha(F)]$ par A[F].

Démonstration

- Si $(S",\alpha)$ est une réalisation de A alors $(S"[\alpha(F)],\alpha)$ est une réalisation de A[F].

- Si S" est une extension conservative de S alors $S"[\alpha(F)]$ est une extension conservative de $S[\alpha(F)]$.

cqfd

Lemme 1

Soit $(S",\alpha)$ une abstraction complète de S par A et φ une formule de A. $(S"[\alpha(\varphi)],\alpha)$ est une abstraction complète de $S[\alpha(\varphi)]$ par $A[\varphi]$.

Démonstration

D'après le lemme 0 il suffit de prouver la complétude. On utilise ici encore le méta-théorème de la déduction, ce qui nécessite de considérer des formules fermées.

Soit $\overline{\varphi}$ la fermeture de φ. $A[\varphi] = A[\overline{\varphi}]$ et $S[\alpha(\varphi)] = S[\alpha(\overline{\varphi})]$.

Soit ψ est une formule de A telle que $\alpha(\psi)$ soit un théorème de $S[\alpha(\overline{\varphi})]$ alors $\vdash_S \alpha(\overline{\varphi}) \supset \alpha(\psi)$, donc $\vdash_A \overline{\varphi} \supset \psi$, dont on tire que ψ est un théorème de $A[\overline{\varphi}]$.

cqfd

Ce résultat se généralise immédiatement par récurrence au cas de n formules.

Lemme 2

Soit $(S",\alpha)$ une abstraction complète de S par A et F un ensemble de formules de A. Alors $(S"[\alpha(F)],\alpha)$ est une abstraction complète de $S[\alpha(F)]$ par A[F].

Démonstration

Avec le lemme 0 il suffit encore de prouver la complétude.

Soit ψ une formule de A telle que $\alpha(\psi)$ soit un théorème de $S[\alpha(F)]$. Dans sa démonstration interviennent des formules $\alpha(\varphi_1) \ldots \alpha(\varphi_n)$ de $\alpha(F)$ qui n'appartiennent pas à S :

$\alpha(\psi)$ est un théorème de $S[\alpha(\varphi_1) \ldots \alpha(\varphi_n)]$, donc ψ est un théorème de $A[\varphi_1 \ldots \varphi_n]$, donc de A[F].

cqfd

Nous pouvons maintenant prouver b) :

Supposons A consistant (sinon la démonstration est terminée). Un modèle $(\mathcal{U},it)$ de A est un modèle d'une extension *complète* A[F]de A.

Alors $S[\alpha(F)]$ est consistante puisque A[F] en définit une abstraction complète. Il existe donc un modèle $(\mathcal{U}",it")$ de $S[\alpha(F)]$, $it"_{\circ}\alpha$ est *le* modèle de A[F] complet qui est le prolongement de la restriction, modulo α, de $(\mathcal{U},it)$ à S.

cqfd

Définition 17

Une abstraction $(S",\alpha)$ de S par A est suffisamment complète pour un ensemble F de formules de A si toute formule φ de F telle que $\alpha(\varphi)$ est un théorème de S est elle-même un théorème de A. □

Alors les théorèmes 14 et 15, 15' et le corollaire se généralisent tels quels.

Il en est de même des théorèmes 10, 12, 16, V et 18 sur le "transport" des solutions des problèmes. Les démonstrations sont tout à fait identiques.

6.3.4.- Abstractions et représentations faibles

La définition précédente d'une abstraction est très "forte" dans la mesure où elle demande que l'image de tout théorème soit un théorème. On peut être ainsi conduit à restreindre beaucoup trop les possibilités de représentation d'un S.I.

Exemple 16

Reprenons l'exemple du S.I. FILE décrit par la figure 1 du chapitre 1.2. et proposons nous de le représenter dans la restriction du S.I. ENTIER* des suites d'entiers caractérisée par le prédicat

$$\varphi(u) = b_i(u) \equiv 1 \wedge b_s(u) \equiv 100$$

Soit donc TAB = ENTIER* | φ (cf. ch. 1.2, § 5.1.)
qui est donc intuitivement le S.I. des tableaux d'entiers sur [1..100].

a) Une première idée de représentation peut consister à utiliser l'opération tr (tranche) et de définir :

$$\forall f \in \text{FILE},\ \mu(f) = tr(f, b_i(f), b_s(f))$$

ce qui a intuitivement pour rôle de "décaler" f de telle manière que sa borne inférieure soit toujours 1.

Mais il est bien connu que cette représentation est très inefficace.

b) Une autre idée peut consister à considérer le tableau comme une liste circulaire : c'est-à-dire que le successeur du dernier élément est le premier (techniquement l'addition des indices est définie modulo 100).

Soit donc FILE1 = <<t : TAB , in : ENTIER , su : ENTIER>> le S.I. cible. On souhaite définir la représentation μ de FILE par FILE1 de la façon suivante :

. $\mu(\overline{\text{FILE}}) = \overline{\text{FILE1}}$

. $\mu(\text{ajtf})(f_1,x) = \underline{si}\ su(f_1) \equiv 100$
$\underline{alors}\ \langle\langle chval(t(f_1),1,x), in(f_1), 1\rangle\rangle$
$\underline{sinon}\ \langle\langle chval(t(f_1), su(f_1)+1, x), in(f_1), su(f_1)+1\rangle\rangle$

. $\mu(\text{retf})(f_1) = \underline{si}\ in(f_1) \equiv 100$
$\underline{alors}\ \langle\langle t(f_1), 1, su(f_1)\rangle\rangle$
$\underline{sinon}\ \langle\langle t(f_1), in(f_1)+1, su(f_1)\rangle\rangle$

. $\mu(\text{tetef})(f_1) = appl(t(f_1), su(f_1))$

. $\mu(<>) = \langle\langle to, 1, 100\rangle\rangle$ où to est l'élément de TAB dont tous les éléments ont pour valeur 0

. $\mu(||)(f_1) = \underline{si}\ in(f_1) \leq su(f_1)$
$\underline{alors}\ su(f_1) - in(f_1) + 1$
$\underline{sinon}\ (100 - in(f_1) + 1) + su(f_1)$

Malheureusement on ne définit pas ainsi une représentation au sens précédent. En effet, soit

f' = retf(ajtf(ajtf(<>,8),9))
f" = retf(ajtf(ajtf(<>,10),9))

Il ressort des axiomes de FILE que f' ≡ f" est un théorème de FILE. Au niveau de la représentation cependant :

$\mu(f') = \langle\langle[8,9,0,\ldots,0],2,2\rangle\rangle$
$\mu(f") = \langle\langle[10,9,0,\ldots,0],2,2\rangle\rangle$

qui sont différents : $\mu(f') \equiv \mu(f")$ n'est pas un théorème de FILE1, ce qui est assez gênant. Il est donc raisonnable d'étendre la définition d'une représentation pour prendre en compte ce genre de situation. □

Il ressort de l'exemple précédent que demander le transport de tous les axiomes du S.I. à représenter est une contrainte trop importante. Mais alors que doit transporter une représentation ? Toujours en nous inspirant de l'exemple précédent on souhaite dire que $\mu(f')$ et $\mu(f")$ sont égaux parce que l'application de l'opération "tetef" sur l'un et l'autre fournit les mêmes résultats. En d'autres termes l'égalité qui nous semble importante est liée à l'*application des opérations dont le but n'est pas une sorte d'intérêt*, ce que l'on appelle les *opérations externes* du S.I. Nous avions déjà fait une remarque très voisine au paragraphe 3 du chapitre 1.2 lorsque nous avions noté que la complétude était une notion trop forte. L'affaiblir conduit au concept de complétude suffisante. Ce concept permet ici encore de donner une définition des représentations plus raisonnable et plus intéressante pratiquement que la précédente.

Définition 18

Soit S un système d'information, extension suffisamment complète du S.I. R.

a) Soit S' une extension suffisamment complète de R. Une *représentation faible* pour R de S par S' est une application μ telle qu'il existe une *extension conservative* S" de S' contenant les images $\mu(u) \equiv \mu(v)$ des théorèmes de S de la forme $u \equiv v$ où les profils de u et v sont dans R.

b) Soit A une extension suffisamment complète de R. Une *abstraction faible* pour R de S par A est une application α telle qu'il existe une extension conservative S" de S contenant les images $\alpha(u) \equiv \alpha(v)$ des théorèmes de A de la forme $u \equiv v$ où les profils de u et v sont dans R. □

Cette définition est semblable à la définition 15 mais on ne demande ici que le transport des théorèmes formés de formules atomiques de R.

Les théorèmes du paragraphe 6.3.2 concernant le transport des solutions d'un problème (théorèmes 10, 12, 16, 17, 18) sont encore valables dans le cadre de la définition précédente si les formules atomiques de l'énoncé strict du problème sont des formules de R.

6.3.5.- Résolution du problème de la transformation de systèmes d'information

La spécification du problème de la transformation de S.I. est très voisine de celle du problème de la transformation d'énoncé (figure 3). Contentons-nous d'en indiquer brièvement la relation entre donnée et résultat dans le cas d'une abstraction.

Abstraction	
P : PROBLEME P_a : PROBLEME	*résultat* $P_a = ((\Sigma, \varepsilon_s, \varepsilon_i), (A, \mathcal{D}op, \mathcal{D}yp))$ *tq* α abstraction de S par A *donnée* $P = ((\Sigma, \alpha(\varepsilon_s), \alpha(\varepsilon_i)), (S, \mathcal{D}op, \mathcal{D}yp))$

Bien sûr, ici encore il faudrait préciser la forme des S.I. A cherchés (cf. ch. 2.2). Mais il faut aussi être capable de *construire* des abstractions et des représentations. Or, si le problème de la synthèse d'énoncés est l'objet de plusieurs études, la synthèse de représentations ou d'abstractions en est encore a ses balbutiements (citons cependant [GAU,78a]). En particulier le *choix* du S.I. cible dans le cas d'une représentation est loin d'être automatisable actuellement. Par exemple en SETL on spécifie (parfois) le choix du S.I. cible et le reste est automatique.

Comme dans le cas des programmes, les principales études sur les représentations de types abstraits ont concerné (et concernent encore!) les techniques de preuves à posteriori (par exemple [GUT,78a],[WUL,75], etc...). Il reste encore beaucoup de travail avant d'arriver à de bonnes techniques de transformation. Nous abordons ce problème au chapitre 2.2.

Signalons pour terminer l'introduction par GUTTAG [GUT,78a] d'un outil permettant de caractériser les représentations faibles : l'interprétation de l'égalité (cf. ch. 2.2 § 5.2).

7.- CONCLUSION.

Dans ce chapitre nous nous sommes efforcés de définir formellement des concepts qui sont d'une importance capitale pour la résolution des problèmes. En ce sens il constitue une étude fondamentale des principales notions associées aux problèmes : donnée, solution, solution calculable, transformation d'énoncé, représentation, abstraction. Il resterait à la poursuivre et à la développer : que peut-on dire de plus sur la représentation de S.I. paramétrés? Comment prendre en compte la représentation de données ? Comment mieux prendre en compte les intermédiaires ? La distinction entre ε_s et ε_i est-elle intéressante ? etc...

Nous disposons maintenant d'un langage de description de problèmes reposant sur une théorie, ce qui va nous permettre de revenir au problème initial de la spécification des problèmes (ch. 1.4).

Mais aussi tout au long de ce chapitre nous nous sommes heurtés à la même lapalissade : un problème étant spécifié formellement, il reste encore beaucoup de chemin à parcourir avant d'en trouver une solution.

Le petit schéma de la figure 4 résume les grandes directions qui peuvent être choisies.

Evidemment, lors d'une résolution particulière, différentes possibilités peuvent être adoptées successivement, de plus dans le cas des transformations

(2ème colonne) l'énoncé de départ peut déjà être explicite (récursif) et on peut chercher à en déduire un énoncé plus efficace (itératif par exemple).

Méthode / Outil	Recherche directe d'une solution : "Interprétation" On part d'une information contenant les données. On obtient une solution s'il en existe.	Transformation d'énoncé "Compilation" On part d'un énoncé implicite. On obtient un énoncé explicite ou sous forme clausale.
Utilisation d'un démonstrateur de théorème "intelligence artificielle"	Méthode de résolution (§4)	Synthèse automatique : heuristiques et choix
Utilisation de règles de calcul (éventuellement déterministe)	Calcul d'une solution (§5)	- Skolémisation - Utilisation de règles (déterministes) de transformation

figure 4 : Comparaison de techniques de résolution

Cette dualité entre point de vue interprétatif et point de vue compilatoire est évidente lorsqu'on utilise les démonstrateurs de théorèmes : dans le premier cas on cherche, pour une valeur des données, à prouver une variante de l'énoncé, dans le deuxième cas on cherche à prouver l'énoncé considéré comme une formule sans variable libre dans laquelle la donnée est quantifiée universellement et le résultat existentiellement.

Signalons encore un gros avantage de la voie interprétative : les données étant connues le calcul d'une solution peut être fait par un algorithme spécifique de ces données. Ceci n'exclut pas l'automatisme de la synthèsede cet algorithme [LLAU,76]. On aborde ainsi une autre voie de recherche : plutôt que de chercher à transformer un énoncé en un autre sur lequel on peut appliquer une méthode de résolution fixée à l'avance, on peut chercher à générer une méthode de résolution adaptée [illegible]

Notons enfin que dans toute cette étude nous nous sommes cantonnés au cas où le problème est spécifié par un énoncé. Une autre possibilité consiste à caractériser un problème par un certain nombre d'exemples [JOU,77], [KOD,78] et à effectuer une synthèse d'un algorithme de résolution à partir de ces exemples. Cette voie semble "orthogonale" à celle que nous détaillons dans la suite et devrait pouvoir lui être associée de manière complémentaire.

CHAPITRE 1.4

UNE MÉTHODE DE SPÉCIFICATION DE PROBLÈMES

L'étude menée aux chapitres précédents nous a permis, d'une part de dégager des bases théoriques sur lesquelles peut reposer l'activité de spécification de problèmes, d'autre part de décrire, à partir de ces bases, un langage de spécification.

Par exemple, la syntaxe de description des données d'un problème a été choisie identique à celle permettant de décrire des paramètres d'un S.I. puisque les données sont des paramètres particuliers.

En revenant au (méta) problème central de cette partie (spécification des problèmes) nous tentons maintenant de dégager, à partir de la théorie et du langage précédents, une méthode de spécification. Elle sera évidemment incomplète et imparfaite comme nous le soulignerons, mais elle doit pouvoir servir de noyau à une étude plus complète.

Au premier paragraphe nous récapitulons les différents concepts sur lesquels repose l'activité de spécification en partant du méta-problème de la spécification. Puis nous nous restreignons au langage de s-description (§ 2) et nous progressons dans la formalisation de l'activité de spécification en proposant un méta-algorithme de spécification. Le déroulement de ce méta-algorithme est décrit sur un exemple.

1.- CONSIDERATIONS GENERALES SUR LA SPECIFICATION.

1.1.- Le problème de la spécification.

Reprenons la discussion commencée au paragraphe 2 du chapitre 1.1.. De manière informelle et intuitive nous avions alors décrit le problème de la spécification comme étant celui du passage d'un énoncé informel à un énoncé formel. La figure 1 en donne une description.

P est une version formelle et structurée de l'énoncé informel Pinf structuré : indique qu'un problème formel est structuré formalise : indique qu'un problème formel formalise un énoncé informel	P : PROBLEME structuré : *fonct* (PROBLEME)BOOL formalise : *op* (PROBLEME, EINFORMEL)BOOL Pinf : EINFORMEL PROBLEME : *M.S.I.* EINFORMEL : *M.S.I.*	*résultat* P *tq* P formalise Pinf ∧ structuré(P) *donnée* Pinf

Figure 1 : Méta-problème de la spécification.

Nous ne pouvons évidemment pas formaliser complètement ce méta problème puisque nous ne savons pas définir ce que signifie "formalise". Les seuls éléments solides sur lesquels peut reposer une amélioration de la spécification précédente sont :

- La définition du concept "structuré"
- La définition du méta système d'information PROBLEME.

Ceci implique que l'on doive préciser le *langage* et la *forme* d'une spécification. La forme est à la fois liée au langage, qui impose certaines contraintes, mais aussi à la façon dont une spécification est obtenue, donc à la *méthode* employée. Cette méthode est par ailleurs le garant essentiel de l'adéquation de la spécification formelle à l'énoncé initial.

Enfin, puisqu'il semble hors de question, en l'état actuel des choses, de construire un méta-algorithme "automatique" permettant de spécifier un problème, il faut chercher à se doter d'outils facilitant la mise en oeuvre de la méthode.

Nous survolons rapidement, dans la suite de ce premier paragraphe, ces trois points fondamentaux : langage, méthode, outils.

1.2.- Langage de spécification.

Il serait de peu d'intérêt, dans l'état actuel de nos travaux, de figer la syntaxe du langage des s-descriptions en le définissant de manière formelle. En revanche, il est beaucoup plus fondamental de rappeler les

caractéristiques structurales de ce langage qui le rendent adapté à la spécification.

a) Structuration et modularité.

Un problème est s-décrit par un ensemble de *modules*. L'un de ces modules est le module principal et il décrit l'énoncé (énoncé strict et énoncé des intermédiaires), le résultat et les données du problème. Les autres modules décrivent soit les S.I. concsituant l'univers du problème, soit le corps d'une opération utilisée dans un autre module (§ 2.3.1 ch. 1.3).

Cette structure des s-descriptions permet, comme nous l'avons déjà indiqué au chapitre 1.2 § 5, la mise en oeuvre d'une double modularité :

- modularité au niveau des univers et entre les univers et l'énoncé
- modularité au niveau de la spécificatien de chaque opération qui apparaît comme celle d'un nouveau problème.

b) Spécification de lexiques et de profils.

Un module est constitué de trois parties :

- Le lexique qui décrit informellement les différents identificateurs (opérations ou S.I.)
- Le profil qui décrit le profil de chaque opération, éventuellement suivi de *de base dans* "nom de S.I." ou de "*donnée*".

Dans cette partie profil , on rencontre aussi les identificateurs de S.I. sous la forme S : *S.I.*.

- L'énoncé formel qui comprend lui-même en général, plusieurs parties :

i) dans le cas de modules décrivant des S.I. :

- Une définition des paramètres
- Une définition des sortes d'intérêt
- Un invariant
- Une définition des opérations.

Certaines de ces parties peuvent être absentes lorsque l'on utilise des opérations d'extension, de restriction, de dérivation etc... (cf § 5 et § 7 ch.1.2)

ii) dans le cas de modules décrivant des énoncés :

- une partie résultat
- une partie définissant des opérations (éventuellement intermédiaires o-aires)
- une partie donnée

Tant pour les modules décrivant des S.I. que pour ceux décrivant les énoncés, l'ordre des différentes parties est non significatif. Cependant, méthodologiquement, il est agréable de commencer à spécifier, les sortes d'intérêt dans le cas des S.I., et le résultat dans le cas des énoncés (cf. § 2).

Lexiques et profils ont un double rôle :

- Ils facilitent la mise en oeuvre d'une méthode de spécification descendante (cf § 2) en permettant de connaître, à tout moment, les identificateurs non encore spécifiés ainsi que leurs particularités (*de base dans..., donnée*).
- Ils facilitent la lecture de la spécification.

c) Quelques qualités du langage des s-descriptions.

Indiquons rapidement quelques propriétés du langage des s-descriptions qui nous incitent actuellement à l'utiliser comme support de la méthode esquissée au paragraphe 2 :

- Il permet d'exprimer dans un même cadre énoncé et univers d'un problème
- Il est de haut niveau (manipulation d'ensembles, de fonctions...) et dispose de mécanismes de constructions d'objets évolués (produit, suites...)
- Il est extensible grâce aux mécanismes de structuration des spécifications et en particulier grâce à la paramétrisation des S.I.. Il est alors possible à un utilisateur s'intéressant à une classe de problèmes particuliers de se doter de constructions de bases (listes, fichiers, graphes...) bien adaptées à la spécification de ces problèmes. On peut ainsi définir l'analogue des prologues de programmathèque en ALGOL 68 [WIN, 75] .
- Il est, à notre sens, assez proche de l'intuition
- Il est formel et repose sur des concepts théoriques solides.
- Il est possible, en paraphrasant les spécifications formelles et en utilisant le lexique, de construire une version informelle (en langue naturelle) mais structurée du problème considéré.

d) Quelques possibilités d'amélioration.

Bien qu'elle repose sur des concepts ensemblistes, on peut envisager de rendre la syntaxe un peu moins "sèche" en introduisant des "mots clés", moins ésotériques. Par exemple on peut préférer la phrase :

α : *suite de* LIAISON
tq α entre(vd, va)
et pour tout β : *suite de* LIAISON *tq* β entre(vd, va)
on a prix(α) ⩽ prix(β) (version suggérée par M. SINTZOF

à la phrase :

α : LIAISON*
tq α entre (vd, va) ∧
(∀β ∈ LIAISON* , β entre(vd, va) ⊃ prix(α) ⩽ prix(β))

figure 3 ch 1.1).

Par ailleurs , il faut expurger la syntaxe de tout ce qui n'est pas fondamental. Pour ce faire, on peut minimiser le nombre de formes de bases et autoriser, comme en ALGOL 68 par exemple, l'utilisation d'un langage étendu (ce que nous avons déjà commencé à faire, par exemple dans le cas des s-descriptions d'opérations, cf. ch. 1.2 § 5.1.1).

1.3.- Méthode de spécification.

a) Enoncé informel, énoncé formel.

La spécification d'un problème que nous cherchons à construire doit être *structurée* et *formelle*. Ces deux qualités seront obtenues en partie par l'utilisation du langage des s-descriptions.

A l'opposé, l'énoncé initial dont on dispose, ne repose, en général, sur aucun langage strict, comporte des incohérences et des ambiguités, est incomplet, mais surtout n'est pas structuré. Notons que, si les ambiguités sont essentiellement liées à la nature du langage, la sous spécification et la plupart des incohérences sont dues, quant à elles, au manque de structuration.

Un guide raisonnable pour une méthode de spécification consiste donc à structurer d'une certaine façon "canonique" le problème considéré ce qui doit permettre de compléter les sous spécifications et d'éviter certaines incohérences ainsi que de lever les ambiguités si le langage cible est formel.

b) Principales idées à la base d'une méthode de spécification"déductive".

La démarche que nous avons illustrée au chapitre 1.1, consiste à spécifier formellement la relation entre donnée et résultats en introduisant éventuellement des objets, fonctions, relations ou S.I. intermédiaires. On considère alors la caractérisation de ces intermédiaires comme autant de nouveaux problèmes qu'il faut formaliser et on reitère le processus. On s'arrête lorsque les objets, fonctions ou relations intermédiaires sont des données ou des éléments primitifs. On peut ainsi préciser conjointement l'énoncé strict et son contexte qui est l'univers. On peut ainsi vérifier pas à pas la cohérence de l'énoncé formel. De plus la spécification des résultats d'une fonction (relation) à l'aide des données peut être effectuée indépendemment des spécifications des autres fonctions : on a ainsi un aspect à la fois *modulaire* et *descendant* de la construction d'une spécification.

Le développement conjoint d'une partie lexique, d'une partie profil et d'une partie énoncé formel permet :

- d'avoir un guide méthodologique : tout nouvel intermédiaire est décri informellement dès son introduction dans une formule, puis il doit être précisé formellement s'il n'est pas de base.

- d'être plus facilement convaincu de l'adéquation de l'énoncé formel à l'énoncé informel en permettant des justifications pas à pas.
- de mettre en évidence les spécifications informelles ambigües, incomplètes et de permettre ainsi de les préciser.
- de faciliter la discussion avec l'utilisateur en ayant des descriptions en langage clair des différentes opérations et des S.I. introduits.
- de faciliter certains contrôles de cohérence, par exemple ceux qui sont associés aux profils.
- de permettre le regroupement, pour chaque S.I. introduit, des différentes opérations s'y rapportant.

Cette approche reprend les principales idées de la méthode déductive de programmation [PAI, 79] , [BEL, 78 a] en généralisant la forme des énoncés et en introduisant des spécifications de S.I. .

c) Méthode de spécification déductive ou associative.

Au niveau de la *construction des programmes* plutôt que d'effectuer une construction complète pour un problème donné on peut chercher à rapprocher ce problème d'un problème déjà résolu et à tenter d'adapter cette solution au cas présent.

C'est une démarche développée en particulier par une équipe de chercheurs de Grenoble [COU, 74] , [LUC, 77] , [SCH, 79] qui a conduit ces chercheurs à définir un catalogue de schémas de programmes. Face à un problème donné la méthode consiste alors à reconnaître un schéma bien adapté à la résolution et ceci peut introduire des résultats intermédiaires, résultats qui seront à leur tour traités de manière identique.

Ce point de vue peut être adapté au problème de la spécification : en présence d'un problème, on peut essayer de l'identifier à un type de problèmes déjà spécifié et utiliser ainsi le *schéma de spécification* correspondant. Ceci nécessite en général des adaptations qui consistent à introduire des sous-problèmes qu'il faut ensuite spécifier. On utilise ainsi une méthode de spécification plus *associative* que déductive. On voit cependant les limites de ce type d'approche : un langage de spécification est beaucoup plus "vaste" qu'un

langage de programmation et le nombre de "schémas de spécification" risque d'être très grand. Néanmoins ce type d'approche peut être valable pour certaines classes de problèmes tels que les problèmes d'informatique de gestion par exemple.

Les deux approches "déductive" et "associative" peuvent être considérées et utilisées de manière complémentaire : au cours de la spécification descendante d'un problème, il faut essayer d'identifier certains sous-problèmes à des schémas de spécification.

d) Encore un exemple.

Avant de proposer une méthode de spécification fondée sur les idées précédentes, nous souhaitons insister sur le fait que la technique générale de spécification consiste à introduire des intermédiaires. C'est en particulier le cas lorsque la structure des données est mal adaptée à une spécification simple du problème, par exemple lorsque la représentation physique des données est éloignée de leur structure logique. L'introduction d'intermédiaires a alors pour objectif de se rapprocher de schémas de spécifications connus.

Exemple 1 :

Evaluation d'une expression arithmétique postfixée (ch. 1.1, § 1.1 1)). Ce problème s'exprime naturellement de manière récursive puisque la forme d'une expression postfixée est définie récursivement, par une grammaire, par exemple. Cette remarque correspond à l'identification d'un certain type de problème . Lors du développement de la spécification on est conduit à décrire l'objet structuré qu'est une expression postfixée. C'est sur ce S.I. "expression postfixée" que repose la spécification de la figure 2.

La spécification obtenue n'est pas satisfaisante si, comme on peut raisonnablement le supposer, la donnée e se présente sous la forme d'une chaîne linéaire de caractères, c'est à dire qu'elle n'est pas de profil EXP mais un profil "expression linéaire d'expression postfixée" qu'il faut transformer en EXP.

Evaluation d'une expression

cherche r, valeur de l'expression arithmétique
ost-fixée e
l : application qui associe sa valeur à toute
xpression arithmétique post-fixée.
fet : application qui à un triplet de la forme
réel, réel, opérateur), associe le résultat de
application de l'opérateur aux deux réels.
uche : associe à une expression, qui n'est pas
réel, sa sous-expression gauche.
oite : idem pour l'expression droite.
era : associe à une expression qui n'est pas
réel son opérateur principal.

- r : REEL
- e : EXP
- val : *fonat*(EXP)REEL
- effet : *fonct*(REEL,REEL, OP)REEL
- gauche : *fonct*(EXP)EXP *de base dans* EXP
- droite : *fonct*(EXP)EXP *de base dans* EXP
- opera : *fonat*(EXP)OP *de base dans* EXP

résultat r
r *tq* r = val(e)
donnée e
val *tq* val(u) = *si* u ∈ REEL *alors* u *sinon* effet(val(gauche(u)), val(droite(u)), opera(u))
effet *tq* effet(x,y, op) = *si* op = '+' *alors* x+y op = '-' *alors* x-y op = '*' *alors* x*y op = '/' *alors* x/y

EXP : S.I.

EXP = REEL U « gauche : REEL, droite : REEL,opera : OP
OP = {'+', '-', '*', '/' }

Figure 2 : évaluation d'une expression post-fixée.

Plus précisément appelons RLEXP le S.I. des représentations linéaires d'expressions postfixées défini par

$$RLEXP = (REEL \cup OP)^* \mid \varphi$$

avec $\varphi(u) = U \in REEL \vee \exists i \in [bi(u)..bs(u)]$ *tq*
$tr(u, bi(u), i-1) \in RLEXP \wedge$
$tr(u, i, bs(u)-1) \in RLEXP \wedge$
$der(u) \in OP$

L'amélioration de la spécification précédente peut s'envisager de différentes façons :

i) Définir une représentation ρ de EXP par RLEXP . La représentation de la spécification de la figure 2 correspond bien au problème initial.

ii) Considérer e comme un objet intermédiaire qu'il s'agit maintenant de définir à partir de l'expression linéaire donnée el .

Dans i) on cherche en fait la fonction faisant passer de e à el, dans ii) on cherche sa réciproque. Développons cette deuxième possibilité :

Notons for la fonction qui transforme un élément u de RLEXP en l'expression postfixée correspondante. Pour la décrire, on l'étend en une fonction définie sur les "sous-suites commençantes" des suites de RLEXP. En général, ces sous-suites ne sont pas elles mêmes des suites de RLEXP. Ainsi :

$$\begin{cases} for\ (u) = u \quad si \quad u \in REEL \\ for\ (u \sim x) = \underline{si}\ \ x \in REEL\ \ \underline{alors}\ \ for(u) \sim x \\ \qquad\qquad\qquad\qquad\ \ \underline{sinon}\ \ for(u_1) \sim\!\!< e_1, e_2, x > \end{cases}$$

où e_1, e_2 sont des éléments de EXP tels que $for(u) = f(u_1)^* e_1^* e_2$

avec for : *fonct* (SRLEXP) EXP*

où SRLEXP = $(REEL \cup OP)^* \mid \psi$

$\psi(u) = \exists v \in (REEL \cup OP)^*$ tq $u * v \in RLEXP$

Il est maintenant possible de compléter la spécification de la figure 2 en remplaçant "*donnée* e

par { e : EXP | e *tq* e = for(el)
for : ~~~~ | } définition de for
el : RLEXP | *donnée* el

Une étape ultérieure, qui peut être considérée comme faisant partie de la résolution du problème, consiste à composer for et val pour éviter l'introduction de l'intermédiaire e . Ceci nécessite d'étendre val aux suites d'expressions :

$$\begin{cases} val : \underline{fonct}\ (EXP^*)\ REEL^* \\ val\ (se) = sr \\ \qquad \underline{tq}\ \ bi(sr) = bi(se) \wedge bs(sr) = bs(se) \wedge \\ \qquad\qquad \forall i \in [bi(se)..bs(se)] \quad , sr_i = val(se_i) \end{cases}$$

Posons alors eval(u) = val (for(u))

avec eval : *fonct* (SRLEXP) REEL*

Une transformation immédiate conduit à la définition récursive directe suivante de eval

$$\begin{cases} eval(x) = x \quad si \quad x \in REEL \\ eval(u \sim x) = \underline{si}\ x \in REEL \\ \qquad\qquad\qquad \underline{alors}\ \ eval(u) \sim x \\ \qquad\qquad\qquad \underline{sinon}\ \ rest(eval(u)) \sim effet(pen(eval(u)), der(eval(u)), x \end{cases}$$

où pen(β) est l'avant dernier élément de β , rest(β) est tel que :

$\beta = rest(\beta) \,.\, pen(\beta) \,.\, der(\beta)$ □

1.4.- Outils de spécification.

Jusqu'à présent nous avons très peu considéré cet aspect, tant parce que le processus de spécification nous semblait trop peu connu que parce que les outils dont on pourrait avoir besoin ne semblent pas très facilement réalisables.

Dans un objectif à long terme, nous essayons de préciser maintenant ce que pourraient être de tels outils :

a) Aide à la vérification de la cohérence :

Il s'agit de prouver la *consistance* d'un système formel du premier ordre. On sait que ce problème est difficile à résoudre dans le cas général. Un projet moins ambitieux concernerait la preuve de "petits morceaux" de spécifications, d'énoncés déjà exprimés sous forme récursive par exemple.

Nous verrons au chapitre 2.2 que, dans le cas de spécifications algébriques, il existe des techniques de preuve de consistance.

b) Aide à la vérification de la fidélité.

Aucune preuve formelle ne peut être faite ; mais on peut penser mettre en oeuvre des techniques de tests ou plus généralement d'évaluation symbolique [KIN, 76] : si on peut disposer d'une méthode de résolution directe de l'énoncé formel, même si elle est très inefficace et non applicable dans tous les cas, il est possible de vérifier que les solutions correspondant à certaines valeurs des données sont bien celles que l'on attendait. On peut même envisager de travailler sur des valeurs symboliques ce qui augmente la puissance de la vérification.

Dans cette optique une double description du problème, d'une part sous forme d'un énoncé informel , d'autre part sous forme d'exemples, est tout à fait souhaitable. On peut alors utiliser, le premier point de vue pour construire méthodiquement une spécification formelle, et le deuxième, pour vérifier à postériori cette spécification. On cherche ainsi, par deux chemins différents, à obtenir le même résultat ce qui est une technique de preuve bien connue.

De plus, il est possible de prolonger cette idée : si la spécification finale a l'aspect modulaire et hiérarchisé fourni par la démarche descendante que nous proposons (§ 1.2, § 2) on peut envisager de vérifier modulairement cette spécification et rechercher d'éventuelles erreurs comme dans le cas de SNOOPY [BEL, 78 d].

Ces idées nécessitent donc la mise au point de méthodes automatiques d'évaluation directe d'énoncés sur un système d'information, fondées sur une méthode de résolution. La construction d'interprète travaillant sur des implantations directes de types algébriques [GUT, 78a] que nous mentionnerons au chapitre 2.2 est un cas particulier de mise en oeuvre de cette idée.

c) Système d'aide à la spécification.

Alors que les outils évoqués en a) et b) permettraient d'effectuer des vérifications à postériori, l'outil le plus utile pour le spécificateur serait certainement un système interactif qui l'aiderait et le guiderait dans la mise en oeuvre de la méthode de spécification. En effet comme nous allons le voir sur l'exemple 2, dès que le problème est d'une taille raisonnable, la tâche du spécifieur est rendue difficile par la multiplication des intermédiaires et des modules. On doit donc lui fournir des aides pour maitriser cette complexité.

Pour décrire les fonctions d'un tel système on peut s'inspirer de quelques unes des idées qui ont été émises pour le développement de systèmes assistés de construction de programmes (par exemple CIP [BAU, 78a] ou SACAD [BEL, 78a]) en les adaptant au problème de la spécification :

- Possibilité de parcourir l'arborescence des modules d'une spécification, éventuellement partielle , pour y rechercher des informations. Cette fonction est à la base du système MENTOR [KAH, 77] qui permet une exploration de l'arborescence qui est la structure abstraite d'un programme PASCAL.

- Possibilité de créer et d'utiliser un catalogue de S.I. prédéfinis contenant un certain nombre de fonctions de base. L'utilisation de systèmes d'information paramétrés permet ainsi de se définir des *constructeurs de S.I.* utilisables pour certaines classes de problèmes.

- Aide à la mise en oeuvre de la méthode de spécification déductive que nous préconisons. A une étape donnée de la spécification, le système peut donner la liste des symboles non encore définis (éventuellement par modules), la liste des fonctions de bases d'un S.I. non encore spécifié, la liste des S.I. non encore décrits etc.... Il permet ainsi d'éviter les oublis de spécifications (à la fin la spécification sera complètement définie) ou les doubles spécifications.

- Possibilité d'utiliser les outils (!) décrits en a) et b) et de faire ainsi, au cours de la spécification, des vérifications locales.

- Possibilité d'avoir des descriptions graphiques des spécifications. Cette idée est loin d'être mûre actuellement mais le grand développement actuel de matériels et logiciels graphiques peut apporter une aide appréciable au problème de la spécification (ch 1.1 § 4.2.1 g)).

g) Langage de description des choix successifs.

Fournir une spécification formelle et affirmer qu'elle rend bien compte d'un énoncé initial n'est pas suffisant et il importe de pouvoir justifier les choix successifs de structuration et de formalisation. Pour cette raison il est intéressant de se doter d'un langage permettant de décrire ces choix, ou, si l'on préfère, d'un langage de *description du méta-algorithme de spécification.*

Un tel langage permet de dialoguer avec la personne qui pose le problème en lui expliquant la raison de tel choix ou au contraire en notant ses intentions. Il permet également au spécifieur de revenir sur les choix antérieurs, ce qui facilite les possibilités de modification d'une spécification.

Nous ne développerons pas ici l'étude d'un tel langage, nous contentant d'en donner un exemple d'utilisation au paragraphe suivant (cf. exemple 2).

2.- UN META-ALGORITHME DE SPECIFICATION.

Il n'est pas possible de formaliser le méta-problème de la spécification. A fortiori il n'est pas possible de le résoudre selon les techniques développées dans la partie 2. La méthode (ou méta-algorithme) de spécification que nous proposons repose donc à la fois sur la structuration inhérente aux s-descriptions et sur les idées générales rappelées au paragraphe 1.3.

Pour être le plus précis et concis possible nous tentons de décrire formellement (§ 2.2) la méthode de spécification déductive en utilisant une syntaxe et un style très voisins de ceux des énoncés explicites présentés dans la deuxième partie (reprenant en cela des suggestions de M. SINTZOFF et M. NIVAT concernant la construction des algorithmes).

Il est évident que le méta-algorithme obtenu est largement perfectible, notamment en ce qui concerne la spécification des univers, mais il n'en constitue pas moins une première étape fondamentale vers l'élaboration d'un système d'aide à la spécification.

Commençons par en donner une description informelle.

2.1.- Description informelle du méta-algorithme de spécification.

Les "opérations élémentaires" du méta-algorithme sont :

i) La spécification d'un objet intermédiaire par l'introduction de nouveaux intermédiaires qui peuvent être des opérations ou des S.I..

ii) La spécification d'un univers par l introduction de nouvelles sortes et par le regroupement des opérations qui s'y rapportent.

La composition de ces opérations élémentaires se fait par une itération, dont l'arrêt est obtenu lorsque tous les intermédiaires ont été spécifiés, et par un choix du prochain intermédiaire à spécifier.

Ce que l'on peut décrire intuitivement de la manière suivante :

1) Commencer par déterminer le ou les résultats, leur profil et leur spécification informelle. Les introduire respectivement dans les parties énoncé, profil et lexique.

2) Jusqu'à ce que tous les identificateurs soient cochés, répéter :

- Choix d'un identificateur non coché dans l'ensemble des parties profil qui n'est pas une opération dite "*de base dans* S" où le S.I. S n'est pas encore spécifié.

- Le cocher dans la partie profil où il se trouve

2-1) Si cet identificateur est une opération

- Choix d'une spécification formelle
- Placer cette spécification formelle dans le module où doit se trouver cet identificateur, à savoir :

- Dans le module contenant le S.I. S si dans la partie profil l'identificateur est dit "*de base dans* S"
- Dans la partie *donnée* du module principal si l'identificateur est dit "*donnée*"
- Dans le module courant dans les autres cas

- Si l'opération n'est pas o-aire spécifier le résultat en fonction des données
- Si l'opération est o-aire introduire, dans le module où vient d'être placé l'identificateur, la spécification informelle des nouveaux intermédiaires et des nouveaux S.I. apparaissant dans leur profil, ainsi que les profils correspondants.

2-2) Si cet identificateur n'est pas une opération

2-2-1) Si ce S.I. S est un paramètre d'un S.I. S' ,

- Choisir la description de S et la placer dans la partie *param* de S'
- Rassembler les différentes opérations qui sont dites "*de base dans* S", les cocher, les spécifier et les placer dans la partie *param* de S à la suite de la description de S .

2-2-2) Si ce S.I. est une donnée
Reprendre la démarche de 2-2-1 mais en plaçant chaque spécification dans la partie *donnée* du module principal

2-2-3) Dans les autres cas
- Créer un nouveau module identifié par le nom S du S.I. considéré
- Choisir la description des sortes et des invariants de S
- Rassembler les différentes opérations qui sont dites "*de base dans* S", les cocher, les spécifier et les placer dans la partie opération de S .

2.2.- Description formelle

a) Forme de la description formelle

La figure 3 propose une version formelle incomplète du méta-algorithme. Le langage de description de l'énoncé est proche du langage MEDEE (cf. ch 2.3).

La structure générale de cet énoncé est itérative : on définit le résultat comme étant le dernier terme d'une suite de suites (on aurait pu choisir des ensembles) de modules. Signalons que les opérations non formalisables (choix d'une spécification pour un intermédiaire) sont placées entre ⊰ et ⊱ .

Cet énoncé est incomplet car, ni le "méta-système d'information" MODULE, ni les opérations permettant de construire des s-descriptions de S.I. ne sont formalisées. Il serait intéressant de poursuivre ce travail en cherchant à obtenir une description la plus précise possible.

b) Insuffisances de cette description

- Tant sous l'aspect informel que sous l'aspect formel, le méta-algorithme ne permet pas de créer un module correspondant à la spécification d'une opération. Pourtant ce sont les post conditions qui permettent d'avoir un deuxième niveau de modularité et il est souvent malcommode de décrire une post condition à l'intérieur d'un module décrivant d'autres opérations. Une extension immédiate du langage et de la méthode consiste donc à accepter, à côté des modules décrivant des S.I., des modules décrivant des pré et post conditions de certaines opérations.

- Un choix de spécification peut être mauvais et on peut souhaiter le modifier. Dans le module itéré sm il faut donc accepter des retours en arrière.

- Mais surtout les choix principaux apparaissent de manière trop anodine. Ce méta-algorithme décrit l'aspect "physique" de la spécification mais il reste les choix cruciaux de savoir comment spécifier formellement un intermédiaire, comment choisir le résultat etc... (ce sont les parties entre ⊰ et ⊱). Nous espérons que cette réflexion peut servir de point de départ à une *étude empirique sur la détermination d'heuristiques de spécification* liées au domaine du problème, à la forme du résultat etc....

méta-algorithme de spécification

Le résultat du méta-algorithme de spécification est une suite de modules qui sont des s-descriptions de l'énoncé du problème donné et de son univers

infor : est l'énoncé informel du problème donné

sm_i : terme général d'une suite dont le dernier terme est sm. C'est une suite de modules dont certains identificateurs restent à expliciter

$ttcoché_i$: terme général d'une suite de booléens qui est faux s'il reste des identificateurs non cochés

MODULE : est le méta-système formel des modules. C'est le produit cartésien
EINFORMEL × PROFIL × EFORMEL

- sm : MODULE*
- sm_i : MODULE*
- $ttcoché_i$: BOOL*
- MODULE : *M.S.I.*
- infor : EINFORMEL

résultat sm
sm = *dernier* sm_i *pour* i *dans* 1 → *jqa* $ttcoché_i$
donnée infor

sm

opération : indique si id est un symbole d'opération ou s'il désigne un S.I.

debase : indique si la déclaration de id dans sm'_j précise qu'il est de base dans le S.I. S

ajoutop : ajoute la spécification de l'opération id dans la place opera(S) du module ind(S). On suppose qu'à chaque S.I. est associé un module ind(S)

opération : *fonct* (SYMBOLE, MODULE)BOOL *de base dans* MODULE
debase : *fonct* (SYMBOLE, MODULE, S.I.) BOOL *de base dans* MODULE
ajoutop : *fonct* (SYMBOLE, ENTIER, PLACE, MODULE*) MODULE*

sm = *si* opération (id, sm'_j)
alors si ∃ S : *S.I.* *tq* de base(id, sm'_j, S)
alors ajoutop(id, ind(S), opera(S),
si id estdonnée sm'_j
alors ajoutop(id, 1, donnée(sm'_j)sm')
si id estparamètre sm'_j
alors ajoutop(id, j,param (sm'_j), sm')
sinon ajoutop(id, j, opera(sm'_j), sm')
sinon si id est paramètre sm'_j

ind : associe à un nom de S.I., le module d'une suite de modules dans lequel il est spécifié

opera : associe à un nom de S.I. où à un module sa partie opération (en particulier lorsque le S.I. est un paramètre)

PLACE : différentes parties d'un module : opérations, paramètres, données

estdonnée : indique si la déclaration de id dans sm'_j précise qu'il s'agit d'une donnée

donnée : associe à un module sa partie donnée

estparamètre : indique si la déclaration de id dans sm'_j précise qu'il s'agit d'un paramètre

param : associe à un module sa partie paramètre

ajoutsys : ajoute un nouveau S.I. soit dans la partie *param..fparam* , soit dans la partie *donnée..fdonnée* d'un module

ajoutmod : ajoute un nouveau module à une suite de modules

cocher : coche l'identificateur choisi

ind : *fonct*(SYMBOLE, MODULE*) ENTIER *de base dans* MODULE

opera : *fonct*(SYMBOLE U MODULE) PLACE *de base dans* MODULE

PLACE : *M.S.I.*

estdonnée : *op* (SYMBOLE, MODULE) BOOL *de base dans* MODULE
donnée : *fonct*(MODULE)PLACE *de base dans* MODULE
estparamètre : *op* (SYMBOLE, MODULE) BOOL *de base dans* MODULE
param : *fonct*(MODULE)PLACE *de base dans* MODULE
ajoutsys:*fonct*(SYMBOLE,ENTIER, PLACE, MODULE*) MODULE*

ajoutmod : *fonct*(SYMBOLE, MODULE*) MODULE*
cocher : *fonct* (MODULE,SYMBOLE) MODULE *de base dans* MODULE

alors ajoutsys(id, j, param(sm'_j), sm')
si id estdonnée
alors ajoutsys(id, j, donnée(sm'_j),s
sinon ajoutmod(id, sm')

sm' = chval($\overline{sm}$, j, cocher($\overline{sm}_j$, id))
<< j, id >> = *choix dans* dom($\overline{sm}$), prof($\overline{sm}_j$)
tq ¬ estcoché(id, $\overline{sm}_j$) ∧
(opération(id, $\overline{sm}_j$)∧ ∃S : S.I. *tq* debase(id, S) ⊃ estcoché(S, $\overline{sm}$))
ttcoché = ∀j ∈ dom(sm), ∀id ∈ prof(sm_j), estcoché(id, sm_j)

§ sm = < minit >
minit = << ⊀ caractérisation informelle de r et des S.I. apparaissant dans son profil ⊁ profil de r et divers S.I. non primitifs apparaissant dans ce profil
résultat r >>
r = ⊀ résultat du problème donné ⊁
§ ttcoché = *faux*

tcoché : indique si un symbole est coché dans un module nit : module initial. Il caractérise informellement le résultat r et décrit son profil est l'identificateur (éventuellement n-uplet d'identificateurs) qui représente le résultat	estcoché : *fonct* (SYMBOLE,MODULE) BOOL *de base dans* MODULE minit : MODULE r : SYMBOLE	
ajoutop(id, j, place, sm') = sm		
cer : introduit un morceau de module (lexique, profil, énoncé formel) à une place donnée d'un module donné : spécification formelle de id : profils des intermédiaires introduits par cette spécification : lexique des nouveaux intermédiaires	placer : *fonct*(PLACE, MODULE, MODULE) MODULE *de base dans* MODULE eid : $\mathfrak{P}^{S}$ (EFORMEL) pid : $\mathfrak{P}^{S}$ (PROFIL) lid : $\mathfrak{P}^{S}$ (EINFORMEL)	sm : *si* $\exists S_1 \cdot S_q\ S_{q+1}$ *tq* profil(id) = $S_1 .. S_q \rightarrow S_{q+1}$ avec $q \geq 1$ *alors* sm = ajoutop(y, j, place, sm_1) sm_1 = chval(sm', j, m) m = placer(place, sm'_j, « ∅, ∅, id $x_1..x_q$ = y *tq*») $x_1,...,x_q$, y = *choix dans* SYMBOLE *sinon* sm = chval(sm', j, m) m = placer(place, sm'_j, « lid,pid,eid ») eid = ‡choix des intermédiaires $id_1..id_r$ de leurs profils et de la spécification formelle de id à partir d'eux‡ pid = ‡profils des $id_1..id_n$ qui ne sont pas déjà dans le profil de sm', on précise s'ils sont de base ou donnés. Introduction des S.I. apparaissant dans ces profils qui ne sont pas primitifs et qui n'apparaissent pas déjà dans le module ‡ lid = ‡ caractérisation informelle des nouveaux intermédiaires et des nouveaux S.I. ‡ .
ajoutsys (id, j, place, sm') = sm		
smp_k : suite de module déduite de sm' en ajoutant les sortes de id et un certain nombre des opérations qui lui sont attachées. Le dernier terme est sm.	smp_k : MODULE* $mcomp_k$: BOOL	sm = *dernier* smp_k *pour* k *dans* 1 → *jqa* mcon
smp		
specifsys : introduit dans un module d'une suite de modules la spécification d'un nouveau S.I. id : ses sortes et son invariant, ou alors l'esquive à l'aide d'opérations de composition de S.I. (cf §5 et §7 du ch. 1.2).	specifsys : *fonct* (SYMBOLE, ENTIER, PLACE, MODULE*) MODULE*	smp = ajoutop (f, j, place, sm1) <<ℓ, f >>= *choix dans* dom($\overline{smp}$, prof($\overline{smp}_\ell$) *tq* ¬ estcoché(f, $\overline{smp}_\ell$)∧ operation f($\overline{smp}_\ell$) ∧ debase(f, $\overline{smp}_\ell$, id) sm1 = chval($\overline{smp}$, ℓ , cocher($\overline{smp}_\ell$, f)) $\mathfrak{S}$ smp = specifsys(id, j, place, sm') mcomp = ∀ ℓ ∈ dom(smp), ∀ f ∈ prof(smp) operation(f, smp_ℓ) ∧ debase(f, smp_ℓ ,id) estcoché(f, smp_ℓ) $\mathfrak{S}$ mcomp = ∀ ℓ ∈ dom(sm'), ∀ f ∈ prof(sm'), opération(f, sm'_ℓ) ∧ debase(f, sm'_ℓ, id estcoché(f, sm'_ℓ)

smq		
creermod : opération de création d'un nouveau module. Introduit ses sortes, son invariant et ses paramètres.	creermod : *fonct* (SYMBOLE) MODULE	smq = ajoutop(f, bs(sm1), opera(id), sm1) <<ℓ, f >> = *choix dans* dom($\overline{smq}$, prof($\overline{smq}_\ell$) *tq* ¬ estcoché(f, $\overline{smq}_\ell$) ∧ opération (f, smq_ℓ) ∧ debase(f, $\overline{smq}_\ell$, id) sm1 = chval($\overline{smq}$, ℓ, cocher($\overline{smq}_\ell$, f)) § smq = sm' ~ creermod(id) fimod = ∀ ℓ ∈ dom(smq), ∀ f ∈ prof(smq_ℓ) opération(f, smq_ℓ) ∧ debase(f, smq_ℓ ,id) = estcoché(f, smq_ℓ) § fimod = ∀ ℓ ∈ dom(sm'), ∀ f ∈ prof(sm'_ℓ), opération(f, sm'_ℓ) ∧ debase(f, sm'_ℓ, id estcoché(f, sm'_ℓ)
ajoutmod(id, sm') = sm		
smq_k : suite de modules dont le dernier élément est associé à id : il comprend la définition des sortes de id et celle d'un certain nombre de ses opérations.	smq_k : MODULE* $fimod_k$: BOOL	sm = *dernier* smq_k *pour* k *dans* 1 → *jqa* $fimod_k$

MODULE : *U.S.I.*		
Un module est formé de trois parties : - Le lexique qui est un ensemble de caractérisations informelles - Le profil qui est un ensemble de s-descriptions de profils (cf ch. 1.2 § 5.1.1) - L'énoncé formel qui est la partie énoncé - soit une s-description de problèmes (cf ch 1.3 § 2.3) - soit une s-description de S.I. (cf ch 1.2 § 5) estcoché : indique si un identificateur est coché dans un module. opération : indique si un identificateur désigne une opération dans un module. ⋮ etc : autres opérations de bases dans MODULE	estcoché : *fonct* (SYMBOLE, MODULE) BOOL opération : *fonct* (SYMBOLE, MODULE) BOOL ⋮ etc	MODULE : << lex : ℱ (EINFORMEL), prof : ℱ (PROFI enf : ℱ (EFORMEL) >>

Figure 3 : Version incomplète d'un méta-algorithme de spécification.

2.3.- Dernier exemple de spécification.

Pour mieux faire comprendre le méta-algorithme précédent décrivons son déroulement sur un exemple.

Exemple 2 : Etat d'avancement de livraisons sur commandes en cours.

La formalisation de cet exemple (cf ch. 1.1 § 1.1 i)) nécessite celle des données et des résultats. A la description donnée par l'énoncé initial et l'annexe du chapitre 1.1 nous préférons la schématisation suivante :

a) données

- le fichier cde-en-cours peut être décrit de la façon suivante :

i) structure hiérarchique : elle précise la forme de chaque enregistrement (figure 4). On précise ainsi que le troisième champ est conditionnel et dépend du type de l'enregistrement.

ii) structure linéaire : elle exprime les règles de succession des différents enregistrements ; elle est nécessaire car le fichier est hétérogène ;

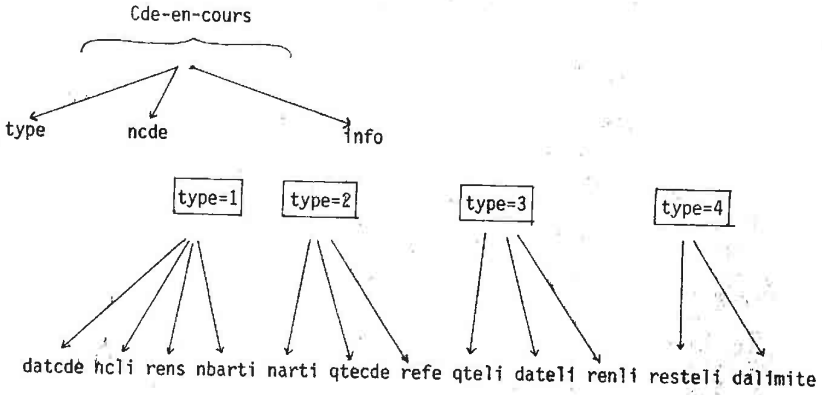


Figure 4 : enregistrements de cde-en-cours

cde-en-cours = { e-cours de type 1, {e-cours de type 2, {e-cours de type 3}* , e-cours de type 4 }+ }*

iii) propriétés : hétérogène;trié (cde-en-cours / ncde ; narti ; dateli)

On décrit ainsi un certain S.I. FICHCOM que nous précisons en termes de s-descriptions dans la suite.

Une description équivalente aurait pu être fournie par une double grammaire [WIN, 75] ou par une c-grammaire munie d'attributs [KNU, 68] , [LOR, 74] , [LIV, 78] exprimant que les enregistrements concernant une même commande sont rangés de manière connexe, que ceux concernant les livraisons d'un même article le sont aussi etc...

- le fichier livraison est homogène. Il suffit d'en écrire sa structure hiérarchique

i) structure hiérarchique (figure 5)

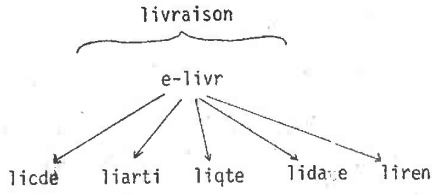


Figure 5

ii) propriétés :
- sur cartes
- homogène
- trié (livraison / licde ; liarti ; lidate)

b) Résultats :

- Le fichier cde-maj prend ses valeurs sur le même ensemble de valeurs que cde-en-cours : dans une description formelle on peut se limiter à écrire

cde-maj : FICHCOM

- Le fichier état-des-cdes peut être décrit par :

i) structure hiérarchique (figure 6) :

Figure 6 :

- structure linéaire

état-des-cdes = $\{\text{entête}, \{\text{article}\}^+\}^*$

- propriétés : hétérogène
  trié (état-des-cdes / ncdei ; num)
  sur imprimante

On peut concevoir les considérations précédentes comme une première phase de la spécification. C'est surtout le support graphique des figures précédentes qui se révèle être utile.

c) Déroulement du méta-algorithme de spécification.

Nous décrivons les choix successifs conduisant à la spécification formelle représentée par les figures 7 et 8.

- *but* détermination du résultat

+ *choix*
  - *résultat r* = cde-maj, état-des-cdes *parce que* ce sont les sortie
  - *profil de r* = FICHCOM, FICHETAT *parce que* noms arbitraires de fi
  - *lexique* = "cf figure 7 premières lignes du lexique jusqu'à description de FICHETAT"

parce que-spécification informelle des sortie
- description de cde-en-cours
- description de la stru ture sur imprimante

Spécification du problème : "état d'avancement des livraisons"

On demande de créer cde-maj à partir de cde-en-cours :
- en ajoutant les livraisons d'articles (enregistrements de type 3)
- en modifiant l'enregistrement récapitulatif (de type 4)

On demande d'imprimer :
- pour chaque commande un en-tête
- pour chaque article un récapitulatif des mouvements

FICHCOM : fichier des commandes, sa description est donnée par la figure 4

FICHETAT : fichier des états de sortie, sa description est donnée par la figure 6

trans1 : prédicat qui précise que θ1 "recopie" dans cde-maj tous les enregistrements de cde-en-cours qui ne sont pas de type 4

trans2 : prédicat qui indique que θ2 "recopie" dans cde-maj sous forme d'enregistrements de type 3 les enregistrements de livraison

LIVRAISON : fichier de livraisons. Sa description se trouve sur la figure 5

-cde-maj : FICHCOM
- état-des-cdes : FICHETAT
- FICHCOM : S.I.
- FICHETAT : S.I.
- trans1 : *fonct* (FICHCOM,FICHCOM, [FICHCOM → FICHCOM]) BOOL
- trans2 : *fonct* (FICHCOM,LIVRAISON, [LIVRAISON → FICHCOM])BOOL
- LIVRAISON : S.I.
- inj : *fonct*($\mathcal{P}$(ENTIER), $\mathcal{P}$(ENTIER)) $\mathcal{P}$ ([ENTIER → ENTIER])
- cde-en-cours : FICHCOM
- livraison : LIVRAISON
- trs1 :*fonct* (FICHETAT, FICHCOM, [FICHCOM → FICHETAT]) BOOL
- trs2 . *fonct*(FICHETAT, FICHCOM, [FICHCOM → FICHETAT]) BOOL

résultat cde-maj, état-des-cdes
cde-maj *tq*
  ∃ θ1 ∈ inj(dom(cde-en-cours), dom(cde-maj)
  ∃ θ2 ∈ inj(dom(livraison), dom(cde-maj))
  tq imag(θ1) ∩ imag(θ2) = ∅ ∧
    imag(θ1) ∪ imag(θ2) = dom(cde-maj) ∧
    trans1(cde-maj, cde-en-cours, θ1) ∧
    trans2(cde-maj, livraison, θ2)

état-des-cdes *tq*
  ∃ γ 1 ∈ inj(com, dom(état-des-cdes))∧
    com = {i ∈ dom(cde-maj) *tq* type(cde-maj, i) = 1) ,
  ∃ γ 2 ∈ inj(art, dom(état-des-cdes)) ∧
    art = {i ∈ dom(cde-maj) *tq* type(cde-maj, i) = 2) ,
  tq imag(γ1) ∩ imag(γ2) = ∅ ∧
    imag(γ1) ∪ imag(γ2) = dom(état-des-cdes)
    trs1(état-des-cdes, cde-maj, γ1) ∧
    trs2(état-des-cdes, cde-maj, γ2)

donnée
cde-en-cours
livraison
fdonnée

inj : associe à deux parties de ENTIER l'ensemble des injections de la première dans la deuxième.

cde-en-cours : fichier donné des commandes en cours

livraison : fichier donné des livraisons

trs1 : prédicat qui indique que les en-têtes de état-des-cdes sont en bijection avec les enregistrements de type 1 de cde-maj (i.e avec les commandes)

trs2 : prédicat qui indique que les articles de état-des-cdes sont en bijections avec les enregistrements de type 4 de cde-maj (i.e avec les récapitulatifs de chaque article)

j : indice dans f de l'enregistrement contenant le numéro d'article correspondant à l'enregistrement d'indice θ(i)

indnuart : fonction qui associe à un enregistrement de type 3 ou 4 de FICHCOM l'indice de l'enregistrement contenant le numéro d'article correspondant.

numcom : associe à tout enregistrement de type article de f le numéro de commande de l'enregistrement de type entête correspondant

h:[ENTIER → ENTIER]
x : ENTIER
y : ENTIER

indnuart : *fonct* (FICHCOM, ENTIER) ENTIER
de base dans FICHCOM

numcom : *fonct* (FICHETAT, ENTIER) ENTIER
de base dans FICHETAT

inj(a, b) = c
tq ∀h ∈ c , dom(h) = a ∧ ∀x, y ∈ a
h(x) = h(y) ⊃ x = y
si a ⊂ b

transl (f, g, θ) = b
tq b = ∀ i ∈ dom(g), type(g_i) ≠ 4 ⊃ g_i = f_θ
si dom(θ) = dom(g)

trans2(f, g, θ) = b
tq b = ∀ i ∈ dom(g),
$f_{\theta(i)}$ = <<3, licde(g_i), liqte(g_i), lidate(g_i), liren(g_i) >> ∧
∃ j *tq* j = indnuart(f, θ(i)) ∧ narti(f_j) = liarti(g_i)
si dom(θ) = dom(g)

trs1 *tq*
trs1(f, g, θ) = ∀ i ∈ dom(θ),
$f_{\theta(i)}$ = <<"entête", rens(g_i), nede(g_i) >>
si dom(0) = {i ∈ dom(g) *tq* type(g_i) = 1}

trs2 *tq*
trs2(f, g, θ) = ∀ i ∈ dom(θ), ∃ j *tq*
j = indnuart(g, i)
$f_{\theta(i)}$ = << narti(g_j), qtecde(g_j), qtecde(g_j)- resteli(g_i), i-j+1, resteli(g_i), dalimite(g_i) >> ∧
ncde(g_i) = numcom(f, θ(i))

Figure 7 : Etat de livraison des commandes en cours

FICH COM : *S.I.*

type : associe à chaque enregistrement son type : 1, 2, 3 ou 4

ncde : numéro de commande

info : information décrite par l'enregistrement ; elle peut prendre 4 formes différentes.

RENCOM : renseignements sur la commande cf figure 4

RENART : renseignements sur l'article cf figure 4

RENLIV : renseignements sur la livraison cf figure 4

RECAP : récapitulatif cf figure 4

invariant : prédicat qui permet de restreindre l'ensemble des fichiers de FICHCOM en précisant que la structure linéaire est correcte, qu'ils sont triés par numéro de commande, numéro d'articles et date de livraisons croissants et que le recapitulatif est correct

trié : prédicat indiquant que le fichier est trié

slcor : prédicat indiquant que le fichier possède une structure linéaire correcte

- RENCOM : *S.I.*
- RENART : *S.I.*
- RENLIV : *S.I.*
- RECAP : *S.I.*
- f : FICHCOM
- trié : *fonct* (FICHCOM)BOOL
- slcor : *fonct* (FICHCOM)BOOL
- recapcor : *fonct* (FICHCOM)BOOL

FICHCOM = << type : {1, 2, 3, 4} , ncde : ENTIER,
info :[RENCOM|(type(x)=1)]U[RENART |(type(x) = 2)] U [RENLIV|(type(x) = 3)]U RECAP|(type(x) = 4)] >> *

invariant
trié(f) ∧ slcor(f) ∧ recapcor(f)

opérations
trié *tq* trié(f) = ∀ i, j ∈ dom(f) i ≤ j ⊃
(ncde(i) < ncde(j)) ∨
(ncde(i) = ncde(j))
∧ type(f_i) = 2 ∧ type(f_j) = 2 ⊃ narti(info(f_i)) ≤ narti(info(f_j))
∧ type(f_i) = 3 ∧ type(f_j) = 3 ∧ (∀ k ∈ [i..j] type(f_k) = 3) ⊃ dateli(info(f_i)) ≤ dateli(info(f_j)))

slcor *tq* slcor(f) =
type(prem(f)) = 1 ∧ type(der(f)) = 4 ∧
∀ i ∈ [bi(f)..bs(f)-1]
(type(f_i) = 1 ⊃ type(f_{i+1}) = 2 ∧
type(f_i) = 2 ⊃ (type(f_{i+1}) = 3 ∨ type(f_{i+1})=
type(f_i) = 3 ⊃ (type(f_{i+1}) = 3 ∨ type(f_{i+1})=
type(f_i) = 4 ⊃ type(f_{i+1}) = 1)

recapcor : prédicat indiquant que chaque récapitulatif est correct indnuart : associe à un enregistrement de type 3 ou 4 de FICHCOM l'indice de l'enregistrement contenant le numéro d'article correspondant	indnuart : *foncť*(FICHCOM,ENTIER) ENTIER	recapcor *tq* recapcor(f) = ∀ j ∈ dom(f) type(f_j)=4 (∃i ∈ dom(f) *tq* type(f_i)= 2 ∧ ∀k ∈[i+1..j-1] type(f_k) = 3 ∧ resteli(info(f_j)) = qtecde(info(f_i)) - Σ qteli(info(f_k)) *pour* k *dans* i+1 → j-1 ∧ ∀ k ∈[i+1..j-1] dateli(info(f_k) ≤ dalimite(info(f_j indnuart (f, i) = j *tq* type(f_j) = 2 ∧ ∀k ∈ [j+1..i-1] type(f_k) = *si* type(f_i) = 2 ∨ type(f_1) = 3
RENCOM : *S.I.*		
DATE : doit être muni de ≤ datcde : date de la commande ncli : numéro de client rens : renseignements divers nbarti : nombre d'articles	- DATE : *S.I.*	RENCOM = << datcde : DATE, ncli : ENTIER, rens : CHAINE, nbarti : ENTIER >>
RENART : *S.I.*		
narti : numéro d'article qtecde : quantité commandée refe : libellé d'article		RENART =<< narti : ENTIER, qtecde : ENTIER, refe : CHAINE >>
RENLIV : *S.I.*		
qteli : quantité livrée dateli : date de livraison renli : renseignements divers		RENLIV =<< qteli : ENTIER, dateli : DATE, renli : CHAINE >>
RECAP : *S.I.*		
resteli : quantité restant à livrer dalimite : date limite de livraison		RECAP = << resteli : ENTIER, dalimite : DATE >>
DATE : *S.I.*		
j : jour m : mois a : année ≤ prédicat de comparaison de deux dates	d : DATE ≤ *op* (DATE, DATE) BOOL	DATE = << j : ENTIER, m : ENTIER, a : ENTIER >> *invariant* 1 ≤ j(d) ≤ 31 ∧ 1 ≤ m(d) ≤ 12 ∧ 1 ≤ a(d) *opérations* ≤ *tq* d1 ≤ d2 = (a(d1) < a(d2)) ∨ (a(d1) = a(d2) ∧ (m(d1) < m(d2) ∨ m(d1) = m(d2) ∧ j(d1) ≤ j(d2))
FICHETAT : *S.I.*		
type : associe à chaque enregistrement son type val : associe à chaque enregistrement sa valeur ENTETE : forme de l'en tête ARTC : renseignements concernant un article invariant : prédicat caractérisant les états corrects : triés par numéro de commande et numéro d'article et chaque entête et suivie d'articles	ENTETE : *S.I.* ARTC : *S.I.*	FICHETAT =<< type : "entête", "article" , val : [ENTETE!(type(x) = "entête") ; U [ARTC!(type(x) = "article")] >>* *invariant* [∀ i, j ∈ dom(f) , i ≤ j ⊃ (type(f_i) = "entête" ∧ type(f_j) ="entête" ⊃ ncdei(val(f_i)) < ncdei(val(f_j)) ∧ (type(f_i) = "article"∧ type(f_j) = "article" ∧ (∀k ∈ [i..j] type(f_k) = "article" ⊃ num(val(f_i))≤ num(val(f_j))) [∀i ∈ dom(f), type(f_i) = "entête"⊃type(f_{i+1}) = "article"] ∧

numcom : associe à tout enregistrement de type "article" le numéro de commande de l'enregistrement de type "entête" correspondant.	numcom : *fonct* (FICHETAT, ENTIER) ENTIER	[type(prem(f)) = "entête" ∧ type(dom(f)) = "article"] *opérations* numcom(f, i) = n *tq* ∃ j ∈ dom(f) *tq* [type(f_j) = "entête" ∧ ncdei(f_j) = n ∧ ∀k ∈ [j+1..i] type(f_k) = "article"] *si* type(f_i) = "article"

ENTETE : *S.I.*

ENTETE : << ent : CHAINE, ncdei : ENTIER >>

ARTC : *S.I.*

num : numéro d'article qart : quantité commandée tot : total des quantités livrées nblivr : nombre de livraisons qrest : quantité restant à livrer dlivr : date limite de livraison		ARTC = << num : ENTIER, qart : ENTIER, tot : ENTIER, nblivr : ENTIER, qrest : ENTIER, dlivr : DATE >>

LIVRAISON : *S.I.*

licde : numéro de commande liarti : numéro d'article liqte : quantité livrée lidate : date de livraison liren : renseignements divers λ : prédicat exprimant que le fichier est trié.	λ : *fonct* (LIVRAISON)BOOL	LIVRAISON = << licde : ENTIER, liarti : ENTIER, liqte : ENTIER, lidate : DATE, liren : CHAINE >>*| λ λ *tq* λ (ℓ) = ∀ i ∈ [bi(ℓ).. bs(ℓ-1)] , licde(ℓ_i) < licde(ℓ_{i+1}) ∧ (licde(ℓ_i)=licde(ℓ_{i+1}) ∧ (liarti(ℓ_i) < liarti(ℓ_{i+1}) ∨ (liarti(ℓ_i)= liarti(ℓ_{i+1}) ∧ lidate(ℓ_i) ≤ lidate(ℓ_{i+1})

Figure 8 : S.I. du problème "état des livraisons sur commandes en cours"

- *rangement* de r, des profils, des lexiques *dans* module principal
on vient ainsi de définir ℘ sm, cf figure 3
- *but* spécification d'un intermédiaire non coché
- *choix* « 1, FICHCOM » *parce que* c'est le profil du premier résultat
- *cocher* FICHCOM *dans* module principal
- *but* ajouter un module FICHCOM
- *but* créer un module FICHCOM
on utilise maintenant l'opération "creermod" qui n'est pas décrite formellement
+ *choix* FICHCOM = "cf figure 8 jusqu'à la description de l'invariant" *parce que* on se refère à la description de la figure 4, au fait que le fichier doit avoir une structure linéaire correcte, qu'il doit être trié, que le récapitulatif doit être cohérent avec ce qui précède. Le fichier n'est pas paramétré.

- *adjonction du module* FICHCOM
- *but* spécification d'un intermédiaire non coché
choix possibles : cde-maj
état-des-cdes
FICHETAT
RENCOM
RENART
RENLIV
RECAP
trié
slcor
recapcor
dans module principal
dans module FICHCOM

- *choix* << 2, trié >> *parce que* on reste au niveau de FICHCOM
- *cocher* trié *dans* FICHCOM
- *but* ajouter l'opération trié *dans* FICHCOM
- *placer* trié *tq* *dans* la partie opération de FICHCOM
- *but* ajouter l'opération trié(f) *dans* FICHCOM
+ *choix* - *de la spécification formelle* trié(f) = "cf figure 8"
  parce que invention et culture
  - *des profils intermédiaires* néant *parce que* pas d'intermédiaires
  - *des spécifications informelles intermédiaires* néant
  parce que pas d'intermédiaires
- *placer* les caractérisations de trié(f) *dans* la partie opération de FICHCOM
- *but* spécification d'un intermédiaire non coché
- *choix* << 2, slcor >> *parce que* on reste au niveau de FICHCOM

etc...

La description complète de cette spécification est fastidieuse mais permet de mettre en évidence les points cruciaux de ce travail. Notons en particulier l'existence de deux types de choix. Les uns sont relatifs aux nouveaux identificateurs à spécifier et pour lesquels on devrait pouvoir développer des heuristiques assez simples (du genre : quand on est dans un module, spécifier toutes les opérations introduites à son niveau). Les autres beaucoup plus fondamentaux (précédés du symbole "+") nécessitent une compréhension profonde du problème.

Signalons pour terminer qu'une autre approche aurait pu consister à introduire la structure logique des données comme intermédiaires, de façon analogue à ce qui est fait dans l'exemple 1. La spécification du résultat en partant de cette structure logique aurait été plus simple mais il aurait fallu alors spécifier la structure logique à l'aide de la structure physique. Cette démarche aurait été un peu plus longue mais aurait simplifié la résolution ultérieure puisque cette dernière peut être fondée sur l'introduction de la structure logique (cf ch 1.3 § 4) [FIN, 79].

3.- BREVE CONCLUSION.

Le problème de la spécification d'un problème informatique est apparu, paradoxalement, très récemment. On a trop longtemps essayé (et on essaye encore trop souvent), de résoudre un problème sans le définir précisément.

La *structure* de l'énoncé cherché est le principal guide dans l'activité de spécification. C'est à partir d'elle que l'on peut préciser une *méthode de spécification* permettant au spécifieur de ne se préoccuper que des choix cruciaux et le guidant dans la construction du problème formel. La description d'une telle méthode est un préalable fondamental à la construction de systèmes interactifs de spécification.

DEUXIEME PARTIE

explicitation
de problèmes

CHAPITRE 2.1

- PROBLÈME DE L'EXPLICITATION DES PROBLÈMES

L'objectif de cette deuxième partie est de définir des méthodes permettant de transformer une spécification initiale d'un problème, souvent plus descriptive que constructive, en une spécification explicite à laquelle pourra être appliquée une méthode de résolution.

Dans ce but il convient tout d'abord de préciser le résultat attendu : qu'est ce qu'une spécification explicite d'un problème ? quelles en sont les propriétés ? dans quelle langage l'exprimer ? Puis de la comparaison entre spécification initiale (en général implicite) et spécification explicite essayer de déduire une méthode d'explicitation.

L'objectif final étant d'obtenir un programme, il est raisonnable ici de s'intéresser essentiellement aux *solutions calculables*. Cette remarque a pour corollaire, en reprenant les notations du paragraphe 2 du chapitre 1.3, la recherche de formes de systèmes d'information $SDi[\varepsilon_S]$ pour lesquelles l'obtention d'un théorème $r \equiv u$ est simple. Ceci pourrait nous inciter à ne considérer que le problème, envisagé au chapitre 2.2, de l'explicitation des S.I.. Cependant, un cas particulièrement simple d'explicitation est celui des énoncés stricts puisque les symboles "inconnus" sont o-aires.

C'est la raison pour laquelle nous faisons une étude détaillée d'une forme importante d'énoncés explicites : les énoncés récurrents (ch 2.3). De plus, il est réconfortant de constater que les techniques d'explicitations sont analogues dans le cas des énoncés et dans celui des S.I., ce qui permet de minimiser, d'un point de vue formel, la distinction entre explicitation de S.I. et explicitation d'énoncé.

Comme dans la partie 1, nous commençons , dans ce premier chapitre, par étudier sur quelques exemples la forme des définitions explicites tout en essayant de les construire systématiquement (§ 1). Ceci permet de dégager quelques idées fondamentales sur l'activité d'explicitation : quels langages et supports formels doit-on utiliser ? Quelles stratégies peut-on suivre ? de quels outils dispose-t-on ? (§ 2) .

De cette étude nous déduisons l'articulation de la suite de cette deuxième partie.

1.- PREMIERES IDEES SUR L'EXPLICITATION D'UN PROBLEME.

1.1.- Forme d'une définition explicite

Le rôle d'une définition explicite est de permettre le calcul de solutions, c'est-à-dire la démonstration de théorèmes de la forme $r \equiv u$.

a) Cas de symboles o-aires.

Avec la remarque précédente, si f est un symbole o-aire , il est raisonnable de décrire une *définition explicite* de f comme une formule atomique du genre :

$$f \equiv u \quad \text{où } u \text{ est un terme.}$$

Le calcul est alors immédiat.

Par exemple $x \equiv a/z$, $y \equiv b/z$, $z \equiv pgcd(a, b)$ sont des définitions explicites de symboles x, y et z dans un S.I. contenant ENTIER et pgcd.

Mais lorsque f n'est pas un résultat mais un simple intermédiaire, il n'est utilisé dans un calcul que comme argument de certaines opérations g. Ceci permet de généraliser la forme d'une définition explicite. Une *définition explicite composée* de f pour un ensemble Θ_p d'opérations est une conjonction de formules du type :

$$\bigwedge_{g \in \Theta_p} g(x_1, \ldots, x_{i-1}, f, x_{i+1}, \ldots, x_q) \equiv u$$

où g est une opération q-aire, u un terme et les x_i des variables. Une telle formule est encore une définition explicite de f, chaque égalité est une définition explicite simple.

Un cas particulièrement important est celui où f est un symbole de profil

$$[\overline{S_1} \times \ldots \times \overline{S_q} \rightarrow \overline{S_{q+1}}] \qquad \text{(cf ch 1.2 § 7.2)}$$

Il peut alors être caractérisé par les deux définitions simples :

$$appl(f, x_1, \ldots, x_q) \equiv u$$
$$dom(f) \equiv v$$

Il en est de même pour une suite s qui peut être caractérisée par :

$$appl(s, i) \equiv u$$
$$bi(s) \equiv v_1$$
$$bs(s) \equiv v_2$$

b) Cas de symboles q-aires.

Si f est un symbole q-aire, une définition explicite de f est une formule $f\, x_1 \ldots x_q \equiv u$

où u peut contenir des occurences de f .

Par exemple

$pgcd(x, y) \equiv$ *si* $x \equiv 0$ *alors* y *sinon* $pgcd(y, reste(x, y))$

$reste(x, y) \equiv$ *si* $x < y$ *alors* y *sinon* $reste(x, x-y)$

sont des exemples de définitions explicites *récursives*.

Dans ces définitions "-" et "reste" doivent simplifier, en un certain sens, les couples (x, y) ; ils seront appelés *simplificateurs* du système d'information considéré.

Mais dans certains cas on ne souhaite pas donner une même définition d'un symbole fonctionnel f pour tous les termes $u_1 \ldots u_q$ de sa source.

Plutôt que d'utiliser des implications, on peut préférer prendre la conjonction de définitions plus restreintes. Ce qui conduit à étendre la notion de définition explicite :

Si f est un symbole q-aire, une définition explicite simple de f est une formule

$$f\, u_1 \ldots u_q \equiv v$$

où $u_1 \ldots u_q$, v sont des termes.

Par exemple ascendant(père(a), a) ≡ vrai,
ascendant(a, père(b)) ≡ ascendant(a, b)
sont des définitions explicites. Ici "père" doit permettre de construire tous les termes du S.I. considéré : c'est un *constructeur* du S.I. . Il faut essayer de caractériser le cas où la conjonction de ces définitions *caractérise complètement le symbole* f ; le concept de forme normale introduit au chapitre 2.2 répond à cette préoccupation (ch 2.2, § 3.3).

Un énoncé explicite est une conjonction de définitions explicites. Un S.I. "explicite" a pour axiomes propres des définitions explicites (si on néglige les préconditions). De manière intuitive le problème de l'explicitation consiste à construire des définitions explicites (tant pour des énoncés que pour des S.I.). Pour mieux le préciser commençons par étudier quelques exemples.

1.2.- Quelques exemples d'explicitation.

Exemple 1 :

Dans l'évaluation d'une expression post-fixée (ch 1.4 § 1.3 exemple 1), la fonction "val" a été caractérisée de manière récursive puisque les expressions étaient définies récursivement à l'aide de constructeurs et de simplificateurs.

Exemple 2 :

Le problème du baguenaudier (ch 1.1, § 1.1 o)) s'énonce comme la recherche d'une définition récursive de la fonction faisant passer de l'état initial (la navette passe dans tous les anneaux) à l'état final (la navette est à l'extérieur de tous les anneaux).

Commençons par formaliser le problème en définissant un baguenaudier à n anneaux comme une suite b définie sur [1..n], d'où la s-description de la figure 1.

BAG : *S.I.*

Un baguenaudier à n éléments est une suite à n éléments sur {0, 1} La seule opération est l'inversion qui change la valeur en un point. On ne peut l'appliquer en i que si le i-1ème élément est 1 et que les i-2 premiers sont égaux à 0.	n : ENTIER b : BAG invers : *fonct* (BAG,ENTIER) ENTIER	*param* n *tq* n ⩾ 1 *fparam* BAG = {0, 1}* *invariant* dom(b) = [1..n] *opérations* invers *tq* invers(b,i) = chval(b,i,1 *si* i > 1 ⊃ [∀j(1 ⩽ j ⩽ i-2 ⊃ b ∧ b_{i-1} ≡ 1]

Figure 1 : s-description du S.I.baguenaudier

Le problème du baguenaudier peut alors s'énoncer informellement.

But P1 : Etant donné une fonction mazt (mise à zéro totale) défini par la s-description :
mazt : *fonct* (BAG) BAG
mazt(b) = b'
tq ∀j(1 ⩽ j ⩽ bs(b') ⊃ b'_j = 0) ∧ bs(b') = bs(b)
si ∀j(1 ⩽ j ⩽ bs(b) ⊃ b_j = 1)
On demande d'en donner une définition récursive à l'aide de invers.

De manière plus formelle, on cherche un terme $\mathcal{C}$ ne contenant que les symboles *si* ... *alors* ... *sinon* , invers et des symboles x et h tel que :

pré mazt(b) ⊃ mazt(b) = $\mathcal{C}$[mazt/h, b/x]

où *pré* u désigne la précondition de u, la substitution étant étendue ici aux symboles fonctionnels quelconques.

Notons que nous cherchons ici à expliciter un axiome d'un S.I. caractérisant un symbole unaire. On pourrait se ramener à un résultat o-aire en cherchant, comme dans [PAR, 77] la suite des anneaux inversés.

Une idée d'explicitation consiste à *généraliser* le problème considéré en faisant apparaitre une nouvelle fonction plus facile à définir récursivement que mazt et dont mazt est un cas particulier. Classiquement cette généralisation, embedding en anglais [BAU, 78 b] ,[DAR, 78] , consiste à introduire un paramètre supplémentaire sur lequel se fait l'induction.

Par exemple ici on peut introduire de façon naturelle un indice k tel que les k premières valeurs d'un baguenaudier soient 0 .

On considère donc un nouveau problème et une nouvelle inconnue maz caractérisés par :

But P2 : Etant donné la fonction maz (mise à zéro) définie par la s-description

maz : *fonct* (BAG, ENTIER) BAG

maz(b, k) = b'

tq $bs(b') = bs(b) \wedge tr(b', k+1, bs(b')) \equiv tr(b, k+1, bs(b)) \wedge \forall j(1 \leq j \leq k \supset b'j \equiv 0)$

si $1 \leq k \leq bs(b) \wedge \forall j(1 \leq j \leq k \supset bj \equiv 1$

On demande d'en donner une définition récursive à l'aide de invers

Le problème initial est alors résolu en posant :

mazt (b) = maz(b, bs(b))

Nous nous intéressons maintenant à P_2 qui est donc un sous-but du but initial.

Pour simplifier les notations introduisons les prédicats :

rest(b, b', k) $\equiv$ (bs(b') $\equiv$ bs(b) $\wedge$ tr(b', k+1, bs(b')) $\equiv$ tr(b,k+1,bs(b)))

zéro(b, k) $\equiv \forall j(1 \leq j \leq k \supset b_j \equiv 0)$

Alors :

(1) $\vdash_{\overline{BAG}}$ *pré* maz(b, k) $\supset$ maz(b, k) $\equiv$ b' $\leftrightarrow$ rest(b, b', k) $\wedge$ zéro(b', k)

(2) $\vdash_{\overline{BAG}}$ *pré* maz(b, k-1) $\supset$ maz(b, k-1) $\equiv$ b' $\leftrightarrow$ rest(b, b', k) $\wedge$ $b'_k \equiv b_k \wedge$ zéro(b', k-1)

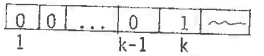
Dans la suite nous négligeronsles $\overline{BAG}$ comme on le fait en mathématiques

But P2' : Exprimer b' = maz(b, k) en fonction de maz(b, k-1) lorsque $k \geq 2$.

Possibilités:

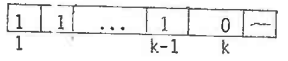
1) Introduire b1 = maz(b, k-1) puis essayer de définir b' à l'aide de b1

C'est-à-dire de manière imagée : t1



2) Introduire b1 tel que $b1_k \equiv 0 \wedge tr(t1, 1, k-1) \equiv tr(b, 1, k-1) \wedge rest(b,b1,k)$

C'est-à-dire ici b1 :



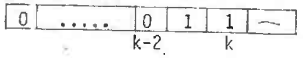
Le choix entre ces deux possibilités dépend de la "facilité" de construire, dans le premier cas la fonction faisant passer de b1 à b', dans le deuxième cas la fonction faisant passer de b à b1 .

Choix de (1).

Dans le premier cas, on passe de b1 à b' en inversant $b1_k$, ce qui nécessite (avec *pré* invers(b)) : $b1_{k-1} \equiv 1 \vee \forall j(1 \leq j \leq k-2 \supset b1_j \equiv 0)$

Pour s'y ramener on doit donc introduire un nouvel intermédiaire b2 tel que *pré* invers (b2) soit vrai.

b2 peut être schématisé par :



c'est-à-dire que b2 est spécifié par : $zéro(b2, k-2) \wedge b2_{k-1} \equiv 1 \wedge b2_k \equiv 1 \wedge$ rest(b, b2, k)

On constate donc que b2 ≡ maz(b, k-2). Ainsi b1 apparait comme un mauvais intermédiaire puisque pour définir b' à partir de lui il faut repasser par b2, plus simple à obtenir directement à partir de b.

Nous ne poursuivrons pas cette solution, car on la retrouve en développant l'autre possibilité.

Choix de (2).

Dans le deuxième cas, *pré* maz(b, k) ⊃

$b' \equiv maz(b, k) \leftrightarrow \exists b1\ tq\ b1_k \equiv 0 \wedge tr(b1, 1, k-1) \equiv tr(b, 1, k-1) \wedge rest(b, b1, k) \wedge b' \equiv maz(b1, k-1)$

Pour construire b1 à partir de b il faut inverser b_k et donc introduire deux intermédiaires :

- b3 *tq* zéro(b3, k-2) $\wedge\ b3_{k-1} \equiv 1 \wedge \wedge b3_k \equiv 0 \wedge$ rest(b, b3, k)

soit schématiquement b3

0	0	...	0	1	0	~~~
1			k-2	k-1	k	

- b2 *tq* zéro(b2, k-2) $\wedge\ b2_{k-1} \equiv 1 \wedge b2_k \equiv 1 \wedge$ rest(b, b2, k)

(cf choix 1)).

Ainsi, sous l'hypothèse *pré* maz(b, k),

(3) $b' \equiv maz(b, k) \leftrightarrow \exists b1, b2, b3\ tq$

$b' \equiv maz(b1, k-1)$

$\wedge\ \forall j(1 \leqslant j \leqslant k-2 \supset b1_j \equiv 1) \wedge tr(b1, k-1, n) \equiv tr(b3, k-1, n)$

$\wedge\ b3 \equiv invers(b2, k)$

$\wedge\ b2 \equiv maz(b, k-2)$

But P3 : Définir récursivement b1 à partir de b3, c'est à dire expliciter la formule :

$\forall j(1 \leqslant j \leqslant k-2 \supset b1_j \equiv 1) \wedge tr(b1, k-1, n) \equiv tr(b3, k-1, n)$

Posons un(b, k) ≡ $\forall j(1 \leqslant j \leqslant k \supset b_j \equiv 1)$

Pour définir b1 à partir de b3 il faut, intuitivement, "mettre à 1" les k-2 premiers éléments de b3.

Choix :

Introduction d'une nouvelle fonction maun caractérisée par la s-description suivante :

maun : *fonct* (BAG, ENTIER)BAG

maun(b, k) = b'

tq bs(b') ≡ bs(b) ∧ tr(b', k+1, bs(b')) ≡ tr(b, k+1, bs(b) ∧ un(b', k)

si 1 ⩽ k ⩽ bs (b) ∧ zéro(b, k)

Cette fonction répond au but P3. De plus, avec cette définition, l'explicitation de maz est terminée puisque (3) devient :

(4) $b' \equiv maz(b, k) \leftrightarrow \exists b1, b2, b3\ tq\ b' \equiv maz(b1, k-1) \wedge b1 \equiv maun(b3, k-2) \wedge b3 \equiv invers(b2, k) \wedge b2 \equiv maz(b, k-2)$

ce qui , si on suppose l'explicitation de "maun" acquise, répond au but P'_2

But P''_2 : Définir explicitement maz(b, k) pour k = 0 et k = 1

Avec la s-description de maz, maz(b, 0) ≡ b et maz(b, 1) ≡ invers(b, 1)

Si l'on veut éviter un cas particulier pour k=1 , il suffit de généraliser (4) à k = 1 en posant maz(b, k) ≡ b si k ⩽ 0

But P4 : Expliciter maun. On peut alors soit refaire le même raisonnement, soit invoquer un argument de symétrie entre 0 et 1 pour obtenir finalement *le système d'équations récursives* suivant :

(5)
$$\begin{cases} maz(b, k) \equiv \underline{si}\ k \leqslant 0\ \underline{alors}\ b\ \underline{sinon}\ maz(maun(invers(maz(b,k-2), k), k-2), k-1) \\ maun(b, k) \equiv \underline{si}\ k \leqslant 0\ \underline{alors}\ b\ \underline{sinon}\ maun(maz(invers(maun(b, k-2), k), k-2), k-1) \end{cases}$$

Remarque 1 :

La démarche suivie pour résoudre P2 est voisine de celle adoptée pour construire une première spécification (ch 1.4 § 2) : on part de la spécification du résultat b' et on cherche à l'expliciter en introduisant des intermédiaires. Ce sont des symboles o-aires (b1, b2, b3) , ou des symboles q-aires (maun). L'introduction du nom b' donne un caractère modulaire à cette explicitation : la forme (4) est plus lisible que la définition purement fonctionnelle finale (5).

Remarque 2 :

Nous nous sommes efforcé, bien qu'utilisant un style informel, de jalonner les étapes successives de la démarche en mettant en évidence les sous-buts successifs et les choix d'explicitation : choix de décomposition, choix d'intermédiaires, choix de stratégies d'explicitation (ici analyse récursive, analyse par cas etc...). Encore plus que dans le cas de la spécification il est nécessaire de disposer d'un langage précis, sinon formel, permettant d'exprimer les différents choix et raisonnements que l'on effectue.

Remarque 3 :

Dans cet exemple nous avons surtout insisté sur le choix des intermédiaires, la stratégie générale consistant à comparer la spécification de maz(b, k) à celle de maz(b, k-1) de manière à exprimer l'une en fonction de l'autre. Pour ce faire nous sommes guidé par le fait que la définition obtenue "doit pouvoir remplacer" la spécification initiale, c'est à dire ici que :

(6) *pré* maz(b, k) ⊃ (*post* (maz(b, k), b) ⊃ D(maz(b, k), b'))

où *post*(maz(b, k), b') est la formule définie par la s-description de maz et D(maz(b, k), b') la définition explicite récursive de maz donnée en (5).

En effet, avec cette hypothèse tout théorème démontré en utilisant D pourra l'être en utilisant *post* . En particulier toute solution d'un problème dont l'énoncé utilise maz sera encore solution du même problème pour lequel maz est caractérisé par *post* . Dans cet exemple nous avons une propriété plus forte puisque nous n'avons utilisé que des équivalences.

En reprenant les termes du paragraphe 6.3 du chapitre 1.3, remplacer *post* (maz(b, k), b') par D(maz(b, k), b') et *post* (maun(b, k), b') par D(maun(b, k), b') revient à faire une *abstraction*.
La formule (6) est une *règle de construction* pour une telle abstraction.

Remarque 4 :

L'invention des intermédiaires n'est pas très simple à partir des seuls énoncés et dans l'exemple du baguenaudier l'introduction de schémas décrivant les différents (états des) baguenaudiers est un support certain pour le raisonnement. Ceci laisse à penser qu'il faut rechercher une syntaxe plus agréable pour le langage des énoncés (ch 1.4) et introduire certains supports graphiques comme le suggère REYNOLDS dans le cas des tableaux [REY, 79] .

Remarque 5 :

Les univers que l'on utilise pour une spécification initiale d'un problème doivent être très "riches" en propriétés ce qui facilite leur utilisation : c'est le cas de s-descriptions des S.I. . En revanche ils apparaissent peu constructifs ou plus exactement sans constructeur évident. Ainsi dans cet exemple nous n'utilisons pas toutes les propriétés des suites : seul nous importe le passage d'un baguenaudier à k-1 éléments à un baguenaudier à k éléments. C'est d'ailleurs tout à fait évident dans la mesure où nous ne nous préoccuppons pas des éléments de k+1 à n (caractérisés par le prédicat rest). En bref, nous nous sommes attachés à caractériser mazt(b~v) à l'aide de mazt : dans le S.I. "suite", le constructeur ~ et ses "réciproques" début et der ont ici une grande importance.

Ainsi l'explicitation d'un problème nécessite la mise en évidence dans l'univers "descriptif" initial de simplificateurs et de constructeurs, L'exemple suivant illustre cette idée.

Exemple 3 :

La figure 2 rappelle, en termes de s-description, la spécification du problème des sous-suites ascendantes de longueur maximum (ch 1.1 § 3.1 a)) :

ℓmax : ENTIER	*résultat* ℓmax *tq* ℓmax = max {k : ENTIER *tq* ∃ β : V* *tq* β ⊑ α ∧ croiss(β) ∧ |β| = k }
α : V*	*donnée* α
⊑ : *fonct* (V*, V*)BOOL	⊑ *tq* β ⊑ γ = (∃ φ : [dom(β) → dom(γ)] *tq* scroiss(φ) ∧ β = γ ∘ φ)
croiss : *fonct*(V*)BOOL	croiss *tq* croiss(β) = (∀i ∈ [bi(β)..bs(β) - 1], $\beta_i \leqslant \beta_{i+1}$)
scroiss : *fonct*(V*)BOOL	scroiss *tq* scroiss(β) = ∀i ∈ [bi(β)..bs(β) - 1] $\beta_i < \beta_{i+1}$)

Figure 2 : sous-suites ascendantes de longueur maximum

But 0 : Expliciter ℓmax

But 1 : Donner des définitions récursives de ⊑ , croiss, scroiss et ||

Choix (1) définir ces fonctions à l'aide des deux simplifications der et début du S.I. V* .

Rappelons que dans V* on connaît un certain nombre de simplificateurs (cf ch 1.2 § 7.2)

dont en particulier

début : *op* (V*) V*
début (u) = u'
  tq bi(u') = bi(u) ∧ bs(u') = bs(u) - 1 ∧
    (∀i, bi(u') ⩽ i ⩽ bs(u') ⊃ $u'_i \equiv u_i$)
  si |u| ⩾ 1

der : *op* (V*) V
der (u) = x
  tq $x = u_{bs}$
  si |u| ⩾ 1

But 1.1. : Définir récursivement croiss(β)

Choix (1.1) analyse par cas

i) β ≡ < > (suite vide)

croiss(< >) ≡ (∀i ∈ [bi(< >)..bs(< >) -1 , $< >_i \leqslant < >_{i+1}$)

≡ *vrai* puisque bs(< >) < bi(< >), cf ch1.2 § 7.2 figure 9

ii) |β| ⩾ 1, alors :

(1) croiss(β) ≡ (∀i ∈ [bi(β)..bs (β) -1] , $\beta_i \leqslant \beta_{i+1}$)

Choix analyse par cas

ii 1)

la formule (1) égale *vrai* si [bi(β)..bs(β) -1] = ∅ (intervalle vide)

c'est à dire si bs(β) -1 < bi(β)

or par hypothèse |β| ⩾ 1, c'est-à-dire bs(β) - bi(β) + 1 ⩾ 1

ou encore bs(β) ⩾ bi(β)

on en tire que (1) est *vrai* si bs(β) ≡ bi(β), c'est-à-dire |β| ≡ 1

ii 2)

Plaçons nous maintenant dans le cas où $|\beta| > 1$

Pour exprimer (1) à l'aide de début (β) et der(β) on explicite croiss(début(β))

croiss(début(β)) ≡ ($\forall i \in [bi(début(\beta))..bs(début(\beta))-1]$, $début(\beta)_i \leqslant début(\beta)_{i+1}$)

soit avec la définition de début :

(2) croiss (début(β)) ≡ ($\forall i \in [bi(\beta)..bs(\beta) - 2]$, $\beta_i \leqslant \beta_{i+1}$)

de (1) et (2) on tire (toujours par unification) :

(3) croiss(β) ≡ croiss(début(β)) ∧ $\beta_{bs(\beta)-1} \leqslant \beta_{bs(\beta)}$

qui a un sens puisque $bs(\beta) > bi(\beta)$

or *pré* der(début(β)) ≡ $bs(début(\beta)) - bi(début(\beta)) + 1 \geqslant 1$

≡ $bs(\beta) - 1 - bi(\beta) + 1 \geqslant 1$

≡ $bs(\beta) - bi(\beta) \geqslant 1$

≡ *vrai*

et der(début(β)) ≡ $\beta_{bs(\beta)-1}$

Ainsi (3) devient :

croiss(β) ≡ croiss(début(β)) ∧ der(début(β)) $\leqslant$ der(β)

Le but 1.1. est donc atteint avec :

croiss(< >) ≡ *vrai*

$|\beta| = 1 \supset$ croiss(β) ≡ *vrai*

$|\beta| \geqslant 2 \supset$ croiss(β) ≡ croiss(début(β)) ∧ der(début(β)) $\leqslant$ der(β)

But 1.2. :

Définir récursivement scroiss(β)

Par analogie avec le raisonnement précédent on obtient :

$$\begin{cases} \text{scroiss}(<>) \equiv \textit{vrai} \\ |\beta| = 1 \supset \text{scroiss}(\beta) \equiv \textit{vrai} \\ |\beta| \geqslant 2 \quad \text{scroiss}(\beta) \equiv \text{scroiss}(début(\beta)) \wedge \text{der}(début(\beta)) < \text{der}(\beta) \end{cases}$$

But 1.3. :

Définir récursivement $\beta \sqsubseteq \gamma$

Cette fonction dépend de deux arguments

Choix 1.3 : on cherche une récursivité portant sur le deuxième argument

Choix : analyse par cas

i) $\gamma = <>$

$\beta \sqsubseteq <> \equiv \exists \varphi$: [dom(β) → dom(< >)] *tq* scroiss(φ) ∧ $\beta \equiv <> \circ \varphi$

or dom(< >) ≡ ∅ , donc :

$\beta \sqsubseteq <> \equiv (\forall j \in \text{dom}(\beta), \beta_j \equiv <>) \equiv (\beta \equiv <>)$

ii) $|\gamma| \geqslant 1$

(1) $\beta \sqsubseteq \gamma \equiv \exists \varphi$: [dom(β) → dom(γ)] *tq* scroiss(φ) ∧ $\beta \equiv \gamma \cdot \varphi$

donc

(2) $\beta \sqsubseteq$ début(γ) ≡ $\exists \varphi'$: [dom(β) → dom(début(γ))) *tq* scroiss(φ') ∧ $\beta \equiv \gamma \cdot \varphi'$

L'unification de (1) et (2) nécessite de déplier la définition de scroiss :

$\exists \varphi$: [dom(β) → dom(γ)] *tq* scroiss(début(φ)) ∧ $\varphi(bs(\beta)-1) < \varphi(bs(\beta))$ ∧ $\beta \equiv \gamma \circ \varphi$

et $\exists \varphi' : [dom(début(\beta)) \rightarrow dom(\gamma)] \wedge \exists k \in dom(\gamma)$ *tq* $scroiss(\varphi') \wedge \varphi'(bs(\beta)-1) < k \quad \wedge \quad \beta \equiv \gamma_{\circ}\varphi'$

En distinguant alors les cas $k < bs(\gamma)$ et $k = bs(\gamma)$:

$$\beta \sqsubseteq \gamma \equiv (\exists \varphi' : [dom(début(\beta)) \rightarrow dom(début(\gamma))] \text{ tq } scroiss(\varphi') \wedge (\forall j \in dom(début(\beta)),\ \beta_j \equiv \gamma_{\varphi'(j)}) \wedge der(\beta) \equiv der(\gamma))$$

$$\vee\ (\exists \varphi : [dom(\beta) \rightarrow dom(début(\gamma))] \text{ tq } scroiss(\varphi) \wedge (\forall j \in dom(\beta),\ \beta_j \equiv \gamma_{\varphi(j)}))$$

ce qui donne :

$$\beta \sqsubseteq \gamma \equiv [début(\beta) \sqsubseteq début(\gamma) \wedge der(\beta) \equiv der(\gamma)] \vee [\beta \sqsubseteq début(\gamma)]$$

et le but 1.3 est atteint.

But 1.4. :

Définir récursivement $|\beta|$

Rappelons que $|\beta| \equiv bs(\beta) - bi(\beta) + 1$ (ch 1.2 § 7.2 figure 9)

choix 1.4 : analyse par cas

i) $|< >| \equiv 0$

ii) $|\beta| \equiv bs(\beta) - bi(\beta) + 1$

$\equiv bs(début(\beta)) - bi(début(\beta)) + 2$ (unification entre $bs(\beta)$ et $bs(début(\beta))$, de même pour bi)

$\equiv |début(\beta)| + 1$ (pliage de ||)

Et le but 1.4. est atteint

Résumons les résultats obtenus jusqu'à présent :

$$(I)\ \begin{cases} \beta \sqsubseteq \gamma \equiv \textit{si}\ \gamma \equiv < > \ \textit{alors}\ \beta \equiv < > \ \textit{sinon}\ \ \beta \sqsubseteq début(\gamma) \vee (der(\beta) \equiv der(\gamma) \wedge début(\beta) \sqsubseteq début(\gamma)) \\ crois(\beta) \equiv \textit{si}\ \beta \equiv < > \vee |\beta| \equiv 1\ \textit{alors}\ \textit{vrai}\ \textit{sinon}\ croiss(début(\beta)) \wedge der(début(\beta)) \leqslant der(\beta) \\ |\beta| \equiv \textit{si}\ \beta \equiv < >\ \textit{alors}\ 0\ \textit{sinon}\ \ |début(\beta)| + 1 \end{cases}$$

Le but suivant consiste alors à définir récursivement &max, cette explicitation étant assez longue nous ne la développerons pas ici. Elle se trouve dans [BRO, 79 a] .

Remarque 1 :

Comme dans l'exemple précédent nous mettons en évidence un certain nombre de *règles de transformation* qui sont utilisées dans l'explicitation d'un énoncé. Ainsi en 1.1 et 1.3 ci-dessus nous avons utilisé :

- le dépliage qui consiste à remplacer un appel de fonction par l'énoncé correspondant (par exemple bi(début (β)) par bi(β), bs(début(β)) par bs(β) -1 etc...)

- l'analyse par cas (par exemple distinction entre les cas $\beta = < >$ et $|\beta| \geqslant 1$)

- l'unification de deux formules (cf ch 1.3 § 4.2).

- le pliage , inverse du dépliage, qui consiste à remplacer un énoncé caractérisant "le corps" d'une fonction par un appel de cette fonction.

Ces règles apparaissent comme des *opérations de bas niveau* qui sont mises en oeuvre par la méthode d'explicitation. Aussi, s'il est important de préciser ce genre de règles et même de les implanter dans le cadre d'un système (automatique [BUR, 77 b] ou du moins interactif) permettant l'explicitation d'énoncés, il est plus important encore de définir *une méthode d'explicitation*.

Un objectif étant de construire un méta-algorithme d'explicitation dans le même esprit que le méta-algorithme de spécification envisagé au chapitre 1.4 (cf § 2 et chapitre 2.4).

En plus des règles de transformation précédente, nous avons utilisé de nombreuses règles de calcul algébrique et ensembliste. Leur utilisation dans un système automatique est difficile et nécessite de se doter d'heuristiques et de calcul formel. □

Un des gros intêrêts de la démarche systèmatique que nous avons suivie est la possibilité de remettre en cause certains choix, et ainsi d'obtenir éventuellement d'autres résultats. Ainsi remettons , par exemple, en cause le choix (1).

Choix(2)

Définir les fonctions intermédiaires à l'aide des constructeurs $< >$ et $\sim$ Rappelons que la fonction d'adjonction en queue, $\sim$, est définie par

$\sim$: *op* (V* , V) V*

$u \sim x = u'$

tq $bi(u') = bi(u) \wedge bs(u') = bs(u) + 1 \wedge$

$\forall i (bi(u) < i \leqslant bs(u) \supset u'_i = u_i) \wedge u'_{bs(u')} = x$

Toute suite finie de V* peut être obtenue à partir de $< >$ en appliquant un certain nombre de fois $\sim$.

But 2.1. : Définir récursivement croiss(β)

Choix (2.1) Analyse par cas

i) $\beta \equiv < >$ alors $croiss(< >) \equiv$ *vrai*

ii) $\beta \equiv \gamma \sim x$

$croiss(\beta) \equiv croiss(\gamma \sim x)$

$\equiv \forall i \in [bi(\gamma \sim x)..bs(\gamma \sim x)-1] , (\gamma \sim x)_i \leqslant (\gamma \sim x)_{i+1}$

(dépliage de croiss)

$\equiv \forall i \in [bi(\gamma)..bs(\gamma) +1-1] , (\gamma \sim x)_i \leqslant (\gamma \sim x)_{i+1}$

(dépliage de bi, bs)

$\equiv \forall i \in [bi(\gamma)..bs(\gamma) -1] , \gamma_i \leqslant \gamma_{i+1} \wedge \gamma_{bs(\beta)} \leqslant x$

(dépliage de $\sim$)

Distinguons alors les cas $\gamma \equiv < >$ et $\gamma \neq < >$:

$croiss(\beta) \equiv$ *si* $\gamma \equiv < >$ *alors vrai*

sinon $croiss(\gamma) \wedge (der(\gamma) \leqslant x)$

et si on ne veut pas utiliser la fonction der nous obtenons :

$croiss(\gamma \sim x) \equiv$ *si* $\exists \gamma', \exists y$ *tq* $\gamma \equiv \gamma' \sim y$ *alors* $croiss(\gamma' \sim y) \wedge y \leqslant x$

sinon vrai

ainsi le but 2.1. est atteint.

Des raisonnements similaires conduisent à l'explicitation de scroiss, $\subseteq$ et $||$. Résumons les diverses définitions obtenues :

(II)
$$\begin{cases} < > \subseteq \beta \equiv vrai \\ \beta \sim x \subseteq < > \equiv faux \\ \beta \sim x \subseteq \gamma \sim y \equiv \beta \sim x \subseteq \gamma \quad \vee \quad (\beta \subseteq \gamma \wedge x \equiv y) \\ crois(< >) \equiv vrai \\ crois(< x >) \equiv vrai \\ crois(\gamma \sim x \sim y) \equiv crois(\gamma \sim x) \wedge x \leqslant y \\ |< >| \equiv 0 \\ |\beta \sim x| \equiv |\beta| + 1 \end{cases}$$

En poursuivant l'explicitation de &max à partir de (II) on retrouve la solution de DIJKSTRA [DIJ, 79a].

La mise en évidence de constructeurs ou de simplificateurs dans le S.I. du problème est donc une étape fondamentale dans le processus d'explicitation.

Remarque 2 :

En reprenant, à peu de choses près, le raisonnement développé dans l'exemple du baguenaudier (exemple 2) et en utilisant les simplificateurs :

- début
- der
- pen (tel que pen(u) = der(début(u)))

nous arrivons au système d'équations récursives suivant :

mazt(b) ≡ *si* b ≡ < >
alors < >
sinon si |b| ≡ 1
alors < 0 >
sinon mazt(maunt(début^2(desc(b))) ~ pen(b))~ der(desc(b))

desc(b) ≡ invers(mazt(début^2(b)) ~ pen(b) ~ der(b), bs(b))

maunt(b) ≡ *si* b ≡ < >
alors < >
sinon si |b| ≡ 1
alors < 1 >
sinon maunt(mazt(début^2(elev(b)))~ pen(b))~ der(elev(b))

elev(b) ≡ invers(maunt(début^2(b)) ~ pen(b) ~ der(b), bs(b)) □

S'il est important de prendre conscience des choix successifs lors d'un processus d'explicitation, il faut également être capable de les *justifier* c'est-à-dire de disposer de critères de décision : pourquoi, par exemple, préférer des simplificateurs à des constructeurs ou inversement ? La découverte de tels critères nécessite certainement une étude expérimentale très poussée.

L'exemple suivant, sans envisager une étude complètement exhaustive, montre dans le cas d'un problème très simple,comment il est possible d'envisager divers choix d'explicitation. L'attitude adoptée n'est certes pas

généralisable puisqu'une exploration systématique de tous les choix (si elle est possible !) conduit rapidement à une explosion combinatoire.

Exemple 4 :

Toujours le problème de la fraction irréductible

L'étude de cet exemple permet de dégager quelques stratégies générales d'explicitation.

Rappelons en l'énoncé formel (cf 1.1 § 2.2 figure 2) :

$$\varepsilon(x, y, a, b) = \underbrace{x * b \equiv y * a}_{propor(x,y,a,b)} \wedge \underbrace{\forall t(t \text{ div } x \wedge t \text{ div } y \supset t \equiv 1)}_{prem(x, y)}$$

But 0 : expliciter ε

Choix (1) : Une idée de résolution consiste (ch 1.3 § 6.2.1. exemple 12) à transformer cet énoncé sous la forme "semi explicite"

$$x \equiv a/z \wedge y \equiv b/z \wedge z \equiv pgcd(a, b)$$

à partir de laquelle il faut encore expliciter pgcd ; ainsi

$$x \equiv a/z \wedge y \equiv b/z \wedge z \equiv pgcd(a, b) \wedge$$

$$pgcd(u, v) \equiv \textit{si } v \equiv 0 \textit{ alors } u \textit{ sinon } pgcd(v, u \textit{ mod } v)$$

est un énoncé récursif du problème considéré.

Choix (2) :

On peut aussi partir directement de ε(x, y, a, b) . Plusieurs approches sont envisageables, parmi elles citons :

- *stratégie d'induction sur les données :* on cherche une valeur particulière des données pour laquelle on sait résoudre le problème et on cherche à s'y ramener.
- *stratégie d'affaiblissement de l'énoncé* : les données restant "fixes", on cherche un résultat vérifiant"partiellement l'énoncé" et on cherche à l'améliorer.

Détaillons sur cet exemple ces deux approches.

Choix (2.1) : Induction sur les données.

Comme dans l'exemple 2 deux voies sont envisageables :

- introduire des constructeurs
- introduire des simplificateurs

Pour simplifier les notations nous introduisons le S.I.

COUPLE = << p1 : ENTIER$^+$, p2 : ENTIER$^+$ >>

auquel seront étenduescertaines opérations sur les entiers. Dans la suite, selon la facilité d'écriture,nous noterons z, c ... des couples ou nous en expliciterons la forme (<< u, v >>, << a, b >>...) .

Choix (2.1.1) : Utilisation de constructeurs :

But :

Introduire une suite de couples d'entiers c telle que :

a) Il existe $n \geq 1$ tel que $c_n = <<$ a, b >>

b) Il existe un constructeur g sur les couples d'entiers tel que

$c_i = g(c_{i-1})$

c) On puisse construire une suite de couples d'entiers z telle que

$\forall i, \quad \varepsilon(z_i, c_i)$

Choix pour c) :

on cherche une définition récurrente de la suite z, il faut donc :

- Définir z_1 tel que

(i) $\varepsilon(z_1, c_1)$

- Trouver une fonction f telle que

(ii) $\varepsilon(z_{i-1}, c_{i-1}) \wedge z_i = f(z_{i-1}, c_i) \supset \varepsilon(z_i, c_i)$

Cette stratégie consiste donc à *généraliser* l'énoncé en introduisant les suites z et c . Les choix de ces suites, donc de f et g , sont à priori très nombreux :

- Choix de c_1 guidé par le fait que $\varepsilon(z_1, c_1)$ doit être facilement soluble, c'est à dire doit fournir une solution z_1 "évidente".

- Choix de g guidé par le fait que f doit pouvoir être définie "simplement".

Dans le cas présent l'écriture des formules (i) et (ii) ne permet pas de "voir" de relation évidente entre f et g . Les choix des constructeurs les plus classiques sur ENTIER(+1, +2 ....) ne permettent pas de définir f . On abandonne ici cette voie, quitte à y revenir si aucune autre solution n'apparaît dans une autre approche.

Choix (2.1.2) : Utilisation de simplificateurs :

But :

Trouver une définition récursive de la forme

$firr(c) = \mathcal{C}(firr(pred(c)), c)$

où pred est un simplificateur sur les couples d'entiers, qu'il faut déterminer, et firr est la fonction associant le résultat << x, y >> à la donnée << a, b >> :

firr : *fonct* (ENTIER$^+$, ENTIER$^+$) << ENTIER$^+$, ENTIER$^+$ >>

firr(a, b) = << x, y >>

tq $\varepsilon(x, y, a, b)$.

On cherche à procéder par unification entre firr(c) et firr(pred(c)) :

(i) $z \equiv firr(c) \longleftrightarrow p1(z) * p2(c) \equiv p2(z) * p1(c) \wedge prem(z)$ (dépliage)

(ii) $z' \equiv firr(pred(c)) \longleftrightarrow p1(z') * pred2(c) \equiv p2(z') * pred1(c) \wedge prem(z')$ (dépliage)

où $pred_i = pi \circ pred$ pour i = 1, 2

Le passage de z' à z et la recherche de pred sont des problèmes trop généraux pour que l'on espère raisonnablement les résoudre de manière systématique.

Une première façon de limiter la généralité concerne la forme de $\mathcal{C}$: on se restreint aux formules $\mathcal{C}$ de la forme :

$\mathcal{C}(firr(pred(c)), c) \equiv$ *si* $\varphi(c)$ *alors* d *sinon* $\mathcal{C}_1(firr(pred(c)), c)$

On décompose ainsi le problème en :

- Recherche de τ_1
- Recherche de la condition d'arrêt φ .

Ensuite on est conduit à faire des "essais". Rien n'assure donc que l'on trouvera une solution, même s'il en existe.

En comparant (i) et (ii) on veut que $prem(z') \supset prem(z)$. Une manière brutale de l'assurer est de poser $z'' \equiv z$, c'est à dire que si

$\varphi(c) \equiv$ *faux* alors $\tau_1(z', c) \equiv z$

Il suffit alors de trouver φ et d tels que

$\varphi(c) \supset \varepsilon(d, c)$

On est donc ramené à chercher une condition sur c pour que le problème initial admette une solution "évidente" :

$\varphi(c) \equiv prem(c)$ ⟨ eurêka ⟩
alors $d \equiv c$ est solution.

En résumé : $firr(c) \equiv$ *si* $prem(c)$ *alors* c *sinon* $firr(pred(c))$
et avec (i) et (ii) pred doit vérifier :

$$p1(c) * pred2(c) \equiv p2(c) * pred1(c)$$

Nous arrêtons ici ce développement car nous allons retrouver ce nouvel énoncé d'une autre manière. Nous terminerons alors l'explicitation.

Choix (2.2) : Affaiblissement de l'énoncé .

Puisque $\varepsilon(x, y, a, b) = propor(x, y, a, b) \wedge prem(x, y)$

L'idée consiste à partir d'une solution d'un seul des deux sous-énoncés puis de l'améliorer jusqu'à ce que le deuxième sous-énoncé soit vérifié. Plus précisément on *affaiblit* l'énoncé en choisissant l'un de ses deux sous-énoncés que l'on *généralise* alors en introduisant une suite z , ou deux suites z et c dont chaque terme est solution du sous-énoncé choisi.

Quelques possibilités :

- choisir z_i tel que $\forall i \geqslant 1$ $propor(p1(z_i), p2(z_i), a, b)$
- choisir z_i tel que $\forall i \geqslant 1$ $prem(z_i)$
- choisir z_i et c_i tels que $\forall i \geqslant 1$ $propor(z_i, c_i)$

Le dernier cas nous rapproche du choix 2.1.1. que nous avons abandonné car il était trop général. Envisageons alors les deux autres cas.

Choix (2.2.1) : On cherche z tel que $\forall i \geqslant 1$, $propor(p1(z_i), p2(z_i), a, b)$.

Pour simplifier les notations posons $z_i = \langle\langle u_i, v_i \rangle\rangle$

$propor(u_i, v_i, a, b)$ est appelé *l'invariant* de la suite.

On veut de plus qu'il existe un élément de la suite solution du problème, c'est à dire que l'on cherche n tel que $prem(u_n, v_n) \equiv$ *vrai*

$prem(u_i, v_i)$ est la *condition d'arrêt* de la suite z .

Pour déterminer la suite z on veut trouver :

- $z_1 = \langle\langle u_1, v_1 \rangle\rangle$
- une relation de récurrence f telle que

$$propor(u_{i-1}, v_{i-1}, a, b) \wedge \neg prem(u_{i-1}, v_{i-1}) \wedge \langle\langle u_i, v_i \rangle\rangle \equiv f(u_{i-1}, v_{i-1}, a, b) \supset propor(u_i, v_i, a, b)$$

- n tel que $prem(u_n, v_n)$

Remarque 1 :

On retrouve bien ainsi la caractérisation du simplificateur "pred" introduit lors du choix 2.1.2 :

$$\langle\langle u_i, v_i \rangle\rangle = pred(u_{i-1}, v_{i-1}) \qquad \square$$

But 1 : Détermination de la relation de récurrence

$propor(u_i, v_i, a, b) \equiv (u_i * b \equiv v_i * a)$ (dépliage)

$propor(u_{i-1}, v_{i-1}, a, b) \equiv (u_{i-1} * b \equiv v_{i-1} * a)$ (dépliage)

Parmi les différentes relations que l'on peut déduire de ces formules, choisissons :

$propor(z_i, z_{i-1})$, c'est à dire $u_i * v_{i-1} \equiv v_i * u_{i-1}$ ⊀ eurêka ⊁

or $propor(u_i, v_i, u_{i-1}, v_{i-1}) \leftrightarrow \exists k_i \neq 0$ tq $u_{i-1} \equiv u_i * k_i \wedge v_{i-1} \equiv v_i * k_i$

⊀ eurêka ⊁

Pour expliciter u_i, v_i de façon récurrente, on introduit une suite k_i telle que $u_i = u_{i-1}/k_i$ et $v_i = v_{i-1}/k_i$, soit, en étendant "/" aux couples d'entiers, $z_i \equiv z_{i-1}/k_i$

Le but 1 est atteint si k_i est explicité ; on cherche donc à expliciter la condition d'arrêt.

But 2 : Explicitation de la condition d'arrêt

Comme il y a de nombreuses suites k_i telles que $propor(z_{i-1}/k_i, z_{i-1})$ on est guidé par le fait que la suite doit converger, c'est-à-dire intuitivement que l'on a gagné quelque chose en passant de z_{i-1} à z_i , quant à la propriété prem. Plus formellement :

$prem(x, y) \equiv \forall t(t \text{ div } x \wedge t \text{ div } y \supset t \equiv 1)$

soit en termes d'ensembles :

$prem(x, y) \equiv (\{t \text{ tq } t \text{ div } x \wedge t \text{ div } y\} \equiv \{1\})$

D'où l'introduction de l'ensemble des diviseurs communs à x_i et y_i :

$dc_i \equiv \{t \text{ tq } t \text{ div } u_i \wedge t \text{ div } v_i\}$ qui s'écrit encore, en étendant div aux couples :

$dc_i \equiv \{t \text{ tq } t \text{ div } z_i\}$

La partie variante s'écrit alors $dc_i \equiv \{1\}$

But 2.1 : Trouver une définition récurrente de dc_i

On cherche à procéder par unification entre dc_i et dc_{i-1}

$dc_i = \{ t \text{ tq } t \text{ div } z_i\}$

$dc_{i-1} = \{ t \text{ tq } t \text{ div } z_{i-1}\}$

alors

$dc_i = \{t \text{ tq } t \text{ div } (z_{i-1}/k_i) \}$ et en utilisant les propriétés de div :

$= \{t \text{ tq } k_i * t \text{ div } z_{i-1} \}$

$= \{t \text{ tq } k_i * t \in dc_{i-1}\}$ et $k_i < dc_{i-1}$

Le but 2.1 est atteint si on admet qu'il s'agit d'une définition explicite.

De plus, on vérifie immédiatement que $dc_i \subsetneq dc_{i-1} \leftrightarrow k_i > 1$

On obtient ainsi la nouvelle version de l'énoncé représentée sur la figure 3.

Remarque 2 :

La présentation adoptée permet de mettre en évidence les étapes successives de l'explicitation :

On part de l'énoncé formel initial (colonne de gauche en haut) que l'on explicite en introduisant des intermédiaires (z_i, n) caractérisés par des énoncés formels (colonne de gauche) et ultérieurement explicités. On peut donc utiliser la même démarche déductive pour la spécification et pour l'explicitation, avec, cependant, en plus, dans ce dernier cas, des règles formelles

Figure 3 : première explicitation du problème de la fraction irréductible.

de construction. Il est donc naturel d'adopter un cadre de description syntaxique analogue pour l'explicitation et pour la spécification. □

L'explicitation précédente de z_i = <<u_i, v_i >> assure :

- que propor(u_i, v_i, a, b) est bien un invariant de la suite, donc que

  propor(u_n, v_n, a, b) ≡ *vrai*

- qu'il n'y a pas de blocage, c'est à dire que pour tout $i < n$, $dc_i \setminus \{1\}$ n'est pas vide, donc k_i existe

- qu'il existe un entier n tel que $dc_n \equiv \{1\}$, c'est à dire tel que

  prem(z_n) ≡ *vrai* ; ceci résulte du fait que la suite dc_i est strictement décroissante.

On assure ainsi à la fois la *correction partielle* et la *terminaison* en reprenant des termes introduits par FLOYD [FLO, 67] et HOARE [HOA, 69] [LIV, 78] dans l'étude de la correction des programmes.

But 2.2. : Première explicitation de k_i

Cet énoncé ε_2 n'est pas encore explicite à cause de la définition des dc_i qui n'est pas constructive : pour passer de dc_{i-1} à dc_i il faut introduire une nouvelle suite d'ensembles intermédiaires initialisée à $\emptyset$ et dont le dernier terme est dc_i . De plus, la définition de dc_1 constitue un nouveau problème à résoudre.

Enfin la spécification de k_i introduit de l'indéterminisme dans la spécification de z_i .

A nouveau plusieurs possibilités se présentent :

- expliciter dans ε_2 ce qui ne l'est pas encore
- chercher à améliorer ε_2 avant d'aller plus loin ce qui peut éviter du travail inutile.

Choix : Amélioration de ε_2

parce que $dc_i \equiv \{1\} \leftrightarrow k_i \equiv \max(dc_{i-1})$ < eurêka >

Dans un raisonnement plus détaillé, cette remarque peut se faire lorsque l'on cherche à déterminer le choix de k_i dans $dc_{i-1} / \{1\}$.

Alors si $k_1 \equiv \max(dc_1)$, $dc_2 \equiv \{1\}$ donc $n \equiv 2$ et ainsi ε_2 se transforme en l'énoncé ε_3 de la figure 4 où k_2 est noté k.

ALors , par définition, k est le pgcd de a et b .

But 2.3. : Explicitation de k

En reprenant le résultat de l'exemple 14 du chapitre 1.3., pgcd est définie récursivement par :

$$pgcd(u, v) \equiv \textit{si}\ \ v \equiv 0\ \underline{\textit{alors}}\ \ u\ \ \underline{\textit{sinon}}\ \ pgcd(v,\ u \bmod v)$$

où $u \bmod v \equiv u - v * (u \div v)$

A ce stade du développement l'explicitation est terminée si on pose $k \equiv pgcd\ (a, b)$.
On retrouve ainsi l'énoncé de l'exemple 14 du chapitre 1.3.

Ainsi le but 0 est atteint.

Recherche d'autres solutions ou amélioration des précédentes :

- On peut étudier d'autres choix que les précédents . Par exemple, remettre en cause le choix 2.2.1 en prenant pour invariant $prem(u_i, v_i)$

  On va se heurter alors à la difficulté liée à la construction de couples d'entiers premiers entre eux.

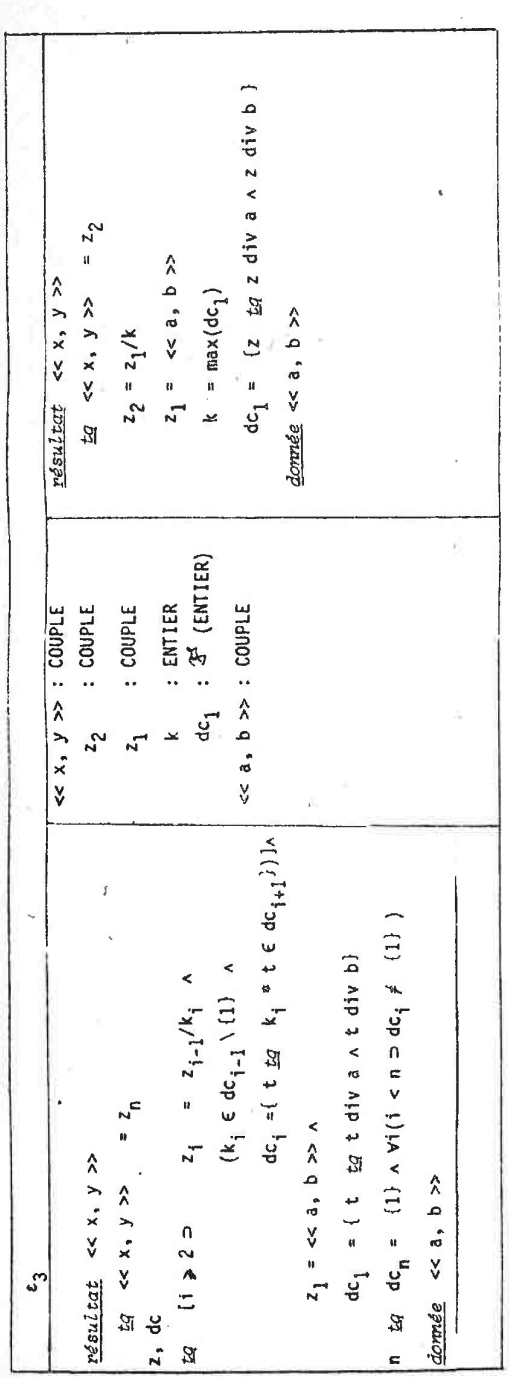


Figure 4 : Deuxième explicitation du problème de la fraction irréductible.

- On peut chercher une définition récurrente de k . Il s'agit alors de mettre en oeuvre une transformation de programmes classique : suppression de récursivité dans le cas de *récursivité terminale*. On obtient ainsi l'énoncé de la figure 5.

ε_4		
<< x, y >> *tq* propor(x, y, a, b) ∧ prem(x, y) k = pgcd(a, b) ∧ pgcd(u, v) = *si* v = 0 *alors* u *sinon* pgcd(v, u mod v) << c, d >> *tq* pgcd(c_1, d_1)= pgcd(c_{i-1}, d_{i-1})	<< x,y >>: COUPLE k : ENTIER << c, d >>: COUPLE* n : ENTIER <<a, b>>: COUPLE	*résultat* << x, y >> *tq* << x, y>> = <<a, b>> /k k = c_n << c, d >> *tq* i ≥ 2 ⊃ <<c_i, d_i >> = <<c_{i-1}, d_{i-1} mod c_{i-1} >>∧<<c_1, d_1 >> = << a, b >> n *tq* d_n = 0 *donnée* << a, b >>

Figure 5 : version récurrente du problème de la fraction irréductible.

Autres possibilités envisageables pour expliciter pgcd :

a) Chercher une induction sur les données a, b à l'aide de constructeurs sur les entiers. On se heurte aux mêmes difficultés que dans le choix 2.1.1. ci-dessus

b) Faire d'autres choix d'affaiblissements en introduisant une suite k_i dont le dernier terme est k. On a alors de nombreux choix d'invariants, par exemple :

invariant : k_i div a

condition d'arrêt : k_i div b ∧ ∀t(t div a ∧ t div b ⊃ t div k_i)

Mais aussi :

invariant : k_i div a ∧ k_i div b

condition d'arrêt : ∀t(t div a ∧ t div b ⊃ t div k_i)

Ou encore :

invariant : k_i = max ({t *tq* t div a∧t div b} ∩ [1..i])

condition d'arrêt : i = min(a, b)

Nous ne développerons pas ces diverses solutions mais on voit tout de suite la quasi impossibilité, même dans un problème aussi simple que celui-ci, d'une étude exhaustive.

Ce qui est d'autant plus regrettable qu'il est possible qu'une telle étude conduise à des solutions intéressantes, par exemple, à des algorithmes de calcul parallèle des diviseurs de a et des diviseurs de b .

Remarque 3 :

La figure 5 donne un exemple de description d'énoncé récurrent. Nous introduisons au chapitre 2.3 le langage MEDEE permettant de construire de tels énoncés. On y précise également quel genre de formule est un énoncé itératif et quelles en sont les propriétés intéressantes.

Remarque 4 :

Nous avons distingué deux méthodes (stratégies) d'explicitation très importantes :

a) Induction sur les données :

Pour ce faire on peut utiliser des constructeurs de données. Lorsqu'un seul constructeur, en plus d'un constructeur o-aire, suffit,on décrit une suite dont le dernier terme est la donnée :

On peut aussi utiliser des simplificateurs .

Cette étude met donc en évidence deux sortes de définitions récursives : les unes utilisant des constructeurs, les autres des simplificateurs. Nous reviendrons sur ces définitions au chapitre 2.2.

b) Affaiblissement.

Il consiste à décomposer l'énoncé initial en une condition d'arrêt et une partie invariante et à introduire une suite intermédiaire dont chaque terme doit vérifier l'invariant et dont le dernier terme vérifie en plus la condition d'arrêt.

Cette technique peut être considérée comme un cas particulier *d'induction sur les résultats* puisqu'on introduit des constructeurs du résultat. (cf chapitre 2.4).

Ces deux techniques nécessitent la mise en oeuvre d'une stratégie de *généralisation*, c'est à dire l'introduction de fonctions ou de suites intermédiaires.

Notons de plus que nous sommes allés plus loin que la première explicitation d'un énoncé puisque nous nous sommes préoccupés de transformation de définitions explicites. Cet aspect de la construction des programmes sera un peu évoqué dans la deuxième moitié du chapitre 2.3 et dans l'exemple 5.

Remarque 5 :

L'étude de l'exemple précédent montre qu'il existe deux niveaux de choix : choix de stratégies et choix d'intermédiaires. Pour faciliter le déroulement de ce genre de raisonnements, et en particulier les retours en arrière, il est nécessaire de se doter d'outils de gestion des choix. Nous associons donc au cadre syntaxique permettant de décrire les raisonnements successifs, un moyen de décrire la structure des choix : parallèlement à la construction de la version explicite nous construisons l'arbre des différents choix.

Remarque 6 :

Un problème se prêtant particulièrement bien à une étude systématique des choix d'explicitations et celui des tris. On trouve dans [GER, 77] une demarche conduisant au tri rapide dans laquelle les principaux choix sont mis en évidence. Des études systématiques [BRO, 78] et orientées vers la synthèse automatique [DAR, 78] ont été développées sur ce sujet.

Exemple 5 : Evaluation d'une expression post fixée.

On peut souhaiter, pour des raisons d'efficacité par exemple, transformer un énoncé explicite en un autre. Les techniques mises en oeuvre sont similaires à celles qui permettent une première explicitation. Ce sont ces "transformations de programmes" qui ont été les plus étudiées jusqu'à présent [ARS, 77] , [BUR, 77 b],[BAU, 78b] etc... .

Une de ces techniques est la *représentation* de tout ou partie du système d'information dans lequel s'exprime l'énoncé.

Ainsi, comme nous l'avons déjà remarqué, dans le problème de l'évaluation d'une expression arithmétique post fixée (ch 1.4 § 1.3 d)) l'énoncé initial a été transformé en un énoncé portant sur une structure linéaire. Cette transformation peut être considérée comme une *représentation* du S.I. EXP par le S.I. RLEXP (cf ch 1.3 § 6.3).

Pour la définir nous avons introduit un intermédiaire constitué d'une suite de suites de réels :

eval(x) ≡ x si x ∈ REEL
eval(u ~ x) ≡ *si* x ∈ REEL
alors eval(u) ~ x
sinon rest(eval(u)) ~ effet(pen(eval(u)), der(eval(u)), x)

Plutôt que de parler en termes de suites de réels il est possible de parler de piles de réels (cf ch 2.2 § 5.1) puisque les seules opérations portant sur ces suites sont ~ , der, début (en effet rest = début^2 et pen = der^2) ,

Finalement, nous obtenons l'énoncé récurrent de la figure 6.

r : REEL p : (PILE [REEL ∪ OP])* empiler : *fonct*(PILE,REEL)PILE *de base* dans PILE depiler : *fonct*(PILE) PILE *de base* dans PILE sommet : *fonct* (PILE)REEL *de base* dans PILE RLEXP : S.I.	*résultat* r *tq* r = sommet(p_n) p *tq* i ⩾ 2 ⊃ p_i *si* $el_i \in$ REEL *alors* empiler(p_{i-1}, el_i) *sinon* empiler(depiler^3(p_i), effet(sommet(depiler^2(p_i)), sommet(depiler(p_i)), el_i)) p_1 = pilevide *donnée* el : RLEXP

Figure 6 : nouvel énoncé de l'évaluation d'une expression post-fixée.

□

2.- ETUDE DU PROBLEME DE L'EXPLICITATION DES PROBLEMES.

L'étude empirique précédente permet de spécifier le (méta) problème de l'explicitation.

2.1.- Vers un méta-algorithme d'explicitation.

2.1.1.- Spécification du problème.

Intuitivement, expliciter un problème consiste à le transformer en un autre problème dont l'énoncé et le S.I. sont constructifs et dont les solutions sont des solutions du problème de départ. Les définitions du paragraphe 6 du chapitre 1.3 permettent de spécifier le méta-problème de l'explicitation (figure 7).

Pour compléter cette spécification puis essayer de construire un méta-algorithme d'explicitation il faut définir les concepts de problème explicite, d'énoncé explicite et de système d'information constructif. Ceci revient à définir l'univers du méta-problème de l'explicitation.

Ce méta-système d'information étant précisé, l'explicitation du méta-problème de l'explicitation consiste à expliciter les relations *vérifie, représentation de, abstraction de.*

Pour expliciter P' *vérif* P on cherche à le décomposer en un ensemble de méta-problèmes plus simples à résoudre. Ces méta-problèmes s'obtiennent, en général, en ajoutant des hypothèses à P' ou P : on cherche une suite $P_1 \ldots P_n$ de problèmes intermédiaires , telle que

P' *vérif* P_n *et* P_n *vérif* P_{n-1} *et* .... *et* P_1 *vérif* P .

On est ainsi conduit à étudier un certain nombre de méta-sous-problèmes de P' *vérif* P de la forme P' *et* $\mathcal{C}'$(P') *vérif* P *et* $\mathcal{C}$(P).
où $\mathcal{C}$ et $\mathcal{C}'$ sont des conditions (méta-prédicats) sur P et P'.

Ces conditions sont liées à la forme d'une partie de l'énoncé et d'une partie de l'univers des problèmes. Ici encore une étude précise des énoncés explicites et des S.I. constructifs doit déboucher sur la mise en évidence d'un certain nombre de méta-sous-problèmes pour lesquels l'explicitation est, en partie au moins, résolue. Ces différents méta-sous-problèmes seront appelés *stratégies d'explicitation.* On retrouve ici un point de vue assez voisin de l'utilisation de "schéma de programmes" [SCH, 79] .

EXPLICITATION	p' : PROBLEME EXPLICITE P : PROBLEME *vérif* : *op* (PROBLEME, PROBLEME) BOOL	*résultat* P' = (E', (S', Dop, Dyp)) *tq* (∃ μ *tq* (S', μ) *représentation de* S ∧ P' *vérif* μ (P)) ∨ (∃ α *tq* (S', α) *abstraction de* S ∧ (∃ Pa : PROBLEME *tq* α (Pa)= P ∧ P' *vérif* Pa)) ∨ (S' = S ∧ P' *vérif* P) *vérif tq* P2 *vérif* P1 = P2 vérifie P1 ∨ (P2 c-vérifie P1 ∧ calcul(P2), exsol(P1)) *donnée* P = (E, (S, Dop, Dyp))
P' est obtenu à partir de P par transformation de l'énoncé et du S.I. *vérif* : indique qu'un problème en vérifie un autre pour toutes les solutions ou pour les solutions calculables (cf ch 1.3 §6 figure 2)		

Figure 7 : spécification du méta-problème de l'explicitation.

Ces diverses stratégies vont,elles aussi,se décomposer,pour finalement se réduire à des *règles de construction* qui apparaissent ainsi comme des opérations élémentaires dans le méta S.I. de l'explicitation. En fait ces règles de construction sont des *théorèmes* du méta-univers dont la conclusion est de la forme P' *vérif* P . D'autres travaux considèrent ces règles de construction comme des règles d'inférence [BUR, 77 b] [BAU, 78 b].

De manière similaire il faut définir des stratégies d'abstraction et de représentation.

Un avantage de ce genre de démarche est que la "correction" du problème obtenu vis à vis du problème de départ est automatiquement assurée si chaque étape de la transformation correspond à l'application d'une règle de construction.

2.1.2. - Stratégies d'explicitation.

La technique générale d'explicitation consiste à introduire des *intermédiaires* (fonctions, objets, ensembles). Pour cela le programmeur dispose des moyens suivants :

a) un guide de l'ordre d'explicitation

- Il est raisonnable de commencer par chercher à expliciter la relation entre données et résultats, puis de recommencer pour les résultats intermédiaires.
- La structuration et la modularité des énoncé et S.I. initiaux permet des explicitations locales.

b) un catalogue de stratégies entre lesquelles il peut choisir Nous avons vu dans les exemples précédents quelques unes des stratégies les plus usuelles

- analyse par cas
- affaiblissement

- induction sur les données
- représentation d'un système d'information par un autre
- abstraction d'un système en un autre (introduction de constructeurs)
- introduction de S.I. intermédiaires.
- suppression de récursivité

etc....

Chaque stratégie est un méta-problème qui s'énonce comme un cas particulier du méta-problème de l'explicitation.

Par exemple si ε est un énoncé de résultat r, la stratégie d'analyse par cas appliquée à (ε, r) consiste à trouver p booléens $b_1, \ldots, b_p$ caractérisés respectivement par les énoncés $\beta_1, \ldots, \beta_p$ et $p+1$ intermédiaires $r_1 \ldots r_{p+1}$ caractérisés respectivement par $\varepsilon_1, \ldots, \varepsilon_{p+1}$ tels que :

$$(\beta_1 \wedge \varepsilon_1) \vee (\beta_2 \wedge \varepsilon_2) \vee \ldots \vee (\beta_p \wedge \varepsilon_p) \vee (\neg(\beta_1 \vee \ldots \vee \beta_p) \wedge \varepsilon_{p+1}) \supset \varepsilon$$

on peut alors *remplacer* ε par

$r \equiv$ *cas* b_1 *alors* r_1
b_2 *alors* r_2
b_p *alors* r_p
sinon r_{p+1}

et par la conjonction des β_i et des ε_i.

Ainsi le *résultat* d'une stratégie d'analyse par cas est formé :

- des énoncés β_i et ε_i
- des intermédiaires b_i et r_i

Bien sûr ces différentes stratégies sont en général, indéterministes et non calculables. Pour les résoudre on restreint le nombre de solutions en effectuant des *choix* qui font appel en général à d'autres stratégies : l'explicitation apparaît donc comme un processus profondément récursif, non seulement à cause de la structure arborescente des énoncés à expliciter, mais aussi par la nature indéterministe des stratégies.

Par exemple, pour effectuer une induction sur les données on pourra chercher des constructeurs, donc faire une abstraction. De même pour trouver une formule de récurrence dans le cas d'un affaiblissement on utilisera souvent l'unification.

Il faut encore noter que ces différentes stratégies n'ont pas la même "taille" :
certaines sont locales (unification, analyse par cas...), d'autres peuvent concerner des inconnues très "complexes" (abstraction, représentation...)

c) un certain nombre d'heuristiques caractérisant les diverses stratégies :
outre les préconditions éventuelles [SIN, 79] à la mise en oeuvre de stratégies, il est utile de donner des critères d'utilisation, un peu comme on le fait par exemple dans la description de techniques de calcul intégral. C'est certainement l'un des points les plus délicat : c'est dans le choix de la stratégie qu'intervient l'intelligence ou l'intuition du programmeur.

2.1.2.- Règles de construction.

Pour mettre en oeuvre ces stratégies le programmeur peut utiliser des *règles de construction* :

- Règles de calcul algèbrique, ensembliste...
- Règles de transformation d'énoncés développées depuis quelques années [ARS, 77], [BUR, 77 b], [BAU, 78 a], [FEA, 79] telles que le pliage, le dépliage, la fusion d'itérations etc...

- Règles permettant d'énoncer certaines stratégies : introduction de suites intermédiaires, introduction de symboles o-aires intermédiaires etc...

L'application de ces règles est conditionnée par certaines propriétés des formules à transformer.

2.1.4.- Expression du méta-algorithme.

Il est nécessaire de disposer d'un *cadre syntaxique* :

- Pour décrire les énoncés et les S.I. : dans un processus d'explicitation il est nécessaire de disposer à la fois de l'ancien énoncé et de l'énoncé que l'on est en train de construire (de même pour le S.I.). Ainsi une présentation sous forme de colonnes, comme pour les spécifications, permet de manipuler les deux formes d'énoncés mais aussi de savoir à tout moment où l'on en est dans l'explicitation : on décrit ainsi les différents résultats partiels du méta-algorithme

- Mais il est tout aussi important de pouvoir décrire la *suite des raisonnements* qui constituent l'explicitation. Plus précisément on veut décrire les "calculs" du méta-algorithme qui sont une suite de stratégies justifiées par des choix. Ceci permet de retrouver, de vérifier les raisonnements et les choix qui ont permis de dériver un énoncé explicite.

- Enfin en même temps que les résultats partiels du méta-algorithme, il faut décrire les états successifs de l'arbre de choix de façon à faciliter les décisions et les retours en arrière. Il va de soit, et on le vérifie aisément sur les exemples précédents, qu'une voie d'avenir est la construction d'un système interactif d'explicitation permettant de soulager le programmeur des calculs intermédiaires, lui réservant les choix de stratégies et lui permettant de cheminer dans l'arbre des choix antérieurs.

2.2.- Enoncés explicites et systèmes d'information constructifs.

Pour préciser les stratégies d'explicitation il faut commencer par caractériser les énoncés et les univers constructifs, ce qui nécessite :

- de définir précisément ces concepts. Nous exprimerons ces définitions en termes de systèmes d'information

- d'étudier leurs propriétés à partir desquelles on peut définir des stratégies de transformation.

- de se doter de langage de description d'un emploi aisé.

Mais nous avons déjà indiqué que nous nous intéressions essentiellement au cas des solutions calculables et donc que la distinction entre énoncé et axiomes d'un S.I. n'avait pas d'importance. Il en résulte qu'on pourrait définir les diverses transformations uniquement en termes d'abstraction et de représentation. Il n'en reste pas moins qu'il faut définir des techniques de transformation adaptées à certains types de formules. La démarche est la même que ces formules soient considérées comme des énoncés ou comme des axiomes d'un S.I..

2.3.- Articulation de la suite de cette deuxième partie.

L'objectif étant la description du méta-algorithme d'explicitation qui est esquissée au chapitre 2.4, nous commençons par étudier la forme et les propriétés des résultats qu'il doit fournir :

- Les S.I. explicites, on dit plutôt constructifs, sont étudiés au chapitre 2.2 en liaison étroite avec les stratégies d'abstraction et de représentation et les définitions récursives.

- Le chapitre 2.3 étudie un cas particulièrement important d'énoncés explicites : les énoncés récurrents. Cette étude débouche sur l'introduction du langage MEDEE qui prolonge le langage des s-descriptions et qui est bien adapté à la description d'énoncés récurrents. Aux énoncés récurrents est associée la stratégie particulièrement importante de l'affaiblissement.

CHAPITRE 2.2

DÉFINITIONS RÉCURSIVES ET DESCRIPTION ALGÉBRIQUE D'UN SYSTÈME D'INFORMATION

La classe la plus générale de définitions explicites est celle des définitions récursives. Nous en distinguons deux types : les définitions s-récursives exprimées à l'aide de simplificateurs et les définitions c-récursives liées à la notion de constructeur. Les définitions s-récursives sont celles que l'on entend habituellement par le vocable "définitions récursives". Après un bref rappel de leurs propriétés (paragraphe 1), on se préoccupe de la construction d'une telle définition à partir de s-descriptions.

Ayant rappelé l'intérêt méthodologique des concepts de constructeurs et de simplificateurs, le paragraphe 2 présente l'étude, à partir de quelques exemples, des descriptions algébriques de système d'information qui forment un cadre bien adapté à l'explicitation des problèmes.

Après avoir caractérisé les définitions c-récursives d'opérations, il est possible, au paragraphe 3, de définir et d'étudier les systèmes d'information algébriques. On peut montrer l'intérêt du concept de forme normale et l'existence de méthodes de preuves de consistance dans ce cadre. De plus, la plupart des opérations de construction de S.I. (cf ch 1.2 § 5 et § 7) peuvent être définies en termes algébriques (paragraphe 4), ce qui assure un caractère de stabilité à ce genre de description.

Il est alors important d'étudier le passage d'une s-description à une description algébrique. Ce passage correspond à une abstraction algébrique et le paragraphe 5 propose un méta-algorithme permettant de construire une telle abstraction.

Le dernier paragraphe concerne le problème de la représentation de systèmes algébriques et propose lui aussi un méta-algorithme de représentation.

1.- NOTION DE DEFINITION RECURSIVE.

Introduites par les logiciens [KLE, 52] , [SHO, 67] pour caractériser la classe des fonctions calculables, les définitions récursives ont été l'objet de très nombreuses études par les informaticiens, autant pour l'introduction d'outils de programmation fondés sur la récursivité (LISP [CAR, 62] CUCH [BOH, 66] , procédures récursives, types abstraits algébriques) que pour la description formelle de leurs propriétés (théorie du point fixe [LIV, 78] , sémantique dénotationnelle [SCO, 71] , sémantique algébrique [COU, 78] , règles de preuves [MIL, 76] , règles de calcul [VUI, 74]) .

Aussi nous contenterons nous ici d'un bref survol des caractéristiques des définitions récursives précisant surtout l'aspect méthodologique associé à leur construction.

1.1.- Définitions s-récursives.

Définition 1 :

Soit S un S.I. d'alphabet A et h un symbole de profil $\overline{S_1}, \ldots, \overline{S_q} \rightarrow \overline{S_{q+1}}$ n'appartenant pas à A ($\overline{S_1}, \overline{S_2} \ldots \overline{S_q}, \overline{S_{q+1}}$ sont des sortes de S). On appelle *symbole de fonctionnelle* (ou simplement *fonctionnelle*) *associé à* h tout schéma fonctionnel $\mathcal{C}$ sur A ∪ {h} □

Par exemple $\mathcal{C}$ = *si* $x_2 \equiv 0$ *alors* x_1 *sinon* $h(x_2, x_1 \bmod x_2)$

est une fonctionnelle associée à h sur un S.I. contenant l'opération mod.

Comme nous l'avons déjà signalé plusieurs fois (ch 1.3 § 2, ch 2.1 § 1) il est possible de définir par récurrence sur la longueur des termes une règle de substitution à h , dans une fonctionnelle $\mathcal{C}$, par un symbole de même profil que h (cf [REM, 74]) .

Définition 2 :

Avec les notations précédentes :

Une *définition s-récursive* d'un symbole fonctionnel f de profil $\overline{S_1}, \ldots, \overline{S_q} \rightarrow \overline{S_{q+1}}$ (pouvant ne pas appartenir à A) est une formule de la forme :

$$f\ y_1 \ldots y_q \equiv \mathcal{C}[f/h, y_{i_1}/x_1, \ldots, y_{i_p}/x_p]$$

où $y_1 \ldots y_q$ sont des variables libres ; $\{x_1 \ldots x_p\}$ est l'ensemble des variables libres de $\mathcal{C}$ et $p \leq q$; $\mathcal{C}$ et h sont comme dans la définition 1. □

Par exemple

pgcd(u, v) ≡ *si* v ≡ 0 *alors* u *sinon* pgcd(v, u mod v)

est une définition s-récursive.

Dans la définition précédente, le terme s-récursive précise que la récursivité est obtenue par l'utilisation de simplificateurs. Nous lui opposerons au paragraphe 3.1 les définitions c-récursives reposant sur l'utilisation de constructeurs.

Il est évident, avec le paragraphe 1 du chapitre 2.1 que les définitions s-récursives sont des définitions explicites. Cependant il n'y a aucune raison pour que les définitions s-récursives soient réservées aux fonctions "totales". Dans le cas général une telle définition sera pré-conditionnée.

Définition 3 : Avec les notations précédentes

Une définition s-récursive pré-conditionnée est une formule du type :

$$\varphi \supset f\ y_1 \ldots y_q \equiv \mathcal{C}[f/h, y_{i_1}/x_1, \ldots, y_{i_p}/x_p]$$

où φ est une formule associée à $f\ y_1 \ldots y_q$: sa pré-condition □

Par exemple

$u \geq 0 \wedge v \geq 0 \supset$ pgcd(u, v) ≡ *si* v ≡ 0 *alors* u *sinon* pgcd(v, u mod v)

est une formule de ce genre.

Lorsqu'il n'y a pas d'ambiguité, on appelle encore définition récursive une définition s-récursive pré-conditionnée.

Notation :

en utilisant le langage des s-descriptions, une définition s-récursive pré-conditionnée s'écrit :

$$f\ x_1 \ldots x_q = \mathcal{C}[\ldots]$$
$$si\ \varphi$$

Définition 4 :

Un système d'équations s-récursives préconditionnées est une conjonction de définitions s-récursives pré-conditionnées de la forme

$$\begin{cases} pré\ (f_1\ y_1^1 \ldots y_{q_1}^1) \supset f_1\ y_1^1 \ldots y_{q_1}^1 = \mathcal{C}_1[\ f_1/h_1, \ldots] \\ \vdots \\ \wedge\ pré\ (f_k\ y_1^k \ldots y_{q_k}^k) \supset f_k\ y_1^k \ldots y_{q_k}^k = \mathcal{C}_k\ [f_k/h_k, \ldots] \end{cases}$$

telle que chaque $\mathcal{C}_i$ soit un schéma fonctionnel sur $A \cup \{h, f_1, \ldots, f_{i-1}, f_{i+1} \ldots f_k\}$

On parle encore de schéma récursif [NIV, 75] . □

1.2.- Bref rappel des propriétés des systèmes d'équations récursives.

Il existe de nombreuses présentations des systèmes d'équations récursives ([COU, 78] , [CAD, 73] , [REM, 74] , [LIV, 78] etc...) aussi nous contentons nous ici de rappeler quelques uns des résultats les plus fondamentaux. Notons cependant que, dans le cadre où nous nous plaçons, nous admettons des fonctions partielles caractérisées par des pré-conditions. Pour se ramener aux théories classiques on devrait les prolonger en des fonctions totales prenant éventuellement la valeur "indéfinie" ω .

a) Il existe plusieurs sémantiques (interprétations) de systèmes d'équations récursives [COU, 78] . De la théorie du point fixe développée par SCOTT, on déduit que chaque fonction d'un système d'équation récursif admet une interprétation qui peut être éventuellement la fonction r (fonction nulle part définie) [LIV, 78] .

b) Ainsi tout système d'équation récursif à k inconnues $f_1 \ldots f_k$ admet toujours une solution (on étend ici le concept de solution du chapitre 1.3) à *condition d'admettre les fonctions nulle part définies* (c'est-à-dire les fonctions f telles que *pré* f = *faux*).

Par exemple

$f(x) = f(x) + 1$ n'est jamais définie

c) Une définition récursive peut admettre des solutions qui ne sont pas calculables.

Par exemple

$r = 2.r - 1$ n'est pas calculable bien que $r = 1$ soit solution.

d) Plusieurs *règles de calcul* (appel par nom, par valeur, par nécessité) ont été définies pour les systèmes récursifs.

Ce sont des restrictions de la règles de substitution qui ont été introduites pour rendre plus déterministe le calcul d'un résultat. Rappelons enfin que les règles de calcul permettent au mieux d'obtenir le plus petit point fixe de la fonctionnelle associée au système [LIV, 78] .

1.3.- Construction de définitions s-récursives.

Nous nous intéressons, dans ce paragraphe, à l'explicitation s-récursive des fonctions. Ce (méta-) problème peut être énoncé informellement de la façon suivante :

Etant donné une s-description d'une fonction f , construire une définition s-récursive de cette fonction.

C'est un problème que nous nous sommes déjà posé lors de l'explicitation de pgcd (ch 1.3 § 6.2) ou encore lors de l'explicitation de maz (ch 2.1 § 1 exemple 2).

Ce problème est donc une *stratégie d'explicitation*. Sa mise en oeuvre peut conduire à introduire de nouvelles fonctions (c'était le cas pour le problème du baguenaudier). Cette stratégie est fondée sur des règles de construction que nous introduisons maintenant.

1.3.1.- Règles de construction de définitions s-récursives.

D'un point de vue méthodologique, la construction d'une définition s-récursive à partir d'une s-description a pour objectif le remplacement de la s-description par la définition récursive. Cette démarche correspond à la construction d'une *abstraction* du S.I. dont la fonction est une opération. C'est donc sur le concept d'abstraction que repose la règle suivante :

(RESR) Règle d'explicitation s-récursive

Soit S un S.I. et f un symbole q-aire défini par la s-description suivante :

$$f\ x_1 \ldots x_q = y \quad tq\ \psi\ si\ \varphi$$

La définition s-récursive prê-conditionnée

$$\varphi \supset f\ x_1 \ldots x_q = \tau$$

est *correcte* vis à vis de ψ si

$$\vdash_S \forall x_1 \ldots x_q \quad (\varphi \supset \psi\ [f\ x_1 \ldots x_q / y\ ]) \supset \forall x_1 \ldots x_q (\varphi \supset f x_1 \ldots x_q = \tau)$$

Théorème 1 :

Avec les notations de la règle précédente, soit A le S.I. déduit de S en remplaçant l'axiome $\varphi \supset \psi\ [f\ x_1 \ldots x_q / y]$ par $\varphi \supset f x_1 \ldots x_q = \tau$. Alors (A, id) est une abstraction de S si id est l'application identité des symboles de A vers les symboles de S □

Résulte immédiatement de la définition d'une abstraction .

Ainsi, tout problème P dont le S.I. est S va s'abstraire en un problème Pa sur A tel que toute solution de Pa soit une solution de P : pour résoudre P il suffit de résoudre Pa (cf théorème 10', théorème 12 ch 1.3)

Exemple 1 :

En reprenant le début d'explicitation du problème des sous-suites ascendantes de longueur maximum, nous avons transformé (ch 2.1 § 1 exemple 3) la s-description :

$$croiss(\beta) = b \quad tq \quad b = (\forall i \in [bi(\beta) \ldots bs(\beta) - 1] \ ,\ \beta_i \leq \beta_{i+1})$$

en la définition récursive :

$$croiss(\beta) = si\ \beta = <> \vee |\beta| = 1\ alors\ vrai\ sinon\ croiss(début(\beta)) \wedge der(début(\beta)) \leq der(\beta)$$

La règle (RESR) était le fil directeur de la stratégie que nous avions alors suivie . □

En réalité dans l'explicitation de croiss nous avons une équivalence entre la s-description et la définition récursive :

$$[\forall \beta\ ,\ croiss(\beta) = (\forall i \in [bi(\beta) \ldots bs(\beta) - 1] \ ,\ \beta_i \leq \beta_{i+1})] \leftrightarrow$$

$$[\forall \beta\ ,\ croiss(\beta) = si\ \beta = <> \wedge |\beta| = 1\ alors\ vrai\ sinon\ croiss(début(\beta)) \wedge der(début(\beta)) \leq der(\beta)\ ]$$

Ainsi, l'abstraction (A, id) définie par cette transformation est complète pour les formules $croiss(\beta) = b$ (définition 13 ch 1.3). Comme par ailleurs elle préserve les autres théorèmes par définition de id, cette abstraction est *complète*. Elle associe de façon bijective à toute solution d'un problème P sur S , une solution de l'abstraction Pa de P à A .

Caractérisons de façon générale cette propriété par la règle suivante :

(RESRT) Règle d'explicitation s-récursive totale

Soit f un symbole q-aire d'un S.I. S , défini par la s-description

$$f\ x_1 \ldots x_q = y \quad \underline{tq}\ \psi\ \underline{si}\ \varphi$$

La définition s-récursive pré-conditionnée

$$\varphi \supset fx_1 \ldots x_q \equiv \tau$$

est *totalement correcte* vis à vis de ψ si

$$\vdash_S \forall x_1 \ldots x_q \ (\varphi \supset \psi[fx_1 \ldots x_q/y]) \longleftrightarrow \forall x_1 \ldots x_q (\varphi \supset fx_1 \ldots x_q \equiv \tau)$$

Théorème 2 :

Avec les mêmesnotations que dans le théorème 1, (A, id) est une abstraction complète de S □

Remarque :

Nous adopterons quelquefois une présentation des règles de construction voisine de celle utilisée pour les règles d'inférence en sémantique axiomatique [LIV, 78] et développée par exemple dans [BAU, 78 b] . Ainsi (RESR) peut s'écrire :

$$\frac{\varphi \supset \psi\ [fx_1 \ldots x_q/y]}{\varphi \supset f\ x_1 \ldots x_q \equiv \tau} \quad \forall\ x_1 \ldots x_q\ (\varphi \supset \psi[fx_1 \ldots x_q/y]) \supset \forall\ x_1 \ldots x_q\ (\varphi \supset fx_1 \ldots x_q \equiv \tau)$$

1.3.2.- Stratégie d'explicitation s-récursive

C'est un cas particulier du problème général de l'explicitation qui concerne les fonctions définies par des s-descriptions.

⊰ SESR ⊱ Stratégie d'explicitation s-récursive

donnée f caractérisée par $f\ x_1 \ldots x_q = y \ \underline{tq}\ \psi$ dans un S.I. S

résultat définition s-récursive $f\ x_1 \ldots x_q = \tau$ *si* φ

$\underline{tq}$ $\varphi \supset f\ x_1 \ldots x_q \equiv \tau$ est correcte vis-à-vis de ψ

Présentation :

i.e transformation de la s-description de S

Dans la nouvelle description du S.I. S on place la s-description initiale de f dans la colonne de gauche et la définition s-récursive dans la colonne de droite.

Heuristique :

- Chercher τ sous la forme :

$$\underline{si}\ b\ \underline{alors}\ \delta\ \underline{sinon}\ \tau 1$$

où δ ne contient pas de symbole inconnu

et b est tel que S soit une solution "évidente" de ψ

- Choisir les simplificateurs de telle manière que, pour tout $x_1 \ldots x_q$ tel que φ soit *vrai* , l'application d'un nombre fini de fois des simplificateurs conduise à un q-uplet $y_1 \ldots y_q$ tel que $b\ [y_1/x_1 \ldots y_q/x_q] \equiv \underline{vrai}$

- Pour déterminer $\tau 1$ essayer d'unifier

$$\psi\ [f\ x_1 \ldots x_q/y] \quad \text{avec} \quad \psi\ [f\ u_1 \ldots u_q/y]$$

où $u_1 \ldots u_q$ est le q-uplet déduit de $x_1 \ldots x_p$ par application des sélecteurs. Ceci peut conduire à l'introduction de nouveaux intermédiaires.

Cette stratégie apparait comme un cas particulier d'induction sur les données. Elle peut faire appel à d'autres stratégies telles que l'unification.

Après avoir étudié le problème de l'explicitation s-récursive d'une fonction, nous nous intéressons maintenant au problème global de l'explicitation d'un système d'information.

2.- NOTION DE SYSTEME D'INFORMATION CONSTRUCTIF.

2.1.- Utilité des abstractions algébriques.

2.1.1.- Explicitation d'énoncé et explicitation de système d'information.

Les S.I. que l'on souhaite utiliser pour une spécification initiale d'un problème doivent être très "riches" en propriétés, ce qui facilite leur utilisation. C'est le cas des s-descriptions de S.I. qui reposent sur l'utilisation de S.I. prédéfinis.

En revanche ces S.I. apparaissent souvent comme peu constructifs, ou plus précisément sans constructeur évident. Or, nous avons constaté sur les exemples du chapitre 2.1, que l'explicitation nécessite la mise en évidence dans le S.I. de référence de simplificateurs et de constructeurs.

Reprenons l'exemple 2 du chapitre 2.1 (problème des sous-suites maximales). L'explicitation des fonctions $\sqsubseteq$, croiss, || en termes de simplificateurs (ou de constructeurs, nous allons revenir sur cette deuxième possibilité) conduit à une *abstraction* du S.I. initial. Cette abstraction consiste simplement à remplacer les axiomes initiaux caractérisant ces fonctions par les définitions récursives correspondantes. Mais pour la résolution ultérieure du problème , on a assez peu gagné si on est obligé de revenir, dans l'abstraction obtenue, aux s-descriptions initiales des simplificateurs. Si donc il était naturel , lors de la spécification initiale du problème, de considérer un S.I. V^* très riche en propriétés, on commence, à ce stade du développement, à savoir quelles sont les propriétés (et donc les fonctions) qui sont les plus intéressantes pour le développement ultérieur de l'énoncé. Ce sont elles que l'on souhaite *abstraire* du système d'information initial et qui seront à la base des raisonnements ultérieurs.
On cherche donc un S.I. "extrait" du S.I. de départ et que l'on pourra prendre comme nouvelle référence en "oubliant' le S.I. initial.

En d'autres termes, et en reprenant les notations du chapitre 1.3, pour expliciter un problème il ne suffit pas d'expliciter l'énoncé du problème et se placer dans l'abstraction de SD $[\varepsilon_i]$ définie par l'explicitation de ε_i , mais encore faut-il expliciter les fonctions de base du S.I. du problème qui sont utilisées dans la version explicite de l'énoncé ε .

2.1.2.- Premier exemple de spécification algébrique.

Exemple 2 :

Reprenons le problème des sous-suites ascendantes maximales. Ayant mis en évidence les constructeurs $\sim$ et $< >$, une première idée consiste à donner une nouvelle définition de V^* en termes de monoïde. Ainsi V^* serait le monoïde libre sur V , c'est à dire un ensemble muni d'une loi de composition interne $\sim$ (la concaténation) associative, admettant un élément neutre $< >$ et contenant un sous-ensemble isomorphe à V . Mais cette définition n'est pas directement constructive et on lui préfère la définition suivante :

V^* est caractérisé par l'existence d'un constructeur binaire :

$$\sim : op\ (V^* , V)\ V^*$$

et d'un élément particulier qui est le constructeur o-aire "$< >$" .

Cette définition revient à dire que tout élément de V^* , différent de la suite vide $< >$, est de la forme $(\ldots(< > \sim x_1) \sim x_2)\ldots) \sim x_n$

noté encore $< > \sim x_1 \sim x_2 \ldots \sim x_n$.

Si on ne donne pas plus d'informations sur V* on obtient une *structure libre* en ce sens qu'il n'y a pas d'axiomes propres.

Une telle structure risque de n'être pas assez riche pour les transformations ultérieures d'énoncés. On l'a d'ailleurs constaté dans l'exemple 2 du chapitre 2.1 où dans la définition de croiss($\gamma \sim x$) on faisait intervenir de manière naturelle der(γ). On peut aussi souhaiter démontrer que les énoncés (I) et (II) obtenus dans cet exemple 2 sont équivalents, ou, mieux encore, que l'on peut passer systématiquement de l'un à l'autre, et ceci sans avoir à revenir aux s-descriptions initiales.

Ainsi on souhaite définir V* en introduisant des axiomes reliant constructeurs et simplificateurs. Reprenons alors les s-descriptions de début, der et $\sim$ (cf ch 1.2 § 7.2) et cherchons à expliciter les deux premières opérations à l'aide de la troisième.

(0) *pré* (début ($\beta \sim x$)) $\equiv |\beta \sim x| > 1$

$\equiv [bs(\beta \sim x) - bi(\beta \sim x) + 1] \geq 1$

$\equiv [bs(\beta) + 1 - bi(\beta) + 1] \geq 1$

$\equiv$ *vrai* puisque $bs(\beta) + 1 \geq bi(\beta)$ pour toute suite β

(1) bi(début ($\beta \sim x$)) $\equiv bi(\beta \sim x) \equiv bi(\beta)$

(2) bs(début ($\beta \sim x$)) $\equiv bs(\beta \sim x) - 1 \equiv bs(\beta)$

(3) $\forall i$, $bi(\beta) \leq i \leq bs(\beta) \supset$ appl(début ($\beta \sim x$), i) $\equiv$ appl($\beta \sim x$, i)

$\equiv$ appl(β, i)

Donc, avec "l'axiome de l'égalité" sur les fonctions, donc sur les suites :

$$(1) \wedge (2) \wedge (3) \longleftarrow \text{début}(\beta \sim x) \equiv \beta$$

De manière analogue on prouve que

$$\text{der}(\beta \sim x) \equiv x$$

On peut donc caractériser V* comme un "nouveau" système d'information, noté encore V* , dont les symboles sont $< >$, $\sim$, der, début (avec les mêmes profils que précédemment) et dont les axiomes sont :

début($\beta \sim x$) $\equiv \beta$

der ($\beta \sim x$) $\equiv x$

et les préconditions :

pré ($\beta \sim x$) $\equiv$ *vrai* ; *pré* $< > \equiv$ *vrai* ;

pré (début(β)) $\equiv (\beta \neq < >)$; *pré* (der(β)) $\equiv (\beta \neq < >)$

Ce nouveau système d'information est une abstraction du système initial.

Les axiomes de V* font apparaître < début, der > comme "réciproque" de la fonction $\sim$. C'est ce genre d'axiomes qui permet de passer d'une définition récursive d'une fonction à une définition itérative.

Notons que les axiomes ne sont pas des définitions s-récursives puisque les opérandes du membre gauche ne sont pas des variables. Il s'agit de définitions récursives portant sur des constructeurs ou définitions c- récursives (cf § 3.1). □

Dans un S.I. tel que le nouveau V* on ne précise pas ce qu'est un objet de la sorte d'intérêt ; c'est-à-dire que l'on n'utilise pas de sorte support. C'est l'une des différences essentielles avec les s-descriptions présentées au chapitre 1.2.

Dans un tel S.I. les axiomes indiquent comment on "passe" d'un objet à un autre (opérations "$\sim$" et "début") ou comment on peut "extraire des informations" d'un objet donné (opération "der"). En ce sens ce genre de spécification est plus abstrait que celui fourni par des s-descriptions : on a abstrait un certain nombre de propriétés et négligé les autres. En revanche, il ne nous semble pas, en général, bien adapté à une spécification initiale d'un problème dans la mesure où il sous-entend déjà une vision constructive de l'univers du problème.

Cette forme de spécification a été introduite par GUTTAG [GUT, 78 a] et le groupe ADJ [GOG, 77 a] . Elle est qualifiée d'*algébrique* car les objets et les opérations apparaissant dans un tel système d'information peuvent être considérés comme définissant une algèbre abstraite [LES, 79] (cf § 3.4).

2.2.- Caractéristiques d'une spécification algébrique.

De manière très schématique, un système d'information algébrique est caractérisé par :

a) le fait qu'il n'y a pas de définition des sortes d'intérêt en termes de sortes support.

b) La forme des axiomes qui ne sont que des définitions récursives, éventuellement pré-conditionnées. Cette forme *explicite* des axiomes caractérisant les fonctions de base est l'un des intérêts majeur de ce genre de spécification (cf § 2.3).

Pour mieux comprendre ces idées donnons une syntaxe possible d'une spécification algébrique, très voisine de celle de GUTTAG, dans le cas des suites sur V (figure 1).

Nous parlons alors de *description algébrique* ou encore de *a-description* de V* .

S.T. V*		
< > : ~ : début : der :	cf s-descriptions ch1.2 figure 9	V : S.I. ~ : *op* (V*, V) V* début : *fonct*(V*)V* der : *fonct*(V*) V < > : *op* V*	*param* *sparam* *opérations* début(u ~ x) = u der(u ~ x) = x *restrictions* *pré* début(u) = (u ≠ < >) *pré* der (u) = (u ≠ < >)

Figure 1 : une partie de la spécification algébrique de V*

Remarque 1 :

Il n'est pas tout à fait exact de prétendre que l'on ne précise rien, dans le cas d'une spécification algébrique, sur la forme des objets considérés. Par exemple, dans le cas des suites on peut écrire :

$\forall u \neq < >$, début : V* , der : V → u (< >)

C'est à dire que l'on considère les opérations début et der comme des projections, le S.I. V* étant construit à partir de "< >" et de l'opération "produit cartésien" (ch 1.2 § 7.1). Ce genre de définition nécessite des précautions puisqu'il s'agit d'une définition récursive : on a affaire à des "domaines réflexifs" et la théorie du point fixe de SCOTT [SCO. 72] peut s'appliquer.

Les concepts de forme normale et d'implémentation directe que nous présentons au paragraphe 3.3. correspondent à un point de vue voisin fondé sur des notions de grammaire.

Si donc on admet, dans le cadre des s-descriptions, des définitions récursives des sortes d'intérêt les descriptions algébriques sont un cas particulier de s-descriptions .

Remarque 2 :

L'introduction de pré-conditions dans la description algébrique résulte de l'existence de ces pré-conditions dans la s-description initiale. Ces pré-conditions nécessitent un certain nombre de précautions lors d'une démonstration. Il faut, en particulier, vérifier qu'à chaque "appel" d'une fonction la pré-condition correspondante est réalisée.

Plutôt que d'utiliser des pré-conditions nous aurions pu décider, tant pour les s-descriptions que pour les a-descriptions, de ne considérer que des opérations totales, quitte éventuellement à introduire un élément indéfini "ω".

Par exemple on peut poser :

$$\begin{cases} \text{début } (< >) = < > \\ \text{der } (< >) = \omega \end{cases}$$

On vérifie, par l'expérience, que le premier axiome ne pose aucun problème, bien au contraire il simplifie les développements ultérieurs en évitant l'introduction de certaines conditionnelles. Inversement si on avait décidé de définir début(< >) comme étant égale à ω on n'aurait rien gagné par rapport aux pré-conditions.

Ici l'introduction de der(<>) = ω permet des simplifications d'écriture à condition, de prendre certaines précautions :
Par exemple puisque V , dans le problème des sous suites maximales, est totalement ordonné on exige que $\forall x \in V, \omega \leqslant x$. Ceci permet alors de redéfinir croiss plus simplement (ch 2.1 exemple 2 énoncé (I)) :

croiss(β) = *si* β = < > *alors vrai sinon* croiss(début(β))
∧ der(début(β)) ⩽ der(β)

car alors cette définition a un sens lorsque |β| = 1 , c'est à dire lorsque

début(β) = < > et on a bien croiss(β) = croiss(< >) ∧ der(< >) ⩽ der(β)
soit croiss(β) = *vrai*

On peut cependant reprocher à la définition de croiss ainsi obtenue d'être moins naturelle que la première.

Ainsi, d'un point de vue méthodologique, se pose la question de savoir s'il est préférable d'avoir des fonctions partiellement définies (pré-conditions) ou si au contraire il vaut mieux n'avoir que des fonctions totales, prenant éventuellement la valeur indéfinie en certains points. Nous aborderons rapidement cette question au paragraphe 3.2.3.

2.3.- Impact méthodologique des systèmes d'information algébriques.

2.3.1.- Exemple de passage d'une s-description à une a-description.

Afin de mieux dégager les intérêts des a-descriptions reprenons l'exemple de la table des symboles (ch 1.? § 7.3 exemple 11).

Exemple 3 :

Spécification algébrique du S.I. "table des symboles" [GUT, 78 a] , [FIN, 78 b] .

pas 1

Il s'agit de *choisir les constructeurs*. Il est assez naturel de choisir "insert" et "debbloc". Mais il faut aussi choisir un "point de départ" que nous appelerons la table vide, notée "tabvide", et qui est définie par :

tabvide = << 0, ∅ >>

pas 2

Comme dans l'exemple des sous-suites croissantes maximales (exemple 2, ch 2.1) , on essaye alors de définir les autres opérations à l'aide de ces constructeurs "primitifs". Donnons deux exemples.

a) Par définition presom(ts, ident) = ∃ a : AT *tq* <<ident, bloc *de* ts, a >>
∈ rel *de* ts

Alors - presom(tabvide, ident) = ∃a : AT *tq* << ident, bloc *de* tabvide, a >>
∈ rel *de* ts
= ∃a : AT *tq* << ident, 0, a >> ∈ ∅
= *faux*

- presom(debbloc(ts), ident) = ∃a : AT *tq* << ident, bloc *de*
debbloc(ts), a >> ∈ rel *de* debbloc(ts)

posons ts' = debbloc(ts).

Pour définition de debbloc : ts' = << bloc *de* ts +1, rel *de* ts >>
alors presom(debbloc(ts),.ident) = ∃a : AT *tq* << ident, bloc *de* ts +1, a >>
∈ rel *de* ts

mais par définition de l'invariant de ts :

$\forall r \in$ rel *de* ts, $0 \leqslant$ bl *de* r $\leqslant$ bloc *de* ts

ainsi ∃a : AT *tq* << ident, bloc *de* ts +1, a >> ∈ rel *de* ts ⊃
∃a : AT *tq* 0 ⩽ bl *de* (<< ident, bloc *de* ts +1, a >>)
⩽ bloc *de* ts ⊃
∃a : AT *tq* 0 ⩽ bloc *de* ts +1 ⩽ bloc *de* ts = *faux*

donc presom (debbloc(ts), ident) = *faux*

b) La démonstration précédente était simple, en particulier parce que les opérations "presom" et debbloc" n'ont pas de préconditions. Dans le cas général il faut faire intervenir les préconditions, comme nous l'avons fait de manière informelle dans l'exemple des sous-suites croissantes maximales Supposons par exemple que nous cherchions l'axiome caractérisant finbloc (insert(ts, ident, a)),

nous aurons comme pré-condition :

$$\underline{pré}\quad \text{insert}(ts, \text{ident}, a) \equiv \text{cardinal}(\text{rel } \underline{de}\ ts) < n \wedge \neg \text{presom}(ts, \text{ident}) \wedge \neg \text{vide}(ts)$$

et $\underline{pré}$ finbloc(insert(ts, ident, a)) doit être *vrai* également. On cherche donc le plus petit prédicat p(ts, ident, a) (au sens où p est plus petit que q si $p \supset q$)

tel que $p(ts, \text{ident}, a) \supset \underline{pré}\ \text{insert}(ts, \text{ident}, a) \wedge \underline{pré}\ \text{finbloc}(\text{insert}(ts, \text{ident}, a))$

soit $p(ts, \text{ident}, a) \supset \text{cardinal}(\text{rel } \underline{de}\ ts) < n \wedge \neg \text{presom}(ts, \text{ident}) \wedge \neg \text{vide}(ts) \wedge \neg \text{vide}(\text{insert}(ts, \text{ident}, a))$

On démontre simplement que $\neg \text{vide}(\text{insert}(ts, \text{ident}, a)) \leftrightarrow \neg \text{vide}(ts)$

On obtient ainsi :

$p(ts, \text{ident}, a) \equiv \text{cardinal}(\text{rel } \underline{de}\ ts) < n \wedge \neg \text{presom}(ts, \text{ident}) \wedge \neg \text{vide}(ts)$

On peut maintenant expliciter finbloc(insert(ts, ident, a)) :

Par définition des s-descriptions de insert et finbloc :

(i) $\underline{pré}$ insert(ts, ident, a) $\wedge$ tsl $\equiv$ insert(ts, ident, a) $\supset$

tsl $\equiv$ << bloc $\underline{de}$ ts, rel $\underline{de}$ ts $\cup$ { << ident, bloc $\underline{de}$ ts, a >>} >>

(ii) $\underline{pré}$ finbloc(tsl) $\wedge$ ts2 $\equiv$ finbloc(ts) $\supset$

ts2 $\equiv$ << bloc $\underline{de}$ tsl -1, rel $\underline{de}$ tsl \ {<< id, x, at >>

$\underline{tq}$ x $\geqslant$ bloc $\underline{de}$ tsl } >>

de (i) et (ii) on tire :

p(ts, ident, a) $\wedge$ ts2 $\equiv$ finbloc (insert(ts, ident, a))

$\supset$ ts2 $\equiv$ << bloc $\underline{de}$ ts -1 , rel $\underline{de}$ ts\ {<<id, x, at >>

$\underline{tq}$ x $\geqslant$ bloc $\underline{de}$ ts } >>

Si, maintenant, on a l'idée (≠ eurêka) d'unifier cette formule avec l'axiome suivant caractérisant finbloc (ts) :

$\underline{pré}$ finbloc(ts) $\wedge$ ts' $\equiv$ finbloc(ts) $\supset$

ts' $\equiv$ << bloc $\underline{de}$ ts-1, rel $\underline{de}$ ts \ {<<id, x, at >>

$\underline{tq}$ x $\geqslant$ bloc $\underline{de}$ ts } >>

On obtient :

p(ts, ident, a) $\wedge$ $\underline{pré}$ finbloc(ts) $\supset$

finbloc(insert(ts, ident, a)) $\equiv$ finbloc (ts)

Remarquons que le sens des implications montre bien que l'on effectue une abstraction en remplaçant les "anciens" axiomes par les "nouveaux". En effet, si on néglige les pré-conditions , la démarche précédente a été guidée par la règle suivante :

$\psi_s \supset \psi_a$ où ψ_s est l'axiome fourni par la s-description et ψ_a l'axiome cherché de la a-description.

pas 3 :

Détermination des pré-conditions algébriques.

Dans un problème quelconque, il est possible qu'en effectuant une abstraction algébrique on cherche de nouvelles pré-conditions. Quelle est la relation qui les lie aux anciennes ?

Soit $\varphi_a \wedge y \equiv f(x) \supset \psi_a$ l'axiome déduit de la a-description de f

et $\varphi_s \wedge y \equiv f(x) \supset \psi_s$ celui déduit de sa s-description .

Par définition d'une abstraction on souhaite que :

$$(\varphi_s \wedge y \equiv f(x) \supset \psi_s) \supset (\varphi_a \wedge y \equiv f(x) \supset \psi_a)$$

Un calcul rapide assure que cette implication est un théorème si :

$$\varphi_a \supset \varphi_s$$

et $\psi_s\ [f(x)/y] \supset \psi_a\ [f(x)/y]$ sont deux théorèmes .

La relation entre φ_a et φ_s assurant que l'on obtient bien une abstraction est donc :

$\vdash_S \varphi_a \supset \varphi_s$ (S est le S.I. de départ)

Ici, ne voulant pas modifier le domaine des différentes opérations il est naturel d'identifier φ_a à φ_s .

La suite de l'explicitation, qui n'est pas détaillée ici, conduit à la a-description de la figure 2. Dans cette figure la colonne de droite contient les axiomes algébriques alors que la colonne de gauche rappelle les s-descriptions des opérations (figure 10 chapitre 1.2).

Seule la fonction "taille" mérite un peu d'attention. Elle n'existait pas dans la s-description initiale et nous avons été conduit à l'introduire pour définir l'opération "pleine". En effet la s-description de pleine utilise l'opération "card" (cardinal) définie sur $\mathcal{P}(E)$ et l'on ne souhaite pas la redéfinir ici. "taille" est donc une "fonction auxiliaire" qui sert à définir d'autres fonctions mais qui n'a pas à être utilisée dans un énoncé :
on dit qu'il s'agit d'une *fonction cachée*. Cet aspect de fonctions "visibles ou invisibles" à l'extérieur d'un S.I. est largement utilisé dans le cadre des types abstraits et contribue à en faire des outils permettant la mise en oeuvre de mécanismes de protection ou de confidentialité [MIN, 79] , ce qui est hors de notre propos ici. L'introduction de telles fonctions est assez naturelle ; ici par exemple, lors de l'explicitation de pleine on est conduit à poser taille (ts)= cardinal (rel *de* ts). □

2.3.2.- Construction et utilisation d'une spécification algébrique.

Les systèmes d'information algébriques apparaissent comme un intermédiaire fondamental dans le processus d'explicitation d'énoncés :

- Le passage du S.I. initial à une spécification algébrique correspond à une *abstraction* des *seules propriétés* utiles au développement ultérieur de l'énoncé. Pour être plus réaliste il se peut que l'on conserve des propriétés "parasites" car inutiles mais on en minimise alors le nombre. En ce

s-description	profil	a-description
S.I. TABSYM		
# la sorte d'intérêt de TABSYM et l'invariant sont définis comme dans la figure 10 du chapitre 1.2 #	AT : *S.I.* n : ENTIER	*param* n *tq* n ⩾ 1 *fparam* *opérations*
-tabvide = << 0, ∅ >> - insert(ts, ident, a) = << bloc *de* ts, rel *de* ts ∪ {<< ident, bloc *de* ts, a >>} >> *si* ¬presom(ts, ident)∧ ¬vide(ts) - debbloc(ts) = << bloc *de* ts+1, rel *de* ts >> -finbloc(ts) = << bloc *de* ts-1, rel *de* ts \ {<< ident, x, a >> *tq* x ⩾ bloc *de* ts} >> *si* ¬vide(ts) - valattr(ts, ident) = *si* ∃y ∈ rel *de* ts *tq* id *de*y = ident ∧ (∀z ∈ rel *de* ts, id *de*z = ident ⊃ bl *de* z ⩽ bl *de* y) *alors* att *de* y *sinon* ω - presom(ts, ident) = ∃a *tq* << ident, bloc *de* ts, a >> ∈ rel *de* ts. - taille(ts) = card(rel *de* ts).	-tabvide : TABSYM -insert : *fonct* (TABSYM, CHAINE,AT) TABSYM -debbloc :*fonct* (TABSYM)TABSYM -finbloc : *fonct*(TABSYM)TABSYM -valattr : *fonct*(TABSYM, CHAINE) AT ∪ { ω } -presom : *fonct*(TABSYM, CHAINE) BOOL -taille : *fonct*(TABSYM) ENTIER	(1) finbloc(debbloc(ts)) = ts (2) finbloc(insert(ts, ident, a)) = finbloc(ts) (3) valattr(tabvide,ident) = ω (4) valattr(debbloc(ts), ident) = valattr(ts, ident (5) valattr(insert(ts, ident, a), ident 1) = *si* ident =ident1 *alors* a *sinon* valattr(ts, ide (6) presom(tabvide, ident) = *faux* (7) presom(debbloc(ts), ident) = *faux* (8) presom(insert(ts, ident, a), ident1) = (ident = ident1) ∨ presom(ts, ident1) (9) taille(tabvide) = 0 (10) taille (debbloc(ts)) = taille (ts) (11) taille(insert(ts, ident, a)) = taille (ts) +

S.I. TABSYM (2)

- pleine(ts) = (card(rel *de* ts) = n) - vide(ts) = (rel *de* ts = ∅ ∧ bloc *de* ts = 0)	- pleine : *fonct* (TABSYM)BOOL - vide : *fonct* (TABSYM) BOOL	(12) pleine(ts) =(taille(ts = n) (13) vide(tabvide) = *vrai* (14) vide(debbloc(ts)) = *faux* (15) vide(insert(ts, ident, a)) = *faux* *restrictions* *pré* finbloc (ts) = ¬vide(ts) *pré* insert(ts, ident, a) = ¬pleine(ts) ∧ ¬presom(ts, ident) ∧ ¬vide(ts)

Figure 2 : a-description du S.I. TABSYM

sens une spécification algébrique apparaît *plus simple* que la spécification dont on part.

- Mais ce passage se traduit également par une *explicitation* de certaines fonctions et donc par l'obtention d'axiomes plus intéressants car récursifs. Ce sont ces deux aspects, "minimalité" et axiomes explicites qui font l'intérêt des spécifications algébriques. Ils permettent en effet :

- d'expliciter l'énoncé du problème dans la mesure où l'on dispose de constructeurs et simplificateurs.

- d'avoir un grand choix de *représentations* ultérieures de l'univers puisque l'on a minimisé les propriétés à transporter. Ces représentations permettant l'explicitation d'énoncés ou servant à e rapprocher de constructions dont on dispose dans un langage de programmation

- de systématiser la construction d'une représentation en utilisant comme guide les axiomes comme nous le verrons au paragraphe 6.

- de disposer de techniques de preuve de consistance ou de complétude suffisante (cf § 3.4) qui ne sont pas applicables dans le cadre trop général des s-descriptions.

On voit donc se dessiner une première icée de méthode d'explicitation d'énoncés.

- Partant de l'énoncé et du S.I. de départ, déterminer les constructeurs et simplificateurs permettant de définir inductivement le résultat. Ceci conduit à définir une induction également sur les données et sur les intermédiaires.

- Construire une spécification algébrique à partir des différents constructeurs ainsi introduits. Cette démarche est guidée par des règles caractérisant les définitions c-récursives, très voisines de celles concernant les définitions s-récursives (§ 1.3). La stratégie d'explicitation c-récursive est présentée au paragraphe 3.1.2.

- Expliciter les intermédiaires et le résultat de l'énoncé en utilisant certaines stratégies. Les stratégies d'explicitation c-récursives et s-récursives en sont des exemples. Nous verrons au chapitre 2.3 d'autres stratégies d'explicitation conduisant à des énoncés récurrents.

Cette méthode générale d'explicitation est précisée au chapitre 2.4.

Notons que les étapes ultérieures de la résolution du problème considéré seront des transformations d'énoncés ou de S.I. conduisant à un programme efficace.

La figure 3 donne une schématisation grossière de cette démarche.

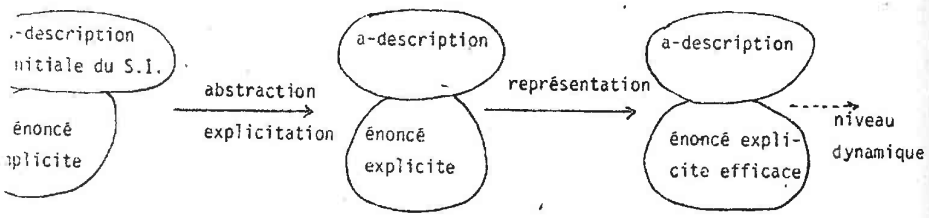


Figure 3 : démarche générale de résolution.

Nous étudions maintenant les systèmes d'information algébriques et définissons un langage de description.

3.- ETUDE ET DESCRIPTION DES SYSTEMES D'INFORMATION ALGEBRIQUES.

3.1.- Définitions c-récursives.

3.1.1.- Définitions.

Les définitions c-récursives forment un deuxième type de définitions récursives et reposent sur la notion de constructeur.

Par exemple : $\begin{cases} |\beta \sim x| = |\beta| + 1 \\ |< >| = 0 \end{cases}$ (exemple 2 ch. 2.1)

est une définition c-récursive de l'opération $|\ |$ à partir des constructeurs $\sim$ et $< >$.

Définition 5 :

Soit S un S.I., $\overline{S_i}$ une sorte de S et g un symbole de S de but $\overline{S_i}$

Soit f un symbole de S de profil $\overline{S_1}\ldots\overline{S_i}\ldots\overline{S_q} \rightarrow \overline{S_{q+1}}$ tel que pour $1 \leqslant j \leqslant q$, $i \neq j \Rightarrow \overline{S_i} \neq \overline{S_j}$ (f n'a qu'un seul argument de profil $\overline{S_i}$).

Une définition *c-récursive simple de* f *à partir de* g est une formule du type

$$f\ y_1\ldots y_{i-1}\ u_i\ y_{i+1}\ldots y_q \equiv \tau\ [f/h,\ t_1/x_1,\ldots,\ t_p/x_p]$$

où : - $y_1,\ldots,y_q$ sont des variables libres de profils respectifs $\overline{S_1},\ldots,\overline{S_q}$

- u_i est un terme de la forme $g\ z_1\ldots z_p$ où les z_i sont des variables libres

- τ est un symbole de fonctionnelle associé à h dont l'ensemble des variables libres est $\{x_1,\ldots, x_p\}$

- chaque t_i est soit un y_j , soit un z_j . □

En général, une seule définition c-récursive simple d'un symbole f ne permet pas de caractériser complètement ce symbole. Pour ce faire il est nécessaire de prendre la conjonction d'un certain nombre de telles définitions. Le cas intéressant est celui où les différentes fonctions g permettent "d'engendrer" tous les termes de profil $\overline{S_i}$. C'est le cas des définitions de "valattr", "presom" etc... à partir des constructeurs "tabvide", "debbloc", "insert" dans la description algébrique de TABSYM (figure 2) .

Nous sommes ainsi conduits à la définitions suivante :

Définition 6 :

Soit S un S.I., $\overline{S_i}$ une sorte de S et $\mathcal{C}_i$ un ensemble de symboles g de S de but $\overline{S_i}$ à chacun desquels est associéeune formule booléenne notée *pré* g et appelée sa pré-condition.

Soit f un symbole S de profil $\overline{S_1}\ldots\overline{S_i}\ldots\overline{S_q} \to \overline{S_{q+1}}$ tel que pour $1 \leqslant j \leqslant q$, $i \neq j \Rightarrow \overline{S_i} \neq \overline{S_j}$ (f n'a qu'un seul argument de profil $\overline{S_i}$)

Une définition *c-récursive préconditionnée de* f *à partir de* g est une formule du type

$$\bigwedge_{g \in \mathcal{C}_i} [(\textit{pré}\ g \wedge \varphi\ [u_i/y_i]) \supset f\ y_1 \ldots y_{i-1}\ u_i \ldots y_q \equiv \tau_g\ [f/h,\ t_1/x_1, \ldots,\ t_p/x_p]]$$

où - y_j, u_i, τ_g, t_j sont comme dans la définition 5

- φ est une formule dont les variables libres sont $y_1, \ldots, y_i, \ldots, y_q$, c'est la pré-condition de f notée encore *pré* f

On nomme pré-condition de $f\ y_1 \ldots y_{i-1}\ u_i \ldots y_q$ la formule

$$\textit{pré}\ f\ y_1 \ldots u_i \ldots y_q \equiv \textit{pré}\ g \wedge \varphi\ [u_i/y_i] \quad \square$$

Une définition c-récursive pré-conditionnée est donc la conjonction d'implications dont les second membres sont des définitions c-récursives simples.

Remarque 1 :

Lorsque *pré* g ∧ φ [u_i/y_i] ≡ *faux* est un théorème de S , on se dispense de définir $f\ y_1 \ldots u_i \ldots y_q$. C'est ainsi que finbloc(ts) n'est pas

défini pour ts = tabvide dans la a-description de TABSYM (Figure 2). □

Remarque 2 :

Les définitions précédentes de c-récursivité sont trop restrictives de deux façons :

- Elles ne considèrent que des ensembles $\mathcal{C}$ relatifs à une seule sorte de S à la fois.
- Elles ne considèrent que des fonctions f n'ayant qu'un seul argument dans $\overline{S_i}$.

Ces restrictions ont été faites dans le souci de simplifier la présentation , de plus c'est souvent le cas dans les a-descriptions.

Elles se suppriment toutes deux en considérant, dans la conjonction des formules, toutes les formes possibles de t-uplets $u_{i_1} \ldots u_{i_k}$ correspondants aux différentes fonctions $g_{i_1} \ldots g_{i_k}$ appartenant respectivement à $\mathcal{C}_{i_1} \ldots \mathcal{C}_{i_k}$.

Par exemple, dans l'énoncé (II) de l'exemple 2 du chapitre 2.1, l'opération ⊑ est définie par la définition c-récursive suivante :

$$\left.\begin{array}{l} <> \sqsubseteq <> \equiv \textit{vrai} \wedge \\ <> \sqsubseteq \beta \sim x \equiv \textit{vrai} \wedge \\ \beta \sim x \sqsubseteq <> \equiv \textit{faux} \wedge \end{array}\right\} \text{regroupées en } <> \sqsubseteq \beta \equiv \textit{vrai}$$

$$\beta \sim x \sqsubseteq \gamma \sim y \equiv \beta \sim x \sqsubset \gamma \vee (\beta \sqsubset \gamma \wedge x = y)$$

Dans la suite nous pourrons adopter ce point de vue plus général. □

Remarque 3 :

La forme des u_i est également restrictive : il arrive que l'on rencontre des définitions de la forme de celles introduites aux définitions 5 et 6 pour lesquelles u_i est un terme plus complexe. Par exemple, toujours dans l'énoncé (II) (exemple 2 ch 2.1), on rencontre la définition :

croiss($\gamma \sim x \sim y$) ⇔ croiss($\gamma \sim x$) $\wedge$ $x \not\leqslant y$.

Nous ne développons pas cette généralisation. □

Notation :

Une description d'une définition c-récursive de f à partir de $\mathcal{C}$ est un couple

$$(\{ fy_1 \ldots u_i^1 \ldots y_q = \tau_1, \ldots, fy_1 \ldots u_i^k \ldots y_q = \tau_k \} , \varphi)$$

où $u_i^j = g_1 z_1 \ldots z_{p_j}$ et φ est la pré-condition de f .

3.1.2.- Construction de définitions c-récursives.

De manière analogue au cas des définitions s-récursives, nous énonçons ici des règles d'explicitations concernant des définitions c-récursives. A ces règles sont associées des stratégies d'explicitation c-récursives utilisées dans la construction de a-descriptions de S.I. à partir de s-descriptions. Ce sont ces stratégies qui ont été mises en oeuvre dans l'exemple 3 (TABSYM).

a) Règle de construction de définitions c-récursives :

Comme pour les définitions s-récursives , on est conduit à construire une définition c-récursive d'un symbole f lorsque l'on effectue une *abstraction* du S.I. dans lequel f est défini par une s-description. C'est sur le concept d'abstraction que repose la règle suivante (avec les notations du § 3.1.1) :

(RECR) Règle d'explicitation c-récursive

soit S un S.I. et f un symbole q-aire s-décrit par :

$$f\, y_1 \ldots y_q = z \Leftarrow \psi \text{ si } \varphi$$

La définition c-récursive pré-conditionnée de f à partir de $\mathcal{C}_i$

$$\bigwedge_{g \in \mathcal{C}_i} [\, pré\ fy_1 \ldots u_i \ldots y_q \supset fy_1 \ldots y_{i-1}\, u_i \ldots y_q = \tau_g]$$

est correcte vis à vis de ψ si, pour tout $g \in \mathcal{C}_i$,

$$\vdash_S \quad \forall y_1 \ldots y_q, z_1 \ldots z_p \ (pré\ fy_1 \ldots y_q \supset \psi[fy_1 \ldots u_i \ldots y_q/z]) \supset$$
$$\forall y_1 \ldots y_q, z_1 \ldots z_p \ (pré\ fy_1 \ldots u_i \ldots y_q \supset fy_1 \ldots y_{i-1}\, u_i \ldots y_q = \tau_g)$$

où $u_i = g\, z_1 \ldots z_p$

Théorème 3 :

Avec les notations précédentes, soit A le S.I. déduit de S en remplaçant l'axiome $\varphi \supset \psi[fy_1 \ldots y_q/z]$ par la définition c-récursive de f à partir de $\mathcal{C}_i$. Alors, si id est l'identité, (A, id) est une abstraction de S □

Le pas 2 de l'exemple 3 donne des exemples d'utilisation de la règle (RECR).

Mais une telle abstraction peut supprimer des solutions d'un problème. C'est en particulier ce qui risque de se passer s'il existe des termes v_i de $\overline{S_i}$ qui ne peuvent s'obtenir à l'aide des g de $\mathcal{C}_i$. Pour énoncer une règle d'explicitation c-récursive totale, analogue à (RESRT) du paragraphe 1.3.1 il faut une *propriété de forme normale*. Aussi repoussons nous au paragraphe 3.3.3 l'énoncé d'une telle règle.

b) Stratégie d'explicitation c-récursive

⊦ SECR ⊣ Stratégie d'explicitation c-récursive

donnée f caractérisée par $fy_1 \ldots y_q = z$ *tq* ψ dans un S.I. S
si φ

réultat $\mathcal{C}_i$ ensemble d'opérations de S de but $\overline{S_i}$ et définition c-récursive

$$\bigwedge_{g \in \mathcal{C}_i} [\ \textit{pré}\ f\ y_1 \ldots u_i \ldots y_q \supset\ fy_1 \ldots y_{i-1}\ u_i \ldots y_q = \tau_g[f/h,\ t_1/x_1 \ldots, t_p/x_p]\ ]$$

tq (RECR) est vérifiée

Présentation :

Dans la nouvelle description du S.I. S on place la s-description initiale de f à gauche et la a-description de f dans la colonne de droite Ceci peut conduire à introduire de nouveaux intermédiaires.

Heuristiques :

- Choix de τ_i (en général constructeurs de $\overline{S_i}$, cf § 5.2.2)
- Ne pas prendre en compte les g pour lesquels $\varphi\ [g\ z_1 \ldots z_p / y_i] =$ *faux*
- Pour déterminer τ_g procéder par unification entre

$\psi\ [fy_1 \ldots gz_1 \ldots z_p \ldots y_q / z]$ et ψ

3.2.- Systèmes d'information algébriques.

Les définitions données ici sont très voisines des définitions classiques de types abstraits algébriques, qu'elles soient énoncées en termes de "systèmes algébriques" [GUT, 78 b] , de théories algébriques [GOG, 77 b] , de description de type abstrait [LES, 79] ou de type abstrait de données [BRO, 79 b] .

3.2.1.- a-description d'un symbole fonctionnel

Avec des notations voisines des s-descriptions, une description algébrique (ou a-description) d'un symbole fonctionnel S à partir d'un ensemble $\mathcal{C}$ de symboles fonctionnels de but $\overline{S_i}$ est un couple formé :

i) d'une description de profil (cf ch 1.2 § 6.1.1. b) i))

ii) d'une description de l'un des deux types suivants :

iii1) description s-récursive qui est un couple (cf § 1.1) :

- $f\ y_1 \ldots y_q = \tau$
- φ (pré-condition de f , c'est une formule dont les seules variables libres sont $y_1 \ldots y_q$)

ii2) description c-récursive à partir de $\mathcal{C}_i$, c'est un couple (cf § 3.1.1) :

- $\{f\ y_1 \ldots u_i^1 \ldots y_q = \tau_1, \ldots,\ f\ y_1 \ldots u_i^k \ldots y_q = \tau_k\}$
- φ (formule de seules variables libres $y_1 \ldots y_q$)

Par exemple :

. finbloc : *fonct* (TABSYM) TABSYM

. $\begin{cases} \text{finbloc(debbloc(ts))} = \text{ts} \\ \text{finbloc(insert(ts, idnt, a))} = \text{finbloc(ts)} \end{cases}$

. *pré* finbloc(ts) = ¬vide(ts)

est une a-description du symbole f à partir de $\mathcal{C}$ = { debbloc, insert, tabvide }

L'axiome défini par une a-description est, selon le cas, la définition s-récursive ou la définition c-récursive correspondante de f .

Par exemple, puisque :

pré finbloc(debbloc(ts)) = ¬vide(debbloc(ts)) ∧ *pré* debbloc(ts) = *vrai*

pré finbloc(insert(ts, ident, a)) = ¬vide(insert(ts, ident, a)) ∧
¬pleine(ts) ∧ ¬presom (ts, ident)
∧ ¬vide (ts)
= ¬pleine (ts) ∧ ¬presom (ts, ident)
∧ ¬vide (ts)

l'axiome défini par la a-description de finbloc est donc :

finbloc(debbloc(ts)) = ts ∧
¬pleine(ts) ∧ ¬presom(ts, ident)∧ ¬vide(ts) ⊃ finbloc(insert(ts, ident,a)) = finbloc(ts).

3.2.2.- Forme d'une a-description d'un système d'information.

La forme des a-descriptions des systèmes d'information est très voisine de celle des s-descriptions (ch 1.2 § 5). Nous nous contentons de la décrire succintement (figure 4).

Commentaires sur cette figure :

a) $\overline{S_1}...\overline{S_k}$ sont les sortes d'intérêt de S. On n'a pas à les définir ici en termes de S.I. supports. Dans le cas de S.I. uni-sorte on continue à noter $\overline{S}$ la sorte d'intérêt de S.

b) Les descriptions des paramètres formels sont les mêmes que dans le cadre des s-descriptions.

c) Les P_i sont les descriptions des profils des opérations f_i introduites par la a-description.

d) Les $s\text{-}A_i$ décrivent les équations de définitions c-récursives ou s-récursives.

- Les fonctions f_i auxquelles ne sont associés aucun $s\text{-}A_i$ sont les *constructeurs* du S.I. S . On note $\mathcal{C}ons(S)$ leur ensemble . Par exemple $\mathcal{C}ons(TABSYM)$ = {tabvide, debbloc, insert } .

- Les autres fonctions f_i sont les *simplificateurs* de S . On distingue deux types de simplificateurs :

  . Les destructeurs qui sont des symboles dont le but est l'une des sortes d'intérêt de S . On note $\mathcal{D}est(S)$ leur ensemble. Par exemple $\mathcal{D}est(TABSYM)$ = {finbloc} .

lexique ou s-description	profil	a-description
S.I. S		
Description des sortes d'intérêt en termes de supports et description des invariants	ST_1 : *S.I.*	*sortes* $\overline{S_1},\ldots,\overline{S_k}$ *param* ST_1 *mani de*
	P^1_1 ⋮ $P^1_{k_1}$	*tq* A^1_1 ⋮ *tq* $A^1_{k_1}$
	ST_2 :	ST_2 *mani de*
	Q_1 ⋮ Q_ℓ	$s\text{-}B_1$ ⋮ $s\text{-}B_\ell$
		fparam *opérations*
s-description des différentes opérations.	P_1 ⋮ P_m ⋮ P_n	$a\text{-}A_1$ ⋮ $a\text{-}A_m$
		restrictions
		R_1 ⋮ R_n

Figure 4 : forme générale d'une a-description.

. Les sélecteurs dont le but n'est pas une sorte d'intérêt de S . On note $\mathcal{S}elt(S)$ leur ensemble. Par exemple

$\mathcal{S}elt(TABSYM)$ = { valattr, presom, taille, pleine, vide }

e) Les P_i sont les pré-conditions des f_i . Ainsi les différents triplets :

$(P_i, s\text{-}A_i, R_i)$ pour $1 \leqslant i \leqslant m$

$(P_i, \emptyset, R_i)$ pour $m+1 \leqslant i \leqslant n$

sont les a-descriptions des opérations introduites par la a-description du S.I.

Comme d'habitude lorsque *pré* $f\ x_1 \ldots x_q$ = *vrai* on se dispense de le faire figurer dans la description.

f) Les S.I. apparaissant dans la description des profils ou en paramètre sont les S.I. *externes* de S .

3.2.3- Système d'information algébrique

Un *système d'information algébrique* est un système d'information défini par une a-description. Sa définition est similaire à celle d'un système d'information défini par une s-description (ch 1.2 § 5.2.2. et § 6.2.2.) à condition de remplacer dans la définition des extensions, "les axiomes définis par les s-descriptions des opérations", par "les axiomes définis par les a-descriptions des opérations ".

3.2.4.- Utilité des pré-conditions.

Définition 7 :

Soit S un S.I. tel qu'à chaque symbole fonctionnel q-aire f soit associéeune pré-condition notée *pré* f ; c'est une formule booléenne dont les seules variables libres sont $x_1 \ldots x_q$ de profils respectifs les constituants $\bar{S}_1, \ldots, \bar{S}_q$ de la source de f . On associe alors à chaque terme u de S une pré-condition *pré* u définie par récurrence sur la longueur de u :

- si u est une variable, *pré* u = *vrai*
- si $u = f\ u_1 \ldots u_n$, *pré* u = *pré* $f[u_1/x_1, \ldots, u_q/x_q] \wedge \bigwedge_{1 \leqslant i \leqslant q}$ *pré* u_i □

Les S.I. définis par des s-descriptions ou des a-descriptions et tels que leurs sous systèmes soient décrits de cette manière vérifient l'hypothèse précédente sur S .

Définition 8 :

Avec l'hypothèse de la définition 7, un terme u de S est *défini dans* S si *pré* u = *vrai* est un théorème de S . □

Nous discutons de l'utilité des pré-conditions à partir d'un exemple.

Exemple 4 :

Les files à taille limitée (extension de l'exemple 1 ch 1.2)

Indiquons rapidement (figure 5) une a-description du S.I. FILE sur un S.I. V, dont les éléments ont une taille limitée par n .

S.I. FILE	
V : *S.I.* n : ENTIER fvide : *op* FILE ajout : *fonct*(FILE, V)FILE retir : *fonct*(FILE)FILE tête : *fonct* (FILE) V vide : *fonct* (FILE)BOOL taille : *fonct*(FILE)ENTIER	*param* n ⩾ 1 *fparam* *opérations* retir(ajout(f, v)) = *si* vide(f) *alors* fvide *sinon* ajout(retir(f), v) tête(ajout(f, v)) = *si* vide(f) *alors* v *sinon* tête(f) vide(fvide) = *vrai* vide(ajout(f, v)) = *faux* taille(fvide) = 0 taille(ajout(f, v)) = taille(f) +1 *restrictions* *pré* retir(f) = ¬vide(f) *pré* ajout(f) = (taille(f) < n) *pré* tête(f) = ¬vide(f)

Figure 5 : a-description du S.I. FILE.

Dans cet exemple les principales opérations sont des fonctions partielles. Mais on aurait pu employerun point de vue opposé en n'adoptant que des fonctions totales grâce à l'introduction d'éléments indéfinis.

Le deuxième point de vue permet d'introduire des diagnostics d'erreurs : à chaque type d'erreur peut être associé un élément indéfini particulier. On impose ainsi une forte contrainte sur l'implantation ultérieure. Il faut d'autre part introduire un ensemble d'axiomes caractérsisant les erreurs, c'est le point de vue de GOGUEN [GOG, 77 c] .

Le premier point de vue est plus simple à mettre en oeuvre d'un point de vue axiomatique et laisse le plus grand choix à l'implanteur. Mais il nécessite un effort théorique pour préciser la forme des pré-conditions admissibles et pour étudier dans quelle mesure les résultats valables pour des fonctions totales le restent pour des fonctions partielles.

C'est un problème important si on veut disposer de systèmes automatiques de preuves tels que le système AFFIRM de MUSSER [MUS, 79] (cf § 3.3).

On peut aussi, comme [GUT, 79] , accepter la cohabitation des deux points de vue dans une spécification : dans l'exemple précédent on peut ne pas définir "tête(fvide)", mais préciser que "ajout (f, v)" provoque un débordement lorsque "taille(f) ⩾ n ". □

3.3.- Le concept de forme normale.

3.3.1.- Induction structurelle dans un système algébrique.

Au paragraphe 2 nous avons introduit les spécifications algébriques pour permettre l'explicitation récursive d'un énoncé. La propriété fondamentale qui était ainsi cherchée était la possibilité de "parcourir" les différents objets de la sorte considérée. On définit plus formellement cette propriété de la façon suivante :

Théorème 4 :

Règle d'induction structurelle.

Soit S un S.I. uni-sorte (pour simplifier les notations) et φ un prédicat sur S (i.e une formule booléenne à une seule variable libre x de profil S).

Si pour tout constructeur ou destructeur f de S de profil

$$\overline{S_1}, \ldots \overline{S_{i-1}}, \overline{S}, \overline{S_{i+1}}, \ldots, \overline{S_q} \rightarrow \overline{S}$$

pour tout q-1 uplet $u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_q$ de profils respectifs

$\overline{S_1}, \ldots, \overline{S_{i-1}}, \overline{S_{i+1}}, \ldots, \overline{S_q}$ et tout terme u de profil $\overline{S}$ on a

pré $f\ u_1 \ldots u_{i-1} u\ u_{i+1} \ldots u_q \wedge \varphi[u/x] \supset \varphi[f\ u_1 \ldots u_{i-1} u\ u_{i+1} \ldots u_q / x]$

alors $\varphi[v/x]$ est un théorème pour tout terme v de profil $\overline{S}$. □

Démonstration :

La justification de cette règle est immédiate par récurrence sur la longueur de v :
s'il existe des termes définis v de profil $\overline{S}$ pour lesquels $\varphi[v/x]$ n'est pas un théorème, il en existe un plus court (au sens large) que les autres noté $f\ u_1 \ldots u_{i-1} u\ u_{i+1} \ldots u_q$.

Puisqu'il est l'un des plus courts , $\varphi[u/x]$ est un théorème, donc par hypothèse de la règle d'induction $\varphi[f u_1 .. u_{i-1} u .. u_q / x]$ en est un également, d'où une contradiction
cqfd

Remarque :

Cette règle d'induction est placée au niveau du méta-raisonnement. Nous avions déjà noté (ch 1.3 § 6.2.2. a)) que si l'on souhaite définir un logiciel permettant d'effectuer des démonstrations dans les systèmes algébriques [MUS, 79] il faut introduire, comme dans [NAK, 79] ou [GOG, 77 a] cette règle comme règle d'inférence.

Signalons également qu'elle est généralisable immédiatement aux S.I. multi-sortes.

Exemple 5 :

Essayons de démontrer que toute file bornée f (exemple 4) vérifie

$\varphi \equiv 0 \leqslant \text{taille}(f) \leqslant n$

(1) φ [fvide/f] est un théorème puisque taille(fvide) ≡ 0 et $n \geqslant 0$

(2) *pré* ajout (f, v) $\wedge \varphi \equiv$ taille(f) $< n \wedge 0 \leqslant$ taille(f) $\leqslant n$
$\equiv 0 \leqslant$ taille(f) $< n$
$\supset 0 \leqslant$ taille(f) +1 $\leqslant n$
$\supset 0 \leqslant$ taille(ajout(f, v)) $\leqslant n$

Mais il faut aussi prouver :

- *pré* retir(f) $\wedge \varphi \supset \varphi$ [retir(f)/f]

soit $0 <$ taille(f) $\leqslant n \supset 0 \leqslant$ taille(retir(f)) $\leqslant n$

Malheureusement on ne dispose d'aucun axiome sur taille(retir(f)). On cherche alors à prouver le lemme suivant :

Lemme :

pré retir(f) $\supset$ taille (retir(f)) ≡ taille(f) -1

On est à nouveau tenté d'effectuer une démonstration par induction. Le résultat est immédiat pour f = fvide ou f = ajout(f_1, v) . Mais on a des difficultés à le démontrer pour f = retir(f_1) car on ne dispose pas d'axiomes caractérisant retir(retir(f_1)).

On peut effectuer une démonstration en raisonnant sur la longueur des schémas fonctionnels mais on sort alors du cadre axiomatique du système algébrique considéré. On peut aussi essayer d'identifier retir(retir(f_1)) à un schéma contenant ajout de façon à pouvoir "supprimer" un retir.

Plus généralement, nous sommes gênés dans cette démonstration parce que les axiomes de FILE ne précisent la valeur des différentes opérations que sur fvide et ajout et non pas sur retir. Il nous faut prouver que cette caractérisation est suffisante pour préciser les propriétés que l'on attend du type FILE : c'est le *lemme de forme normale* que nous étudions maintenant. □

3.3.2.- Forme normale et implantation directe

Les constructeurs d'un système défini par une a-description ont un rôle privilégié : ce sont eux qui permettent d'obtenir, par composition, tous les termes des sortes d'intérêt. En reprenant la définition de GUTTAG [GUT,78a] :

Définition 9

Un S.I. algébrique uni-sorte S possède la *propriété de forme normale* si, pour tout terme défini u de profil $\bar{S}$, il existe un terme v de S ne contenant pas de simplificateur de S, défini et tel que

$u \equiv v$ soit un théorème de S. □

Ainsi, dans un S.I. défini par une a-description et possédant la propriété de forme normale, les classes d'équivalence pour ≡ des termes ayant pour profil une sorte d'intérêt, ont un représentant ne contenant que des constructeurs.

Cette définition se généralise immédiatement aux S.I. algébriques multi-sortes.

Exemple 6

Le système FILE(V,n), défini à l'exemple 4, possède la propriété de forme normale :

Pour tout terme f de profil FILE, il existe $v_1 \ldots v_k \in V$ tels que

$f \equiv \text{ajout}^k(\text{fvide}, v_1, \ldots, v_k)$ (†)

Par hypothèse si $f \equiv \text{fvide}$ la propriété est vérifiée.

si $f \equiv \text{ajout}(f_1, v)$ la propriété est encore vraie par hypothèse d'induction.

si $f \equiv \text{retir}(f_1)$ alors par hypothèse :

$f_1 \equiv \text{ajout}^{k_1}(\text{fvide}, v_1, \ldots, v_{k_1})$

donc $\text{retir}(f_1) \equiv \text{ajout}^{k_1-1}(\text{fvide}, v_2, \ldots, v_{k_1})$ en appliquant k fois l'axiome 1 de FILE.

De manière analogue on peut prouver que TABSYM (exemple 3) possède la propriété de forme normale [DER,79].

(†) $\text{ajout}^k(\text{fvide}, v_1, \ldots, v_k)$ représente $\underbrace{\text{ajout}(\ldots(\text{ajout}}_{k \text{ fois}}(\text{fvide}, v_1), v_2) \ldots v_k)$

et $\text{ajout}^0(\text{fvide}, v_1, \ldots, v_0) \equiv \text{fvide}$.

Pour tout t_s : TABSYM, $t_s \equiv$ tabvide ou

il existe t'_s , $k \geq 0$, $id_1, \ldots, id_k$: CHAINE, $a_1, \ldots, a_k$: AT tels que

$t_s \equiv \text{insert}^k(\text{debbloc}(t'_s), id_1, a_1, id_2, a_2, \ldots, id_k, a_k)$. □

De cette définition on tire immédiatement le théorème suivant qui est la règle d'induction structurelle dont les hypothèses sont restreintes aux constructeurs.

Théorème 5

Soit S un S.I. algébrique uni-sorte possédant la propriété de forme normale. Soit φ un prédicat sur $\bar{S}$ de variable libre x.

Si pour tout constructeur f de S de profil $\bar{S}_1, \ldots, \bar{S}_{i-1}, \bar{S}, \bar{S}_{i+1}, \ldots, \bar{S}_q \rightarrow \bar{S}$ et tout q-uplet $u_1, \ldots, u_{i-1}, u, \ldots, u_q$ de profils respectifs les éléments de la source de f

$\vdash_{\bar{S}} pré\ f\ u_1 \ldots u_{i-1}\ u\ u_{i+1} \ldots u_q \wedge \varphi[u/x] \supset \varphi[f\ u_1 \ldots u_{i-1} u \ldots u_q / x]$

alors $\varphi[v/x]$ est un théorème pour tout terme v de profil $\bar{S}$. □

De ce théorème on déduit la validité de la démonstration de l'exemple 5. Cette propriété de forme normale, outre qu'elle a des implications très importantes à la fois dans la démonstration de propriétés d'un système algébrique (§ 3.4) ou la construction de représentations (§6), permet de donner une vue plus "opératoire" à une spécification algébrique : c'est le concept d'*implantation directe* introduit par GUTTAG.

Dans un système d'information algébrique S possédant la propriété de forme normale et tel que la forme normale d'un terme quelconque soit unique, on peut associer à chaque terme d'une sorte d'intérêt une représentation arborescente de sa forme normale. Notons $\hat{S}_i$ l'ensemble des arborescences associées à la sorte $\bar{S}_i$. Alors on définit une *interprétation libre*, ou *interprétation de HERBRAND* de S (cf. ch. 1.3 § 4.1.3) en associant à chaque symbole f de profil $\bar{S}_1 \ldots \bar{S}_q \rightarrow \bar{S}_{q+1}$, qui n'est pas un constructeur, une fonction $\hat{f}$ de $\hat{S}_1 \times \ldots \times \hat{S}_q$ dans $\hat{S}_{q+1}$ vérifiant les axiomes correspondant à f. Cette interprétation est appelée *implantation directe* de S. LESCANNE propose dans [LES,79] une définition de l'implantation directe d'un type abstrait algébrique en termes de *ramifications*. Ceci fournit une représentation canonique de tout S.I. défini par une a-description et possédant la propriété de forme normale.

Exemple 7

Le terme ajout(retir(ajout(ajout(ajout(fvide,v_1),v_2),v_3)),v_4) se représente sous la forme de l'arbre de la figure 6.

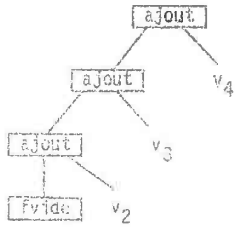


figure 6 □

Ce concept est utile d'un point de vue méthodologique. D'une part il permet de "dessiner" la forme d'un objet du type. D'autre part il permet la construction d'un système validant les opérations, ce qui est un début de réponse au problème de la validité d'une spécification (ch. 1.4 § 1.4).

C'est ainsi que MUSSER [MUS,79], développe le système AFFIRM permettant de vérifier qu'un type abstrait possède la propriété de forme normale. Ce système repose sur les notions suivantes :

L'ensemble des axiomes d'une description algébrique d'un S.I. peut être considéré, si on oriente les égalités, comme un système de réécriture [HUE,77]. Un tel système possède la *propriété de CHURCH-ROSSER* si, de manière intuitive, l'ordre d'application des règles pour passer d'un terme u à un terme v n'a pas d'importance. Une forme normale peut alors être définie comme un terme *irréductible*, c'est-à-dire invariant par application des règles de réécriture.

Un S.I. algébrique est *convergent* [MUS,78] si :

1) Il possède la propriété de CHURCH-ROSSER. Pour vérifier cette propriété on dispose de l'algorithme de KNUTH BENDIX [KNU,70].

2) Il possède la propriété de *terminaison finie*, c'est-à-dire que toute suite de réécriture d'un terme quelconque est finie. MUSSER [MUS,78] propose un critère pour vérifier cette propriété qui malheureusement n'est pas applicable dans tous les cas (cf. [LES,79]).

Si un S.I. est convergent il possède la propriété de forme normale et la forme normale d'un terme quelconque est unique.

De plus, l'utilisation de l'algorithme de KNUTH BENDIX permet d'ajouter des axiomes à un S.I. non confluent pour le rendre confluent. C'est donc un outil méthodologique puissant.

Ces idées sont développées et étudiées dans [LES,79].

Remarque 1

La méthode de spécification de type abstrait introduite par PARNAS [PAR,77] et fondée sur le concept de *traces* est une utilisation particulière des formes normales : les axiomes caractérisant les sélecteurs ne sont définis que sur les formes normales (assertions de valeurs) et des *assertions d'équivalences* axiomatisent les destructeurs. Enfin des *assertions de légalité* jouent, en partie, le rôle des préconditions en indiquant quelles compositions de symboles fonctionnels ont un sens.

3.3.3.- Forme normale et définitions c-récursives

La propriété de forme normale permet de caractériser les définitions c-récursives qui conservent intégralement les propriétés de s-descriptions correspondantes.

RECRT Règle d'explicitation c-récursive totale

Soit S un S.I. algébrique possédant la propriété de forme normale et f un symbole q-aire de S défini par la s-description suivante :

$f\ y_1 \ldots y_q = z$ *lq* ψ *ai* φ

La définition c-récursive préconditionnée de f à partir de l'ensemble $\mathcal{C}ons(S)$ des constructeurs de S

$$\bigwedge_{g \in \mathcal{C}ons(S)} [\underline{pré}\ f\ y_1 \ldots u \ldots y_q \supset f\ y_1 \ldots u_i \ldots y_q \equiv \mathcal{C}g]$$

est *totalement correcte* vis-à-vis de ψ si, pour tout $g \in \mathcal{C}ons(S)$,

$$\vdash_S \forall y_1,\ldots,y_q,z_1,\ldots,z_p\ (\underline{pré}\ f\ y_1 \ldots u_i \ldots y_q \supset \psi[f\ y_1 \ldots u_i \ldots y_q / z]) \leftrightarrow$$
$$\forall y_1,\ldots,y_q,z_1,\ldots,z_p\ (\underline{pré}\ f\ y_1 \ldots u_i \ldots y_q \supset f\ y_1 \ldots u_i \ldots y_q \equiv \mathcal{C}g)$$

où $u_i = g\ z_1 \ldots z_p$

Comme pour les définitions s-récursives, toute abstraction définie par une explicitation c-récursive vérifiant cette règle conserve les solutions de tout problème. Plus précisément :

Théorème 6

Avec les notations du théorème 3, si la définition c-récursive de f vérifie la règle précédente, l'abstraction (A,id) de S est *complète*. □

Avec (RECRT) on peut donc construire pas à pas une abstraction d'un S.I. préservant les solutions de tout problème.

3.3.4.- Forme normale et s-description

La construction d'une abstraction d'un S.I. S possédant la propriété de forme normale peut être facilitée si S est défini par une s-description. Il s'agit en effet de choisir les constructeurs de telle manière qu'ils engendrent tous les termes de S. Illustrons cette idée sur l'exemple de TABSYM (exemple 3) :

Exemple 8

D'après la s-description de TABSYM (ch. I.2 exemple 11, figure 10), tout terme de sorte TABSYM est un couple :

<<b,re>> où re est un ensemble de triplets <<x,nb,a>> vérifiant en particulier $0 \leq nb \leq b$.

Si l'on cherche à exprimer ce couple à l'aide de certaines opérations de TABSYM, il est naturel de distinguer trois cas :

- b = 0, et alors nécessairement <<b,re>> = <<0,$\emptyset$>>, d'où le constructeur 0-aire tabvide
- b > 0 et,tout nb vérifie nb < b alors <<b,re>> = debbloc(<<b-1,re>>)
- b > 0 et il existe <<x,nb,a>> tel que nb = b
  alors <<b,re>> = insert(<<b,re{<<x,nb,a>>}>>,x,a).

Ici encore, plutôt que de chercher *à prouver à posteriori* qu'un S.I. vérifie la propriété de forme normale (cf. exemple 6), il est plus intéressant de *construire* ce S.I. de telle manière qu'il possède cette propriété. Cette construction consiste, en fait, à choisir les constructeurs du S.I.

3.3.5.- Comparaison des a-descriptions et des s-descriptions

La propriété de forme normale permet de considérer tout S.I. algébrique uni-sorte la possédant comme étant défini par une s-description pour laquelle les termes de la sorte support de la sorte d'intérêt sont les formes normales.

Exemple 9

A la description algébrique de TABSYM donnée à l'exemple 3 on peut substituer la s-description suivante

- le S.I. support de la sorte d'intérêt TABSYM est l'ensemble T solution unique de l'équation :

  $T = \{tabvide\} \cup insert(T,\overline{CHAINE},\overline{AT}) \cup debbloc(T)$

- Les s-descriptions des opérations sont :

. presom *tq*

$$presom(t_s,ident) = \underline{si}\ t_s = tabvide\ \underline{alors}\ faux\ \underline{sinon}\ \underline{si}\ \exists\, t_{s_1}\ \underline{tq}\ t_s = debbloc(t_{s_1})\ \underline{alors}\ faux\ \underline{sinon}\ \exists\, t_{s_1}, ident_1, a_1\ \underline{tq}\ t_s = insert(t_{s_1}, ident_1, a_1) \wedge ident = ident_1 \wedge presom(t_s, ident$$

. valattr *tq*

$$valattr(t_s,ident) = \underline{si}\ t_s = tabvide\ \underline{alors}\ \omega\ \underline{sinon}\ \underline{si}\ \exists\, t_{s_1}\ \underline{tq}\ t_s = debbloc(t_{s_1})\ \underline{alors}\ valattr(t_{s_1},ident)\ \underline{sinon}\ \exists\, t_{s_1}, ident_1, a_1\ \underline{tq}\ (t_s = insert(t_{s_1},ident_1,a_1) \wedge \underline{si}\ ident = ident_1\ \underline{alors}\ a_1\ \underline{sinon}\ valattr(t_{s_1},ident))$$

. finbloc *tq*

$$finbloc(t_s) = \underline{si}\ \exists\, t_{s_1}\ \underline{tq}\ t_s = debbloc(t_{s_1})\ \underline{alors}\ t_{s_1}\ \underline{sinon}\ \exists\, t_{s_1}, ident_1, a_1\ \underline{tq}\ t_s = insert(t_{s_1}, ident_1, a_1) \wedge finbloc(t_{s_1})$$

$\underline{si}\ \neg vide(t_s)$

- Nous ne détaillons pas la description des autres opérations.

Pour compléter cette s-description il faut en préciser l'invariant. En effet dans la description précédente nous identifions une table des symboles à un mot du langage T. Mais il est clair que tout élément de ce langage n'est pas une représentation d'une table des symboles : il faut tenir compte des préconditions dans la composition des schémas fonctionnels (une c-grammaire ne suffit pas à décrire TABSYM). Les restrictions proviennent ici uniquement des préconditions de insert :

$$I_a(t_s) = (t_s \equiv \text{insert}(t_{s_1}, \text{ident}_1, a_1)) \Rightarrow$$
$$\neg\text{pleine}(t_{s_1}) \wedge \neg\text{reprasom}(t_{s_1}, \text{ident}_1) \wedge \neg\text{vide}(t_{s_1}))$$

que l'on peut encore écrire :

$$I_a(t_s) = t_s \equiv \text{insert}^k(\text{debbloc}(t_{s_1}), id_1, a_1, id_2, a_2, \ldots, id_k, a_k) \Rightarrow$$
$$(\text{taille}(t_s) \leq n) \wedge (\forall i, j \in [1 \ldots k], i \neq j \Rightarrow id_i \not\equiv id_j)$$
$$\wedge\ t_{s_1} \not\equiv \text{tabvide}$$

I_a est l'*invariant abstrait* de TABSYM. □

On voit sur cet exemple se dessiner une méthode de traduction systématique d'une description algébrique d'un système d'information possédant la propriété de forme normale en une s-description. Nous ne la formaliserons pas davantage.

Cette étude met bien en lumière la "quantité d'abstraction" obtenue lors du passage d'une s-description à une description algébrique : on ne donne plus d'axiomes sur les constructeurs, ce qui permet un plus grand choix de représentations (§ 6).

3.4.- Propriétés d'un système d'information algébrique.

3.4.1.- Semantique

De nombreuses études [GOG,77a], [GUT,78b], [BRO,79b], [LES,79] ont précisé les modèles associés aux systèmes algébriques, aussi nous contentons-nous ici encore de donner quelques idées fondamentales.

Un modèle d'un système d'information algébrique est une *algèbre hétérogène* [LES,79], encore appelée *structure de donnée* [BRO,79b]. Rappelons qu'un S.I. (consistant) non complet admet plusieurs modèles (ch. 1.2 § 3.3). Il importe, d'un point de vue méthodologique, de savoir quels sont les modèles les plus intéressants d'un système algébrique.

Exemple 10

Considérons le S.I. "séquences de chiffres" défini par la a-description de la figure 7.

S.I. SEQC

SEQC = CHIFFRE*	< > : *op* SEQC ∿ : *op*(SEQC,CHIFFRE) SEQC ≃ : *op*(SEQC,SEQC)BOOL	*opérations* < > ≃ < > = *vrai* u ∿ x ≃ v ∿ y = (u ≃ v) ∧ (x ≃ y)
≃ *tq* u ≃ v = ∀ i ∈[b_i(u) .. b_s(u)] u_i ≃ $v_{b_i(v)-b_i(u)+1}$		

figure 7 : a-description de SEQC

Alors si A_1 admet pour seul ensemble {0} et pour fonctions

id : {0} × {0} → {0} telle que id(0,0) = 0

eg égalité sur les entiers

l'interprétation (A_1, r) où

r(<>) = 0, r(*) = id, r(≡) = eg

est un modèle de SEQC.

Mais l'ensemble IN des entiers ayant pour fonction l'addition et la multiplication associé à r :

r(<>) = 0, r(u ∿ x) = r(u) × 10 + x, r(≡) = eg.

définit aussi un modèle de SEQC. □

Exemple 11

Les multi-ensembles (ch. 1.2, exemple 2).

Considérons la a-description du S.I. MULTENS de la figure 8.

Cette spécification est incomplète puisqu'on ne sait pas comparer u_1 = ajout(ajout(m,v_1),v_2) et u_2 = ajout(ajout(m,v_2), v_1). C'est d'ailleurs ce que nous cherchons dans le passage aux S.I. algébriques (cf. ch. 1.2, § 3.3.2). □

S.I. MULTENS

VAL : *S.I.* # peut être muni d'un ordre comme au chapitre 1.2 # mvide : *op* MULTENS ajtm : *fonct*(MULTENS,VAL)MULTENS apt : *op*(VAL,MULTENS)BOOL supm : *fonct*(MULTENS,VAL)MULTENS	*param* *fparam* *opérations* v apt mvide = *faux* v apt ajtm(m,v_1) = *si* v ≃ v_1 *alors vrai* *sinon* v apt supm(ajtm(m,v),v_1) = *si* v ≃ v_1 *alors* m *sinon* ajtm(supm(m,v_1), v) *restrictions* *ssi* supm(m,v) = v apt m

figure 8 : a-description de MULTENS

Dans le cas, souvent souhaité, de spécification algébrique incomplète que doit-on décider des formules qui ne sont pas des théorèmes et dont la négation n'est pas un théorème ? Par exemple que doit-on décider pour les termes u_1 et u_2 de l'exemple 11 ?

a) On peut supposer que, s'ils ne peuvent être démontrés égaux, ils doivent être considérés comme différents et, en particulier, ils doivent être représentés par deux objets différents. Ceci revient à définir la sémantique d'un S.I. algébrique comme étant l'*algèbre initiale* associée. L'algèbre initiale est intuitivement celle qui comporte "le plus d'éléments". De manière plus précise, rappelons qu'un *morphisme* d'une algèbre A vers une algèbre B est une application g telle que $g(f(x)) = g(f)(g(x))$ pour tout terme f(x) de A. Une algèbre I dans une classe d'algèbre $\mathcal{C}$ est dite initiale si, pour toute algèbre A de $\mathcal{C}$, il existe un unique morphisme surjectif de I sur A (NIVAT [NIV,75], parle de magma initial sur $\mathcal{C}$). Dans le cas d'un système algébrique, l'algèbre initiale associée est le quotient de l'algèbre des mots (ensembles des termes du système) par la relation d'équivalence associée aux axiomes ; nous avons vu que des représentants privilégiés étaient les formes normales.

Admettre que la sémantique d'un système algébrique en est l'algèbre initiale c'est rejoindre le point de vue de ZILLES [ZIL,75] et du groupe ADJ [GOG,77a]. Bien qu'il soit satisfaisant puisque l'algèbre initiale existe toujours il est contraignant d'un point de vue informatique car il impose que le spécificateur précise tous les objets qu'il souhaite identifier. Ainsi une représentation de MULTENS [ENTIER] en termes de tableau associant à chaque entier son nombre d'occurrences sera incorrecte.

Adopter ce point de vue fait donc perdre l'un des avantages du passage au niveau algébrique, qui était de ne plus prendre en compte des contraintes superflues.

b) De façon duale on peut considérer que deux termes qui ne peuvent être démontrés différents sont égaux. Ceci revient à choisir pour sémantique d'un système algébrique l'*algèbre terminale*. Intuitivement c'est celle qui contient le "moins d'éléments". Elle est caractérisée par le fait qu'il existe, pour toute algèbre A de la classe considérée, un morphisme surjectif de A vers elle. Malheureusement elle n'existe pas toujours [BRO,79b].

c) Finalement, pour laisser la plus grande latitude à l'implanteur du système il est raisonnable de se rallier au point de vue de GUTTAG et al [GUT,78a]. BROY et al [BRO,79b] ou LESCANNE [LES,79] qui consiste à prendre pour sémantique possible l'ensemble des modèles.

3.4.2.- Consistance

Dans le cas de spécifications algébriques la forme plus simple des axiomes permet d'avoir des critères de consistance.

Exemple 12

La consistance du système algébrique MULTENS (exemple 11) provient du fait que le membre gauche d'une équation quelconque ne peut se réduire au membre gauche d'une autre équation ; autrement dit chaque équation caractérise l'effet d'une fonction sans qu'il y ait de contradictions possibles entre elles. En revanche si on ajoute à cette description l'axiome :

(4) v_1 apt supm(m,v_2) ≡ *si* $v_1 \equiv v_2$ *alors faux sinon* v_1 apt m

on obtient un système inconsistant puisque, par exemple :

v apt supm(ajtm(ajtm(mvide,v),v),v) ≡ (3)

v apt ajtm(mvide,v) ≡ *vrai* (2)

or v apt supm(ajtm(ajtm(mvide,v),v),v) ≡ *faux* (4) □

L'utilisation des propriétés des systèmes de réécriture donne un critère de consistance : tout système convergent (possédant la propriété de forme normale) est consistant [MUS,78]. La justification intuitive est celle donnée dans le cas de l'exemple précédent ; une démonstration précise se trouve dans [MUS,78].

3.4.3.- Complétude suffisante

Pour définir le concept de complétude suffisante (ch. 1.2 § 4.2.1. déf.20) d'un S.I. algébrique, considérons ce dernier comme une extension de la réunion de ses S.I. externes.

Définition 10

Soit S un S.I. algébrique. S est *suffisamment complet* si tout terme défini u, dont le profil est une sorte externe S_i de S, admet une forme normale ne contenant que des opérations de S_i. □

Exemple 13

Le S.I. MULTENS (figure 8) est suffisamment complet. Il s'agit de prouver que, pout tout v ∈ V, pour tout terme m de profil MULTENS, on peut démontrer l'un des deux théorèmes : v apt m ≡ *vrai* ou v apt m ≡ *faux*.

Par induction structurelle :

$$\text{- } m \equiv mvide \supset v \text{ apt } m \equiv \underline{faux} \qquad (1)$$

$$\text{- } m \equiv ajtm(m_1,v_1) \wedge (v \text{ apt } m_1 \equiv \underline{vrai} \vee v \text{ apt } m_1 \equiv \underline{faux})$$
$$\supset (\underline{si}\ v \equiv v_1\ \underline{alors}\ v \text{ apt } m \equiv \underline{vrai}\ \underline{sinon}\ v \text{ apt } m \equiv v \text{ apt } m_1) \wedge$$
$$(v \text{ apt } m_1 \equiv \underline{vrai} \vee v \text{ apt } m_1 \equiv \underline{faux})$$
$$\supset \underline{si}\ v \equiv v_1\ \underline{alors}\ v \text{ apt } m \equiv \underline{vrai}\ \underline{sinon}\ (v \text{ apt } m \equiv \underline{vrai} \vee v \text{ apt } m \equiv \underline{faux}) \quad \square$$

Bien entendu le problème de la complétude suffisante d'un système quelconque est indécidable [GUT,78b], mais on trouve dans cet article l'énoncé d'un critère de complétude suffisante lié à la forme des termes et leur degré d'imbrication. Mais là encore le système AFFIRM de MUSSER [MUS,79] permet de démontrer, dans un grand nombre de cas, la complétude suffisante d'un système et, lorsqu'elle n'est pas vérifiée, de compléter suffisamment l'axiomatisation (toujours à l'aide de l'algorithme de KNUTH BENDIX). On dispose donc là d'un outil de spécification permettant d'éviter certaines formes de sous-spécification.

4.- OPERATIONS SUR LES DESCRIPTIONS ALGEBRIQUES.

Il est intéressant de définir les opérations sur les systèmes d'information (ch. 1.2 § 5 et § 7) en termes algébriques, ce qui confère une propriété de stabilité à ce genre de description. Ainsi les systèmes obtenus par application de ces opérations à partir de systèmes algébriques sont encore algébriques et possèdent donc les propriétés étudiées précédemment.

Les opérations "incrémentales" sur les S.I. (extension, restriction, dérivation, introduction de sous-sortes, réunion) peuvent être évidemment définies en termes algébriques, si l'on restreint les définitions de symboles fonctionnels à des a-descriptions. C'est d'ailleurs en termes d'extension que l'on pourrait caractériser le S.I. défini par une a-description (§ 3.2) de façon analogue à ce que nous avons fait dans le cas des s-descriptions (ch. 1.2 § 5.2). C'est aussi de cette manière que le groupe ADJ définit "enrich", "derive", "combine" [BUR,77a].

Nous montrons simplement ici comment les différentes opérations sur le S.I. définies à l'aide de la paramétrisation peuvent s'exprimer en termes de a-description.

4.1.- Produit cartésien.

Le S.I. PROCAR décrit en termes algébriques sur la figure 9 est, en reprenant les notations du chapitre 1.2 § 7.1.1, égale à $<<p_1:S_1,p_2:S_2>>$

S.I. PROCAR	
S_1 : *S.I.*	*param*
S_2 : *S.I.*	*fparam*
<< >> : *fonct*(S_1,S_2) PROCAR	*opérations*
p_1 : *fonct*(PROCAR)S_1	$p_1(<<x_1,x_2>>) = x_1$
p_2 : *fonct*(PROCAR)S_2	$p_2(<<x_1,x_2>>) = x_2$
eg : *fonct*(PROCAR,PROCAR)BOOL	$eg(<<x_1,x_2>>,<<y_1,y_2>>) = (x_1 = x_2) \wedge (y_1 = y_2)$

figure 9 : a-description du S.I. produit cartésien

Dans cette description, la notation $<<x_1,x_2>>$ est préférée à $<<>>(x_1,x_2)$

Nous introduisons un prédicat d'égalité pour lequel, avec les conventions faites au chapitre 1.2 § 2, la notation "=" sera préférée à "eg".

Notons enfin que dans cette description les noms des projections sont fixés. En fait on peut imaginer qu'il y a autant de S.I. PROCAR que de symboles de projections.

4.2.- Ensemble des parties.

Montrons simplement que l'on peut définir les principales opérations de ce S.I. paramétré de manière algébrique (figure 10). Ici encore, comme au chapitre 1.2 § 7.1.2. la liste des opérations n'est pas exhaustive. La a-description nécessite de déterminer des constructeurs, ici $\emptyset$ et adj. Nous continuons, bien entendu, à utiliser les notations introduites au chapitre 1.2 § 7.1.2 ; en particulier $\mathcal{P}(S)$ est toujours préféré à ENS(S).

S.I. ENS

$a \subset b = \forall x, x \in a \supset x \in b$

$x \in a \cap b = x \in a \wedge x \in b$

$x \in a \cup b = x \in a \vee x \in b$

$x \in \{\ \}\ \varphi\ = appl(\varphi, x)$

S : *S.I.*

param
fparam
opérations

adj : *fonct*(ENS,S)ENS
$\in$: *op*(S,ENS)BOOL
$\emptyset$: *cp* ENS

$x \in \emptyset$ = *faux*
$x \in adj(a,y)$ = *si* x = y *alors vrai* *sinon* $x \in a$

$\subset$: *op*(ENS,ENS)BOOL

$\emptyset \subset a$ = *vrai*
$adj(a,x) \subset b = x \in b \wedge a \subset b$

$\cap$: *op*(ENS,ENS)BOOL

$\emptyset \cap a = \emptyset$
$adj(a,x) \wedge b$ = *si* $x \in b$ *alors* $adj(a \cap b, x)$ *sinon* $a \cap b$

$\cup$: *op*(ENS,ENS)BOOL

$\emptyset \cup a = a$
$adj(a,x) \cup b = adj(a \cup b, x)$

{} : *op*([S → BOOL])ENS

{} fvide = $\emptyset$
{} ajout(f,x,v) = *si* v *alors* adj({} f,x) *sinon* {} f

card : *fonct*(ENS)ENTIER

card($\emptyset$) = 0
card(adj(a,x)) = *si* $x \in a$ *alors* card(a) *sinon* card(a)+1

figure 10 : a-description du S.I. ensemble des parties

S.I. SUITE

V : *S.I.*

<> : *op* SUITE
$\sim$: *op*(SUITE,V)SUITE
début : *fonct*(SUITE)SUITE
der : *fonct*(SUITE)V
reste : *fonct*(SUITE)SUITE
prem : *fonct*(SUITE)V

b_i : *fonct*(SUITE)ENTIER

b_s : *fonct*(SUITE)ENTIER

|| : *op* (SUITE)ENTIER

* : *op* (SUITE)SUITE

tr : *fonct*(SUITE,ENTIER,ENTIER)SUITE

vide : *fonct*(SUITE)BOOL

param
fparam
opérations

début(u $\sim$ x) = u
der(u $\sim$ x) = x
reste(u $\sim$ x) = *si* vide(u)
alors <>
sinon reste(u) $\sim$ x
prem(u $\sim$ x) = *si* vide(u)
alors x
sinon prem(u)

b_i(<>) = 1
b_i(u $\sim$ x) = b_i(u)
b_s(<>) = 0
b_s(u $\sim$ x) = b_s(u) + 1
|<>| = 0
|u $\sim$ x| = |u| + 1
u * < > = u
u * (u' $\sim$ x) = (u * u') $\sim$ x
tr(<>,i,j) = <>
tr(u $\sim$ x,i,j) = *si* j < i
alors <>
sinon si j=b_s(u)+1
alors tr(u,i,j-1)$\sim$x
sinon tr(u,i,j)
vide (<>) = *vrai*
vide(u $\sim$ x) = *faux*

restrictions

pré début(u) = $\neg$vide(u)
pré der(u) = $\neg$vide(u)
pré reste(u) = $\neg$vide(u)
pré prem(u) = $\neg$vide(u)

figure 12 : a-description du S.I. SUITE

S.I. FONC		
	S_1 : *S.I.* S_2 : *S.I.* fvide : *op* FONC ajout : *fonct*(FONC,S_1,S_2)FONC appl : *fonct*(FONC,S_1)S_2 dom : *fonct*(FONC)$\mathcal{P}(S_1)$ imag : *fonct*(FONC)$\mathcal{P}(S_2)$ supr : *fonct*(FONC,S_1)FONC chval : *fonct*(FONC,S_1,S_2)FONC égal : *fonct*(FONC,FONC)BOOL	*param* *fparam* appl(ajout(f,x,y),x') = *si* x = x' *alors* y *sinon* appl(f,x') dom(fvide) = ∅ dom(ajout(f,x,y)) = adj(dom(f),x) imag(fvide) = ∅ imag(ajout(f,x,y)) = adj(imag(f),y) supr(ajout(f,x,y),x') = *si* x = x' *alors* f *sinon* ajout(supr(f,x'),x,y) chval(fvide,x,y) = fvide chval(ajout(f,x,y),x',y') = *si* x = x' *alors* ajout(f,x,y') *sinon* chval(f,x',y') égal(fvide,fvide) = *vrai* égal(fvide,ajout(f,x,y)) = *faux* égal(ajout(f,x,y),fvide) = *faux* égal(ajout(f,x,y),ajout(f',x',y')) = *si* x = x' *alors* y *sinon* égal(supr(f,x'),supr(f',x))∧appl(f,x') = y' appl(f',x *restrictions* *pré* appl(f,x) = x ∈ dom(f) *pré* supr(f,x) = x ∈ dom(f) *pré* ajout(f,x,y) = x ∉ dom(f)
dom : comme ch. 1.2 §7.2.1 imag: comme ch. 1.2 §7.2.1 supr: comme ch. 1.2 §7.2.1 chval:comme ch. 1.2 §7.2.1 égal(f,f') = (dom(f) = dom(f')) ∧ ∀ x ∈ dom(f) appl(f,x) = appl(f',x)		

figure 11 : a-description du S.I. ensemble des fonctions

4.3.- Ensemble des fonctions.

Ici encore il est nécessaire d'introduire des constructeurs. Nous choisissons la fonction vide fvide et l'adjonction d'un élément. La figure 11 propose une a-description de ce S.I.

4.4.- Ensemble des suites.

Nous ne considérons ici que les suites finies. Par abus de notation nous noterons encore V leur S.I. La figure 12 donne une a-description de ce S.I.; nous n'y faisons pas figurer la s-description correspondante qui se trouve sur la figure 9 du ch. 1.2.

5.- ABSTRACTION ALGEBRIQUE.

La motivation du passage d'une s-description d'un S.I. à une a-description a été largement développée aux paragraphes précédents. Aussi nous contentons-nous ici d'une brève étude théorique des propriétés particulières des abstractions algébriques (§ 5.1) avant de proposer une méthode de construction d'une telle abstraction à partir d'une s-description (§ 5.2).

5.1.- Définition et propriétés d'une abstraction algébrique.

Définition 11

Soit S un S.I.. Une *abstraction algébrique* de S est un S.I. algébrique A tel qu'il existe une application α faisant de (A,α) une abstraction de S.

Un cas particulièrement intéressant est celui où S est défini par une s-description et α est l'identité sur les opérations introduites par cette s-description.

Outre les divers résultats valables dans le cas d'une abstraction quelconque (ch. 1.3 § 6.3), les abstractions algébriques possèdent des propriétés spécifiques qui lient les concepts d'abstraction complète pour un ensemble de formules (ch. 1.3 § 6.3.2 d)) et d'algèbres initiale et finale.

Théorème 7

Soit A un S.I. algébrique, S un S.I. complet et F l'ensemble des formules égalitaires sans variable de F. Supposons que (A,α) soit une abstraction simple de S. Alors (A,α) est une abstraction complète pour F si et seulement si le modèle "algèbre initiale" de A peut être étendu, modulo α, en un modèle de S. □

Démonstration

a) Supposons que le modèle initial de A puisse être étendu, modulo α, en un modèle de S.

Soit $(\mathcal{U}, \text{int})$ ce modèle initial. Par définition, pour tout couple φ, φ' de formules de A :

$$\text{int}(\varphi) = \text{int}(\varphi') \Leftrightarrow \vdash_A \varphi \equiv \varphi'$$

Appelons int' son extension à S : int = int' ∘ α.

Alors si $\vdash_S \alpha(\varphi) \equiv \alpha(\varphi')$, int'(α(φ)) = int'(α(φ')), donc int(φ) = int(φ') c'est-à-dire $\vdash_A \varphi \equiv \varphi$.

b) Supposons que (A,α) soit complète pour F.

Puisque S est complet, il a un seul modèle $(\mathcal{U}, \text{it}')$. Alors $(\mathcal{U}, \text{it}'\circ\alpha)$ est un modèle de A.

Si it'∘α(φ) = it'∘α(φ'), alors $\vdash_S \alpha(\varphi) \equiv \alpha(\varphi')$, donc $\vdash_A \varphi \equiv \varphi'$, ce qui prouve que $(\mathcal{U}, \text{it}'\circ\alpha)$ est l'algèbre de A.

cqfd

Ce théorème se généralise immédiatement au cas des formules positives, c'est-à-dire les formules ne contenant que les connecteurs "∧" et "∨" (éventuellement quantifiées) et sans variable libre.

Théorème 8

Avec les mêmes notations que dans le théorème 7, soit F_1 l'ensemble des formules positives de A.

(A,α) est une abstraction complète pour F_1 si et seulement si le modèle initial de A peut être étendu, modulo α, en un modèle de S. □

Démonstration

- La condition est nécessaire puisque, si (A,α) est complète pour F, elle l'est pour F_1 ; on utilise alors le théorème 7.

- Réciproquement, supposons que (A,α) soit une abstraction complète pour F_1.

Lemme

Soit $(\mathcal{U}, \text{it})$ un modèle de A, $(\mathcal{U}, \text{int})$ son modèle initial et φ une formule positive. Si int(φ) est vrai pour certaines valeurs des variables de φ, it(φ) est vrai pour ces valeurs.

Soit $\varphi \in F_1$ telle que α(φ) soit un théorème de S, alors int'(α(φ)) est toujours vrai (int' est l'extension de int à S). Alors int(φ) = int'(α(φ)) est toujours vrai ; donc avec le lemme it (φ) toujours vrai pour une interprétation quelconque. Ainsi φ est un théorème de A.

cqfd

Ces théorèmes se généralisent immédiatement au cas des abstractions généralisées (ch. 1.3 § 6.3.3).

Théorème 9

Soit A un S.I. algébrique, consistant qui est une extension suffisamment complète du S.I. R complet. Alors A admet un modèle final $(\mathcal{U}, f_n)$ caractérisé par $f_n(\varphi) = f_n(\varphi') \Leftrightarrow \varphi \not\equiv \varphi'$ n'est pas un théorème de A. □

Démonstration

Soit la congruence $\varphi \sim \varphi' \Leftrightarrow$ pour toute opération externe f de A,

$$\vdash_A f(\varphi) \equiv f(\varphi').$$

Elle définit un modèle f_n de A.

$\varphi \not\sim \varphi' \Leftrightarrow$ pour un f, f(φ) ≡ f(φ') n'est pas un théorème de A.

$\varphi \not\sim \varphi \Leftrightarrow$ pour un f, $\vdash_A f(\varphi) \not\equiv f(\varphi')$

Mais $\varphi \equiv \varphi' \supset f(\varphi) \equiv f(\varphi')$ est un théorème de A, donc :

$$\vdash_A \varphi \not\equiv \varphi'$$

Ainsi $\varphi \not\sim \varphi' \Rightarrow \vdash_A \varphi \not\equiv \varphi'$

d'où, si $\varphi \not\equiv \varphi'$ n'est pas un théorème de A, $f_n(\varphi) = f_n(\varphi')$

La réciproque est vraie pour tout modèle.

cqfd

Théorème 10

Soit (A,α) une abstraction simple du S.I. S complet. Supposons, comme au théorème 9, que A est algébrique, consistant et que c'est une extension suffisamment complète du S.I. R complet. Soit enfin F l'ensemble des formules $\varphi \not\equiv \varphi'$ de A, sans variable.

Alors (A,α) est complète pour F si et seulement si le modèle final de A peut être étendu, modulo α, en un modèle de S. □

La démonstration est analogue à celle du théorème 8.

Ce théorème se généralise au cas des formules négatives (qui sont les négations des formules positives) de A. Il se prolonge également au cas des abstractions généralisées.

5.2.- Construction d'une abstraction algébrique.

5.2.1.- Passage d'une s-description à une a-description

Dans ce paragraphe nous cherchons à systématiser le passage d'un S.I. S défini par une s-description à une abstraction algébrique A de S.

Dans la suite nous supposons que :

- A et S ont mêmes S.I. externes et mêmes sortes d'intérêt.
- L'ensemble des opérations de A définies par la a-description se décompose en:
  - $\mathcal{O}_{p_1}$: sous-ensemble des opérations définies par la s-description de S.
  - $\mathcal{O}_{p_2}$: opérations de A qui ne sont pas des symboles définis par la s-description de S (fonctions cachées).
- L'abstraction α de S par A est l'identité sur les opérations de $\mathcal{O}_{p_1}$ et sur les S.I. externes.

Ces propriétés sont obtenues lorsque l'on définit une abstraction en remplaçant les axiomes de la s-description par des définitions récursives (cf. exemples 2 et 3). Le théorème suivant est alors immédiat.

Théorème 11

Si A et S vérifient les propriétés précédentes, (A,α) est une abstraction de S si :

1) α est l'identité sur $\mathcal{O}_{p_1}$ et les S.I. externes

2) pour tout $f \in \mathcal{O}_{p_1}$ et toute égalité $f\ u_1 \ldots u_q = v$ de la a-description de A, on a $\vdash_S \underline{pré}_S\ \alpha(f\ u_1 \ldots u_q) \supset \alpha(f\ u_1 \ldots u_q) = \alpha(v)$ où $\underline{pré}_S u$ est la précondition de u dans S.

3) pour tout $f \in \mathcal{O}_{p_1}$, $\vdash_S \alpha(\underline{pré}_a f) \supset \underline{pré}_S f$
où $\underline{pré}_a f$ est la précondition dans A de f

4) l'image par α des axiomes $\underline{pré}\ f\ u_1 \ldots u_q \supset f\ u_1 \ldots u_q \equiv v$, où $f \in \mathcal{O}_{p_2}$, est un théorème de S. □

Démonstration

Ce théorème résulte immédiatement de la définition d'une abstraction et du fait que les axiomes caractérisant tout symbole f de $\mathcal{O}_{p_1}$ sont de la forme (définition des a-descriptions) :

$\underline{pré}_a f\ u_1 \ldots u_q \supset f\ u_1 \ldots u_q \equiv v$
car alors $\vdash_S \alpha(\underline{pré}_a f\ u_1 \ldots u_q) \supset \underline{pré}_S \alpha(f u_1 \ldots u_q)$
et $\vdash_S \underline{pré}_S \alpha(f\ u_1 \ldots u_q) \supset \alpha(f\ u_1 \ldots u_q) \equiv \alpha(v)$
conc $\vdash_S \alpha(\underline{pré}_a f\ u_1 \ldots u_q \supset f\ u_1 \ldots u_q \equiv v)$
cqfd

Remarque

On peut rapprocher les abstractions algébriques de la notion de représentation d'un type abstrait algébrique en termes de procédures que donne GUTTAG dans [GUT,78c]. De manière évidente son critère de correction est voisin du théorème 11 puisqu'il exige que, si $u = v$ est un axiome algébrique, alors $POST(u) \wedge POST(v) \supset r_u = r_v$, où POST(u) est la post-condition de l'appel de procédure représentant le terme u, r_u est le résultat correspondant.

GUTTAG indique également des "règles" permettant de dériver les post-conditions des axiomes algébriques (elles ne sont pas trop éloignées des règles (RESR) et (RECR)).

Cependant le langage de représentation est dynamique et donc le passage est brutal du niveau algébrique à cette implantation. De plus ce caractère dynamique nécessite une sémantique axiomatique (et donc des règles de preuves) élaborées.

On trouve une démarche voisine, en termes de pré et post-conditions, dans ALPHARD [WUL,75].

5.2.2.- Méthode de construction d'une abstraction

La construction d'une abstraction A d'un S.I. S défini en termes de s-description et spécifiant l'univers d'un problème P = (E,S) est guidée par les idées suivantes :

- tout axiome de A doit être un théorème de S
- la précondition algébrique de toute opération f de A qui est aussi une opération de S doit impliquer la précondition correspondante dans la s-description
- A doit posséder la propriété de forme normale. Ceci est un guide du choix des constructeurs
- outre les constructeurs, les opérations définies dans la a-description de A sont, soit des opérations de S, soit des fonctions auxiliaires nécessaires à leur explicitation
- de plus on essaye de rendre l'abstraction complète en procédant par équivalences pour dériver les axiomes algébriques des post-conditions (règles (RESRT) et (RECRT)).

Les exemples 2 et 3 illustrent ces idées.

Décrivons informellement les grandes lignes de cette méthode d'abstraction (qui est une explicitation du méta-problème de l'abstraction).

Résultat

(A,α) abstraction de S ; S est la donnée, c'est un S.I. défini par une s-description.

1) Choix des constructeurs

a) Ce choix est guidé :
- par le type de la stratégie d'explicitation qui fait appel à l'abstraction (induction sur les données, induction sur les résultats, etc...)
- par le souci d'obtenir un S.I. possédant la propriété de forme normale. On cherche alors à exprimer un terme (dont le profil est une des sortes d'intérêt) à l'aide de certains termes plus simples et de certaines opérations (cf. § 3.3.4). Il s'agit donc d'un problème de décomposition de termes exprimés en termes de sortes supports. Ceci conduit, en général à un raisonnement par cas.

Notons que certains constructeurs peuvent ne pas être des opérations introduit❍par la s-description de S.

b) On place les profils des constructeurs dans la colonne centrale (et on les coche).

c) Pour chaque constructeur g, détermination d'un prédicat $\underline{pré}_a g$ tel que $\underline{pré}_a g \supset \underline{pré}_s g$ soit un théorème de S.

Ceci peut introduire de nouvelles opérations dont le profil est placé dans la colonne centrale.

2) Choix des autres opérations

Ce sont les opérations de S apparaissant dans l'énoncé que l'on veut expliciter.

On place le profil de ces opérations dans la colonne centrale.

3) Explicitation des autres opérations

Pour chaque opération f non cochée de la colonne centrale, faire
- Cocher l'opération f dans la colonne centrale
- Choix entre
  . Application de ⌈SESR⌉ # explicitation s-récursive, ceci nécessite que des simplificateurs aient déjà été explicités #
  . Application de ⌈SECR⌉ # c'est au moins le cas des simplificateurs "réciproques" des constructeurs #
- Détermination d'un prédicat $\underline{pré}_a f$ tel que $\underline{pré}_a f \supset \underline{pré}_s f$ soit un théorème de S

Notons que chaque explicitation peut introduire de nouveaux intermédiaires qui sont d'abord définis par une s-description avant d'être à leur tour explicités.

Il reste à préciser cet algorithme en le formalisant et en le détaillant. Nous nous contentons d'en montrer la mise en oeuvre sur un nouvel exemple.

5.2.3.- Un exemple d'application

Exemple 14

Dans [MAJ,77] MAJSTER montre certaines limites des spécifications algébriques. Si nous sommes d'accord avec sa critique concernant l'inadéquation des spécifications algébriques à la spécification initiale d'un problème, nous pensons que l'introduction de fonctions auxiliaires (cachées) est tout à fait naturelle. Aussi sans reprendre la polémique qu'a pu susciter cet article [MAR,77] montrons ici comment le passage par les pré-post conditions permet d'introduire une fonction cachée servant à la spécification algébrique.

Rappelons brièvement le problème : il s'agit de spécifier une pile avec lecture interne. Outre les opérations classiques d'empilement et de dépilement il est possible de lire à l'intérieur de la pile, à une hauteur précisée par un index. On dispose d'une opération permettant de descendre pas à pas l'index (desc) et d'une opération (rem) permettant de le remonter au sommet.

Nous indiquons directement dans la colonne de gauche de la figure 12 une s-description de ce S.I. PILEL.

Utilisons le méta-algorithme du paragraphe précédent pour construire une abstraction de PILEL :

1) Choix des constructeurs

a) Nous ne sommes pas dans le cadre de l'explicitation d'un problème. Le seul guide est la recherche d'une propriété de forme normale.

Soit p un terme de profil PILEL ; $p = \langle\langle u,h\rangle\rangle$ où $1 \leq h \leq b_s(u)$
Plusieurs cas sont envisageables :

. $b_s(u) = 0$, i.e. $u = \langle\rangle$. Alors nécessairement $p = \langle\langle \langle\rangle,0 \rangle\rangle$, c'est une constante que l'on note pvide. C'est un constructeur 0-aire

. $b_s(u) > 0$ et $h < b_s(u)$. Parmi les opérations de la s-description, seule desc permet de rendre $b_s(u)$ et h inégaux. Donc nécessairement :

$$\langle\langle u,h\rangle\rangle = desc(\langle\langle u,h+1\rangle\rangle)$$

. $b_s(u) > 0$ et $h = b_s(u)$. Trois opérations fournissent un résultat vérifiant cette propriété :
  - emp
  - ret
  - dep

ret ne permet pas de modifier la première composante d'une pile ; dep ne permet que de simplifier une pile d'où le choix de emp.

b) Détermination des préconditions des constructeurs non 0-aires.

. $\underline{pré}_s$ emp(p,v) = (i(p) = b_s(e(p)))
d'où l'introduction d'une fonction auxiliaire :
indsom : *fonct*(PILEL) BOOL # indice au sommet #
indsom *tq* indsom(p) = (i(p) = b_s(e(p)))
alors $\underline{pré}_a$ emp(p,v) = indsom(p)

. $\underline{pré}_s$ desc(p,v) = (i(p) > 1) = (p ≠ pvide)

2) Choix des autres opérations

Ici on prend toutes les opérations de la s-description

3) Explicitation des autres opérations

a) ret : on utilise la stratégie ⟨SECR⟩ ; ce qui conduit à la recherche de 3 fonctionnelles puisqu'il y a 3 constructeurs.

- ret(pvide) = ret(<< <>,0 >>) = << <>,0 >> = pvide
- ret(desc(p)) = ret(<<e(p),i(p)-1>>) = <<e(p),b_s(e(p))>> = ret(p)
- ret(emp(p,v)) = ret(<<e(p) ∿ v,i(p)+1>>) =<<e(p) ∿ v,b_s(e(p))+1>> = emp(p,v)

en utilisant la précondition de emp(p,v).

b) L'explicitation de dep et indsom est immédiate, elle se trouve sur la figure 12.

c) En revanche l'explicitation, en utilisant ⟨SECR⟩, de val pose des problèmes :

- val(emp(p,v)) = appl(e(p) ∿ v,b_s(e(p))+1) = v
- val(desc(p)) = appl(e(p),i(p)-1)

ce qu'on ne sait pas exprimer à l'aide des seules opérations algébriques. On cherche p_1 = dep^k(p) telle que i(p)-1 = b_s(p_1) car alors appl(e(p),i(p)-1) = appl(p_1,b_s(p_1)) = val(p_1).

L'idée est d'introduire une nouvelle fonction cachée decr qui fournit la "sous pile" d'une pile dont le sommet est la valeur de l'index.

⟨eureka⟩ decr : *fonct*(PILEL)PILEL
decr(p) = p'
tq b_s(e(p')) = i(p) ∧ ∀ j(1≤j≤i(p) ⊃
appl(e(p'),j) = appl(e(p),j)) ∧ i(p') = i(p)

S.I. PILEL	V : S.I.	*param* / *fparam* / *opération*
PILEL = <<e : V*, i : ENTIER>> *invariant* 1 ≤ i(p) ≤ b_s(e(p)) ∧ b_i(e(p)) = 1 pvide = << <>,0>>	pvide : *op* PILEL	
emp *tq* emp(p,v) = <<e(p) ∿ v,i(p)+1>> *si* i(p) = b_s(e(p)) desc *tq* desc(p) = <<e(p),i(p)-1>> *si* i(p) > 1 ret *tq* ret(p) = <<e(p),b_s(e(p))>>	emp : *fonct*(PILEL,V)PILEL desc : *fonct*(PILEL)PILEL ret : *fonct*(PILEL)PILEL	ret(pvide) = pvide ret(desc(p)) = ret(p) ret(emp(p,v)) = emp(p,v) dep(emp(p,v)) = p
dep *tq* dep(p) = <<début(e(p)),i(p)-1>> *si* i(p) = b_s(c(p)) ∧ i(p) ≥ 2 val *tq* val(p) = appl(e(p),i(p)) *si* i(p) ≥ 1	dep : *fonct*(PILEL)PILEL val : *fonct*(PILEL)V	indsom(pvide) = *vrai* indsom(desc(p)) = *faux* indsom(emp(p,v)) = *vrai* decr(pvide) = pvide decr(emp(p,v)) = emp(p,v) decr(desc(p)) = decr(decr(p)) *restrictions* *pré* emp(p,v) = indsom(p) *pré* dep(p) = (p ≠ pvide) ∧ indsom(p) *pré* desc(p) = (p ≠ pvide) *pré* val(p) = (p ≠ pvide)
indsom *tq* indsom(p) = (i(p) = b_s(e(p)))	indsom : *fonct*(PILEL)BOOL	
decr(p) = p' *tq* b_s(e(p')) = i(p) ∧ i(p) = i(p') ∧ ∀j(1≤j≤i(p) ⊃ appl(e(p'),j) = appl(e(p),j))	decr : *fonct*(PILEL)PILEL	

figure 12 : pile avec lecture interne

Alors ∀ p, appl(e(p),i(p)) = appl(e(decr(p)),i(p))
donc val(desc(p)) = appl(e(decr(p)),i(decr(p)) = val(decr(p))
- *pré* val(p) = (p ≠ pvide)

d) La construction de l'abstraction se termine par l'explicitation c-récursive de decr (figure 12).

6.- REPRESENTATION D'UN SYSTEME ALGEBRIQUE.

Les objets de systèmes algébriques tels que TABSYM, MULTENS, ne sont pas, en général, exprimables directement dans un langage de programmation. Dans le processus de résolution d'un problème on cherche donc à se "rapprocher" d'objets informatiques de base tels que les structures, les tableaux, etc... Ce problème de *représentation* d'un système algébrique par un autre a été l'objet de nombreuses études [GUT,78a],[PAR,77], [LES,79], [REM,79],[GOG,77a]. Dans la plupart des cas était posé le problème de la *correction d'une représentation* (preuve a posteriori).

Nous nous sommes préoccupés au paragraphe 6.3 du chapitre 1.3 de définir et étudier ce concept de représentation. Aussi nous restreignons-nous ici à l'aspect méthodologique de la construction des représentations. C'est ainsi qu'au paragraphe 5.1 nous étudions, à partir d'un exemple, la construction de représentations généralisées et au paragraphe 5.2 celle de représentations faibles. Le paragraphe 5.3 propose, quant à lui, un méta-algorithme de représentations.

6.1.- Construction de représentations généralisées.

Exemple 14. Toujours la table des symboles

BUT 0 : trouver un S.I. S et une application μ tels que (S,μ) soit une représentation généralisée (ch. 1.3 § 6.3.4) de TABSYM[AT,n]

① Choix du S.I. représentant et représentation des constructeurs

La dualité des rôles joués par debbloc et finbloc (axiome 1 figure 2) peut faire penser à utiliser une *pile*. A un niveau de bloc donné, à chaque identificateur est associé un seul attribut : les objets empilés sont des *fonctions* dont les domaines sont formés d'identificateurs et les codomaines d'attributs. La figure 11 propose une a-description du S.I.PILE(VAL).

On choisit donc de représenter TABSYM[AT,n] par PILE[[CHAINE → AT]]. Pour être plus précis il faudrait considérer un couple formé de PILE[...] et

S.I. PILE	
VAL : *S.I.* pilevide : *op* PILE empiler : *fonct*(PILE, VAL) PILE depiler : *fonct*(PILE) PILE sommet : *fonct*(PILE) VAL ∪ {ω} estpvide : *fonct*(PILE) BOOL	*param* *parsm* *opérations* depiler (empiler (p,v)) = p sommet (pilevide) = ω sommet (empiler(p,v)) = v estpvide (pilevide) = *vrai* estpvide (empiler(p,v)) = *faux* *restriction* *pré* depiler (p) = ¬estpvide(p)

Figure 11 : le S.I. des piles

de n. Nous supposerons n implicite dans la suite.

Le choix de ce S.I. représentant va de pair avec celui de la représentation des constructeurs primitifs.

μ (tabvide) = pilevide
μ (debbloc(ts)) = empiler (μ (ts), fvide)
μ (insert(ts, ident,a)) = modifsommet (μ(ts), ident,a)

où modifsommet (p,x,a) est une abréviation de
empiler (depiler(p), ajout (sommet(p),ident,a))

BUT 1 : définir μ sur les autres opérations de TABSYM de façon que (PILE [[CHAINE → AT]], μ) soit une représentation généralisée de TABSYM [AT,n]

Pour définir μ sur les simplificateurs de TABSYM on est guidé par le fait que les axiomes de ce S.I. doivent être des théorèmes du S.I. représentant. On met ainsi en evidence des *règles de représentation* analogues aux règles d'explicitation (règles d'abstraction) étudiées précedemment. Developpons cette idée dans le cas de finbloc:

② Choix de la représentation de finbloc

a) Représentation des axiomes égalitaires

La représentation de finbloc est caractérisée par le fait que les formules suivantes doivent être des théorèmes de PILE [[CHAINE → AT]]

(1) μ (finbloc (debbloc(ts))) = μ(ts)

(2) μ (finbloc (insert(ts,ident,a))) = μ(finbloc(ts)).

Ces égalités s'écrivent encore, en utilisant la définition de μ sur les constructeurs:

(1) μ (finbloc) (empiler(μ(ts)fvide)) = μ (ts)

(2) μ (finbloc) (modifsommet(μ(ts),ident,a)) = μ(finbloc) (ρ(ts))

Ceci est une définition récursive de μ (finbloc) sur les piles, non vides, qui sont des représentations de tables des symboles. Dans ce cas précis, il est possible de supprimer la récursivité et d'obtenir une définition de μ (finbloc) directement en termes de fonctions du S.I. représentant.

Nous avons vu au paragraphe 3.3.2 exemple 6 que toute table des symboles non vide ts se met sous la forme normale:

ts = insert^k (debbloc(ts'),id1,a1,id2,a2,..., idk,ak)

On en déduit que la représentation μ (ts) est égale à

μ(ts) = modifsommet^k (empiler(μ(ts'),tvide),id1,a1,id2,...idk,ak)

alors en utilisant k fois (2) et une fois (1) :

μ(finbloc) (μ(ts)) = μ(finbloc) (empiler(μ (ts'),tvide)) =μ (ts')

et en notant, « eurêka », que :

depiler (μ(ts))= depiler (modifier^k (empiler (μ(ts'),tvide),id1,a1, id2,a2 ...idk,ak)

= μ(ts')

on en tire que pour toute pile p représentation d'une table ts non vide

μ (finbloc) (p) = depiler (p).

b) représentation de la pré-condition

On demande (cf cas des abstractions) que

μ ($pré_{TABSYM}$ finbloc (ts)) $\supset$ $pré_{PILE}$ finbloc(ts)

ceci est necessaire pour que les axiomes *pré* f $\supset$ $fu_1..u_q \equiv \mathcal{C}$ se transforment par μ en des théorèmes.

pré finbloc(ts) = $\neg$vide(ts)

alors μ (*pré* finbloc(ts)) = $\neg\mu$ (vide(ts)) = $\neg$estvide(μ (ts))

or μ(finbloc(ts)) = depiler(μ(ts)) si μ(ts) n'est pas pilevide, autrement dit si $\neg$estvide(μ(ts)).

Ainsi on complète la représentation de finbloc en posant

μ(finbloc) = depiler

et $pré_{PILE}$ finbloc (ts) $\equiv$ *vrai*

Remarque 1

Il existe un symbole fonctionnel image par μ de finbloc. Cette situation n'est cependant pas générale et il se peut que l'on ne puisse donner qu'une définition récursive de l'image de finbloc. C'est le cas par exemple de valattr (cf ch 1.3 § 6.3.3) :

μ(valattr) (p,ident) = *si* p = pilevide
alors ω
sinon si ident $\in$ dom(sommet(p))
alors appl(sommet(p),ident)
sinon μ (valattr) (depiler(p),ident)

C'est pour cette raison que nous avons introduit le concept de représentation généralisée. □

Remarque 2

La démarche suivie est un premier pas vers une "synthèse de représentation" fondée sur les mêmes idées que la "synthèse d'abstraction" du paragraphe précédent. Elle est plus amplement developpée dans [GAU,78a]. □

On trouve dans [GUT, 78a] et [FIN, 78b] la représentation des autres fonctions. □

6.2.- Construction de représentations faibles

Ici encore nous étudions le problème de représentation sur un exemple.

Exemple 15 Représentation du S.I. PILE

BUT 0 : trouver une représentation (S',μ) de PILE

① Choix du S.I. représentant et des constructeurs

Choisissons, de manière tout à fait classique de représenter une pile par un couple

- tableau
- hauteur

Plus formellement, PILE (VAL) va être représenté par << t : TABLEAU, h: ENTIER>> où le S.I. TABLEAU est a-décrit par la figure 2

La représentation des constructeurs de PILE est donnée par:

- μ(pilevide) = << tvide,0 >>
- μ(empiler(p,v)) = << assigne(t*de* μ(p),h *de*μ(p)+1,v), h *de* μ(p)+1>>

ou encore - μ(empiler)(<<tab,hau >>,v) = << assigne(tab,hau+1,v), hau+1>>

S.I. TABLEAU	
VAL: *S.I.*	*param* *fparam*
tvide : *op* TABLEAU	*opérations*
assigne : *fonct* (TABLEAU, ENTIER, VAL) TABLEAU	
accès : *fonct*(TABLEAU, ENTIER) VAL ∪ {ω}	accès (tvide,i) = ω accès (assigne(t,i,v),j) = *si* i = j *alors* v *sinon* accès(t,j)

Figure 12 : le S.I. tableau

BUT 1 : Représentation des simplificateurs

a) Cas de sommet:

En reprenant la règle de représentation evoquée au paragraphe 5.1 :

μ(sommet) (<<tvide,o>>) = ω

μ(sommet) (<<assigne (tab, hau +1, v), hau+1>>) = v

Choix : μ(sommet) (<<tab, hau>>) = accès (tab, hau)

b) Cas de depiler:

Un raisonnement identique conduit à:

(1) μ(depiler) (<<assigne (tab, hau+1, v), hau+1>>) =<< tab, hau >>

Mais, comme dans l'exemple 16 du chapitre 1.3 (représentation de FILE), il est clair que cette représentation est inefficace: on ne souhaite pas retrouver le tableau initial lors d'un dépilement, seule la hauteur doit diminuer. C'est donc la notion de *représentation faible* qui doit être utilisée ici: on ne demande que le transport des théorèmes concernant les S.I. externes.

Dans cet exemple, le S.I. externe est VAL et on veut donc que les théorèmes qui concernent sommet soient des théorèmes dans la représentation.

La représentation de depiler doit donc vérifier:

si sommet (depiler (p)) = v alors μ (sommet (μ(depiler) (μ (p))) = v
ou accès (μ(depiler) (μ (p))) = v

or sommet (depiler (p)) = v exige qu'il existe une pile p1 et une valeur v1 tels que p = empiler (empiler (p1,v),v1)

et donc il existe un couple <<tab1, hau1 >>tel que

μ(p) = <<assigne (assigne(tab1 hau1+1, v), v1, hau1+2), hau+2 >>

alors accès (assigne (assigne(tab1, hau1+1, v), v1, hau1+2), hau1+1) = v

et on souhaite avoir:

accès (μ(depiler) (μ(p))) = v

qui sera vérifié si

(2) μ (depiler) <<tab, hau >> = <<tab, hau-1>>

Avec (2), le premier membre de (1) vérifie:

μ (depiler) (<< assigne (tab, hau+1, v), hau+1>>) =
<< assigne (tab, hau+1, v), hau >>

Mais d'après (1),<< tab, hau >>est une autre représentation possible du premier membre.

GUTTAG [GUT, 78a] introduit alors une relation de congruence notée ≡ dans <<t: TABLEAU, h: ENTIER >> qu'il appelle *interprétation de l'égalité* pour μ. Cette relation est un nouveau prédicat du système produit défini par:

$$\langle\langle t,i\rangle\rangle \equiv \langle\langle t',i'\rangle\rangle \Leftrightarrow (i=i') \wedge (\forall k,\ 1\leq k\leq i \Rightarrow \text{accès}(t,k) = \text{accès}(t',k))$$

Intuitivement, on ne se préoccupe pas des valeurs de t et t' dont l'indice est supérieur à i.

Ainsi pour que ρ transporte les théorèmes portant sur les objets externes, il suffit que pour tout axiome u ≡ v du S.I. PILE tel que profil (u) = PILE, on ait μ(u) ≡ μ(v) dans <<t: TABLEAU, h: ENTIER>> .

Ceci donne un guide méthodologique pour la définition d'une représentation: on commence par chercher une interprétation de l'égalité dans le type représentant et pour cela on est guidé par la préservation des théorèmes sur les objets externes.

Pour tout terme u, la classe d'équivalence de ρ (u) pour l'interprétation ≡ de l'égalité est l'ensemble des *représentations admissibles* de u.

Nous arrêtons là cet exemple et revenons au cas général des représentations faibles. □

Théorème 11

Soit S et S' deux S.I. algébriques supposés uni-sortes.

(S', μ) est une représentation simple faible de S si et seulement si il existe une relation d'équivalence $\approx$ sur les termes de profil $\overline{S}'$ telle que:

1) l'image par μ de tout axiome $u \equiv v$ de S, ou profil(u) = $\overline{S}$, vérifie $\mu(u) \approx \mu(v)$.

2) pour toute opération f de S dont le but est une sorte externe, pour tous termes u,v de S de profil $\overline{S}_i$ composant du profil de f, si $\mu(u) \approx \mu(v)$ alors $\mu(f)\, x_1..x_{i-1}\, \mu(u)..x_q \equiv \mu(f)\, x_1..u_{i-1}\, \mu(v)..x_q$ est un théorème de S'.

3) pour toute opération f de S,

$\vdash_{S'} \mu(\underline{pré}_S\, fx_1..x_q) \supset \underline{pré}_S\, \mu(f)x_1..x_q$ □

On généralise ce théorème au cas des représentations faibles généralisées en considérant une extension conservative S" de S' à la place de S' dans 2) et 3) précédents.

Démonstration

a) Avec 2) et 3), tout théorème $\varphi \supset u \equiv v$ de S, où profil(u) est une sorte externe, a pour image un théorème de S'.

b) Réciproquement, les deux relations sur les termes de profil $\overline{S}'$ de S' définies par:

$u' \approx_1 v' \iff \exists u,v \in \mathcal{G}ch(\overline{S})$ *tq* $\mu(u)=u'$ *et* $\mu(v)\approx v'$ *et* $\vdash_S u \equiv v$

$u' \approx_2 v' \iff \forall f \in \mathcal{G}ym(S), \vdash_S \mu(f)x_1..x_{i-1}u'..x_q \equiv \mu(f)x_1..x_{i-1}\, v'..x_q$

sont des équivalences. Leur intersection en est encore une.
cqfd

6.3.- Méthode de construction d'une représentation

La possibilité de définir de façon systématique des représentations est une condition indispensable à la mise en oeuvre d'une méthode d'analyse de problèmes portant sur des structures abstraites. Les représentations peuvent aussi servir d'outils d'explicitation d'énoncé dans la mesure où elles permettent de remplacer certains constructeurs par d'autres mieux adaptés à la résolution du problème.

6.3.1.- Composantes d'une méthode de représentation

Le problème de la représentation d'un S.I. algébrique S est double:
- Détermination du S.I. représentant S
- Détermination de la fonction de représentation μ

Les exemples 14 et 15 ont montré l'existence de plusieurs étapes:

a) Choix de S' et de la représentation des constructeurs

La recherche de S' dépend du but poursuivi et du problème considéré. C'est là aussi que sont prises en compte des questions d'*efficacité* dont nous n'avons pas parlé jusqu'à présent. Ainsi, représenter un ensemble par une liste n'est pas toujours très intéressant et, s'il est muni d'un ordre total, on peut lui préférer une représentation sous forme d'arbre binaire [REM, 79].

D'un point de vue méthodique, on peut supposer disposer d'un catalogue de systèmes "concrets" (tableaux, piles, files, arbres équilibrés, fichiers, structures,etc..) et de systèmes primitifs (ENTIER, REEL,..), on cherche à définir le S.I. représentant comme composé d'un certain nombre d'entre eux (en utilisant les opérations de composition de systèmes d'information). Ceci peut s'appliquer plusieurs fois et on peut supposer disposer de plusieurs niveaux de catalogues. Au niveau le plus bas le problème a été étudié depuis longtemps déjà par les informaticiens: si on ne dispose comme système concret que des tableaux (mémoire centrale), on dispose de *techniques de représentation* pour les S.I. classiques (listes linéaires, arbres piles, etc..) [KNU, 68], [GAU, 78b].

Le choix du système cible va de pair avec le choix de la représentation des constructeurs: les axiomes liant constructeurs et destructeurs peuvent servir de guide. Notons également que, S' etant choisi, on peut chercher à engendrer automatiquement la représentation des constructeurs: l'ordre associé à l'induction structurelle est alors un guide appréciable [TER, 78].

Il est important d'étudier le problème du choix d'une représentation et de se doter de critères de choix.

b) Représentation des simplificateurs

La définition de ces représentations est guidée, dans le cas de représentations simples, par la nécessité de préserver les théorèmes concernant les opérations externes.

6.3.2.- Esquisse d'un méta-algorithme de représentation

Décrivons informellement un méta-algorithme permettant de guider la construction de représentations (généralisées) simples.

Résultat (S', μ) représentation généralisée simple de S donné

(1) Choix du S.I. représentant et de la représentation μ des constructeurs

Représenter les pré-conditions des constructeurs

(2) Représentation des sélecteurs

Pour chaque sélecteur f de S faire

- Pour chaque axiome $fu_1..u_q \equiv \mathcal{C}$ dans S
  Construire l'équation $\mu(f)u'_1..u'_q \equiv \mathcal{C}'$
  déduite de $\mu(fu_1..u_q) \equiv \mu(\mathcal{C})$ en remplaçant chaque image par μ de constructeur par sa définition donnée en (1)
- Chercher $\mu(f)$ vérifiant le système d'équations ainsi défini.

Représentation de la pré-condition de f guidée par:

$$\vdash_{S'} \mu(\underline{pré}_S\ fx_1..x_q) \supset \underline{pré}_{S'}\ fx_1..x_q$$

Ceci peut conduire à enrichir le S.I. S' en introduisant de nouveaux symboles $\mu(f)$ définis par des équations

3 Représentation des destructeurs

Pour chaque destructeur f de S

(3.1) Pour chaque axiome $ax = fu_1..u_q \equiv \mathcal{C}$

(3.1.1) Construire comme en (2) l'équation $\mu(f)u'_1..u'_q \equiv \mathcal{C}'$

(3.1.2) Pour chaque sélecteur s de S écrire l'équation

$$\mu(s)\ (v_1..\mu(f)u'_1..u'_q..v_p \equiv \mu(s)\ v_1..\mu(f)u'_1..u'_q..v_p$$

on note $\Pi(f, ax)$ l'ensemble d'équations ainsi obtenu

(3.2) Chercher $\mu(f)$ vérifiant $\Pi(f, ax)$ pour tous les ax précédents

(3.3) Chercher $\mu(f)$ vérifiant les équations $\mu(f)u'_1..u'_q \equiv \mathcal{C}'$ pour tous les ax

(3.4) déduire de (3.2) et (3.3) une interprétation de l'égalité dans S'. Montrer qu'il s'agit d'une équivalence

(3.5) Représenter la pré-condition de f

Cette esquisse de méthode montre que les spécifications algébriques se prêtent assez bien à la construction de représentations en permettant de distinguer les étapes relevant d'un choix de celles qui peuvent être systématisées.

Remarque: La recherche des représentations des sélecteurs ou destructeurs apparait dans cette approche comme un *nouveau problème à résoudre*. Ainsi la résolution d'un problème initial P fait apparaître des sous-problèmes non plus issus directement de P par une démarche descendante mais liés à l'approche "transformationnelle" adoptée.

6.3.3.- Un exemple d'application

De nombreux problèmes informatiques sont des problèmes de représentation: c'est le cas de la pagination à la demande pour une mémoire virtuelle (ch 1.1,§ 1.1 h)).

En effet on peut définir le système d'information algébrique "mémoire virtuelle" comme étant de type fonction d'un ensemble d'adresses virtuelles dans un ensemble de valeurs. Les opérations privilégiées sont:

affvirt: $\overline{MEMVIR}$, $\overline{ADV}$, $\overline{VAL}$ ⟶ $\overline{MEMVIR}$ # affectation #

accèvirt: $\overline{MEMVIR}$, $\overline{ADV}$ ⟶ $\overline{VAL}$

De manière analogue on peut définir le système "mémoire physique" à deux sortes d'intérêt:

- mémoire principale $\overline{MEMP}$
- mémoire secondaire $\overline{MEMS}$

et des opérations:

affp : $\overline{MEMP}$, $\overline{ADP}$, $\overline{VAL}$ ⟶ $\overline{MEMP}$ # affectation #

accep: $\overline{MEMP}$, $\overline{ADP}$ ⟶ $\overline{VAL}$

char : $\overline{MEMP}$, $\overline{MEMS}$, $\overline{PAP}$, $\overline{PAS}$ ⟶ $\overline{MEMP}$ # chargement d'une page secondaire PAS sur une page principale #

dechar: $\overline{MEMP}$, $\overline{MEMS}$, $\overline{PAP}$, $\overline{PAS}$ ⟶ $\overline{MEMS}$ # opération inverse #

Si on caractérise alors les différentes adresses comme des couples (page, déplacement), le problème posé est donc un problème de représentation du système MEMVIR par le système "mémoire physique". Dans la construction de la représentation de affvirt et accèvirt apparaît naturellement le test: la page concernée est-elle présente en mémoire principale? dont on déduit la nécessité d'une table des pages présentes. De même apparaissent naturellement les questions relatives à la politique de gestion de cette table. Nous ne développerons pas davantage cet exemple ici.

CHAPITRE 2.3

LES ÉNONCÉS RÉCURRENTS

Au chapitre précédent nous nous sommes préoccupé de l'explicitation des systèmes d'information dans le but d'introduire des constructeurs et des simplificateurs. Nous nous intéressons maintenant à l'explicitation d'énoncés. Nous restreignons cette étude à un cas d'énoncés explicites particulièrement important : les *énoncés récurrents*. De manière intuitive, les énoncés récurrents définissent des ensembles de suites. Leur rôle fondamental en informatique résulte des remarques suivantes :

- Leur domaine d'application est très large puisque, à condition d'introduire des intermédiaires du genre pile, tout système d'équations récursives peut être transformé de façon systématique en un énoncé récurrent.
- Toute solution d'un énoncé récurrent est calculable
- Il existe une méthode de résolution (cf ch 1.3 § 3) pour les énoncés récurrents
- Il existe des critères (de terminaison) assurant l'existence de solution pour un énoncé récurrent.

Ainsi, dans un premier paragraphe, nous définissons et étudions les énoncés récurrents. De cette étude nous dérivons, au paragraphe 2, un langage de description d'énoncés récurrents , *MEDEE*, dont les principales constructions sont les définitions itératives . Les règles de construction d'énoncés MEDEE que nous introduisons alors sont justifiées par la *sémantique statique* du langage et permettent de définir des *stratégies d'explicitation récurrente* dont la plus importante est la *stratégie d'affaiblissement*.

Mais la forme des définitions itératives introduites au paragraphe 2 ne mettent pas bien en évidence le domaine de l'itération lorsque celui-ci est une suite définie par ailleurs (itération sur des fichiers par exemple). Le

paragraphe 3 propose une généralisation de la forme des définitions itératives ainsi que des règles et des stratégies de construction associées.

Le paragraphe 4 est une brève incursion dans le domaine des transformations d'énoncés récurrents. Nous expliquons comment de telles transformations peuvent être déduites de la sémantique statique des définitions itératives. Il s'agit ici de *transformations sémantiques* au sens d'ARSAC [ARS, 77] , c'est-à-dire de transformations permettant d'améliorer l'efficacité d'un énoncé récurrent.

Au paragraphe 5 nous montrons comment le fait de considérer les suites comme des "objets" permet de résoudre de classiques problèmes dans lesquels il y a un conflit entre la structure des données et celle des résultats (par exemple problèmesde rupture en algorithmique de gestion).

Nous terminons cette étude des énoncés récurrents par un bref tour d'horizon d'autres travaux sur ce sujet. Il faut cependant bien remarquer que la plupart des études sur la synthèse et la transformation de programmes se contentent de considérer les énoncés récurrents, à la fois, comme un cas particulier d'énoncés (point de vue de la construction) et comme le domaine cible de la suppression de la récursivité (point de vue de la transformation) [BUR, 77 b] , [BAU, 78 b] .

Notre point de vue rejoint celui d'ARSAC [ARS, 77] , d'ACHROFT et WADGE [ASH, 77] , DIJKSTRA [DIJ, 79 a] et SCHOLL [SCH, 79] qui soulignent l'importance des énoncés récurrents, la nécessité de définir des langages bien adaptés à l'expression de tels énoncés ainsi que des méthodes de constructions spécifiques.

1.- ETUDE DES ENONCES RECURRENTS.

1.1.- Définitions.

Intuitivement un énoncé récurrent a pour inconnues des suites ou des symboles o-aires. Pour nous ramener au cadre du chapitre 1.3 , dans lequel les résultats d'un énoncé sont des symboles o-aires, nous considérons les suites

comme des symboles o-aires . Ce point de vue est d'ailleurs tout à fait compatible avec la réalité informatique : les "résultats" ou les données d'un problème sont , en général, des suites représentées sur carte, sur fichier magnétique, etc...

Ainsi, dans tout ce paragraphe, S désigne un système d'information contenant au moint - BOOL
- ENTIERC (cf ch 1.2. § 5 3)
- Pour tout sous-système V de S , V* appartient à S . Ainsi S contient les suites d'entiers, les suites de suites d'entiers..... Rappelons qu'alors si $u \in (\underbrace{(V*)* \ldots}_{n \text{ fois}})* = V^{n*}$,

alors $u_{i_1 i_2 \ldots i_p}$ désigne appl(appl(appl(u, i_1), i_2...), ip) $\in V^{(n-p)*}$

pour $0 \leqslant p \leqslant n$ avec pour convention $V^{o*} = V$ (ch 1.2 § 7.2)

Définition 1 :

Un *énoncé récurrent élémentaire* de p suites $u^1 \ldots u^p$ d'un S.I. S est une formule du type :

$$\bigwedge_{1 \leqslant j \leqslant p} bi(u^j) \equiv k \ \wedge bs(u^j) \equiv n \ \wedge$$

$$u^j_i \equiv \underline{si} \quad i \equiv k \ \underline{alors} \ c^j \ \underline{sinon} \ \mathcal{C}^j[u^1_{i-1}/x_1, \ldots, u^p_{i-1}/x_p]$$

où

- k et n sont des termes de profils respectifs ENTIER et ENTIERC
- pour tout $1 \leqslant j \leqslant p$, c^j est un terme ne contenant pas la variable libre i
- pour tout $1 \leqslant j \leqslant p$, $\mathcal{C}^j$ est un terme dont les variables libres sont incluses dans l'ensemble $\{x_1, \ldots, x_p\}$

Notons que, d'après la définition de la conditionnelle et celle du domaine de définition d'une suite, on peut remplacer la définition de u_i^j par :

$$u_k^j \equiv c^j \wedge \forall i\ (k < i \leqslant n \supset u_i^j \equiv \mathcal{C}^j\ [u_{i-1}^j/x_1, \ldots, u_{i-1}^p/x_p]$$

Par exemple (cf ch 2.1 exemple 4 figure 5).

$$bi(c) \equiv 1 \wedge bs(c) \equiv n \wedge c_1 \equiv a \wedge \forall i\ (2 \leqslant i \leqslant n \supset c_i \equiv d_{i-1}) \wedge$$

$$bi(d) \equiv 1 \wedge bs(d) \equiv n \wedge d_1 \equiv b \wedge \forall i\ (2 \leqslant i \leqslant n \supset d_i \equiv c_{i-1} \bmod d_{i-1})$$

est un énoncé récurrent élémentaire.

Dans la définition précédente, on peut toujours se ramener à $p = 1$ en utilisant des S.I. produits. De plus, chaque u^j est un *terme* de profil "suite", il peut être de la forme $v_{i_1 \ldots i_p}$.

En particulier $\mathcal{C}^j$ peut être de l'une des deux formes suivantes :

i) $\mathcal{C}^j$ = *si* b *alors* $\mathcal{C}_1$ *sinon* $\mathcal{C}_2$

$u_i^j \equiv \mathcal{C}^j$ est alors appelée *définition conditionnelle*

ii) $\mathcal{C}^j = \mu i.\ b_i$ où i est une variable de profil ENTIERC et b un terme de profil BOOL* . Intuitivement cette expression désigne le "plus petit entier i tel que $b_i \equiv$ *vrai* soit un théorème". On l'utilise en particulier pour définir les bornes supérieures de suites.

μ est l'opérateur de minimisation introduit dans la définition des fonctions récursives [SHO, 67] . Définissons le maintenant :

Soit μ de profil BOOL* , ENTIER $\longrightarrow$ ENTIER défini par

pré $b_i \supset \mu(b, i) \equiv$ *si* b_i *alors* i *sinon* $\mu(b, i + 1)$

Alors $\mu i.b_i = \mu(b, bi(b))$.

Nous supposons désormais que le S.I. S dans lequel nous travaillons contient ce symbole μ et l'axiome correspondant.

Définition 2 :

Une *définition simple* d'un symbole o-aire t est une formule atomique :

$t \equiv \mathcal{C}$ où $\mathcal{C}$ est un terme de même profil que t

□

Définition 3 :

Soit $\mathcal{E} = \{D_1, \ldots, D_p\}$ un ensemble de définitions simples ou d'énoncés récurrents élémentaires. On appelle *relation de dépendance* sur $\mathcal{E}$ la relation Γ sur l'ensemble des symboles fonctionnels de $\mathcal{E}$ définie par :

u Γ v $\longleftrightarrow$ [u est membre gauche d'une définition simple $u \equiv \mathcal{C}$ de $\mathcal{E}$ et $\mathcal{C}$ contient un terme de la forme $v_{i_1 \ldots i_p}$] *ou*

[il existe un énoncé récurrent de $\mathcal{E}$ dans lequel

$u_{i_1 \ldots i_q} \equiv$ *si* $i_q \equiv k$ *alors* c *sinon* $\mathcal{C}\ [w^1_{q-1}/x_1 \ldots]$

et c ou $\mathcal{C}$ contient un terme $v_{j_1 \ldots j_p}$ avec $q \leqslant p$

ou $q > p$ et $i_1 = j_1 \ldots i_p = j_p$] □

intuitivement u dépend de v s'il est nécessaire de "calculer" v pour "calculer" u :

Définition 4 :

Un ensemble $\mathcal{E}$ de définitions simples ou d'énoncés récurrents élémentaires est *séquentiel* si la fermeture transitive de sa relation de dépendance Γ est sans circuit. □

Définition 5 :

Un *énoncé récurrent* est une conjonction de définitions simples et d'énoncés récurrents élémentaires dont l'ensemble est séquentiel □

Par exemple

$$x \equiv a/k \wedge y \equiv b/k \wedge k \equiv c_n \wedge$$

$$bi(c) \equiv 1 \wedge bs(c) \equiv n \wedge c_1 \equiv a \wedge \forall i(2 \leqslant i \leqslant n \supset c_i \equiv d_{i-1}) \wedge$$

$$bi(d) \equiv 1 \wedge bs(d) \equiv n \wedge d_1 \equiv b \wedge \forall i(2 \leqslant i \leqslant n \supset d_i \equiv c_{i-1} \bmod d_{i-1}) \wedge$$

$$n \equiv \mu i.(d_i \equiv 0)$$

est un énoncé récurrent.

Par contre

- $x = 3x + 2y \wedge y = 8x - y$
- $bi(x_i) = 1 \wedge bs(x_i) = n_i \wedge x_{ij} =$ *si* $j = 1$ *alors* y_i *sinon* $x_{i\,j-1+1}$ $\wedge$ $bi(bi) = 1 \wedge bs(b_i) = n_i \wedge b_{ij} = (x_{ij} \geq 0) \wedge bi(y) = 1 \wedge bs(y) = 25 \wedge y_i = x_{ij}$

n'en sont pas .

Remarque :

a) Cette définition diffère de celle donnée par ARSAC dans [ARS, 77] : un algorithme récurrent est de la forme :

$\underline{v}(i) \leftarrow R(\underline{v}(i-1), i, \underline{u}, \underline{d})$

$t(i) \leftarrow T(\underline{v}(i), i, \underline{u}, \underline{d})$

$\underline{u}, \underline{v}(0) \leftarrow I(d)$

$k \leftarrow \mu i.\ t(i)$

$\underline{r} \leftarrow F(\underline{v}(k), k, \underline{u}, \underline{d})$

où $\underline{v} = v^1, \ldots, v^p$ est un p-uplet de suites ; r est le résultat ; t la suite des conditions d'arrêt, $\underline{d}$ le q-uplet des données. R, T, I, et F sont "des algorithmes récurrents quelconques".

Dans cette définition on n'exprime pas de propriété de séquentialité : R ne dépend que de $\underline{v}(i-1)$ et pas de $\underline{v}(i)$. Il est clair que l'on peut toujours se ramener à ce cas si l'énoncé initial possède la propriété de séquentialité : on résoud, module par module, le système triangulaire

$$v^1(i) = \alpha\,(v^1(i-1))$$

$$v^2(i) = \beta\,(v^1(v),\ v^1(i-1),\ v^2(i-1))$$

Mais surtout cette définition implique une structuration de l'énoncé lorsque l'on travaille sur des suites à plusieurs indices, contrairement aux définitions précédentes où toutes les formules d'un énoncé récurrent sont au même niveau. Cette différence résulte du fait qu'ARSAC se préoccupe surtout

de décrire et de construire des énoncés récurrents. Ici, conformément à la démarche développée depuis le début de ce travail, nous distinguons l'étude fondamentale des propriétés d'un énoncé récurrent (qui se situe au niveau du système formel sous jacent) de la construction des énoncés récurrents (qui nécessite un langage de description structuré).

b) Elle diffère également de celle donnée par BURSTALL et DARLINGTON [BUR, 77 b] qui considèrent un énoncé récurrent comme un ensemble de définitions de la forme :

$$u^j(i_1 \ldots i_{n_j}) = \mathcal{T}^j$$

avec
- $\mathcal{T}^j$ ne contient aucun u^k
- ou $\mathcal{T}^j$ est de la forme $u^k(e_1, \ldots, e_{n_k})$, aucun e_ℓ ne contenant de u^i
- ou $\mathcal{T}^j$ est une conditionnelle dont les alternatives sont de l'une des formes précédentes.

Dans cette définition il n'y a pas de structuration, ce qui est regrettable pour un langage de descriptions. On ne fait également pas apparaître la propriété de séquentialité.

1.2.- Propriétés des énoncés récurrents :

Nous nous intéressons ici aux solutions d'un énoncé récurrent.

Théorème 1 :

Toute solution d'un problème, dont l'énoncé est récurrent, est calculable □

Pour démontrer ce théorème nous pouvons :

- soit décomposer ε en un énoncé strict et un énoncé des intermédiaires
- soit considérer ε comme un énoncé strict dont le résultat est un p-uplet de valeurs (formé du résultat proprement dit et des intermédiaires). (cf ch 1.3 § 1)

La démonstration se fait alors par récurrence sur le nombre d'inconnues de ε , en effectuant un raisonnement par cas selon que le $p^{ième}$ intermédiaire est défini par un énoncé récurrent élémentaire ou par une définition simple , et en utilisant la propriété de séquentialité.

Ce théorème met en évidence le principal avantage des énoncés récurrents par rapport aux définitions récursives générales.

Corollaire :

Un énoncé récurrent admet au plus une solution □

De plus il est possible d'associer aux énoncés récurrents une *règle de calcul* plus simple que dans les systèmes d'équations récursives :

Soit $r^1 \ldots r^p$ les inconnues de ε , on calcule $r^j_{j_1 \ldots j_{q_j}}$ avant $r^i_{i_1 \ldots i_{q_i}}$

si : - il existe k , $1 \leq k \leq \min(q_j, q_i)$ tel que

$j_1 = i_q \ldots j_{k-1} = i_{k-1}$ et $j_k < i_k$

- ou si pour tout k , $1 \leq k \leq \min(q_j, q_i)$ on a

$j_k = i_k$ et $r^i \Gamma r^j$

La justification de cette règle est semblable à la démonstration du théorème 1.

Son application permet de calculer la solution de ε si elle existe.

Il faut aussi remarquer que cette règle de calcul constitue une méthode de résolution des énoncés récurrents, *déterministe* et *indépendante des données* (ch 1.3 § 5.2.2). Ceci ne veut, bien entendu, pas dire qu'un seul ordre de calcul est possible puisque la règle précédente demande uniquement

de respecter la relation de dépendance Γ lorsque les indices communs sont les mêmes. On retrouve la notion de calcul collatéral d'ALGOL 68 [WIN, 75].

On peut ainsi associer à un énoncé récurrent ε un ensemble de calculs, c'est-à-dire l'ensemble des démonstrations de théorèmes $r^1 \equiv s^1 \ldots\ldots r^p \equiv s^p$. C'est l'idée à la base de la *sémantique calculatoire* [FIN, 76 a] , [FIN, 78 c] .

Il est également possible de préciser les conditions d'existence de solutions d'énoncés récurrents :

Théorème 2 :

Soit $P = (E, S)$ un problème dont l'énoncé ε est récurrent. P admet une solution pour la donnée (D, ρ) si :

(1) Pour toute définition $n \equiv \mu i.b_i$, il existe un k tel que $\vdash_{SD\ [\rho(\varepsilon)]} b_k \equiv \underline{vrai}$

(2) Pour toute définition $x \equiv f\, u_1 \ldots u_p$, si $\vdash_{SD[\ \rho(\varepsilon)]} u_i \equiv s_i$ pour $i = 1, \ldots, p$ alors $\vdash_{SD\ [\ \rho(\varepsilon)]} \underline{pré}\ f u_1 \ldots u_p \equiv \underline{vrai}$ □

(1) apparaît comme un cas particulier de (2) relativement à μ .

(1) exprime la propriété classique de terminaison, (2) est une règle de non blocage.

La démonstration de ce théorème résulte également de la séquentialité. Elle s'effectue encore par récurrence sur le nombre d'inconnues de E .

1.3.- Sémantique d'un énoncé récurrent

Comme il s'agit d'un cas particulier d'énoncé récursif toutes les sémantiques développées dans ce cadre sont utilisables. Une des plus naturelles est certainement la sémantique fonctionnelle (ou encore sémantique point fixe). On se place dans un modèle du système d'information S et on prolonge l'interprétation aux symboles inconnus. L'image de l'énoncé ε par l'interprétation

devient alors un système d'équations dont les inconnues sont les images des symboles inconnus de $\mathcal{E}$. L'existence d'une solution (différente de l'indéfini ω) est liée à la vérification des règles du théorème 2.

1.4.- Nécessité d'un langage de description.

Bien que les énoncés récurrents présentent de nombreux intérêts pour la résolution d'un problème, leur expression sous forme d'une formule d'un système d'information, comme nous l'avons décrite ci-dessus (et utilisée dans l'exemple 4 du chapitre 2.1) est malcommode. Ceci est dû en partie aux différents indices de suites, mais aussi au *manque de structuration d'une formule*. Pour éviter ces inconvénients nous introduisons maintenant le langage MEDEE dont les qualités de modularité et de simplicité d'écriture donnent une plus grande lisibilité mais aussi facilitent la construction descendante et systématique d'un énoncé récurrent. C'est sur ce langage que sont décrites les règles de construction d'énoncés récurrents et les stratégies associées.

2.- LE LANGAGE MEDEE ET LA CONSTRUCTION D'ENONCES RECURRENTS

Introduit comme langage support de la méthode déductive de programmation [PAI, 79] , [BEL, 78 a] , [BEL, 78 b] , MEDEE est un langage *définitionnel* (on dit encore *applicatif*) permettant de décrire de manière modulaire et descendante des énoncés récurrents ou récursifs. L'aspect description de définitions récursives étant tout à fait classique nous allons nous limiter essentiellement au caractère itératif de MEDEE.

2.1.- Description sommaire de MEDEE.

2.1.1.- Exemples.

MEDEE est un langage essentiellement conçu pour la description d'énoncés itératifs. Son utilisation doit normalement s'accompagner de celle d'un langage de description de systèmes d'information. Nous utilisons ici une

syntaxe légèrement différente de celle présentée dans [HUC, 77] ou [FIN, 78 c] , encore un peu plus éloignée des aspects dynamiques des langages classiques et donc mieux adaptée à une construction à partir d'un énoncé implicite.

Brièvement, une description d'énoncé en MEDEE (on parle de programme MEDEE) est formée d'un module principal définissant un résultat, qui refère différents modules de niveau inférieur, définissant à leur tour des résultats et ainsi de suite.

En règle générale, un module est identifié par le nom de son résultat.

Cette syntaxe apparaît comme une extension des langages de description de problèmes (s-descriptions et a-descriptions) introduits auparavant.

Exemple 1 :

L'énoncé de la figure 5 du chapitre 2.1 (version récurrente du problème de la fraction irréductible) s'écrit, en MEDEE, sous la forme de la figure 1.

	profil	énoncé en MEDEE
<< x, y >> *tq* propor(x, y, a, b)∧ prem(x,y) k = pgcd (a, b) << c, d >> *tq* ∀i, pgcd(c_i, d_i) = pgcd(a, b)	<< x,y >> : COUPLE k : ENTIER << c,d >> : COUPLE* << a,b >> : COUPLE	*résultat* << x, y >> x = a/k y = b/k k = *premier* c_i *pour* i *dans* 1 → *tel que* d_i = 0 *donné* << a, b >>
<< c, d >>		

		c = $\bar{d}$
		d = $\bar{c}$ mod $\bar{d}$
		$ c = a
		$ d = b

Figure 1 : un énoncé MEDEE

La colonne de gauche de cet exemple contient les spécifications (implicites le plus souvent) des résultats et des intermédiaires. La colonne centrale précise les profils des différents symboles et la colonne de droite l'énoncé MEDEE proprement dit.

Le double trait horizontal permet de distinguer le module principal du module " *<< c, d >>" .

Ce dernier définit de manière récurrente les suites c et d . L'imbrication des modules permet de laisser sous entendu l'indice courant : c est utilisé pour c_i , $\overline{c}$ pour c_{i-1} et $ c pour c_1 . □

Si l'on compare ce langage au langage des s-descriptions, on y note simplement la mise en oeuvre effective de la modularité liée à la définition du terme général d'une suite (cf ch 1.3 § 2.3.1).

Exemple 2 :

La figure 2 représente un énoncé en MEDEE qui résoud le problème des sous-suites ascendantes de longueur maximum [BRO, 79 a] .

Cet énoncé donne un exemple d'imbrication de modules . On voit ainsi que les conventions de suppression d'indice courant sont bien adaptées à une analyse par niveau :
à une étape donnée on peut"oublier" le niveau d'imbrication où l'on se trouve. Ainsi ind, maxim sont définis comme des entiers alors qu'au niveau de l'énoncé complet ce sont des suites d'entiers ; md, maxi et mdint sont définis comme des suites alors que ce sont des suites de suites. Il faut alors être bien conscient des abréviations utilisées : $\overline{md}(k)$ représente md(i-1, k) et non pas md(i, k-1) ! □

2.1.2.- Syntaxe de MEDEE.

Nous nous contentons de donner une description informelle d'un sous-langage de MEDEE.

Un énoncé MEDEE est un ensemble de modules ayant une structure arborescente pour la relation "le module x utilise le module y" .

Un module MEDEE est un ensemble de *définitions* ; il est en général identifié par le nom, précédé de " * " , d'un de ses résultats. Une définition est de l'un des deux types suivants :

a) Définition simple

de la forme $x = f(x_1, \ldots, x_p)$ où x, $x_1 \ldots x_p$ sont

i) soit des identificateurs
ii) soit des identificateurs surlignés (sauf x , qui peut être $ y)
iii) soit des éléments de suites de la forme $u_{i_1 \ldots i_k}$

et f une opération du S.I. considéré

Un cas particulier important est celui où f est une conditionnelle.

b) Définition itérative

b1) Expression itérative

L'expression u_i *pour* i *dans* k → n

définit suite de terme général u_i , de borne inférieure k et supérieure n ; si n = + ∞ on admet de ne pas le faire figurer dans l'expression.
Une telle description de suite est appelée *expression itérative*.

La définition de md(cf figure 2) est un exemple de définition d'une suite à l'aide d'un tel constructeur.

b2) Opérations sur les suites :

Les figures 1 et 2 donnent quelques exemples des opérations portant sur les suites que l'on utilise dans les énoncés MEDEE :

. *der* fournit le dernier terme d'une suite (cf ch 1.2 § 7.2.2)

. *jqa* permet de tronquer une suite à partir du premier élément vérifiant une certaine condition.

énoncé implicite	profils	énoncé en MEDEE
$\ell = \max\{k \mid tg\ \exists \beta\ \underline{tg}\ \beta \trianglelefteq \alpha \wedge croiss(\beta) \wedge \|\beta\| = k\}$ $\ell\ \max\ \underline{tg}$ $\ell max_i = \max(k\ \underline{tg}\ \exists \beta\ \underline{tg}\ \beta \trianglelefteq tr(\alpha, 1, i) \wedge croiss(\beta) \wedge \|\beta\| = k)$	ℓ : ENTIER ℓmax : ENTIER* n : ENTIER α : V*	*résultat* ℓ ℓ = *der* ℓmax_i *pour* i *dans* 1 → n n = $\|\alpha\|$ *donnée* α
* ℓmax		
$id \in [1..\ell max_{i-1}+1] \wedge md_{i,id-1} \leqslant \alpha_i < \bar{md}_{i,id}$ $md_{i,k} = \min\{der(\gamma)\ \underline{tg}\ \gamma \trianglelefteq tr(\alpha, 1, 1) \wedge croiss(\gamma) \wedge \|\gamma\| = k-1\}$ $md = mdint$	id : ENTIER md : (V ∪ {−∞,+∞})* maxi : ENTIER* arret : BOOL* mdint : (V ∪ {−∞,+∞})*	ℓmax = *si* $\overline{id}$ $\overline{\ell max}$ < id *alors* id *sinon* $\overline{\ell max}$ $ 1 id = *der* $maxi_j$ *pour* j *dans* 1 → *jqa* $arret_j$ md = $mdint_k$ *pour* k *dans* 1 → ℓmax $ md = tr(α, 1, 1)
*maxi		
$md_{i,mini-1} \leqslant \alpha_i < md_{i,maxi} \wedge 1 \leqslant mini_{i,j} \leqslant maxi_{i,j} \leqslant \ell max_{i-1} + 1$	mini, maxi : ENTIER dicho : ENTIER	<< maxi, mini >> = *si* $\alpha_i < \overline{md}_{dicho}$, *alors* << dicho, $\overline{mini}$ >> *sinon* << $\overline{maxi}$, dicho +1 >> dicho = $(\overline{maxi} + \overline{mini}) \div 2$ $ maxi, $ mini = ℓmax +1 , 1 ⟩ arrêt = (mini ⩾ maxi) $ arrêt = *faux*
* mdint		
		mdint = *si* k = ind *alors* id *sinon* $\overline{md}_k$

Figure 2 : énoncé récurrent du problème des sous suites ascendantes maximales.

b3) Forme d'une définition itérative :

Une définition itérative se présente de manière analogue à une définition simple :

$x = [\underline{der}]^{0/1}\ u_i\ \underline{pour}\ i\ \underline{dans}\ k \rightarrow n\ [\underline{jqa}\ c_i]^{0/1}$

où $[f]^{0/1}$ indique que f peut être absent.

D'autres formes de définitions itératives ont été utilisées précédemment :

- $x =$ *premier* u_i *pour* i *dans* k → n *tel que* c_i qui est identique à :
  $x =$ *der* u_i *pour* i *dans* k → n *jqa* c_i
- $x = \Sigma\ u_i$ *pour* i *dans* k → n *jqa* c_i qui est identique à :
  $x =$ *der* s_i *pour* i *dans* k → n *jqa* c_i

si le module *s est égal à *u ∪ {s = $\bar{s}$ + u , $ s = 0}

u ∈ ENTIER* ∪ REEL*

Nous ne développerons pas davantage l'introduction et l'utilisation de telles opérations ; de même, nous ne décrirons pas la manipulation de fonctions en MEDEE.

Terminons cette brève description syntaxique de MEDEE en indiquant qu'un énoncé MEDEE doit posséder la propriété de séquentialité (définition 4) qui se définit simplement dans ce cadre [FIN, 78 c] . Il suffit de définir la relation de dépendance pour chaque module : Si x et y sont deux identificateurs d'un même module m ,

$x\ \Gamma\ y \iff$ y occurre en partie droite de la définition de x ou $\bar{x}$ occurre en partie droite de la définition de y .

2.2.- Sémantique statique de MEDEE.

Le langage MEDEE permettant des abréviations d'écriture et une grande modularité il faut être capable de caractériser précisément l'énoncé récurrent

décrit par un énoncé MEDEE.

La *sémantique statique* est une fonction qui associe à tout énoncé MEDEE une formule dans le S.I. correspondant [FIN, 78 c] . Le terme sémantique n'est pas utilisé ici dans son acception usuelle puisqu'il s'agit plutôt d'une traduction d'un langage formel dans un langage d'expression de système formel. Pour définir une sémantique de MEDEE, au sens classique, il faut alors composer la sémantique statique avec une sémantique quelconque (cf figure 3).

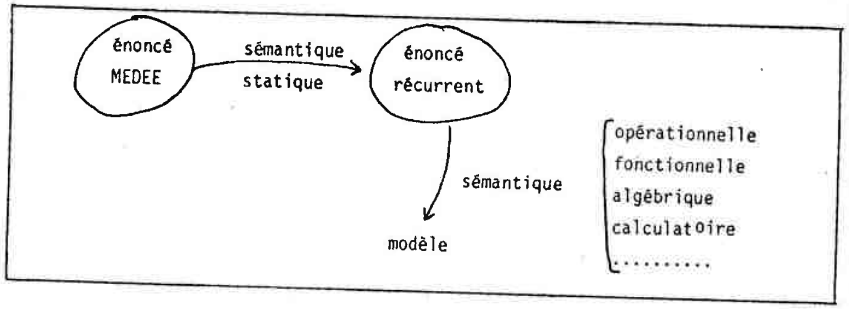


Figure 3 : rôle de la sémantique statique

On trouve dans [LES, 78] et [FIN, 78 d] une sémantique point-fixe directe de MEDEE appelée sémantique descriptive.

2.2.1.- Conventions de notations.

a) Nous supposons associé à tout énoncé MEDEE un prédicat appelé *dépend de* entre identificateurs et noms de modules, défini par

x *dépend de* m si et seulement si x est défini dans le module m ou dans un des sous-modules de m.

Ainsi dans l'exemple 2, md *dépend de* *&max, dicho *dépend de* * &max mais md *dépend de* * maxi est faux.

Notons qu'à partir de cette relation de dépendance il est possible de définir des améliorations d'énoncés consistant à "sortir" des constantes de modules itérés [BEL, 79] .

b) Nous introduisons une fonction ind permettant "l' indiçage" des symboles fonctionnels de suites. Elle est définie sur les identificateurs par :

$ind(x_{i_1 \ldots i_p}, i, m) =$ si x *dépend de* m alors $x_{i i_1 \ldots i_p}$

sinon $x_{i_1 \ldots i_p}$

$ind(\overline{x}_{i_1 \ldots i_p}, i, m) =$ si x *dépend de* m alors $x_{i-1 i_1 \ldots i_p}$

sinon $x_{i_1 \ldots i_p}$

$ind(\&x_{i_1 \ldots i_p}, i, m) =$ si x *dépend de* m alors $x_{bi(x_{i_1 \ldots i_p}) i_1 \ldots i_p}$

sinon $x_{i_1 \ldots i_p}$

ind se prolonge naturellement aux termes puis aux formules du S.I. sous-jacent.

c) Nous supposons que les identificateurs, et en particulier les indices de définitions itératives, d'un énoncé MEDEE sont tous différents. Plus précisément, le langage MEDEE permet des définitions locales d'identificateurs avec une règle de portée associée à la structure modulaire. Nous supposons ici que la sémantique statique ne s'applique qu'à des énoncés résultant d'une opération de transcription sur les identificateurs qui évite toute double définition d'un même identificateur.

d) Nous notons généralement $*u$ le module associé à la suite u : $\$m$ est l'ensemble des définitions d'un module m dont le membre de gauche est précédé de $\$$. $*m$ est le complémentaire de $\$m$ dans m .

ident (m) désigne l'ensemble des identificateurs apparaissant soit en membre gauche d'une définition d'un module m soit en membre gauche d'un sous module de m.

Ainsi , $\$ * \ell max = \{ \$ \ \ell max = 1, \$ md = tr(\alpha , 1, 1)\}$,

ident $(* \ell max) = \{\ell max, id, md, maxi, arrêt, mdint, mini, dicho\}$

dans l'exemple de la figure 2.

2.2.2.- Définition de la sémantique statique.

Soit $\mathcal{E}$ un énoncé MEDEE et S le S.I. sous-jacent (i.e le S.I. du problème dont $\mathcal{E}$ est l'énoncé). Nous supposons que S est comme au paragraphe 1.1 (il contient ENTIERC, BOOL, les suites, μ,etc...).

La sémantique statique SEMST associe à $\mathcal{E}$ une formule de S définie récursivement sur la structure de $\mathcal{E}$ de la manière suivante :

a) La sémantique statique de $\mathcal{E}$ est celle de son module principal

b) La sémantique d'un module m est la conjonction des définitions $D_1, \ldots, D_p$ de m :

$$SEMST\ [\![m]\!] = SEMST\ [\![D_1]\!] \wedge \ldots\ldots \wedge SEMST\ [\![D_p]\!]$$

c) La sémantique d'une définition simple $x = f(x_1, \ldots, x_p)$ est la définition elle même :

$$SEMST\ [\![x = f(x_1, \ldots, x_p)]\!] = x \equiv f(x_1, \ldots, x_p)$$

d) Pour définir la sémantique statique d'une définition itérative il suffit de caractériser chaque constituant de son membre de droite :

d1) Sémantique de l'expression itérative u_i *pour* i *dans* $i \rightarrow n$:

$$SEMST\ [\![u_i \ \underline{pour}\ i \ \underline{dans}\ k \rightarrow n]\!] = DOM(u, k, n) \wedge MOD(u, i)$$

avec

$$MOD(u, i) = (bi(u) < i \leq bs(u)) \supset ind(SEMST\ [\![* *u]\!], i, *u)) \wedge ind(SEMST\ [\![\$ *u]\!], bi(u), *u)$$

et

$$DOM(u, k, n) = \bigwedge_{y \in ident(*u)} (bi(y) \equiv k-1 \wedge bs(y) \equiv n)$$

Dans cette définition d'une part, on exprime le domaine (DOM) des différentes suites définies par $*u$, d'autre part, on précise la définition du terme général de chacune de ces suites ainsi que la définition du premier terme.

d2) Définition de l'opération *iqa*

u *iqa* a $=v$

tq $bi(v) = bi(u) \wedge bs(v) = inf(bs(u), \mu i.a_i) \wedge$

$\forall i\ (bi(v) \leq i \leq bs(v) \supset v_i = u_i)$

où inf est une opération sur ENTIERC.

d3) Définition itérative d'une suite

$$SEMST\ [\![v = u_i \ \underline{pour}\ i \ \underline{dans}\ k \rightarrow n]\!] = (v \equiv u) \wedge SEMST\ [\![u_i \ \underline{pour}\ i \ \underline{dans}\ k \rightarrow n]\!]$$

d4) Définition complète de la sémantique d'une définition itérative : Elle résulte de d1), d2), d3) et de la définition de l'opération *der* :

$$SEMST\ [\![x = \underline{der}\ u_i \ \underline{pour}\ i \ \underline{dans}\ k \rightarrow n \ \underline{iqa}\ a_i]\!] =$$

$$DOMA(u, k, n) \wedge MOD(u, k, n) \wedge x \equiv u_{bs(u)}$$

où

$$DOMA(u, k, n) = \bigwedge_{y \in ident(*u)} (bi(y) \equiv k-1 \wedge bs(y) \equiv inf(n, \mu i.a_i))$$

Exemple 3 :

Sémantique statique de l'énoncé de la figure 2.

$SEMST\ [\![\mathcal{E}]\!] = SEM1 \wedge SEM2$

$SEM2 = SEMST\ [\![n = |\alpha|]\!] = n \equiv |\alpha|$

$SEM1 = SEMST\ [\![\ell = \underline{der}\ \ell max_i \ \underline{pour}\ i \ \underline{dans}\ 1 \rightarrow n]\!]$

$= bi(\ell max) \equiv 0 \wedge bi(ind) \equiv 0 \wedge bi(maxi) \equiv 0 \wedge bi(arrêt) \equiv 0 \wedge bi(md) \equiv 0$

$\wedge$ bi(mini) = 0 $\wedge$ bi(dicho) = 0 $\wedge$ bs(ℓmax) = n $\wedge$ bs(ind) = n $\wedge$
bs(maxi) = n $\wedge$ bs(arrêt) = n $\wedge$ bs(md) = n $\wedge$ bs(mini) = n $\wedge$
bs(dicho) = n $\wedge$
$(1 \leqslant i \leqslant n \supset$ ind(SEM1', i, * ℓmax)) $\wedge$
$\ell max_0 = 1 \wedge md_0 = tr(\alpha, 1, 1)$ $\wedge$

$\ell = \ell max_n$

SEM1' = SEM11 $\wedge$ SEM12 $\wedge$ SEM13

SEM11 = SEMST [ℓmax = *si* $\overline{\ell max} <$ ind *alors* ind *sinon* $\overline{\ell max}$]

= ℓmax = *si* $\overline{\ell max} <$ ind *alors* ind *sinon* $\overline{\ell max}$

SEM12 = SEMST [ind = *der* $maxi_j$ *pour* j *dans* 1 $\rightarrow$ *jqa* $arrêt_j$]

= bi(maxi) = 0 $\wedge$ bi(mini) = 0 $\wedge$ bi(dicho) = 0 $\wedge$ bi(arrêt) = 0 $\wedge$
bs(maxi) = d $\wedge$ bs(mini) = d $\wedge$ bs(dicho) = d $\wedge$ bs(arrêt) = d $\wedge$
d = μ j.$arrêt_j$ $\wedge$
$(1 \leqslant j \leqslant d \supset$ ind(SEM12' , j, * maxi)) $\wedge$
$maxi_0 = \overline{\ell max} + 1 \wedge mini_0 = 1 \wedge arrêt_0 =$ *faux* $\wedge$
ind = $maxi_d$

SEM13 = SEMST [$md = mdint_k$ *pour* k *dans* 1 $\rightarrow$ ℓmax]

= (md = mdint) $\wedge$ bs(mdint) = ℓmax $\wedge$ bi(mdint) = 0 $\wedge$
$(1 \leqslant k \leqslant \ell max \supset$ ind(SEM13', k, * mdint))

SEM12' = SEM121 $\wedge$ SEM122 $\wedge$ SEM123

SEM121 = << maxi, mini >> = *si* $\alpha_i < \overline{md}_{dicho}$ *alors* << dicho, $\overline{mini}$ >>
sinon << $\overline{maxi}$, dicho+1 >>

SEM122 = dicho = $(\overline{maxi} + \overline{mini}) \div 2$

SEM123 = arrêt = (mini $\geqslant$ maxi)

SEM13' = mdint = *si* k = ind *alors* ind *sinon* $\overline{md}_k$

ind(SEM12', j,* maxi) = (<< $maxi_j$, $mini_j$ >> = *si* $\alpha_i < \overline{md}_{dicho_j}$
alors << $dicho_j$, $mini_{j-1}$ >>
sinon << $maxi_{j-1}$, $dicho_{j+1}$ >>) $\wedge$
$dicho_j = (maxi_{j-1} + mini_{j-1}) \div 2$ $\wedge$
$arrêt_j = (mini_j \geqslant maxi_j)$

ind(SEM13', k, * mdint) = $mdint_k$ = *si* k = ind *alors* ind *sinon* $\overline{md}_k$

d'où finalement :

SEMST [$\mathcal{E}$] = n = |α| $\wedge$
$\ell = \ell max_n$ $\wedge$
bi(ℓmax) = 0 $\wedge$ ....$\wedge$ bi(dicho) = 0 $\wedge$ bs(ℓmax) = n $\wedge$...$\wedge$ bs(dicho) = n $\wedge$
$\ell max_0 = 1 \wedge md_0 = tr(\alpha, 1, 1)$
$1 \leqslant i \leqslant n \supset$

SEM11 ---> ℓmax_i = *si* $\ell max_{i-1} <$ ind_i *alors* ind_i *sinon* ℓmax_{i-1} $\wedge$

SEM12 ---> $bi(maxi_i) = 0 \wedge$ ....$\wedge$ $bi(arrêt_i) = 0 \wedge bs(maxi_i) = d_i \wedge$...$\wedge$
= $bs(arrêt_i) = d_i \wedge$
$d_i = \mu$ j.$arrêt_{i,j} \wedge maxi_{i,0} = \ell max_{i-1} \wedge mini_{i,0} = 1 \wedge$
$arrêt_{i,0} =$ *faux* $\wedge$ $ind_i \equiv maxi_{di} \wedge$
$1 \leqslant j \leqslant d_i \supset$

ind(SEM12', j,*maxi) ----->
[<< $maxi_{i,j}$, $mini_{i,j}$ >> = *si* $\alpha_i < md_{i-1}$'$dicho_{i,j}$
alors .....
sinon .....

SEM13 ---> $md_i = mdint_i \wedge bs(mdint_i) = \ell max_i \wedge bi(mdint_i) = 0$ $\wedge$
$1 \leqslant k \leqslant \ell max_i \supset$

SEM13'--- > $mdint_{i,k}$ = *si* k = ind_i *alors* ind_i *sinon* $md_{i-1,k}$

L'absence de structuration d'une telle formule en rend la compréhension peu aisée. □

Ainsi MEDEE apparaît comme un langage de description structurée (modules) et simplifiée (omission d'indices) d'énoncés explicites. Sa restriction aux définitions simples et aux définitions itératives concerne la description des seuls énoncés récurrents.

2.3.- Construction d'énoncés MEDEE.

Comme dans le cas des définitions récursives (ch 2.2) nous cherchons des règles de construction d'énoncés récurrents sur lesquels reposeront des stratégies d'explicitation récurrentes.

Nous sommes guidé par le fait que l'énoncé MEDEE $\mathcal{E}'$ construit à partir d'un énoncé ε doit *vérifier* cet énoncé (ch 1.3 § 6.2). Pour être plus précis, $\mathcal{E}'$ n'étant qu'une description , il faut démontrer

$$\vdash_{S} \text{SEMST}(\mathcal{E}') \supset \varepsilon$$

Evidemment, pour faire ce genre de démonstration , on souhaite :

- pouvoir effectuer des vérifications locales(module par module, définition par définition)
- ne pas avoir à utiliser directement la sémantique statique trop compliquée.

Nous somme ainsi conduit à introduire des règles de preuves, nous préférons, comme au chapitre 2.2, parler de règles de construction, pour le langage MEDEE.

Ces règles sont justifiées à l'aide de la sémantique statique.

2.3.1.- Règles de construction d'énoncés MEDEE.

Soit $\mathcal{E}$ un énoncé MEDEE de résultat r . ε l'énoncé du S.I. sous-jacent caractérisant r . Nous supposons qu'à chaque identificateur a de $\mathcal{E}$ est associé un sous énoncé ε_a de ε . On note alors ε_y l'énoncé suivant :

- si y est un identificateur u de suite apparaissant dans une définition itérative :

  ... u_i *pour* i *dans* k → n *faq* c_i

  alors $\varepsilon_y = \varepsilon_u \ [u/u_i]$

- si y est de la forme $\bar{u}$, où u est comme ci-dessus :

  $\varepsilon_y = \varepsilon_u \ [\bar{u}/u_i, \ i\text{-}1/i]$

- si y est de la forme $\$u$, où u est comme ci-dessus :

  $\varepsilon_y = \varepsilon_u \ [\$u \ / \ u_i \ , \ k\text{-}1/i]$

Avec ces conventions , toujours en notant S le S.I. sous-jacent, nous pouvons énoncer les règles suivantes :

a) Introduction de définition simple :

(RIDS) Règle d'introduction d'une définition simple

La définition simple $x = f(x1 \ldots, xp)$ est *correcte* vis à vis des énoncés $\varepsilon_x, \varepsilon_{x1} \ldots \varepsilon_{xp}$ si

$$\vdash_S \varepsilon_{x1} \wedge \ldots \wedge \varepsilon_{xp} \supset \textit{pré} \ f_{x1 \cdots xp} \wedge \varepsilon_x \ [fx_1 \ldots x_p/x]$$

Par exemple dans l'énoncé MEDEE de la figure 1,
<< c, d >> est caractérisé par $\text{pgcd}(c_i, d_i) = k$. Pour que la définition
<< c, d >> = << $\bar{d}$, $\bar{c}$ mod $\bar{d}$ >> soit correcte vis à vis de

$\varepsilon_{<< c,d >>} = (\text{pgcd}(c, d) = \text{pgcd}(a, b))$ et $\varepsilon_{<< \bar{c},\bar{d} >>} = (\text{pgcd}(\bar{c}, \bar{d}) = \text{pgcd}(a, b))$

il faut et il suffit que

$$\text{pgcd}(\bar{c}, \bar{d}) = \text{pgcd}(a, b) \supset \textit{pré} \ \bar{c} \mod \bar{d} \wedge \text{pgcd}(\bar{d}, \bar{c} \mod \bar{d}) = \text{pgcd}(a, b)$$

Théorème 3 :

Avec les notations précédentes, si $x = f(x1 \ldots xp)$ est correcte vis à vis de $\varepsilon_x, \ldots, \varepsilon_{xp}$ □

$$\text{SEMST} \ [\![x = f_{(x1 \ldots xp)}]\!] \ \text{vérifie} \ \varepsilon_x \ \text{dans S} \ [\varepsilon_{x1}, \ldots, \varepsilon_{xp}]$$

□

La restriction de cette règle à $\varepsilon_{x1} \wedge \ldots \wedge \varepsilon_{xp} \supset \textit{pré} \ fx1 \ldots xp$ constitue une *règle de non blocage* (cf chapitre 2.4 exemple 1).

b) Introduction d'une définition itérative.

(RIDI) Règle d'introduction d'une définition itérative

La définition itérative

x = *der* u_i *pour* i *dans* $k \to n$ *iqa* a_i

est *correcte* vis à vis de ε_x , ε_u (l'invariant de la suite u), π_a(la condition d'arrêt) si

$$\vdash_S \pi_a \wedge nl = \inf(n, \mu i.a_i) \wedge \varepsilon_u[nl/i\ ,\ u_{nl}/u] \supset \varepsilon_x\ [u_{nl}/x]$$

Toujours en reprenant l'exemple 1, $\varepsilon_u = (pgcd(c, d) = pgcd(a, b))$,

$\varepsilon_x = (k = pgcd(a, b))$

$\pi_c = (d_i = 0)$ alors on a bien :

$$nl = \inf(+\infty, \mu i.(d_i = 0)) \wedge pgcd(c_{nl}, d_{nl}) = pgcd(a, b) \supset (c_{nl} = pgcd(a, b))$$

Théorème 4 :

Avec les notations précédentes, si x = *der* u_i *pour* i *dans* $k \to n$ *iqa* a_i est correcte vis à vis de ε_x, ε_u, π_c alors

SEMST $[\![x$ = *der* u_i *pour* i *dans* $k \to n$ *iqa* $a_i]\!]$ vérifie ε_x dans S $[\varepsilon_u, \pi_a]$.

Ceci résulte immédiatement du fait que

SEMST $[\![x$ = *der* u_i *pour* i *dans* $k \to n$ *iqa* $a_i]\!] \supset bs(u) = \inf(n, \mu i.a_i) \wedge x = u_{bs(u)}$

On remarque ainsi que la justification des règles ne nécessite pas l'utilisation de toute la sémantique. Un point de vue voisin consiste à constater l'inexistence d'une règle de récurrence. En fait "la règle de récurrence" se trouve à un autre niveau dans un processus de démonstration : pour démontrer que ε_u est invariant on prouve, avec les notations précédentes , que :

$$k \leq i \leq bs(u) \wedge \varepsilon_u[i\text{-}1/i] \wedge ind(SEMST\ [\![** u]\!], i, *u) \supset \varepsilon_u$$

et $$ind(SEMST\ [\![\S * u]\!], k\text{-}1, *u) \supset \varepsilon_u\ [k\text{-}1/i]$$

Ces propriétés se démontrent à l'aide des règles précédentes (en considérant u à la place de u_i, $\bar{u}$ à la place de u_{i-1} et $\S u$ à la place de u_{k-1}).

Il est immédiat alors, par récurrence sur le nombre d'idéntificateurs, que si chaque définition d'un énoncé $\mathcal{E}$ est correcte vis à vis des sous-énoncés de ε correspondants, la sémantique statique de $\mathcal{E}$ vérifie ε .

c) Correction totale

Du théorème 2 (§ 1.2) on déduit une règle assurant l'existence d'une solution pour une définition itérative.

(RTDI) Règle de terminaison d'une définition itérative.

La définition itérative

x = *der* u_i *pour* i *dans* $1 \to n$ *iqa* a_i

se *termine* si $\vdash_{SI\ \varepsilon_u,\ \pi_a]} \exists i$ *tq* a_i

2.3.2.- Utilisation des règles précédentes.

Nous pouvons maintenant préciser les stratégies générales de construction d'énoncés MEDEE.

a) Explicitation simple

◁ SES ▷ Stratégie d'explicitation simple

donnée x caractérisé par ε_x

résultat f symbole fonctionnel p-aire, p symboles $x1 \ldots xp$ caractérisés par p énoncés $\varepsilon_{x1}, \ldots, \varepsilon_{xp}$

tq $\varepsilon_{x1} \wedge \ldots \wedge \varepsilon_{xp} \supset$ *pré* $fx1 \ldots xp \wedge \varepsilon_x\ [fx1 \ldots xp/x]$

Présentation des résultats :

- $\varepsilon_{x1}, \ldots, \varepsilon_{xp}$ sont placés dans la colonne de gauche
- les profils de f, x1,..., xp sont placés dans la colonne centrale
- x = f x1...xp est placée dans la colonne de droite
- x est coché

Heuristiques :

a) Eventuellement décomposer la recherche des xi et de f en commençant par considérer l'énoncé $\varepsilon_{x1} \wedge \ldots \wedge \varepsilon_{xp} \supset \varepsilon_x[fx1\ldots xp/x]$ puis vérifier à postériori le non blocage : $\varepsilon_{x1} \wedge \ldots \wedge \varepsilon_{xp} \supset$ *pré* fx1...xp

b) C'est la position de ε dans un processus général d'explicitation qui guide la recherche des intermédiaires. Un cas particulier important est celui où ε_x est un invariant caractérisant une suite. On utilise alors une *stratégie d'unification* en comparant

$$\varepsilon_x \quad \text{et} \quad \varepsilon_x \ [\bar{x}/x]$$

et en cherchant π tel que $\varepsilon_x \longleftrightarrow \varepsilon_x \ [\bar{x}/x] \wedge \pi$

c) Une autre possibilité pour rechercher les xi consiste à utiliser les constructeurs f de la sorte de x , dans le cas où ceux-ci sont en nombre limité. Pour chaque constructeur f on essaye de séparer $\varepsilon_x[fx1\ldots xp/x]$ en p énoncés nouveaux $\varepsilon_{x1} \ldots \varepsilon_{xp}$.

Remarque :

Cette stratégie est un cas particulier de la stratégie qui introduit conjointement les nouveaux symboles xi et f .

b) Affaiblissement.

◂ SEA ▸ stratégie d' explicitation par affaiblissement

donnée x caractérisé par ε_x

résultat u symbole o-aire de suite caractérisé par ε_u (invariant)

a symbole o-aire de profil BOOL* caractérisé par π_a (condition d'arrêt)

k et n termes de profil ENTIER

tq $\vdash_S$ (n1 ≡ inf(n, μ i. π_a) ∧ ε_u [n1/i] ⊃ $\varepsilon[u_{n1}/x]$) ∧

∃k *tq* π_a [k/i] ≡ *vrai*

Présentation :

- ε_u et π_a sont placés dans la colonne de gauche
- les profils de u et a sont placés dans la colonne centrale
- x = *der* u_i *pour* i *dans* k → n *iqa* a_i est placé dans la colonne centrale
- x est coché.

Heuristiques :

a) Chercher une décomposition de ε_x en deux formules ε_1 et ε_2 telles que :

- $\varepsilon_x = \varepsilon_1 \wedge \varepsilon_2 \ [x/u_i \ , \ n1/i]$
- $\varepsilon_1[d/x]$ soit facile à vérifier pour une certaine valeur d
- on ait des chances raisonnables de trouver une récurrence conservant $\varepsilon_1 \ [u_i/x]$

(C'est ce qui a été fait en (22) de l'exemple 3 ch 2.1).

Un cas particulièrement important est celui où ε contient une variable entière libre n. On peut alors choisir :

$$\varepsilon_1 = \varepsilon\ [k/n] \quad ; \quad \varepsilon_2 = (k \equiv n)$$

C'est le cas de nombreux problèmes portant sur des suites (par exemple problèmes de tris, problème du drapeau tricolore etc...)

b) Chercher une induction sur les données en utilisant un constructeur (cf chapitre 2.1 (21) a) exemple 4 ou (2) exemple 3). On peut aussi dans ce cas chercher une définition itérative portant sur une suite (§ 3.2) et obtenir les résultats cherchés en transformant par la règle (RCSDI) (§ 5.2.1.) le résultat de la stratégie d'induction sur une suite complète ⊀ SISC ⊁ (§ 3.3.2).

La validité de ces stratégies résulte des règles (RIDS) et (RIDI). On peut l'exprimer de la manière suivante :

Tout énoncé MEDEE $\mathcal{E}'$ obtenu à partir d'un énoncé initial $\mathcal{E}$ par application des stratégies ⊀ SES ⊁ et ⊀ SEA ⊁ vérifie

$$\varepsilon_1 \wedge \ldots \wedge \varepsilon_k \wedge \text{SEMST } [\![\mathcal{E}']\!] \supset \varepsilon$$

où les $\varepsilon_1, \ldots, \varepsilon_k$ sont les énoncés résultats des stratégies précédentes et qui n'ont pas encore été explicités.

Comme cas particulier on obtient, lorsque tous les énoncés intermédiaires ont été explicités.

$$\vdash\!\!\!\!\!\!{}_{S}\ \text{SEMST } [\![\mathcal{E}']\!] \supset \varepsilon$$

3.- DEFINITIONS ITERATIVES DE PLUS HAUT NIVEAU.

3.1.- Insuffisance des définitions itératives précédentes

Dans une définition itérative MEDEE.

x = *der* u_i *pour* i *dans* k ⟶ n *jqa* a_i

la suite u est définie, par l'intermédiaire du module ⁎u, à partir d'autres suites. Par exemple, la résolution du problème de la planification des charges d'une entreprise (ch 1.1 §3.2 a)) necessite de parcourir le fichier des commandes. Le

résultat est donc caractérisé par une définition itérative "portant" sur ce fichier. Cette remarque nous invite à considérer le couple formé de l'expression itérative "u_i *pour* i *dans* k ⟶ n" et du module ⁎u comme étant une *fonction* associant au fichier donné d la suite u.

Avec ce point de vue la forme précédente d'une définition itérative présente deux inconvénients :

a) D'une part, elle ne met pas en évidence la suite d à partir de laquelle est définie u. Plus précisément, cette suite d n'est mentionnée que dans le cas particulier où il s'agit de la suite des entiers commençant à k et se terminant à n1 = inf(n, μi.a_i) ; c'est-à-dire lorsque u_i est définie par un calcul. Ce qui est le cas, en général, pour les suites "mathématiques" (arithmétiques, géométriques, etc...)

b) D'autre part, elle recouvre deux façons logiquement différentes de définir le *dernier terme* de u :

i) La définition de l'arrêt a porte sur la suite u, ce qui est naturel pour de nombreux problèmes numériques dans lesquels on calcule une valeur avec une certaine précision.

Exemple 4 :

Calcul de $\sqrt{a}$ à eps près

résultat rac2 rac2 = *der* r_i *pour* i *dans* 1 ⟶ *jqa* fin$_i$ *donnée* eps
⁎r

r = (r̄ + a/r̄)/2
$ r = 1
fin = |r - r̄| < eps
$ fin = *faux*

□

ii) La définition de l'arrêt porte sur la suite "donnée". C'est le cas de la plupart des problèmes portant sur des fichiers

Exemple 5 :

calcul du maximum d'une liste de nombres positifs se terminant par 0 .

résultat maximum maximum = *der* m_i *pour* i *dans* 1 ⟶ *iqa* $arret_i$. *donnée* d
*m
m = max($\overline{m}$, d_i) arret = (d_i = 0) ∮ m = 0 ∮ arret = *faux*

Cette forme d'itération est peu agréable puisqu'elle nécessite d'indicer d et non pas m . De plus le programmeur doit penser, lors de l'explicitation, à la fois, à la "gestion du parcours" de la suite donnée (définition du premier élément, définition de l'arrêt) et à la définition du "traitement" (définition de m). Or on pourrait souhaiter traiter ces deux sous-problèmes de manière indépendante. □

En conclusion, si la forme précédente des définitions itératives en MEDEE permet d'exprimer tous les énoncés récurrents, elle est en contre-partie de trop "bas niveau" lorsque la suite u est définie à partir d'une suite donnée d . Nous nous attachons, dans ce paragraphe, à introduire de nouvelles formes de définitions itératives mieux adaptées à une explicitation mettant en évidence une suite donnée.

3.2.- Définitions itératives de haut niveau [PAI, 77] , [BEL, 78 c]

3.2.1.- Introduction

Il résulte des constatations précédentes que, lorsqu'une définition itérative porte sur une suite donnée d , nous ne souhaitons pas décrire la gestion du parcours de d dans le module itéré *u .

Introduisons donc une nouvelle forme d'expression itérative :

u_i *pour* i *dans* d

où i parcourt le domaine de la suite d .

Ceci induit une nouvelle forme de définition itérative.

Exemple 6 :

Le calcul du maximum d'une suite, envisagé dans l'exemple 5, s'écrit :

résultat maximum maximum = *der* m_i *pour* i *dans* d *donnée* d
*m
m = max ($\overline{m}$, d) ∮ m = 0

ce qui a le même sens que l'énoncé de l'exemple 5 □

Cependant, dans de nombreux problèmes, il s'agit de ne parcourir qu'une sous-suite d'une suite donnée. Parfois la borne supérieure de cette sous-suite est caractérisée par un prédicat (exploration de sous-fichiers, recherche d'un élément dans un fichier etc...).

Dans d'autres cas cette borne dépend du terme général de la suite calculée. Nous nous attachons donc maintenant à généraliser la forme des expressions itératives pour rendre compte simplement de ce genre de problèmes ; la généralisation de la forme des définitions itératives s'en déduira immédiatement.

3.2.2.- Expression itérative sur une suite complète.

a) syntaxe

Soit d une suite, une expression itérative sur d est de la forme

u_i *pour* i *dans* d

d peut être appelée la *suite de départ* de u

b) sémantique

SEMST $[\![u_i \ \textit{pour}\ i \ \textit{dans}\ d]\!]$ = DOMC (u, d) ∧ MOD(u, i)

où

- MOD(u, i) est comme au paragraphe 2.2.2. d) : c'est une formule caractérisant le terme général et le premier terme de la suite u . Pour qu'elle ait un sens il suffit que l'opération d'indiçage ind (§ 2.2.1 b)) indice les occurrences de d dans le module *u . Ceci résulte immédiatement de l'extension de la relation *dépend de* obtenue en posant, dans le cas de l'expression considérée en a), d *dépend de* *u .

- DOMC(u, d) caractérise le domaine de u :

$$DOMC(u, d) = \bigwedge_{y \in ident(*u) \setminus \{d\}} (bi(y) = bi(d)-1 \ , \ bs(y) = bs(d))$$

Notons qu'il est possible que d soit une suite infinie, il en est alors de même de u . Cependant, dans de nombreux cas, d aura une représentation physique (fichier par exemple) et sera donc fini.

De cette sémantique nous déduisons immédiatement le sens d'une définition itérative dont l'expression itérative est de la forme a).

Exemple 7 :

La définition itérative de l'exemple 6 a pour sémantique :

$$x = m_{bs(m)} \wedge bi(m) = bi(d) \wedge bs(m) = bs(d) \wedge$$
$$(bi(m) < i \leqslant bs(m) \supset m_i = \max(m_{i-1}, d_i)) \wedge$$
$$m_0 = 0$$

3.2.3.- Expression itérative sur une sous-suite connexe.

Dans de nombreux problèmes on ne doit parcourir qu'une sous-suite d'une suite donnée (impression des salaires des employés dont le nom commence par e, f ou g ; recherche du premier enregistrement d'un fichier séquentiel vérifiant une propriété ; plus généralement parcours de sous-fichier etc...). Il s'agit donc de définir précisément la notion de sous-suite connexe d'une suite donnée.

a) Sous-suites connexes

Une sous-suite connexe d'une suite ℓ est, intuitivement, une tranche de ℓ dont la tête et la queue sont caractérisées par des prédicats t et q (t et q sont des formules à une seule variable libre notée i dans la suite). On distingue les sous-suites connexes dont le premier élément vérifiant q fait partie (on parle de sous-suite fermée) de celles dont il est exclu (sous-suites ouvertes), d'où les définitions suivantes :

Définition 6 :

On appelle sous-suite connexe fermée, de tête t et de queue q , de la suite ℓ , la suite notée ℓ *de* t *à* q *inclus* définie par :

$$\ell \ \textit{de}\ t\ \textit{à}\ q\ \textit{inclus} = tr(\ell, deb, fin) \wedge deb = \mu i.(bi(\ell) \leqslant i) \wedge (i = bs(\ell) \vee t)$$
$$fin = \mu i.(deb \leqslant i \wedge (i = bs(\ell) \vee q)$$

On peut en donner une définition récursive

Théorème 5 :

Avec les notations précédentes,

ℓ *de* t *à* q *inclus* = *si* ℓ = < >
alors < >
sinon si t[bi(b)/i]
alors si q [bi(ℓ)/i]

alors < prem(ℓ) >
sinon < prem(ℓ) > *·(reste(ℓ) *de vrai à* q *inclus*)
sinon reste(ℓ) *de* t *à* q *inclus*

où < u > représente < > ~ u □

On définit de manière analogue les sous-suites ouvertes qui sont d'une grande utilité pratique : par exemple c'est le cas des fichiers se terminant par un "drapeau". C'est aussi le cas des problèmes de traitements de chaines de caractères pour lesquels, en général, la fin d'un mot est déterminée par le fait que le caractère suivant est soit un blanc, soit un caractère de séparation.

Définition 7 :

On appelle sous-suite connexe ouverte, de tête t et de queue q , de la suite ℓ , la suite notée ℓ *de* t *à* q *exclu* définie par :

ℓ *de* t *à* q *exclu* ≡ tr(ℓ, deb, fin) ∧ deb ≡ μi.(bi(ℓ) ≤ i) ∧ (i ≡ bs(ℓ) ∨ t)
∧ fin ≡ μi.(deb ≤ i+1) ∧ (i+1 ≡ bs(ℓ)
∨ q [i+1/i])

Théorème 6 :

ℓ *de* t *à* q *exclu* ≡ *si* ℓ ≡ < >
alors < >
sinon si t[bi(ℓ) /i]
alors si q [bi(ℓ)/i]
alors < >
sinon < prem(ℓ) > * rest(ℓ) *de vrai à* q *exclu*
sinon rest(ℓ) *de* t *à* q *exclu* □

Théorème 7 :

|ℓ *de* t *à* q {*exclu* / *inclus*} | < +∞ ⟷ (|ℓ| < + ∞) ∨
∃ i *tq* (i ≥ bi(ℓ) ∧ t))
∧ (∃j *tq* j ≥ i ∧ q [j/i]) □

Ces trois théorèmes résultent immédiatement de la définition récursive de μ et de la définition de | | .

Enfin lorsque t est de la forme i = a , on admettra l'abréviation suivante :

ℓ *de* a *à* q {*exclu* / *inclus*} .

b) Expression itérative sur une sous-suite

Avec les définitions précédentes, et la généralisation de la forme des expressions itératives introduites au paragraphe 3.2.2, nous pouvons désormais utiliser des expressions itératives de la forme :

u_i *pour* i *dans* d *de* t *à* q {*exclu* / *inclus*}

Leur sémantique résulte immédiatement des définitions 6 et 7 et du paragraphe 3.2.2 b).

Notons cependant qu'une application stricte de cette sémantique nécessite que, dans le module *u, le terme courant de la sous-suite se présente sous la forme :

d *de* t *à* q {*exclu* / *inclus*}

et il faut que l'opération d'indiçage ind puisse s'appliquer à cette expression.

Cette écriture étant trop lourde nous conviendrons désormais de noter encore d le terme courant de cette sous-suite .

Caractérisons directement ces sémantiques :

Théorème 8 :

SEMST ⟦ u_i *pour* i *dans* d *de* t *à* q *exclu* ⟧ = DOMPE(u, t, q, i) ∧ MOD(u, i)

avec

DOMPE(u, t, q, i) = $\bigwedge_{y \in *ident}$ (bi(y) ≡ deb-1 ∧ bs(y) ≡ fin)
∧ fin ≡ μi.(deb ≤ i + 1 ∧ (i + 1 ≡ bs(d) ∨ q[i+1/i]))
∧ deb = μi.((bi(d) ≤ i) ∧(i ≡ bs(d) ∧ t)) □

et MOD(u, i) est comme au paragraphe 2.2.2 d)

Démonstration :

Elle résulte immédiatement du lemme suivant :

Lemme :

Avec les notations précédentes :

bi(d *de* t *à* q *exclu*) = 1 ∧ bs(d *de* t *à* q *exclu*) = fin-deb+1 ∧

∀ i(deb ⩽ i ⩽ fin ⊃ appl(d, i) = appl(d *de* t *à* q *exclu* , i-deb+1))

résulte immédiatement de la définition d'une tranche

cqfd.

Théorème 9 :

SEMST ⟦u_i *pour* i *dans* d *de* t *à* q *inclus*⟧ = DOMPI(u, t, q, i) ∧ MOD(u, i)

avec

$$DOMPI(u,t,q,i) = \bigwedge_{y \in ident(*u)} (bi(y) = deb\text{-}1 \wedge bs(y) = fin)$$
$$\wedge\ fin = \mu i.(deb \leqslant i \wedge (i = bs(d) \vee q))$$
$$\wedge\ deb = \mu i.(bi(d) \leqslant i \wedge (i = bs(d) \vee t))$$

et MOD(u, i) est comme au paragraphe 2.2.2 d) □

La démonstration est analogue à celle du théorème 8.

3.2.4.- Forme générale d'une définition itérative.

En guise de résumé, indiquons la forme générale que peut prendre une définition itérative :

$$x = \textit{der}\ \ u_i\ \textit{pour}\ \ i\ \textit{dans}\ d\ \textit{de}\ t\ \textit{à}\ q\ \left\{\begin{matrix}\textit{inclus}\\ \textit{exclu}\end{matrix}\right\}\ \textit{jqa}\ \ a_i$$

Ceci résulte du paragraphe précédent et de l'utilisation des opérations *der* *jqa* .

Indiquons, par exemple, sa sémantique dans le cas "*exclu* " :

SEMST ⟦ x = *der* u_i *pour* i *dans* d *de* t *à* q *exclu jqa* a_i⟧ =

DOMPA(u, t, q, d, i, a) ∧ MOD(u, i) ∧ $x = u_{bs(u)}$

avec

$$DOMPA(u,t,q,d,i,a) = \bigwedge_{y \in ident(*u)} (bi(y) = deb\text{-}1 \wedge bs(y) = inf(fin, \mu i.a_i))$$
$$\wedge\ fin = \mu i.\ (deb \leqslant i+1 \wedge (i+1 = bs(d) \vee q[i+1/i]))$$
$$\wedge\ deb = \mu i.\ ((bi(d) \leqslant i) \wedge (i = bs(d) \vee t))$$
$$MOD(u,i) = (bi(u) < i \leqslant bs(u) \supset ind(SEMST\ ⟦**u⟧\ , i, *u)) \wedge$$
$$ind(SEMST\ ⟦\$\ *u⟧, bi(u)\ , *u)$$

3.2.5.- Quelques généralisations.

a) On utilise fréquemment les définitions itératives sur des sous-suit afin de parcourir un fichier""par morceaux". Il est alors nécessaire de retenir à une étape donnée, la valeur du dernier indice atteint, cet indice étant le point de départ de l'itération suivante. Dans ce but nous introduisons une nouvelle opération sur les suites :

dernier(u) = << der(u), bs(u) >>

on écrit alors :

<< x, p >> = *dernier* u_i *pour* i *dans* d *de* t *à* q $\left\{\begin{matrix}\textit{exclu}\\ \textit{inclus}\end{matrix}\right\}$ *jqa* a_i

dont la sémantique est immédiate.

(Notons qu'il est équivalent d'écrire

<< x, p >> = *der*<<u_i, i >> *pour* i....)

b) Un autre problème très fréquent est la recherche d'un premier élément dans une suite (ou un fichier) qui vérifie une certaine propriété. Il est donc utile d'introduire une syntaxe particulière pour ce genre de recherche.

p = *premplace* i *dans* d *depuis* t *telle que* q

Elle est équivalente à :

p = *der* i *pour* i *dans* d *de* t *à* q

en particulier *i est le module vide

3.3.- Construction de définitions itératives de haut niveau.

Comme au paragraphe 2.3, nous nous attachons ici à définir des règles et des stratégies de construction de définitions itératives de l'une des formes introduites ci-dessus.

3.3.1.- Règles de construction.

Elles résultent immédiatement de la sémantique associée. Chacune de ces règles assure que la conjonction de la définition itérative et des sous-énoncés obtenus vérifie l'énoncé initial. Nous nous contentons de les énoncer.

a) Définition itérative sur une suite complète.

(RIDISC) Règle d'introduction d'une définition itérative sur une suite complète

Soit x caractérisé par ε_x dans le système S . La définition itérative

x = *der* u_i *pour* i *dans* d

est *correcte* vis à vis de ε , de l'invariant ε_u caractérisant u si

$\vdash_S \; n_1 = bs(d) \wedge \varepsilon_u [nl/i \; , \; u_{nl}/u] \supset \varepsilon_x [u_{nl}/x]$

b) Définition itérative sur une sous suite ouverte

(RIDISO) Règle d'introduction d'une définition itérative sur une sous-suite ouverte

La définition itérative

x = *der* u_i *pour* i *dans* d *de* t *à* q *exclu* *jqa* a_i

est *correcte* vis à vis de ε_x , ε_u,ε_d,et π_a si

$\vdash_S \; nl = \mu i.((deb \leqslant i + 1) \wedge (i+1 = bs(d) \vee q[i+1/i] \vee \pi_a) \wedge \varepsilon_d$

$deb = \mu i.((bi(d) \leqslant i) \wedge (i = bs(d) \vee t)) \wedge \varepsilon_u[u_{nl}/u \; , \; nl/i] \supset \varepsilon_x[u_{nl}/x]$

c) Définition itérative sur une sous-suite fermée

(RIDISS) Règle d'introduction d'une définition itérative sur une sous-suite fermée

La définition itérative

x = *der* u_i *pour* i *dans* d *de* t *à* q *inclu* *jqa* a_i

est *correcte* vis à vis de ε_x , ε_u et π_a si

$\vdash_S \; nl = \mu i. ((deb \leqslant i) \wedge (i = bs(d) \vee q \vee \pi_a)) \wedge$

$deb = \mu i. ((bi(d) \leqslant i) \wedge (i = bs(d) \vee t) \wedge \varepsilon_u[u_{nl}/u, \; nl/i] \supset \varepsilon_x[u_{nl}/x]$

d) Dernier élément tel que

(RIDIPT) Règle d'introduction d'une définition itérative du premier tel que.

Soit p caractérisé par ε_p . La définition itérative

p = *premplace* i *dans* d *depuis* t *telle que* q

est *correcte* vis à vis de ε_p si

SEMST $[\![$ p = *premplace* i *dans* d *depuis* t *telle que* q $]\!] \supset \varepsilon_p$

Cette règle se réduit à la définition de "vérifie" puisqu'il n'y a pas d' intermédiaires.

3.3.2.- Stratégies d'induction sur des suites données.

Nous nous contentons de décrire une stratégie fournissant une itération sur une suite complète. Les autres cas sont très analogues.

ǂ SISC ǂ stratégie d'induction sur une suite complète

donnée x caractérisé par ε_x

résultat u symbole o-aire de profil suite caractérisé par un invariant ε_u

d symbole o-aire de profil suite caractérisé par l'invariant ε_d

tq $\vdash_S \varepsilon_d \wedge nl = bs(d) \wedge \varepsilon_u \; [u_{nl}/u, \; nl/i] \supset \varepsilon[u_{nl}/x]$

Présentation :

- ε_u , ε_d sont placés dans la colonne de gauche
- les profils de u et d sont placés dans la colonne centrale
- x = *der* u_i *pour* i *dans* d est placé dans la colonne de droite
- x est coché

Heuristiques :

- choix d'une suite d finie
  - souvent c'est la suite donnée
  - parfois c'est une suite intermédiaire associée à la suite donnée (§ 5.2).
- détermination de ε_u en essayant de le prendre égal à ε_x sur tr(d, bi(d), i)

C'est encore un cas d'affaiblissement

Exemple 8 :

Plutôt que d'obtenir l'énoncé de la figure 1 (§ 2.1.1) , l'utilisation de la stratégie ⊲SISC⊳ dans le cas du problème des sous-suites croissantes de longueur maximum nous aurait conduit à :

résultat ℓ
ℓ = *der* ℓmax$_i$ *pour* i *dans* α
donnée α

l'invariant caractérisant ℓmax$_i$ étant la restriction de l'énoncé à tr [α, 1, i] □

4.- QUELQUES REGLES DE TRANSFORMATION DE DEFINITIONS ITERATIVES.

4.1.- Passage d'une itération de haut niveau à une itération générale.

Dans ce paragraphe nous introduisons des règles permettant de traduire les expressions itératives de haut niveau décrites au paragraphe 3.2. en l'itération de base du langage MEDEE (§ 2.1). On peut les utiliser pour traduire un énoncé MEDEE de haut niveau en un programme d'un langage de programmation algorithmique style ALGOL.

On peut remarquer que ces règles sont des équivalences : les sémantiques statiques des deux membres sont équivalentes.

4.1.1.- Expression itérative sur une suite complète.

(RTSC) Règle de traduction d'une expression itérative sur une suite complète

$$\frac{u_i \ \textit{pour} \ i \ \textit{dans} \ \alpha \ ; \ *u}{u_i \ \textit{pour} \ i \ \textit{dans} \ bi(d) \rightarrow bs(d) \ ; \ *u\,[d_i/d]}$$

Cette notation signifie que l'on peut remplacer dans un énoncé MEDEE, l'expression itérative du haut par celle du bas à condition de substituer d_i à d dans le module *u

Cette règle se justifie immédiatement avec la sémantique statique si on remarque que le rôle de l'opération ind est précisément d'effectuer la substitution de d_i à d.

4.1.2.- Expression itérative sur une sous-suite ouverte.

(RTSO) Règle de traduction d'une expression itérative sur une sous-suite ouverte

$$\frac{u_i \ \textit{pour} \ i \ \textit{dans} \ d \ \textit{de} \ deb \ \textit{à} \ q \ \textit{exclu} \ \textit{jqa} \ a_i \ ; \ *u}{u_i \ \textit{pour} \ i \ \textit{dans} \ deb \rightarrow bs(d) \ \textit{jqa} \ \ q[i+1/i] \vee a_i \ ; \ *u\,[d_i/d \ , \ a_i/a]}$$

Nous nous restreignons ici au cas où t est de la forme i = deb Le cas général nécessitant d'introduire en plus la définition itérative de deb caractérisé comme le premier indice tel que t soit vrai .

4.1.3.- Expression itérative sur une sous suite fermée.

(RTST) Règle de traduction d'une expression itérative sur une sous-suite fermée

u_i *pour* i *dans* d *de* deb *à* q *inclu jqa* a_i ; *u

u_i *pour* i *dans* deb → bs(d) *jqa* q v a_i ; *u[di/d, a_i/a]

4.2.- Opérations sur les définitions itératives.

Nous présentons ici quelques règles de transformation qui permettent d'améliorer, en un certain sens que nous précisons dans chaque cas, des énoncés MEDEE. Cet ensemble de règles demande à être complété dans le sens développé dans [BEL, 79] .

4.2.1.- Fusion d'itérations.

La démarche d'explicitation que nous développons, reprenant les idées de la méthode déductive, consiste à partir de la spécification des résultats et à expliciter indépendemment des sous problèmes disjoints. Il arrive alors fréquemment que deux résultats soient les derniers termes de deux suites définies sur la même suite donnée. Il faut regrouper ces deux définitions de manière à ne parcourir qu'une seule fois la suite considérée (c'est une question d'efficacité lorsque cette suite est un intervalle d'entiers ou est représentée en mémoire centrale, c'est presque une nécessité physique si cette suite est en mémoire sécondaire séquentielle, c'est obligatoire sur cartes).

Dans ce but on introduit une nouvelle règle de transformation d'énoncés MEDEE permettant de fusionner deux définitions portant sur la même suite :

(RFDI) Règle de fusion de définitions itératives

x = *dernier* u(i) *pour* i *dans* d ; *u

y = *dernier* v(i) *pour* i *dans* d ; *v

<< x, y >> = *dernier* << u(i), v(i) >> *pour* i *dans* d ; *u U *v

Dans cette définition, on suppose que les modules *u et *v sont fusionnés .

Cette règle se justifie immédiatement par la sémantique statique (§ 3.2.2. b)) .

On peut la généraliser au cas où x et y sont définis en parcourant deux sous-suites d'une suite d dont l'intersection n'est pas vide. On peut alors fusionner ce qui concerne cette intersection en utilisant la règle suivante (§ 4.2.2) sur la concaténation d'itérations.

4.2.2.- Concaténation de définitions itératives.

Il est fréquent en programmation de gestion qu'une suite résultat soit définie par les parcours successifs de deux suites données sur lesquelles s'effectuent les mêmes traitements (problèmes de mise à jour de fichiers par exemple). Il est alors intéressant de regrouper ces deux définitions en parcourant la concaténée des deux suites données.

Inversement, comme nous l'avons mentionné au paragraphe précédent, il peut être intéressant, par exemple pour des problèmes d'efficacité, de décomposer une définition itérative en deux.

(RCDI) Règle de concaténation de définitions itératives

x = *der* u_i *pour* i *dans* d1

y = *der* v_i *pour* i *dans* d2

$\forall u = a \in *u$ *et*

*v = *u [x/a, v1/u1, ..., vk/uk]

$v^1 \ldots v^k$ identificateurs de *v

$u^1 \ldots u^k$ ....de *u

y = *der* u_i *pour* i *dans* d1 * d2

Autrement dit cette transformation n'est licite que si les modules *v et *u sont identiques à une permutation sur les identificateurs près et au remplacement de la valeur initiale de u par x

$$\text{SEMST}[\![\ y = \underline{der}\ u_i\ \underline{pour}\ i\ \underline{dans}\ d1 * d2\ ]\!] =$$

$$y = u_{di} \wedge di = bs(d1 * d2) \wedge$$

$$(bi(di * d2) \leqslant i \leqslant di \supset ind(\text{SEMST}[\![** u]\!], i, *u)) \wedge$$

$$ind(\text{SEMST}[\![\$ * u]\!], bi(d1 * d2)) \wedge$$

$$\bigwedge_{u^i \in ident(*u) \setminus \{d1\}} (bs(u^i) = di \wedge bi(u^i) = bi(d1 * d2) - 1)$$

En utilisant la définition de d1 * d2

$$= y = u_{di} \wedge di = bs(d1) + (bs(d2) - bi(d2)) \wedge$$

$$(bi(d1) \leqslant i \leqslant bs(d1) \supset ind(\text{SEMST}[\![** u]\!], i, *u)) \wedge$$

$$ind(\text{SEMST}[\![\$ * u]\!], bi(d1)) \wedge$$

$$\bigwedge_{u^i \in ident(*u)} (bs(u^i) = di \wedge bi(u^i) = bi(d1 * d2))$$

Les noms des identificateurs sont des variables libres qui peuvent être changées par substitution:

$$= y \equiv v_{di} \wedge di \equiv bs(d1) + (bs(d2) - bi(d2)) \wedge$$

$$(bs(d1) \leq i \leq di \supset ind(\text{SEMST}[\![** v]\!], i, * v)) \wedge$$

$$\bigwedge_{v^i \in ident(*v)} (bs(v^i) \equiv di \wedge bi(v^i) \equiv bs(d1)) \wedge$$

$$v_{bs(d1)} \equiv x \wedge x \equiv u_{bs(d1)}$$

$$(bi(d1) \leq i \leq bs(d1) \supset ind(\text{SEMST}[\![** u]\!], i, * u)) \wedge$$

$$ind(\text{SEMST}[\![\$ * u]\!], bi(d1), *u) \wedge$$

$$\bigwedge_{u^i \in ident(*u)} (bs(u^i) \equiv bs(d1) \wedge bi(u^i) \equiv bi(d1))$$

Enfin on remarque que les 5 premières conjonctions sont invariables "par translation des indices" : on peut remplacer partout bs(d1) par bi(d2) et bs(d1) + (bs(d2) - bi(d2)) par bs(d2) car
(bs(d1) + bs(d2) -bi(d2)) - bs(d1) = bs(d2) - bi(d2)

d'où le résultat.

5 - INTRODUCTION DE SUITES INTERMEDIAIRES

5.1 - Le problème

La mise en évidence d'une suite de départ dans une définition itérative (§ 3) est un guide méthodologique important puisqu'elle simplifie la définition du résultat. Mais, si dans certains énoncés comme celui du problème des sous-suites ascendantes de longueur maximum, la suite de départ de la définition itérative est une suite donnée, il n'en est pas de même dans le cas général et la suite de départ doit à son tour être définie.

Nous avons donné (ch 1.2 § 7.2.2) un certain nombre d'opérations permettant de construire des suites : adjonction en queue (~), concaténation (*), tranche etc... L'application repétée de ces opérations permet de construire une suite par approximations successives : nous nous intéressons donc aux suites qui sont définies par une expression itérative.

$$v = u_i \quad \underline{pour}\ i\ \underline{dans}\ \ldots$$

avec les définitions des paragraphes précédents, v est une suite qui est le dernier terme d'une suite de suites u.

Dans certains cas v devra être "construite effectivement", c'est-à-dire qu'au niveau de la programmation il faudra en donner une représentation physique en mémoire. C'est par exemple le cas où v est obtenue par un tri d'une suite donnée d. En revanche, dans d'autres cas, on peut *composer* la définition du résultat r et celle de v en une unique définition itérative portant sur d et dont le résultat est r :

$$r = \underline{der}\ x_j\ \underline{pour}\ i\ \underline{dans}\ v$$
$$v = u_j\ \underline{pour}\ j\ \underline{dans}\ d$$

se composent en

$$r = \underline{der}\ y_j\ \underline{pour}\ j\ \underline{dans}\ d.$$

Examinons tout d'abord un exemple.

Exemple 9 Traitements de chaines de caractères

La plupart des problèmes de traitement de chaines de caractères (construction de l'index d'un texte, recherche des contextes d'un mot, problème des télégrammes (exemple f)ch1.1.) peuvent être résolus en utilisant des algorithmes voisins de ceux de l'analyse syntaxique. On parle encore d'algorithmes fondés

sur le concept d'automate d'état fini : la suite (de caractères) donnée est traitée élément par élément. On a donc une méthode de résolution orientée par les données (on utilise une définition itérative dont la suite de départ est la suite donnée) qui n'est pas toujours simple à mettre en oeuvre lorsqu'il y a un conflit de structure entre les données et les résultats.

Par opposition à ce mode de résolution, une autre possibilité consiste à être guidé par la structure du résultat en introduisant une suite intermédiaire permettant de résoudre les conflits de structure, comme le montre l'exemple du "comptage" des mots d'un texte (ch 1.1. § 1.1. p)):

Rappelons que l'on cherche le nombre de mots de longueur 1, 2, 3,..., 20 d'un texte pour lequel 20 est la longueur maximale d'un mot.

a) Spécification du problème (figure 4)

lexique	profils	spécification formelle
nbmot est la suite à 20 éléments formée du nombre de mots de longueur 1,2...20 du texte donné texte est le texte donné mot est un prédicat indiquant qu'une chaîne est un mot d'une autre chaine	nbmot : ENTIER* m : CAR* texte : CAR* mot : *fonct*(CAR*, CAR*) BOOL	*résultat* nbmot *tq* dom (nbmot) = [1..20] ∧ $nbmot_i$ = card {m *tq* mot (m, texte) ∧ | m | = i} mot *tq* mot (m,α) = ∃ i,j *tq* m = tr (α,i, j)∧ (i=bi (α)∨ α_i = '␣') ∧ (j=bs (α)∨ α_j = '␣') *donnée* texte

Figure 4 : s-description du problème du "comptage de mots"

Notons que le prédicat "mot" est très voisin de $\sqsubset$ introduit pour le problème des sous-suites croissantes maximales.

b) Explicitation du problème

But : Expliciter l'énoncé de la figure 4

Choix (1) Une première solution consiste à chercher une induction sur la donnée texte ; donc à utiliser la stratégie ⟨ S I S C ⟩ (§ 3.3.2.) sur la suite "texte". Les recherches de l'invariant et de la formule de récurrence sont assez difficiles car la structure de "texte" correspond à des caractères alors que la suite nbmot est définie à partir de mots. Une résolution possible est donnée dans [QUE,79].

Choix (2) La remarque précédente incite à raisonner non plus sur une suite de caractères mais sur la suite des mots de "texte"

But 2 1 En supposant que l'on dispose de cette suite, notée motexte, la mise en oeuvre de la stratégie ⟨ S I S C ⟩ fournit la première partie de la figure 5, car :

$$nl = bs(motexte) \wedge \varepsilon_{motexte} \wedge \varepsilon_{nb}[nl/j, nb_{nl}/nb] \supset$$

$$dom(nb_{nl}) = [1..20] \wedge nb_{nl,i} = card\{m \ tq \ mot(m,texte) \wedge |m| = i\}$$

But : Expliciter la définition de la suite nb

Choix (2 1 1) Appliquer une stratégie d'analyse récurrente.

But 2 1 1' Trouver une relation de récurrence entre nb et $\overline{nb}$.

On chercher à vérifier

$$\varepsilon_{nb} \text{ avec } \varepsilon_{nb}[j-1/j] .$$

Le domaine restant [1..20] on se restreint à la deuxième partie de la conjonction :

$$nb_{j-1,i} = card\{m \in motexte([1..j-1]) \wedge |m| = i\}$$

et $$nb_{j,i} = card\{m \in motexte([1..j]) \wedge |m| = i\}$$

d'où $$nb_{j,i} = si \ |motexte_j| = i \ alors \ nb_{j-1,i}+1 \ sinon \ nb_{j,i}$$

ou encore $nb_j = chval(nb_{j-1}, i, nb_{j-1,i}+1)$ avec $i = |motexte_j|$

(chval est défini au ch. 1.2 § 7.2.1).

- - Pour expliciter i on utilise la définition de | |

But 2 1 1" : Pour définir la valeur initiale nb_o on utilise $\varepsilon_{nb}[o/j]$:

$$nb_{o,i} = card(\emptyset) = 0$$

on suppose disposer de la suite [o] dont tous les éléments sont o.

On obtient finalement la figure 5.

énoncé implicite	profil	énoncé explicite
nbmot : ε_{nbmot} comme sur la figure 4 colonne de droite $\varepsilon_{motexte}$=imag (motexte) ≡ { m *tq* mot (m,texte)} ∧ bi (motexte) ≡ 1 ε_{nb_j} = dom (nb_j)≡[1..20]∧ $nb_{j,i}$ ≡ card { m ∈ motexte ([1..j]) *tq* | m | ≡ i }	-nbmot : ENTIER* - nb : ENTIER** - motexte : CAR** - texte : CAR*	*résultat* nbmot nbmot= *der* nb_j *pour* j *dans* motexte *donnée* texte
* nb		
(2 1 1 1) i = | motexte | ε_s = s_k ≡ | tr (motexte, 1, k) | (2 1 1 2)	- i : ENTIER - s : ENTIER*	nb = chval ($\overline{nb}$,i,$\overline{nb}_{i+1}$) i = *der* s_k *pour* k *dans* motexte $nb = tr ([o] , 1, 20)
* s		
		s = $\overline{s}$ + 1 $ s = 0

Figure 5 : explicitation incomplète du problème

But 2 2 Expliciter la définition de motexte caractérisé par :

$\varepsilon_{motexte}$ = imag (motexte) ≡ { m *tq* mot (m,texte) } ∧ bi (motexte)≡ 1

Chercher une induction directe sur texte nous ramène à l'inconvénient mentionné en (1) : on souhaite "construire" motexte mot par mot.

Choix (2 2) Stratégie d'affaiblissement ⦃ S E A ⦄

Choix (2 2 1) On vérifie que l'invariant ε_{mt} (cf figure 6) convient.

But 2 2 1 : Expliciter cet invariant.

Choix (2 2 2) : Faire une analyse par cas.

But 2 2 2' : Définir mt en fonction de $\overline{mt}$.

On compare ε_{mt} et ε_{mt} [ℓ-1/ℓ]

∃ $f_{\ell-1}$ *tq* imag ($mt_{\ell-1}$) ≡ { m *tq* mot (m, tr (texte,1, $f_{\ell-1}$)) } ∧ card (imag ($mt_{\ell-1}$)) ≡ ℓ-1

∃ f_ℓ *tq* imag (mt_ℓ) ≡ { m *tq* mot (m, tr (texte,1, f_ℓ)) } ∧ card (imag (mt_ℓ))≡ ℓ

Intuitivement : on passe de $mt_{\ell-1}$ à mt_ℓ en ajoutant un mot de texte.
Par le calcul : mt_ℓ ≡ $mt_{\ell-1}$ ∼ mc_ℓ ∧ mot (mc_ℓ,tr (texte, $f_{\ell-1}$ +1, f_ℓ)).
Il faut alors expliciter f_ℓ : on le choisit comme le dernier indice de mc dans texte et pour le définir onintroduit d_ℓ premier indice de mc_ℓ dans texte.

énoncé implicite	profils	énoncé explicite
$\varepsilon_{motexte}$: comme figure 5 colonne gauche ε_{mt} = (∃ f ∈ dom (texte) *tq* imag (mt_ℓ)≡{ m *tq* mot (m,tr (texte, 1, f))) ∧ card (imag (mt_ℓ)) ≡ ℓ ∧ $fint_\ell$ ≡ (der (mt_ℓ) (1)='*')	mt : CAR***	motexte = *der* mt_ℓ *pour* ℓ *dans* 1 → *jqa* $fint_\ell$
* nt		

d_ℓ,f_ℓ *tq* mc_ℓ = tr (texte, d_ℓ, f_ℓ)
∧ mot (mc_ℓ,texte)
	mc : CAR*
	d : ENTIER
	f : ENTIER
		mt = $\overline{mt}$ ∼ mc
		mc = tr (texte, d, f)
		d = *premplace* i *dans* texte *depuis* f+1 *telleque* $texte_\ell$ ≠'␣'
		f = *premplace* i *dans* texte *depuis* d *telleque* $texte_{\ell+1}$='␣'
		$ mt = < >
		$ f = 0
		fint = ($texte_f$ = '*')
		$ fint = *faux*

figure 6 : Complèment de la figure 5

On effectue ainsi une restriction d'énoncé :

$mt_\ell = mt_{\ell-1} \frown mc_\ell \wedge mc_\ell = tr\ (texte, d_\ell, f_\ell) \wedge mot\ (mc, texte) \wedge card\ (mt_\ell) = \ell$

But 2 2 2 1 : explicitation de f_ℓ et d_ℓ.

Ils sont caractérisés par :

$$\cdot\ f_{\ell-1} < d_\ell \leq f_\ell \wedge \forall k\ (f_{\ell-1} < k < d_\ell \supset texte_k = '\sqcup') \wedge (texte_{d\ell} = '\sqcup' \vee d_\ell = bi\ (texte))$$

$$\cdot\ \forall k\ (d_\ell \leq k \leq f_\ell \supset texte_k \neq '\sqcup') \wedge (texte_{f\ell+1} = '\sqcup' \vee f_\ell + 1 = bs\ (texte)).$$

On reconnait ici (⌇ eurêka ⌇) les spécifications de la construction *premplace*.

La règle (RIDIPT) est immédiatement vérifiée, ce qui justifie l'explicitation de la figure 6.

But 2 2 2" ε_{mt} [o/ℓ] conduit à initialiser mt à la liste vide.
De même on initialise f à o.

En regroupant les définitions des modules principaux des figures 5 et 6 et en ajoutant les différents modules secondaires, on obtient une version explicite complète de l'énoncé.

On peut s'arrêter là et il n'est pas difficile de passer de cet énoncé à un programme Algol par exemple. Mais on peut également remarquer que l'introduction de la suite motexte complète, comme intermédiaire, est inutile et conduira à un programme inefficace. En fait, plutôt que de *construire* motexte, il nous importe seulement de pouvoir la *parcourir* , c'est-à-dire d'expliciter le passage de $\overline{mt}$ à mt ou encore la *définition* de mc. On peut donc remplacer dans la définition de nbmot l'itération

der nb_j *pour* j *dans* motexte

par l'itération décrivant le parcours de motexte à partir de texte qui se trouve sur la figure 6. Ceci revient donc à *composer* les définitions de nbmot et de motexte au sens de la composition des fonctions, si on considère qu'une définition itérative est une fonction associant à une suite de départ une certaine valeur.

On obtient alors :

nbmot = *der* nb_j *pour* j *dans* [*der* mt_ℓ *pour* ℓ *dans* 1↣ *jqa* fint]

Mais on constate, ici, que cette expression se simplifie en *une seule itération* à condition de réunir les modules itérés *nb et *mt (dans lesquels on supprime les définitions de mt).

On obtient ainsi l'énoncé de la figure 7 (seule la 3ème colonne est représentée) dans lequel on a explicité mc = tr (texte , d, f) en une itération sur texte. On peut alors constater sur ce nouvel énoncé qu'il est possible à nouveau de composer la définition de i et celle de mc. D'autre part, on note que les définitions itératives de f et mc portent sur la même suite de départ qui est une sous-suite de texte de premier indice d ; on peut donc appliquer la règle (R F D I) de fusions de définitions itératives. Finalement on obtient l'énoncé de la figure 8 .

énoncé explicite
résultat nbmot nbmot = *der* nb_ℓ *pour* ℓ *dans* 1↣ *jqa* $fint_\ell$ *donnée* texte
*nb ∪ *mt

nb = chval ($\overline{nb}$, i, $\overline{nb}_{f+1}$)
i = *der* s_k *pour* k *dans* mc
mc = *der* mcp_p *pour* p *dans* texte *de* d *à* f
d = *premplace* ℓ *dans* texte *depuis* $\overline{f}$ + 1 *telleque* $texte_p$ ≠'⊔'
f = *premplace* ℓ *dans* texte *depuis* d *telleque* $texte_{\ell+1}$ ='⊔'
$ f = 0
$ nb = tr ([o] , 1, 20)
fint = (textef = '*')
$ fint = *faux*

* s
s = $\overline{s}$ + 1 $s = 0
* mcp
mcp = $\overline{mcp}$ ~ texte $mcp = < >

Figure 7 : nouvel énoncé du problème du comptage des mots.

énoncé explicite

résultat nbmot
nbmot = *der* nb_ℓ *pour* ℓ *dans* 1→ *iqa* $fint_\ell$
donnée texte

* nb

nb = chval ($\overline{nb}$, i, $\overline{nb}_i$ + 1)
<<i,f >>= *der* s_p *pour* p *dans* texte *de* d *à* $texte_{\ell+1}$ = '␣'*exclu*
d = *premplace* ℓ *dans* texte *depuis* $\overline{f}$ + 1 *telleque* $texte_\ell \neq$ '␣'
$ f = 0
$ nb = tr ([o] ,1, 20)
fint = ($texte_f$ = '*')
$ fint = *faux*

* s

s = $\overline{s}$ + 1
$ s = 0

Figure 8 : énoncé final du problème du comptage des mots

Remarquons que la démarche suivie, si elle est systématique, est assez lourde : nous avons défini complètement les suites intermédiaires (motexte, mc) et ceci indépendamment des traitements s'y rapportant. Et, une fois seulement les différentes définitions posées, nous avons effectué des simplifications par composition et fusion. C'est la rançon de la démarche guidée par les résultats.

Enfin pour que la suite de transformations précédente ait un sens, il faut que l'on puisse caractériser les définitions itératives pour lesquelles l'opération est applicable.

5.2 Mise en oeuvre de la stratégie d'introduction de suites intermédiaires

Il résulte de l'exemple précédent qu'il est nécessaire de définir une opération de composition d'itération (§ 5.2.1). Nous nous attacherons alors à préciser cette stratégie fondée sur l'introduction de suites comme nouveaux intermédiaires.

5.2.1 Composition de définitions itératives

a) Composition simple

Si une suite ℓ est définie à l'aide d'une expression itérative, on peut exprimer toute définition itérative dont ℓ est la liste de départ à l'aide de cette expression itérative.

(R C S D I) Règle de Composition Simple de Définitions Itératives

x = *der* u_i *pour* i *dans* ℓ ; * u
ℓ = v_j *pour* j *dans* k → n *iqa* a_j ; * v

---

x = *der* u_j *pour* j *dans* k → n *iqa* a_j ;
* u [v/ℓ] ∪ * v

Intuitivement, le module itéré de la nouvelle définition caractérise l'élément courant v_j de la suite v (c'est le rôle de *v) et, à partir de cet élément, il spécifie la valeur courante u_j.

Justification de la règle

Soit

S_1 = SEMST [x = *der* u_i *pour* i *dans* ℓ] =

$x = u_{bs(\ell)} \wedge (\bigwedge_{y \in ident(*u)\setminus\{\ell\}} (bi(y) = bi(\ell)-1 \wedge bs(y) = bs(\ell))) \wedge$

$(bi(\ell)<i\leq bs(\ell) \supset ind(SEMST[**u],i,*u)) \wedge ind(SEMST[\$*u], bi(\ell)-1, *u)$

S_2 = SEMST [ℓ = v_j *pour* j *dans* k → n *iqa* a_j] =

$\ell = v \wedge \bigwedge_{z \in ident(*v)} (bi(z) = k-1 \wedge bs(z) = inf(n, \mu j.aj)) \wedge$

$(k \leq j \leq bs(v) \supset ind(SEMST[** v],j,*v)) \wedge ind(SEMST[\$*v], k-1, *v)$

$S_1 \wedge S_2 = x \equiv u_{bs(v)} \wedge (\bigwedge_{y \in ident(*u)} (b_i(y) = k-1 \wedge bs(y) = inf(n, \mu_j.a_j))) \wedge$

$(k \leq i \leq bs(v) \supset ind(SEMST[\![**u]\!], i, *u)[v/\ell]) \wedge ind(SEMST[\![\$*u]\!], k-1, *u)$

$\wedge (\bigwedge_{z \in ident(*v)} (bi(z) = k-1 \wedge bs(z) = inf(n, \mu_j.a_j))) \wedge$

$k \leq j \leq bs(v) \supset ind(SEMST[\![**v]\!], j, *v) \wedge ind(SEMST[\![\$*v]\!], k-1, *v)$

Soit en réunissant * v et * u [v/ℓ]

$S_1 \wedge S_2 = x \equiv u_{bs(u)} \wedge (\bigwedge_{y \in ident(*u\ [v/\ell]\ \cup\ *v)} bi(y) \equiv k-1 \wedge bs(y) = inf(n, \mu_j.a_j)) \wedge$

$(k \leq i \leq bs(u) \supset ind(SEMST[\![**u\ [v/\ell]\ \cup\ **v]\!], i, *u\ [v/\ell]\ \cup\ *v)) \wedge$

$ind(SEMST[\![\$*u\ [v/\ell]\ \cup\ \$*v]\!], k-1, *u\ [v/\ell]\ \cup\ *v))$

ce qui termine la justification.

Remarque 1

Cette règle est encore valable si ℓ est défini par :

ℓ = *der* v_j *pour* j *dans* k → n *jqa* a_j si, dans le module * v, v est défini par v = $\overline{v} \sim m$ (cf exemple 9).

Remarque 2

Cette règle se prolonge, de manière évidente, au cas où ℓ est défini par une expression itérative sur une suite complète ou partielle, mutadis mutandis.

b) Composition multiple

Dans la règle précédente la suite ℓ est identique à la suite v qui est elle-même définie "élément par élément" sur un sous-intervalle de [k..n] . Il est utile d'étudier le cas général où v est défini "par morceaux". Intuitivement v_i se déduit de v_{i-1} par concaténation d'une certaine sous-suite m_i : v est une suite de suites. L'exemple 10 (§ 5.3) donne une illustration de cette situation. Dans ce cas, il faut généraliser la règle précédente (ou plus précisément son analogue résultantde la remarque 1) en "plaçant" dans le module * v le module * u "aux endroits" où sont définis les éléments dont la concaténation définira v. Commençons donc par définir une opération *composition de modules* dont la réunion utilisée dans la règle (R C D S I) est un cas particulier.

b1) Composition de modules

C'est une opération qui "place" un module *u dépendant du terme général d'une suite ℓ , au niveau de chaque terme qui contribue à définir dans le module itéré *v de la définition itérative de ℓ . Cette opération remplace également u par un nouvel identificateur u^j dans la jième occurence du module *u placée dans *v, si ce nombre d'occurrences est supérieur à un.

Commençons par définir une fonction mes (mesure) associant à une suite w définie dans un module *v le nombre d'éléments de w obtenus par des adjonctions ou des concaténations, mais non pas par l'intermédiaire d'une définitic itérative :

w1 = w2 ~ a ∈ *v ⇒ mes (w1,*v) = mes (w2) +1

w1 = w2 * w3 ∈ *v ⇒ mes (w1,*v) = mes (w2) + mes (w3)

w1 = < > ⇒ mes (w1,*v) = 0

w1 = *der* q_i ..∈*v ⇒ mes (w1,*v) = 1

Par exemple si

w = w1 ~ a

w1 = w2 * w3

w2 = w4 ~ b

w3 = *der* ....

w4 = < >

alors mes (w) = 2

mes (*v) est le nombre d'indices j qu'il faut introduire pour remplacer u par u^j lors des déplacements de *u dans *v.

Nous pouvons maintenant définir l'opération COMP de composition de deux modules.

Définition 8

Soit x= *der* u_i *pour* i *dans* ℓ.

ℓ= *der* v_j *pour* j *dans* k → p *jqa* a_j

On suppose que $1 \leq j \leq n$ et $n \geq 2$.

i) v = <b> ∈ *v alors

COMP (*v,*u, j, n) =

$$\begin{cases} \text{si } j = 1 & \text{alors } * \tilde{v} \cup * u^1 [\, z^1/z,\ b/\ell \,] \\ \text{si } j = n & \text{alors } * \tilde{v} \cup * u^n [\, z^{n-1}/\bar{z},\ b/\ell \,] \\ \text{si } 1 < j < n & \text{alors } * \tilde{v} \cup * u^j [\, z^{j-1}/\bar{z},\ z^j/z,\ b/\ell \,] \end{cases}$$

où $* u^j$ est déduit de $* u$ en remplaçant chaque identificateur y local à $*u$ non récurrent par y^j ; z désigne un identificateur récurrent de $*u^j$ et $* \tilde{v} = * v \setminus \{ v = < b > \}$

ii) $v = \underline{der}\ vl_{j1}\ \underline{pour}\ j1\ \underline{dans}\ k1 \to p1\ \underline{jqa}\ al_{j1} \in * v$

et $\$ * vl = \{ \$ vl = < > \} \cup \$ * \tilde{v}1 \in * vl$ (i.e. que le premier terme de vl est $< >$).

COMP (*v, *u, j, n) =

$$\begin{cases} \text{si } 1 \leqslant j < n \text{ alors } u^1 = \underline{der}\ uu^j_{j1}\ \underline{pour}\ j1\ \underline{dans}\ k1 \to n1\ \underline{jqa}\ al_{j1} \\ \text{si } j = n \quad \text{alors } u = \underline{der}\ uu^n_{j1}\ \underline{pour}\ j1\ \underline{dans}\ k1 \to n1\ \underline{jqa}\ cl_{j1} \end{cases}$$

Avec

$$\begin{cases} \text{si } j = 1 & \text{alors } * uu^1 = \$ * \tilde{v}1 \cup \{ \overline{uu}^1 = \bar{u} \} \cup UN\ (uu^1, 1) \\ \text{si } 1 < j \leqslant n & \text{alors } * uu^j = \$ * \tilde{v}1 \cup \{ \overline{uu}^j = uu^{j-1} \} \cup UN\ (uu^1, j) \end{cases}$$

et

$$UN\ (uu,j) = \begin{cases} ** \tilde{v}1 \cup * u^j [\, uu/u,\ \overline{uu}/\bar{u},\ b/\ell \,] & \text{si } vl = \overline{vl} * < b > \in * vl \\ COMP\ (** \tilde{v}1,\ u^j [\, uu/u,\ \overline{uu}/\bar{u}, m/\ell \,], 1, mes(m, *v)) & \text{si } vl = \overline{vl} * m \in * vl \end{cases}$$

où $* \tilde{v}1 = * vl \setminus \{ v = \underline{der}\ vl_{j1} \ldots\ldots \}$

et $* u^j$ est défini comme en i)

iii) $v = vl * v2 \in * v$ où vl et v2 ne sont pas réduits à la suite vide

COMP (*v,* u,j,n) = COMP (*vl,*u,j,j+mes(vl,*u))∪

COMP (*v2,*u,j+mes (vl,*v)+1,n)\ {v = vl*v2 }

où *vl (resp *v2) est le sous module de * v définissant vl (resp. v2). □

Intuitivement cette opération place un module *u à chaque endroit de *v où est défini un élément intervenant dans la constitution de ℓ. Il faut effectuer des changements d'identificateurs pour éviter des définitions multiples de u qui doivent être cohérents de manière à prendre en compte la définition récurrente de u.

Nous pouvons maintenant décrire la règle de composition dans le cas général.

2) Composition multiple de deux itérations

(R C M D I) Règle de composition multiple ce définitions itératives

$x = \underline{der}\ u_i\ \underline{pour}\ i\ \underline{dans}\ \ell \quad ; * u$

$\ell = \underline{der}\ v_j\ \underline{pour}\ j\ \underline{dans}\ k \to p\ \underline{jqa}\ a_j \ ; * v \qquad v = \bar{v} * vl \in * v$

$\$ v = < > \in * v$

$x = \underline{der}\ u_j\ \underline{pour}\ j\ \underline{dans}\ k \to p\ \underline{jqa}\ a_j$;

COMP ($* \tilde{v}$, * u, 1, mes (vl, * v))

La justification de cette règle s'effectue par récurrence sur mes (vl,*v) et repose sur un raisonnement par cas faisart intervenir la définition 6.

Remarque 3

Cette règle est assez compliquée à décrire car elle envisage toutes les définitions possibles de la suite intermédiaire v. Plutôt que de l'introduire pour améliorer l'énoncé final, on peut imaginer la mise en oeuvre de type *coroutines* [CLI,73] : une coroutine fournissant les éléments ℓ_i, l'autre les utilisant pour définir u_i. Si on admet des zones communes non réduites à un élément on peut résoudre des problèmes plus généraux que ceux traités ici : il n'est pas nécessaire alors d'avoir une correspondance bijective entre les éléments ℓ_i "engendrés" et les éléments u_i.

5.2.2 Application à une stratègie d'introduction de suites intermédiaires

On peut utiliser les règles précédentes pour décrire une stratègie d'explicitation consistant à introduire des suites de départ intermédiaires. Cette stratègie est la composée d'un certain nombre de stratègies élémentaires et elle peut être considérée comme un cas particulier de la stratègie ⊰ S E A ⊱ d'explicitation par affaiblissement,

⊰ S E A S I S D ⊱ stratègie explicitation par approximation successive en introduisant une nouvelle suite de départ

donnée x caractérisée par ε

résultat u symbole o-aire de suite, * u' module explicite associé

a symbole o-aire de suite booléenne, k et p entiers

$$\underline{tq} \quad \text{il existe } \ell\ ,\ \varepsilon_\ell\ ,\ \varepsilon_u,\ v,\ \varepsilon_v,\ *u,\ *v,\ \Pi_a$$

$$\underline{tq} \quad \not\vdash S\ I\ C\ S \not\vdash \begin{bmatrix} x & \ell & u & a \\ \varepsilon_x\ ; & \varepsilon_\ell\ , & \varepsilon_u\ , & \Pi_a \end{bmatrix}$$

$$\underline{et} \quad \not\vdash S\ E\ A \not\vdash \begin{bmatrix} \ell & v & , c & , k & , p \\ \varepsilon_\ell\ ; & \varepsilon_v \end{bmatrix}$$

$$\underline{et} \quad \not\vdash S\ E\ X\ P \not\vdash [\ \varepsilon_u\ ;\ *u\ ]$$

$$\underline{et} \quad \not\vdash S\ E\ X\ P \not\vdash [\ \varepsilon_v\ ;\ *v\ ]$$

$$\underline{et} \quad *u' =$$

$$\begin{cases} \text{si } \begin{bmatrix} v = \overline{v} \frown m \\ \$\ v = <> \end{bmatrix} \in *v \text{ alors } *u\ [\ m/\ell\ ] \cup\ *v/\ \{\ v = \overline{v} \frown m,\ \$\ v = <>\ \} \\ \text{si } \begin{bmatrix} v = \overline{v} * v1 \\ \$\ v = <> \end{bmatrix} \in *v \text{ alors } \mathrm{COMP}\ (*\hat{v},\ *u,\ 1,\ \mathrm{mes}\ (v1,\ *v)) \end{cases}$$

Dans cette définition $\not\vdash ST \not\vdash\ [\ d\ ;\ b\ ]$ désigne une stratégie $\not\vdash ST \not\vdash$ de donnée d et de résultat b. $\not\vdash S\ E\ X\ P \not\vdash$ est la stratégie générale d'explicitation esquissée au chapitre 2.4.

On peut définir de manière analogue une stratégie dont le résultat serait une définition itérative sur une suite complète ou partielle d. Il s'agit essentiellement de remplacer dans la définition ci-dessus

$$\not\vdash S\ E\ A \not\vdash \begin{bmatrix} \ell\ , & v,\ c,\ k,\ p \\ \varepsilon_\ell\ ; & \varepsilon_v \end{bmatrix}$$

par

$$\not\vdash S\ I\ C\ S \not\vdash \begin{bmatrix} \ell & d & v \\ \varepsilon_\ell ; & \varepsilon_d\ , & \varepsilon_v \end{bmatrix}$$

Heuristique

Le gros problème est d'inventer la liste intermédiaire ℓ. On la choisit de telle manière qu'il existe une bijection entre elle et u : leurs domaines de définition seront alors les mêmes.

Exemple 10 Structures de sortie superposées

Cet exemple est dû à ABRIAL et a été repris par PAIR [PAI,77] .

Soit f un fichier dont chaque enregistrement est formé :

- d'une clé
- d'une longueur ℓg ($\geqslant 0$)
- d'une suite de ℓg valeurs numériques val.

On demande d'éditer pour chaque article :

. pour chaque valeur v une ligne (clé, v)

. une ligne finale (clé, somme des valeurs).

On exige de plus que l'édition débute par un saut de page et qu'il y ait un saut de page toutes les 4 lignes.

a) Spécification formelle du problème

lexique	profil	spécification formelle
On cherche une suite de lignes res définie comme ci-dessus LIGNE : *de base* mepgcorr: prédicat indiquant que la mise en page est correcte convention de notation : $clé_k$ note clé (f_k) val_k note val (f_k) etc...	res : LIGNE* f : << clé : CHAINE, ℓ_g : ENTIER, val : ENTIER*>>* mepgcorr: LIGNE*→ BOOL	*résultat* res *tq* [∀ k ∈ dom (f)] (∃ i *tq* imp (res_i)=($clé_k$,Σ$val_{k,j}$ *pour* j *dans* 1 → ℓg_k))∧ (∀ j ∈ [1..ℓg_k] , ∃ i *tq* imp (res_i) = ($clé_k$, $val_{k,j}$))]∧ mepgcorr(res) mepgcorr(r) : b *tq* b = (mep (r_1) = st) ∧ (∀ j ⩾ 4, mep (r_{j-3}) ≠ st ∧ mep (r_{j-2})≠st∧mep (r_{j-1}) ≠ st ∧ ⊃ mep (rj) = st) *donnée* f
une ligne est un couple formé de la valeur imprimée et d'une instruction de mise en page : st pour saut sq pour séquence		S.I. LIGNE = << imp :<<clé : CHAINE, vℓ : ENTIER >>, mep : { st } ∪ { sq } >>

Figure 9 : Impressions avec sauts

Notons que cette spécification est incomplète :
- on ne précise pas qu'il s'agit là des seuls res_i
- on ne précise pas que l'ordre des indices i de r est lié à celui des indices k de f_k et j de val (f_k).

b) Exploitation

① Choix de la stratègie ⊰ S E A S I S D ⊱ :

On veut dissocier, comme le suggère la forme de l'énoncé, le problème des sauts de celui de la définition des lignes "significatives".

⊰ EURÊKA ⊱ : introduction de la suite intermédiaire des couples (clé, v) où v est soit l'une des valeurs d'un article, soit la somme des valeurs d'un article

⑪ ⊰ S I C S ⊱

on introduit ℓ , ε_ℓ , u, ε_u tels que :

ε_ℓ : ∀ k ∈ dans (f)

$(\exists\ i$ *tq* $\ell_i = \ll clé_k, \Sigma\ val_{k,j}$ *pour* j *dans* $1 \to \ell g_k \gg \wedge$

$(\forall\ j \in [\,1..\ \ell g_k\,],\ \exists\ i$ *tq* $\ell i = (clé_k, val_{k,j}))$

ε_u : imp (u_p) = tr $(\ell, 1, p) \wedge$ mepgcorr(u_p)

ce qui permet de commencer l'explicitation (cf figure 10).

⑪① explicitation de u : analyse récurrente

⑪①① p ⩾ 1

ε_u [p-1/p] : imp (u_{p-1}) = tr (ℓ,1,p-1) ∧ mepgcorr(u_{p-1})

ε_u : imp (u_p) = tr $(\ell, 1, p) \wedge$ mepgcorr(u_p)

unification: $u_p = u_{p-1} \frown ec \wedge$ imp (ec) = ℓ_p ∧

mep (ec) = *si* $(p \geqslant 4) \wedge$ mep$(u_{p-1,p-1}) \neq st \wedge$

mep $(u_{p-1,p-2}) = st \wedge$ mep $(u_{p-1,p-3}) \neq st$

alors st

sinon sq

⑪①①① On veut définir la condition précédente de manière récurrente :

cd = ∀ i, $1 \leqslant i \leqslant 3$ mep $(u_{p-1,p-i}) \neq st$.

ⓐ L'application d'une stratègie d'affaiblissement classique portant sur 3 conduit à :

ε_{pr} : ∀ i, $1 \leqslant i \leqslant pr$ mep $(u_{p-1,p-i}) \neq st$

car on a bien

$pr = 3 \wedge \varepsilon_{pr}\ [\ 3/pr\ ] \supset$ cd = *vrai*

C'est-à-dire que l'on est ainsi conduit à calculer cd par récurrence. Ceci nécessite de disposer des mep $(u_{p-1,p-i})$.

ⓑ Autre idée : utilisation de la relation d'ordre sur les entiers

cd = (sup { i ⩾ 1, mep $(u_{p-1,p-i}) \neq st$ } = 3 }

affaiblissement :

pr = sup { i ⩾ 1, mep $(u_{p-1,p-i}) \neq st$ }

D'où une définition récurrente de pr portant sur p :

pr_p = *si* mep $(u_{p,p}) \neq st$ *alors* $pr_{p-1}+1$ *sinon* 0

⑪①② p = 0, u_0 = < >

pr_0 = 3 pour que mep (u_1) = st.

⑫ Explicitation de ℓ

⊰ S I C S ⊱ portant sur la suite f donnée

on introduit v, ε_v tels que

ε_v : vℓ (tr $(v_k$, bs (v_k)-ℓg_k, bs $(v_k))) = val_k \frown (\Sigma\ val_{k,j}$ *pour* j *dans* $1 \to \ell g_k)$

∧ ∀ e ∈ tr $(v_k$, bs (v_k)-ℓg_k, bs $(v_k))$, clé (e) = $clé_k$

121 explicitation de v : analyse récurrente

⑫①① k ⩾ 1

unification :

$v_k = v_{k-1} * \ell m \wedge \ell m = \ell m1 * \ll (clé_k, \Sigma\ val_{k,j}$ *pour* j *dans* $1 \to \ell g_k) \gg$

$\wedge\ (\forall\ j,\ 1 \leqslant j \leqslant \ell g_k \quad \ell m1_j = (clé_k, val_{k,j}))$

Il faut encore expliciter $\Sigma\ val_{k,j}$ ce qui est immédiat.

On peut aussi immédiatement fusionner la définition itérative de $\Sigma\ val_{k,j}$ et de $\ell m1$ (Règle (R F D I) § 4.2) ce qui donne la partie ** v

du module * v. L'explicitation étant déjà faite on obtient le module * ℓmp

(1212) k = 0, on initialise v à < >

d'où la partie $ * v du module * v

ce qui termine la première explicitation représentée pour la figure 10.

énoncé implicite	profil	énoncé explicite
u *tq* ϵ_u cf (11) ℓ *tq* ϵ_ℓ cf (11)	res : LIGNE* u : LIGNE** ℓ : <<clé:CHAINE, vℓ:ENTIER>>* v : <<clé:CHAINE, vℓ:ENTIER >>**	*résultat* res res = *der* u_p *pour* p *dans* ℓ ℓ = *der* v_k *pour* k *dans* f *donnée* f
* u		

ec *tq* imp (ec) ≡ ℓp ∧
(mep(ec) ≡ *si* p ⩾ 4 ∧ cd
alors st *sinon* sq)
pr *tq* pr = sup { i ⩾ 0
tq mep($u_{p,p-i}$) ≠ st }
	ec : LIGNE
	pr : ENTIER
		u = ū ~ ec
		<<ec,pr>>= *si* $\overline{pr}$ = 3 *alors*
		<<<< ℓ ,st>>,0 >>
		sinon<<<< ℓ , sq << , $\overline{pr}$ +1 >>
		$ u = < >
		$ pr = 3

* v		
ℓm *tq* ϵ_v ℓm1 *tq* ϵ_v som = Σ $val_{k,j}$ *pour* j *dans* 1 ⟶ ℓg$_k$	ℓm ; <<clé:CHAINE, vℓ:ENTIER >>* ℓm1 : <<clé:CHAINE, vℓ:ENTIER >>* ms : <<clé:CHAINE, vℓ:ENTIER >> som:ENTIER	v = v̄ * ℓm ℓm = ℓm1 * < ms > ms =<< clé, som >> <<ℓm1,som>>= *der*<<ℓmp$_i$, somp$_i$>> *pour* i *dans* 1 → ℓg $ v = < >

* ℓmp		
	ℓmp : << clé:CHAINE, vℓ:ENTIER>>* somp: ENTIER clé : CHAINE val : ENTIER	ℓmp =$\overline{ℓmp}$ ~ << clé, val >> somp = $\overline{somp}$ + val $ somp = 0 $ ℓmp = < >

Figure 10 : première explicitation

(13) Pour terminer l'application de ⫞ S E A S I S D ⫟ on compose la définition de x et celle de ℓ . Nous sommes ici dans le cas où les éléments de ℓ sont définis à plusieurs endroits de * v :

- dans la définition de ms
- dans la définition de ℓmp dans le module * ℓmp.

En appliquant la règle de composition multiple (R C M D I) on obtient l'énoncé de la figure 11 (dans laquelle on ne place que l'énoncé explicite).

résultat res res = *der* u_k *pour* k *dans* f *donnée* f
COMP (* $\tilde{v}$, u, 1, 2)

$u = u^1 \sim ec^2$
<< ec^2,pr^2>>= *si* pr^1 = 3 *alors*<<<<<< clé,som >>,st >> , o >>
sinon<<<<<< clé,som >>,sq >>,pr^1+1 >>
u^1, som = *der* << uu^1_i, $somp_i$ >> *pour* i *dans* 1 → ℓg
$ u = < >
$ pr = 3

* uu^1
$uu^1 = \overline{uu}^1 \sim ec^1$ << ec^1, pr^1 >> = *si* $\overline{pr}^1$ = 3 *alors* <<<<<< clé,val >>,st >>,o>> *sinon* <<<<<< clé,val >>,sq >>,$\overline{pr}^1$+1>>

$ $uu^1 = \bar{u}$
$ $pr^1 = \overline{pr}$
somp = $\overline{somp}$ + val
$ somp = 0

figure 11 : énoncé après composition

5.2.3 Application à l'algorithme de gestion

La stratégie précédente permet de résoudre deux des principaux problèmes algorithmiques qui apparaissent en informatique de gestion dans la manipulation des fichiers séquentiels :

- Les "problèmes de rupture" : c'est le cas où une suite à plusieurs indices est représentée par une suite à un seul indice, certains critères permettant le parenthèsage. Le problème des télégrammes (ch.1.1 § 1.1 exemple f) ou celui des comptagesde mots (exemple 8) en sont des exemples.
- Les problèmes de mise à jour de fichiers dont les classiques problèmes d'interclassement sont des cas particuliers.

Enfin, dans le cas général les problèmes d'algorithmique de gestion sont un mélange de ces deux types, c'est le cas du problème de l'état d'avancement des livraisons sur commandes en cours (ch. 1.4 exemple 2).

Précisons ce qu'est un problème de rupture :

Définition 9 : une suite finie f *possède la propriété de rupture* (brièvement f est à rupture), s'il existe une fonction booléene car définie sur le co-domaine de f et une suite id d'indices tels que:

$$imag(id) \subset dom(f) \text{ et } prem(id) = 1 \text{ et } der(id) = bs(f)$$

et $\forall\ i \in [\ bi(id)..\ bs(id)-1\ ]\ ,\ \forall\ j1,j2 \in [\ id(i)..id(i+1)\ [$
$$car(f_{j1}) = car(f_{j2})$$

et $\forall\ i \in [\ bi(id)..\ bs(id)-2\ ]\ ,$
$\forall\ j_1 \in [id(i)..id(i+1)[\ ,\ \forall\ j_2 \in [\ id(i+1)..id(i+2)\ ]\ ,$
$$car(f_{j1}) \neq car(f_{j2})$$

On a alors la propriété suivante

Propriété

A toute suite f ∈ V* possèdant la propriété de rupture on peut associer une suite fℓ ∈ V** telle que

$$imag(f) = \{\ f\ell_{ij} \text{ tq } j \in [\ bi(fi)..bs(f\ell_i)]\ ;\ i \in [bi(f\ell)..bs(f\ell)]\ \}$$

et $\forall\ i \in [bi(f\ell)..bs(f\ell)]\ ,\ \forall\ j1, j2 \in [\ bi(f\ell_i) .. bs(f\ell_i)\ ]$
$$car(f\ell_{i,j1}) = car(f\ell_{i,j2})$$

Plus précisément on définit fℓ de la façon suivante (avec les notations de la définition 7) :

. bi (fℓ) = bi (id) ; bs (fℓ) = bs (id) -1
. $\forall i \in [\ bi(f\ell)..\ bs(f\ell)\ ]$, $bi(f\ell_i) = 1$, $bs(f\ell_i) = id(i+1)-id(i)$
. $\forall i \in [\ bi(f\ell)..\ bs(f\ell)\ ]\ ,\ \forall\ j \in [\ bi(f\ell_i)..bs(f\ell_i)]$ $\quad f\ell_{i,j}=f_{id(i)+j}$

Mais lorsque l'on veut définir fℓ de manière constructive à partir de f, on ne dispose pas de id. On vérifie alors simplement que l'énoncé de la figure 12 est correct.

Ainsi, l'explicitation de n'importe quel énoncé portant sur f se réduit à l'explicitation du même énoncé sur fℓ , qui est en général beaucoup plus simple. L'obtention de l'énoncé complet s'obtient alors par composition avec l'énoncé de la figure 12.

fin est un prédicat caractérisant la fin de la suite f	fℓ = *der* v_i *pour* i *dans* 1 → *iga* a_i
	* v

	v = $\overline{v}$ ~ sf
	<< sf,d >>= *der* << sf_j,j >> *pour* j *dans* f *de* $\overline{d}$ *à* ar_j *exclu*
	a = fin (f_d)
	ca = car ($f_{\overline{d}}$)
	$ v = < >
	$ a = fin (prem (f))
	$ d = bi (f)

	* u
	sf = $\overline{sf}$ ~ f ar = (car (f) ≠ ca) $ sf = < >

Figure 12 : définition d'une suite à deux niveaux associés à une suite à rupture f.

Le problème de "l'état d'avancement des livraisons sur commandes en cours" (ch 1.4 exemple 4)peut être explicité de manière systématique en introduisant des fichiers "logiques" intermédiaires (cf [FIN,79]).

6 - COMPARAISON AVEC D'AUTRES TRAVAUX

L'étude précédente des énoncés récurrents n'est que très partielle et nécessiterait d'être développée, notamment dans la recherche de stratégies d'explicitation.

Les différents aspects des énoncés récurrents que nous avons abordés ici : description, construction, introduction de suites intermédiaires et transformations sont l'objet de nombreuses recherches actuelles, aussi nous contentons nous maintenant d'un bref survol partiel de quelques travaux sur ce domaine.

6.1 Langages à assignation unique

Comme nous l'avons déjà indiqué au paragraphe 1, ARSAC propose dans [ARS,77] un langage statique (ie n'utilisant pas le concept de variable informatique) fondé sur les mêmes notions que celles développées ici. La différence essentielle entre ce langage et MEDEE réside en l'utilisation explicite, dans le premier, d'indices de suites ce qui complique un peu les notations. Et il ne nous apparaît pas évident que cette utilisation explicite d'indices facilite les preuves si, comme nous le suggèrent les règles (R I D S) et (R I D I) la correction est vérifiée de façon modulaire. Pour concrétiser notre propos, indiquons comment peut s'écrire dans ce langage l'énoncé de la figure 1 (problème de la fraction irréductible) en reprenant un exemple de [ARS,77] :

```
        x, y ← FRAIRR (a, b) a, b ∈ ℕ   a > 0    b > 0
        x, y ← a/k, b/k
         k   ← EUCL (a, b)   a, b ∈ ℕ   a > 0    b > 0   *calcul du pgcd *
c(i) , d(i)  ← R (c (i-1), d (i-1))                      *définition itérative *
      t (i)  ← d (i) = 0                                 *condition d'arrêt *
c(o) , d(o)  ← a, b
```

6.2 Construction d'algorithmes récurrents

Ce problème est étudié par [ARS,77] et par DIJKSTRA [DIJ,76] mais aussi par SCHOLL [SCH,79] , qui parle alors de traitement séquentiel.

Dans ces deux derniers cas cependant la construction se situe tout de suite à un niveau dynamique ce qui complique les choses. Il n'en reste pas moins qu'on trouve dans ces diverses approches des techniques d'affaiblissement, d'analyse par cas etc... et qu'il serait très profitable de reprendre chaque exemple traité dans ces différents cadres pour essayer de préciser les heuristiques et les conditions de mise en oeuvre des stratègies.

6.3 Introduction de structures intermédiaires

De nombreux problèmes de traitements de chaines de caractères ou de manipulations de fichiers peuvent être résolus en utilisant des techniques d'analyse syntaxique : la suite en entrée est décrite par une grammaire et donc, à partir d'un analyseur de cette grammaire [AHO ,72] , il est possible de construire un algorithme de traitement quelconque, par exemple en introduisant des attributs [LOR,74] . Il s'agit ici d'un traitement "orienté par la structure des données".

Plusieurs autres approches sont fondées sur l'introduction d'intermédiaires (cf § 5) : la structure "physique" des données et des résultats étant décrite, on "invente" (ou on "abstrait") une structure logique associée. Le problème se ramène aux trois sous-problèmes suivants (⫞ S E A S I S D ⫟ en est un cas particulier) :

(1) passage de la structure physique des données à leur structure logique

(2) passage de la structure logique des données à la structure logique des résultats

(3) passage de la structure logique des résultats à leur structure physique.

C'est une démarche proposée par WARNIER [WAR,75] , JACKSON [JAK,75], [HUG,79] ou SCHOLL [SCH,79] .

Nous proposons ici d'aller un peu plus loin : non seulement il faut introduire des intermédiaires, mais il est nécessaire :

- de disposer de règles permettant de spécifier puis de construire ces intermédiaires
- de systématiser le passage d'un intermédiaire à l'autre
- de disposer de règles "de composition" des différents algorithmes ainsi obtenus.

Notons enfin, que les sous-problèmes (1) et (3) peuvent être considérés respectivement comme des problèmes d'abstraction et de représentation.

6.4 Transformation d'énoncés

[ARS,77] propose un certain nombre de transformations sémantiques d'énoncés récurrents. FEATHER [FEA,79] partant des idées à la base du système de BURSTALL et DALINGTON [BUR,77 b] introduit également un ensemble de règles de transformation dont certaines concernent les énoncés récurrents. Ceci lui permet de résoudre le problème du télégramme (ch 1.1 § 1.1 f) en adoptant une démarche voisine de celle évoquée au paragraphe précédent [FEA,78] .

Il nous semble nécessaire de compléter ces études par la mise en évidence de critères de transformation : décrire la forme du résultat comme dans [FEA,79] est un premier pas dans cette direction, il serait plus important encore de dire quelles sont les propriétés que doit posséder le résultat (complexité de l'algorithme, occupation mémoire etc...).

CHAPITRE 2.4

UNE METHODE D'EXPLICITATION DE PROBLEMES

Au chapitre 2.1 nous avons spécifié le meta-problème de l'explicitation des problèmes et nous avons montré que sa résolution nécessitait :

i) de se doter d'un langage de descriptions explicites

ii) de définir des stratégies d'explicitation

iii) de se munir d'un langage permettant de décrire la construction d'une version explicite d'un problème.

Le premier point est certainement le plus facile à satisfaire car il est lié à la notion bien connue de solution calculable. Les a-descriptions de S.I. et le langage MEDEE répondent à ce besoin.

La mise en évidence de stratégies d'explicitation (ii) n'en est qu'à ses balbutiements. Et, comme le souligne SINTZOFF dans [SIN,79], si de nombreux exemples ont été étudiés, la nature profonde des méthodes d'explicitation n'est pas encore bien connue.

Pour construire un catalogue de stratégies d'explicitation il faut donc mener de front les deux activités suivantes :

a) Commencer par dégager et structurer ces stratégies à partir des idées que nous avons mises en évidence dans les chapitres précédents :

. Une stratégie d'explicitation consiste à introduire des intermédiaires. Elle repose donc sur une *règle de construction* qui en est l'énoncé implicite. C'est ce que nous avons décrit dans la partie "*donnée* - *résultat* .. *tq* ... des stratégies.

. Il n'existe pas, en général, de version explicite d'une stratégie. Plus précisément pour résoudre (expliciter) une stratégie il faut faire appel à d'autres stratégies. D'où une mise en oeuvre récursive du processus d'explicitation, récursivité liée à la complexité du problème traité.

Nous avons décrit quelques possibilités de résolution d'une stratégie dans la partie "*heuristique*".
Ainsi, décrire le rôle d'une stratégie revient à la spécifier, décrire des heuristiques consiste à la résoudre.

b) Pour chaque stratégie, étudier les conditions de mise en oeuvre (ce que l'on peut appeler heuristiques d'utilisation par opposition aux heuristiques de résolution évoquées ci-dessus). Nous n'avons pratiquement pas abordé cet aspect du choix d'une stratégie sinon sous l'angle de la forme syntaxique des énoncés à expliciter. Par exemple la stratégie d'explicitation s-récursive, ⊦ SESR ⊣ , porte sur une s-description d'une fonction et doit fournir une définition s-récursive.

Il est donc très important de se doter de critères de choix *sémantiques* portant sur certaines propriétés de l'énoncé à expliciter et de propriétés attendues du résultat de l'explicitation.

Enfin le point iii) n'a été que très peu envisagé dans les chapitres précédents.

Néanmoins, nous avons pu constater sur quelques exemples, qu'il n'était pas simple de décrire de manière linéaire les choix d'explicitation successifs, notamment lorsqu'il est nécessaire d'effectuer des retours en arrière. Ainsi, il est raisonnable de construire, en même temps que le tableau à 3 colonnes (énoncé implicite, profil, énoncé explicite) qui contiendra le résultat final de l'explicitation, l'*arbre des choix successifs*. Le langage de description du processus d'explicitation doit donc permettre de décrire :

- Les buts successifs.
- Les choix successifs, leur justification et correlativement la construction de l'arbre de choix.
- Les différents calculs algébriques ou ensemblistes utilisés dans une construction.

Dans ce chapitre nous essayons de dégager quelques nouvelles stratégies d'explicitation (§ 1) avant de proposer une structure générale d'un meta-algorithme d'explicitation et un langage de description de construction (§ 2).

Nous terminons en étudiant l'explicitation du problème du tri topologique (§ 3).

1. QUELQUES AUTRES STRATEGIES D'EXPLICITATION

1.1. Induction sur les données et induction sur les résultats

Dans les heuristiques de résolution des stratégies d'explicitation s-récursive (⊦ SESR ⊣) et c-récursive (⊦ SECR ⊣) étudiées au chapitre 2.2, le guide essentiel était la structure des données. L'idée fondamentale est de chercher à simplifier les arguments de la fonction à expliciter, soit par utilisation de constructeurs, soit par utilisation de simplificateurs. Nous pouvons parler dans ce cas de *stratégie d'induction sur les données*.

Plutôt que de travailler sur les données, il est possible de commencer par étudier la nature du résultat y. Lorsque la sorte d'intérêt de ce dernier n'admet qu'un nombre restreint de constructeurs, nous pouvons chercher à exprimer y à l'aide de ces consctruteurs : pour chaque constructeur p-aire g il faut introduire p intermédiaires $y_1 \dots y_p$ tels que $y = g(y_1 \dots y_p)$

Mais par définition, $y = f(x_1 \dots x_q)$
il s'agit donc finalement de chercher à exprimer $y_1 \dots y_p$ à l'aide de $x_1 \dots x_q$.

Par exemple lorsque g n'admet qu'un seul argument, disons y_1, de la sorte de y, nous souhaitons définir une fonction h telle que :

$$f(x_1, \dots, x_q) = g(f(h(x_1, \dots, x_q), x_1 \dots x_q), y_2 \dots y_p)$$

Nous parlerons alors d'*induction sur les résultats*.

Remarque :

La stratégie d'affaiblissement peut être considérée, à la fois comme une induction sur les données (on utilise le constructeur $i \to i+1$ sur les indices) ou comme une induction sur les résultats (constructeur $\curlyvee$ sur les suites).

1.2. Généralisation

C'est une technique d'explicitation classique qui consiste à introduire une fonction intermédiaire.

⊦ S G ⊣ Stratégie de généralisation
donnée x caractérisé par ε_x

résultat - f symbole fonctionnel p-aire caractérisé par une s-description
$f(x_1 .. x_p) = y$ *tq* Ψ *si* φ
- p symboles p-aires $y_1 \ldots y_p$ caractérisés respectivement par les énoncés $\varepsilon y_1 \ldots \varepsilon y_p$.

tq $(\varepsilon y_1 \wedge .. \wedge \varepsilon y_p \supset \varphi [y_1/x_1 \ldots y_p/x_p]) \wedge$
$(\varepsilon y_1 \wedge .. \wedge \varepsilon y_p \wedge \Psi [f y_1 .. y_p / y] \supset \varepsilon_x [f y_1 .. y_p / x])$

Présentation

- $\varepsilon y_1 .. \varepsilon y_p$, $f(x_1 .. x_p) = y$ *tq* Ψ *si* φ sont placés dans la colonne de gauche
- les profils de $y_1, .., y_p$ et f sont placés dans la colonne centrale
- $x = f(y_1 .. y_p)$ est placé dans le colonne de droite
- x est coché

Heuristiques

a) Choisir, parmi les intermédiaires dont dépend x, les $y_1 .. y_p$. Ceci permet de caractériser Ψ à partir de ε_x. C'est par exemple ce qui est mis en oeuvre dans l'explicitation du problème des sous suites croissantes maximales lorsque le résultat lmax est "transformé" en une fonction lmax(α) de la donnée α [BRO,79a].

b) Choisir, comme en a) les $y_1 .. y_{p-1}$ et introduire un intermédiaire supplémentaire y_p qui n'apparaît pas dans ε_x. C'est ce que nous avons fait dans l'exemple 2 du chapitre 2.1 lorsque nous avons introduit la fonction maz à partir de mazt. C'est également ce que suggère la stratégie d'affaiblissement : l'affaiblissement peut être considéré comme la généralisation restreinte aux suites.

1.3. Analyse récurrente

C'est la stratégie qui est souvent mise en oeuvre pour expliciter l'invariant obtenu par affaiblissement.

⊦ SAR ⊦ Stratégie d'analyse récurrente

donnée u suite caractérisée par ε_u (invariant)
s suite booléenne caractérisée par π_a (condition d'arrêt)
k, n symboles de profils respectifs ENTIER et ENTIERC

résultat - f fonction de profil $\bar{S}_1, .., \bar{S}_p \rightarrow \bar{S}$ où $\bar{S}$ est le profil de u et l'un au moins des $\bar{S}_i$ est égal à $\bar{S}$. Pour fixer les idées posons $\bar{S}_1 = \bar{S}$.
- φ et Ψ respectivement pré-condition et post-condition de
$f x_1 \ldots x_p = y$
- p - 1 intermédiaires $y_2 \ldots y_p$ caractérisés par $\varepsilon y_2 .. \varepsilon y_p$
- t interne caractérisé par ε_t

tq $(\varepsilon_u [\bar{u}/u] \wedge \varepsilon y_2 \wedge .. \wedge \varepsilon y_p \wedge \neg \pi_a \supset \varphi [u/x_1, .., y_p/x_p]) \wedge$
$(\varepsilon_u [\bar{u}/u] \wedge \varepsilon y_2 \wedge .. \wedge \varepsilon y_p \wedge \neg \pi_a \wedge \Psi [u/x_1, .., y_p/x_p] \supset$
$\varepsilon_u [f \bar{u} y_2 .. y_p / u]) \wedge$
$(\varepsilon_t [\$u/t] \supset \varepsilon_u [\$u/u])$

Présentation

- Créer un nouveau module *u si nécessaire
- $\varepsilon y_2 \ldots \varepsilon y_p$, ε_t, $f x_1 .. x_p = y$ *tq* Ψ *si* φ sont placés dans la colonne de gauche
- les profils de $y_2, .., y_p$, t, f sont placés dans la colonne centrale
- $u = f(\bar{u}, y_2, .., y_p)$ et $\$u = t$ sont placés dans la colonne de droite
- u est coché

Heuristiques

Effectuer une analyse par cas en distinguant $\varepsilon_u[k-1/i]$ des autres cas
- Chercher à unifier ε_u et $\varepsilon_u [\bar{u}/u]$, ce qui doit conduire à f, $y_2 .. y_p$
- Chercher à expliciter $\varepsilon_u [k-1/i]$ ce qui doit conduire à t.

Remarque :

Cette stratégie peut être considérée comme un cas particulier d'explicitation s-récursive (cf. ch. 2.2 § 1.3.2).

1.4. Stratégies plus ponctuelles

Nous ne décrivons pas un certain nombre de stratégies de très bas niveau qui sont en fait des opérations indéterministes associées à des règles de calcul booléen, arithmétique ou ensembliste. Parmi elles, citons la disjonction (faire apparaître 2 énoncés ε_1 et ε_2 tels qu'un énoncé donné ε s'écrive $\varepsilon = \varepsilon_1 \wedge \varepsilon_2$),

la conjonction ($\varepsilon = \varepsilon_1 \wedge \varepsilon_2$) souvent associée à une analyse par cas, le passage d'une forme logique à une forme ensembliste (on remplace ε_x par $x \in \{z \ tq \ \varepsilon_x[z/x]\}$) etc ...

2. META-ALGORITHME D'EXPLICITATION

2.1. Description du méta-algorithme

Nous n'en donnons qu'une description semi-formelle et partielle (figure 1) qui ne doit être considérée que comme une esquisse d'un véritable meta-algorithme de construction de problèmes explicites.

Il est indispensable de préciser et de formaliser cet algorithme si l'on souhaite se doter d'un système d'aide à la construction des programmes. Mais même dans le cas d'une construction "à la main" il faut structurer l'ensemble des stratégies ce que nous n'avons fait que de façon embryonnaire. En particulier il faut situer les mécanismes généraux d'abstraction et de représentation par rapport aux stratégies plus ponctuelles (explicitation s-récursive, généralisation etc...).

De plus, une formalisation du meta-algorithme nécessite de décrire l'aspect syntaxique des énoncés et S.I. obtenus : il faut décrire le meta-système d'information du meta-problème de l'explicitation, comme nous avons commencé à le faire pour celui du meta-problème de la spécification (ch. 1.4). En particulier il faut introduire des constructeurs de modules, des opérations telles que "cocher".

Enfin, comme nous l'avons mentionné dans l'introduction, il faudrait définir le meta-système d'information des "arbres de choix de construction" avec des opérations d'accès permettant des retours en arrière. Ici encore nous le décrivons de manière informelle en même temps que nous précisons un langage permettant d'exprimer le déroulement d'un processus d'explicitation.

2.2. Description de la construction d'un énoncé explicite

La construction d'une version explicite d'un problème à partir d'une spécification formelle, est une suite de décisions et de calculs.

Les calculs sont la partie la plus classique de la demande, malheureusement ils sont encore, de manière générale, difficilement automatisables puisqu'ils font souvent appel à des "astuces", des heuristiques de simplification etc ...

P' et P décrits sur la figure 7 du ch. 2.1 $ttcoché_i$: booleen qui indique si tous les identificateurs sont cochés	P' : PROBLEME EXPLICITE P : PROBLEME ttcoché : BOOL* Pb : PROBLEME*		*résultat* P' = (E',(S', Dop, Dyp)) P' = *der* Pb_i *pour* i *dans* $1 \rightarrow iga$ $ttcoché_i$ *donnée* P = (E, (S, Dop, Dyp))
• Pb			
initialiser : créer un tableau à 3 colonnes en plaçant dans la colonne de gauche P (module par module), dans la colonne centrale le *résultat* r de P avec son profil, dans la colonne de droite "*résultat* r" ♯ SEXP ♮ : stratégie d'explicitation ♯ SREP ♮ : stratégie de représentation, décrite informellement au chapitre 2.2 ♯ SABS ♮ : stratégie d'abstraction décrite au chapitre 2.2	initialiser : *fonct*(PRORLEME) PROBLEME ♮ SEXP ♮, ♮ SRER ♮, ♮ SABS ♮ : *fonct*(PROBLEME, BOOL) PROBLEME		Pb, ttcoché = *choix entre* ♮ SEXP ♮ ($\overline{Pb}$, $\overline{ttcoché}$) ♮ SREP ♮ ($\overline{Pb}$, ttcoché) ♮ SABS ♮ ($\overline{Pb}$, ttcoché) $ Pb = initialiser(P) $ ttcoché = *faux*
♮ SEXP ♮ (P_1, b) = P_2			
♮ SEXPG ♮ : stratégie générale d'explicitation de l'énoncé caractérisant un identificateur dans une suite de modules	P_1, P_2 : PROBLEME ♮ SEXPG ♮ : *fonct*(MODULE*, ENONCE, SYMBOLE), MODULE*		P_2 = (E_2, (S_1, Dop, Dyp)) E_2 = ♮ SEXPG ♮ (E_1, E_x, x) x = *choix dans* {y ∈ colonne centrale (E_1) *tq* ¬ coché (y)}

⊢ SEXPGD $(E_1, \epsilon_x, x) = E_2$		
⊢ SESR ⊢ : explicitation s-récursive (ch. 2.2 § 1.3.2) ⊢ SECR ⊢ : explicitation c-récursive (ch. 2.2 § 3.1.2) ⊢ S G ⊢ : généralisation (§ 1.2) ⊢ SAR ⊢ : analyse récurrente (§ 1.3) ⊢ SES ⊢ : explicitation simple (ch. 2.3 § 2.3.2) ⊢ SEA ⊢ : affaiblissement (ch. 2.3 § 2.3.2) ⊢ SISC ⊢ : induction sur une suite complète (ch. 2.3 § 3) ⊢ SAC ⊢ : analyse par cas		E_2 = *choix entre* - retour en arrière - *cas* x : *fonction alors choix entre* - ⊢ SESR ⊢ (E_1, ϵ_x, x) - ⊢ SECR ⊢ (E_1, ϵ_x, x) - ⊢ S G ⊢ (E_1, ϵ_x, x) x : *suite alors choix entre* - ⊢ SAR ⊢ (E_1, ϵ_x, x) - ⊢ SEA ⊢ (E_1, ϵ_x, x) - ⊢ SISC ⊢ (E_1, ϵ_x, x) - ⊢ SES ⊢ (E_1, ϵ_x, x) - ⊢ S G ⊢ (E_1, ϵ_x, x) *sinon choix entre* - ⊢ SES ⊢ (E_1, ϵ_x, x) - ⊢ S G ⊢ (E_1, ϵ_x, x) - ⊢ SEA ⊢ (E_1, ϵ_x, x) - ⊢ SISC ⊢ (E_1, ϵ_x, x) - ⊢ SAC ⊢ (E_1, ϵ_x, x)

Figure 1 : Esquisse d'un meta-algorithme d'explicitation

Dans une description d'une construction, nous décidons de faire précéder chaque calcul, ou suite de calculs, du mot *calcul*. La nature de chaque opération élémentaire (instantiation, pliage, dépliage, utilisation d'un axiome arithmétique, algébrique, bolléen, etc ...) est indiquée explicitement.

Mais la partie la plus délicate d'une construction est certainement celle des choix qui peuvent être de différents types :

i) Choix de stratégie : ce sont les plus incertains parce que, d'une part nous sommes loin de disposer d'un catalogue de stratégies complet et bien organisé, et d'autre part les quelques critères associés à chaque stratégie pour caractériser son domaine d'application sont loin d'être complets.

ii) Choix d'intermédiaires : la mise en oeuvre d'une stratégie permet de caractériser les intermédiaires à introduire. Bien que la démarche suivie ait pour but de réduire l'indéterminisme sur les intermédiaires, il reste cependant en général un effort d'invention à faire.

iii) Choix, dans l'ensemble des candidats possibles, de l'identificateur à expliciter à une étape donnée.

A une étape donnée, deux choix de type i) ou ii) différents sont exclusifs l'un de l'autre, ils correspondent au "choix d'un élément x dans un ensemble". Inversement les différentes possibilités apparaissent dans un choix de type iii) devront toutes être traitées : il s'agit d'un "pour tout x dans un ensemble". Ainsi un arbre de choix est décrit à l'aide de plusieurs constructeurs :

- un constructeur indiquant que les différents fils d'un noeud sont exclusifs l'un de l'autre, ce que l'on représente parfois sous la forme

et que nous notons simplement

- un constructeur indiquant que les différents fils d'un noeud devront être tous traités, ce qui est parfois représenté par :

et que nous notons

La figure 3 donne un exemple d'arbre de choix, on y distingue les différents types de noeuds :

S T : choix de stratégie

⟨ u = ε ⟩ : choix d'intermédiaire

(expliciter p) : choix d'identificateur à expliciter

Cet arbre permet d'effectuer des retours en arrière, dans le langage de description d'une construction nous utiliserons un nouveau type de choix : celui de l'endroit où l'on souhaite retourner.

Chaque choix est précédé de *choix de* et suivi d'une justification informelle "*parce que*".

Outre les choix et les calculs, il est utile de préciser, à la suite d'un choix, quelques possibilités de résolution ; nous les ferons précéder de :

"*Différents choix*" ou "*Quelques possibilités pour*"

Le meta-algorithme construit une description d'un problème explicite ; l'introduction d'objets ou formules dans le tableau à trois colonnes en cours de construction est précédée de "*Action*"

Enfin il est utile, après chaque choix, de rappeler l'objectif visé. Chaque nouvel objectif est précédé de "*But*".

L'exemple suivant montre une utilisation de ce "*langage*".

3. DERNIER EXEMPLE D'EXPLICITATION

Dans ce paragraphe nous appliquons le meta-algorithme précédent au problème du tri topologique. Nous partons donc de la spécification représentée sur le figure 1.2 du chapitre 1.1. En reprenant les idées précédentes nous construisons conjointement la suite des descriptions de problèmes partiellement explicités (figure 2) ainsi que l'arbre des choix successifs (figure 3). La démarche que nous suivons est décrite en utilisant le langage envisagé au paragraphe 2.2.

But : Expliciter le problème du tri topologique spécifié sur la figure 1.2 du chapitre 1.1.

Action : Initialisation de la description de la figure 3
- Placer la spécification obtenue au chapitre 1.1.en partie gauche
- Placer α et son profil dans la colonne centrale
- Ecrire "*résultat* α" en partie droite
- $ttcoché = *faux*

Notation : Tout prédicat φ : *foncț*(ENTIER)BOOL est étendu aux intervalles d'entiers par $\varphi\,[m..n] = \forall\, i \in [m..n]\ \varphi\,(i)$

Différents choix :
- stratégie d'explicitation ⟊ SEXP ⟋
- stratégie de représentation ⟊ SREP ⟋
- stratégie d'abstraction ⟊ SABS ⟋

d'après la définition du meta-algorithme

Choix de ⟊ SEXP ⟋ *parce que* seul le résultat α n'est pas coché.

Différents choix :
- ⟊ SAR ⟋ # inutilisable car ε_α n'est pas un invariant #
- ⟊ SEA ⟋

etc...

autres stratégies correspondant au cas où x est une suite, cf. figure 1

d'après la définition du meta-algorithme

Choix de ⟊ SEA ⟋ # affaiblissement # *parce que* α dépend de n

But : Trouver u symbole de suite, ε_u invariant, a symbole de suite booléenne et π_a condition d'arrêt tels que (RIDI) soit vérifiée

Différents choix : - décomposition de ε_α en deux formules
- induction fondée sur un constructeur des données

d'après les heuristiques de ⊦ SEA ⊣

Choix de ⊦ décomposition ⊣ *parce que* ε_α semble décomposable et dépend de n

But : Trouver ε_u et π_a tels que $\exists n1,\ \varepsilon_\alpha = \varepsilon_u \wedge \pi_a\ [\alpha/u_\ell,\ n1/\ell]$

Quelques possibilités pour π_a :
- $\ell \equiv n$
- $\ell \equiv 1$
- $u_\ell \in inj(n)$
- $u_\ell \in inj(\ell)$
- $u_n \in inj(\ell)$
- $\forall i \in [1..n],\ \forall j \in [1..n]\ j < i \supset u_{\ell,i}\ \hat{\Gamma}\ u_{\ell,j}$ etc ...

d'après la forme de ε_α

Choix de $\pi_a = (\ell \equiv n)$ *parce que* l'un des plus simple et conduit à une forme simplifiée de définition itérative

Conséquence : $\varepsilon_\alpha \equiv \varepsilon_u\ [\alpha/u_\ell,\ n/\ell]$

But : Trouver ε_u tel que l'égalité ci-dessus soit vérifiée

Introduction intermédiaire : $cpt\ (i,j) = j < i \supset \alpha_i\ \hat{\Gamma}\ \alpha_j$

Quelques possibilités pour ε_u :
- a) - $u_\ell \in inj(\ell) \wedge cpt([1..n], [1..n])\ [u_\ell/\alpha]$
- b) - $u_\ell \in inj(\ell) \wedge cpt([1..\ell], [1..n])\ [u_\ell/\alpha]$
- c) - $u_\ell \in inj(\ell) \wedge cpt([1..\ell], [1..\ell])\ [u_\ell/\alpha]$
- d) - $u_\ell \in inj(\ell) \wedge cpt([1..n], [1..\ell])\ [u_\ell/\alpha]$
- e) - $u_\ell \in inj(n) \wedge cpt([1..\ell], [1..n])\ [u_\ell/\alpha]$
- f) - $u_\ell \in inj(n) \wedge cpt([1..\ell], [1..\ell])\ [u_\ell/\alpha]$
- g) - $u_\ell \in inj(n) \wedge cpt([1..n], [1..\ell])\ [u_\ell/\alpha]$

Propriétés
- a) ⇏ b) ⇏ c) ⇏ d) car le domaine est réduit à $[1..\ell]$
- c) ⇏ f) car $\forall i \in [1..\ell], \forall j \in [1..n], j < i \supset u_{\ell,i}\ \hat{\Gamma}\ u_{\ell,j}$
  ⇏ $\forall i \in [1..\ell], \forall j \in [1..n], j \leq \ell \wedge j < i \supset u_{\ell,i}\ \hat{\Gamma}\ u_{\ell,j}$
  ⇏ $\forall i \in [1..\ell], \forall j \in [1..\ell], j < i \supset u_{\ell,i}\ \hat{\Gamma}\ u_{\ell,j}$

Conséquence : il ne reste que 3 possibilités parmi les précédentes
- a1) $u_\ell \in inj(\ell) \wedge cpt([1..\ell], [1..\ell])\ [u_\ell/\alpha]$
- b1) $u_\ell \in inj(n) \wedge cpt([1..\ell], [1..\ell])\ [u_\ell/\alpha]$
- c1) $u_\ell \in inj(n) \wedge cpt([1..n], [1..\ell])\ [u_\ell/\alpha]$

Choix ① $\varepsilon_u = u_\ell \in inj(n) \wedge cpt([1..\ell], [1..\ell])\ [u_\ell/\alpha]$ *parce que* au hasard

Finbut

Finbut ⊦ décomposition ⊣

Action : - Placer ε_u dans la colonne de gauche ainsi que cpt
- Placer les profils de u et n dans la colonne centrale
- introduire la définition itérative de α en partie droite
- cocher α

Finbut ⊦ SEA ⊣

But : Expliciter l'un des identificateurs non coché, ici u ou n

Choix de u *parce que* c'est la suite de ⊦ SEA ⊣

But : expliciter u

Différents choix - ⊦ SAR ⊣
- ⊦ SEA ⊣
- etc...

d'après la définition du meta-algorithme puisque u est une suite

Choix de ⊦ SAR ⊣ # analyse récurrente # *parce que* ε_u est un invariant

But 1 : Trouver une définition explicite de u_ℓ en fonction de $u_{\ell-1}$ avec $1 \leq \ell \leq n$

Différents choix - ⊦ SES ⊣
- ⊦ SG ⊣
- etc...

d'après le meta-algorithme puisque u_ℓ est un élément de suite (bien qu'ici ce soit encore une suite).

Choix de ⊦ SES ⊣ # définition explicite simple # *parce que* on veut définir u_ℓ comme une amélioration de $u_{\ell-1}$

But : Trouver une fonction f telle que $u = f(u_{\ell-1})$

Choix de ⟨ unification ⟩ *parce que* c'est l'une des heuristiques de ⟨ SES ⟩ lorsque ε_u est un invariant

Calculs :

a) *dépliage*

$\varepsilon_u = u_\ell \in inj(n) \wedge \forall i \in [1..\ell], \forall j \in [1..\ell], j < i \supset u_{\ell,i} \hat{\Gamma} u_{\ell,j}$

b) *instantiation*

$\varepsilon_u [\ell\text{-}1/\ell] = u_{\ell-1} \in inj(n) \wedge \forall i \in [1..\ell-1], \forall j \in [1..\ell-1], j < i \supset u_{\ell-1,i} \hat{\Gamma} u_{\ell-1,j}$

c) *introduction d'un intermédiaire*

$\varepsilon_u [\ell\text{-}1/\ell, u_\ell/u_{\ell-1}] = u_\ell \in inj(n) \wedge \forall i \in [1..\ell-1], \forall j \in [1..\ell-1], j < i \supset u_{\ell,i} \hat{\Gamma} u_{\ell,j}$

d) *unification entre* a) et c)

$\varepsilon_u = \varepsilon_u [\ell\text{-}1/\ell, u_\ell/u_{\ell-1}] \wedge \varphi$

$\varphi = \forall j \in [1..\ell-1], j < \ell \supset u_{\ell,\ell} \hat{\Gamma} u_{\ell,j}$

e) *simplification*

$\varphi = \forall j \in [1..\ell-1], u_{\ell,\ell} \hat{\Gamma} u_{\ell,j}$

f) *hypothèse* Γ est sans circuit

$\forall x \in E, x \hat{\Gamma} x$

g) *conjonction* de e) et f)

$\varphi = \forall j \in [1..\ell], u_{\ell,\ell} \hat{\Gamma} u_{\ell,j}$

h) *unification entre* b) et c)

$u_\ell \in inj(n) \wedge \forall i \in [1..\ell-1], u_{\ell-1,i} \equiv u_{\ell,i}$

i) *conjonction* de g) et h)

$u_\ell \in inj(n) \wedge (\forall j \in [1..\ell], u_{\ell,\ell} \hat{\Gamma} u_{\ell,j}) \wedge (\forall i \in [1..\ell-1], u_{\ell-1,i} \equiv u_{\ell,i})$

Choix de $u_\ell \equiv trsp(u_{\ell-1}, \ell, m)$ (+)

$\wedge$ m *tq* m $\in [\ell..n] \wedge (\forall j \in [1..\ell]\ u_{\ell-1,m} \hat{\Gamma} u_{\ell-1,j}$

parce que d'après i) la propriété supplémentaire de u_ℓ par rapport à $u_{\ell-1}$ est φ

(+) Rappelons que trsp est l'opération de transposition de deux élément d'une suite, cf. Ch. 1.2 § 7.2.2.

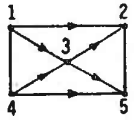
impasse # Il n'y a aucune raison pour que la spécification précédente de ℓ admette une solution. On trouve précisément ce résultat lorsque l'on essaye d'expliciter m : une manière peut être peu efficace mais sûre de le définir consiste à explorer l'intervalle $[\ell..n]$ et à vérifier la condition $u_{\ell-1,m} \hat{\Gamma} u_{\ell-1,j}$. Mais on ne peut pas démontrer que la fonction rech ainsi définie fournit un résultat.

On n'a pas :

$\varepsilon_u [\ell\text{-}1/\ell] \supset$ *pré* rech(ℓ)

la condition de *non blocage* (ch. 2.3 § 2.3.1 règle (RIDS) n'est pas vérifiée)

On peut d'ailleurs trouver des contre-exemples simples :



si $u_3 = 3, 2, 5$
on ne peut trouver ℓ = rech(4) #

Quelques possibilités pour le retour. # on utilise ici l'arbre de choix #

a) choix de u_ℓ
b) choix de ⟨ unification ⟩
c) choix de ⟨ SES ⟩
d) choix de ⟨ SAR ⟩
e) choix de ε_u
f) choix de π_a

Choix de retour en ⟨ $\varepsilon_u = u_\ell \subset \ldots$ ⟩

parce que a) est difficile à remettre en cause directement car il faut de toute manière trouver m pour vérifier

b) est directement lié à ⟨ SES ⟩

c) ne peut être changé sous peine d'avoir une définition plus complexe de u_ℓ à partir de $u_{\ell-1}$, par exemple en utilisant une itération. C'est envisageable mais contraire à l'idée d'affaiblissement où l'on cherche un passage simple d'un terme à l'autre.

d) est lié étroitement au fait que ε_u soit un invariant. Il n'est pas immédiat ici de donner une définition non récurrente de u_ℓ.

Choix ② $\varepsilon_u = u_\ell \in inj(n) \wedge cpt([1..n], [1..\ell])\ [u_\ell/\alpha]$
parce que c'est l'invariant le "plus proche" de ε_α parce qu'il ne contient qu'une seule occurrence de ℓ en paramètre.

Finbut

Finbut ⧼ décomposition ⧽

Action : - Placer ε_u
- ⎫ comme ci-dessus
- .... ⎭

Finbut ⧼ SEA ⧽

But : expliciter l'un des identificateurs non coché, ici u ou n

Choix de u *parce que* c'est la suite de ⧼ SEA ⧽

Choix de ⧼ SAR ⧽ # analyse récurrente # *parce que* ε_u est un invariant

But 1 : Trouver une définition explicite de u_ℓ en fonction de $u_{\ell-1}$ avec $1 \le \ell \le n$

Choix de ⧼ SES ⧽ *parce que* comme ci-dessus
⎧ comme dans la définition précédente, mutatis-mutaudis
e) $\varphi = \forall i \in [\ell..n]\ u_{\ell,i}\ \Gamma\ u_{\ell,\ell}$
⎩ comme précédemment

Choix de $u_\ell \equiv trsp(u_{\ell-1}, \ell, m)$
$\wedge\ m\ tq\ m \in [\ell..n] \wedge \forall i \in [\ell..n]\ u_{\ell-1,i}\ \hat{\Gamma}\ u_{\ell-1,\ell}$

Finbut ⧼ unification ⧽

Finbut But 1

But 2 : Trouver une définition de u_0

Calculs :

a) *instantiation*
$\varepsilon_u\ [o/\ell] = u_o \in inj(n) \wedge \forall i \in [1..n], \forall j \in [1..o]\ j < i \supset u_{o,i}\ \hat{\Gamma}\ u_{o,j}$

b) *simplification*
$\varepsilon_u\ [o/\ell] = u_o \in inj(n)$

Choix de u_o = suite quelconque formée des n éléments de E

Finbut But 2

Finbut ⧼ SAR ⧽

Finbut ⧼ expliciter u ⧽

Action : - Introduire un nouveau module *u
- Placer la spécification de m en colonne gauche
- Placer le profil de m au centre
- Placer les définitions de u et $ u en colonne droite
- Cocher u

But : Expliciter l'un des identificateurs non coché, ici n et m

Choix de n *parce que* on termine ainsi le module spécial
⎩ on introduit ainsi n, dont la spécification est déjà explicite, E et Γ

But : Expliciter l'un des identificateurs non coché, ici il ne reste plus que m

Quelques possibilités pour m
- exploration systématique de l'intervalle $[\ell..n]$, utilisation de la construction "*premplace .... telle que*" ; c'est-à-dire encore utiliser une version de ⧼ SISC ⧽
- introduire comme intermédiaire l'ensemble des m solution et en chercher une définition itérative que *l'on peut fusionner* avec la définition de u.
ici il y a plus qu'une première explicitation puisqu'on se préoccuppe d'éfficacité

Choix de $m \in rac$
$\wedge\ rac_\ell = \{z \in [\ell..n]\ tq\ \forall i \in [\ell..n]\ u_{\ell-1,i}\ \Gamma\ u_{\ell-1,z}\}$
parce que c'est une bonne solution si elle existe !

Propriétés : - rac_ℓ est l'ensemble des racines du graphe partiel $(u_{\ell-1}\ [\ell..n], \Gamma)$
- théorèmes de théorie des graphes : tout graphe non vide sans circuit admet une racine

Conséquence : $rac_\ell \neq \emptyset$ # il n'y a pas de blocage #

Finbut

Action : - Placer la définition de m en partie droite
- Placer la spécification de rac en colonne gauche
- Introduire le profil de rac au centre
- cocher m

But : Expliciter un identificateur non coché, ici uniquement rac

Choix de ⊦ SAR ⊣ *parce que* rac_ℓ est caractérisé par un invariant

But 1 : Trouver une définition explicite de rac_ℓ en fonction de $rac_{\ell-1}$ avec $1 \leq \ell \leq n$

Différents choix - ⊦ SES ⊣
- ⊦ SG ⊣
etc ...

d'après le meta-algorithme, puisque rac_ℓ est un élément d'une suite

Choix de ⊦ SES ⊣ *parce que* on veut définir rac_ℓ comme une amélioration de $rac_{\ell-1}$

But : Trouver une fonction f telle que $rac_\ell = f(rac_{\ell-1})$

Choix de ⊦ unification ⊣ *parce que* c'est une heuristique de ⊦ SES ⊣ lorsque ε_u est un invariant

Calculs

a) *dépliage*

$rac_\ell = \{z \in [\ell..n] \ tq \ \forall i \in [\ell..n] \ \neg u_{\ell-1,i} \ \Gamma \ u_{\ell-1,z}\}$

b) *introduction d'un intermédiaire*

$rac_\ell = [\ell..n] \cap com_\ell$

$com_\ell = \{z \ tq \ \forall i \in [\ell..n] \ \neg u_{\ell-1,i} \ \Gamma \ u_{\ell-1,z}\}$

c) *instantation*

$com_{\ell-1} = \{z \ tq \ \forall i \in [\ell-1..n] \ \neg u_{\ell-2,i} \ \Gamma \ u_{\ell-2,z}\}$

d) *décomposition* de com

$com_\ell = \{z \ tq \ (\forall i \in [\ell-1..n] \ \neg u_{\ell-1,i} \ \Gamma \ u_{\ell-1,z}) \vee [(\forall i \in [\ell..n] \ \neg u_{\ell-1,i} \ \Gamma \ u_{\ell-1,z}) \wedge u_{\ell-1,\ell-1} \ \Gamma \ u_{\ell-1,z}]\}$

e) *propriété*

$u_{\ell-1}([\ell-1..n]) = u_{\ell-2}([\ell-1..n])$

f) *unification* de com_ℓ et $com_{\ell-1}$

$com_\ell = com_{\ell-1} \cup truc_\ell$

$truc_\ell = \{z \ tq \ \forall i \in [\ell..n] \ \neg u_{\ell-1,i} \ \Gamma \ u_{\ell-1,z}\} \cap \{z \ tq \ u_{\ell-1} \ \Gamma \ u_{\ell-1,z}\}$

g) *dépliage* rac_ℓ b) et f)

$rac_\ell = ([\ell..n] \cap com_{\ell-1}) \cup ([\ell..n] \cap truc_\ell)$

h) *pliage* et *intermédiaire*

$rac_\ell = (rac_{\ell-1} \setminus \{\ell-1\}) \cup nouv_\ell$

$nouv_\ell = pred_{\ell-1} \cap suc_{\ell-1}$

$pred_{\ell-1} = \{z \in [\ell..n] \ \underline{tq} \ \forall i \in [\ell..n] \ \neg u_{i-1,i} \ \Gamma \ u_{\ell-1,z}\}$

$suc_{\ell-1} = \{z \ \underline{tq} \ u_{\ell-1,\ell-1} \ \Gamma \ u_{\ell-1,z}\}$

Finbut

Finbut But 1

But 2 : Définir rac_0

a) *instantiation*

$rac_0 = \{z \in [0..n] \ \underline{tq} \ \forall i \in [0..n] \ \neg u_{-1,i} \ \Gamma \ u_{-1,z}\}$

ce qui n'est pas défini

b) *instantiation*

$rac_1 = \{z \in [1..n] \ \underline{tq} \ \forall i \in [1..n] \ \neg u_{0,i} \ \Gamma \ u_{0,z}\}$

$= nouv_1$ si on pose $suc_0 = E$ et $rac_0 = \emptyset$

Finbut But 2

Action :
- Placer les spécifications de pred et suc dans la colonne gauche
- Placer les profils de nouv, pred et suc dans la colonne centrale
- Placer les définitions de rac et nouv dans la colonne de droite
- Cocher rac et nouv

Finbut Explicitation de rac

Nous arrêtons ici cette explicitation. Une version plus complète mais moins systématique de cette démarche se trouve dans [FIN,78a].

Figure 2 : début d'énoncé explicite du problème du tri topologique

Figure 3 : Arbre de choix du tri topologique

CONCLUSION

CONCLUSION

1 - BILAN DE CE TRAVAIL

Nous nous sommes efforcé , tout au long de ce travail, d'unifier un certain nombre de concepts jusqu'ici assez disjoints : types abstraits, spécification de problèmes, solution d'un problème etc... Nous espérons que cette synthèse aidera à l'élaboration d'une véritable *théorie de la construction des programmes*, complément indispensable aux diverses *théories des programmes* développées jusqu'ici (syntaxe, sémantique, preuves etc...). Les résultats théoriques que nous avons obtenus (caractérisation des solutions et des solutions calculables, théorèmes sur les extensions de systèmes d'information, théorèmes sur les abstractions et représentations etc...) nous encouragent à penser que ce sont de telles études qui permettront de dégager les aspects essentiels de l'activité de construction des programmes et donc de la faciliter.

L'autre objectif de cette étude était de décrire des méthodes, des outils et des langages adaptés à la construction des programmes. Les principales idées mises en évidence :

- Utilité d'adopter une démarche déductive (et descendante),
- Intérêt de la décomposition Univers-Enoncé,
- Nécessité de procéder en plusieurs étapes résultant des oppositions statique-dynamique, spécification-résolution, solution-solution efficace,

nous ont permis d'esquisser *une méthode de construction de programmes*.

C'est ainsi que nous avons dissocié l'aspect spécification, où malheureusement les choix sont tellement nombreux qu'il n'a pas été possible de dégager de stratégies de spécification locales, de l'aspect explicitation où il apparaît clairement qu'un gros effort de structuration est d'orès et déjà possible.

Supportant cette méthode de construction, le langage que nous avons développé apparaît comme un outil de structuration des différentes versions d'un problème et sa sémantique statique nous ramène au cadre bien connu du

calcul des prédicats du premier ordre typé.

Enfin, il est utile de souligner l'intérêt, mais aussi la difficulté, de sortir d'exemples-jouets : dès qu'un problème atteint un degré de complexité "sérieux" sa spécification puis sa résolution deviennent très longs. Il suffit pour s'en convaincre d'étudier les quelques travaux s'appuyant sur de tels exemples ([BAL, 79b], [FEA, 79], [ABR, 77] etc...). Nous sommes cependant persuadé que c'est de l'étude de tels exemples que pourront être dégagés, à la fois, concepts fondamentaux, langages et méthodes (cf le problème des liaisons aériennes mettant en évidence le fait qu'une donnée peut être un S.I.).

2 - PROLONGEMENTS ET APPROFONDISSEMENTS

2.1 Etapes suivantes de la construction

a) Transformations au niveau statistique.

Nous avons signalé à plusieurs reprises que cet aspect est actuellement l'objet de nombreuses études, un effort important étant fait pour dégager des règles (sémantiques) de transformation ([ARS, 77], [ARS, 79], [BAU, 78b], [BAU, 79], [FEA, 79], [BEL, 79] etc...). Il convient de le compléter par l'introduction de *critères d'amélioration* [REM, 79]. Ainsi les différents travaux sur la complexité apparaissent comme un complément indispensable à ces recherches (en particulier parce qu'il peut être plus facile de mesurer l'efficacité d'un algorithme en restant au niveau statique plutôt qu'en travaillant à un niveau dynamique).

b) Passage au niveau dynamique.

L'utilisation de variables informatiques permet de nouvelles améliorations d'un algoritme. Le calcul d'une solution n'est plus considéré maintenant comme une suite de théorèmes dont le dernier est de la forme $r \equiv u$, mais comme une suite d'enrichissements successifs du S.I. initial S Di (on ajoute à S Di les différentes définitions de l'énoncé ε). Pour compléter la "théorie de construction des programmes" que nous avons esquissée, il faut donc introduire un cadre formel prenant en compte les notions dynamiques : on peut ainsi définir une *structure d'information* comme un couple formé d'un système d'information et d'un ensemble de *modifications* faisant passer d'un S.I. à un autre. Un programme peut alors être considéré comme un *ensemble de calculs* (un calcul est une suite de modifications). Les langages algorithmiques classiques permettent de décrire structures d'informations et ensembles de calculs. On trouve un développement de ce point de vue dans [PAI, 74], [FIN, 74], [PAI, 75], [FIN, 76a].

Ayant décrit le cadre "cible" il faut ensuite développer une méthode

systématisant le passage d'une version "statique" d'un problème à un programme. C'est, par exemple, à ce niveau que se posera la question de savoir si une suite va être représentée par une variable (cas où elle est définie par une récurrence "simple" $u_i = f(u_{i-1})$ ou par un tableau (récurrence multiple $u_i = f(u_{i-1}, u_{i-2} \ldots)$). De même il faut alors décider si un constructeur d'un S.I. va être représenté par une opération (schéma fonctionnel) ou par une modification de la structure cible. Quelques idées sur ces transformations sont émises dans [FIN, 78], [REM, 79] et [LES, 79].

2.2 Approfondissements théoriques

De nombreux points fondamentaux n'ayant été qu'effleurés dans cette thèse, indiquons brièvement quelques développements possibles :

- Généralisation de la notion de solution de problème au cas de résultats q-aires, voire au cas où le résultat serait un S.I.. Corrélativement étudier l'utilité théorique du concept d'énoncé strict : peut-on ramener toute explicitation de problème à l'explicitation de son S.I. ?

- L'introduction d'intermédiaires lors d'une explicitation (cf problème de l'évaluation d'une expression arithmétique, chapitre 1.4) semble être de même nature qu'une représentation. Peut-on ainsi ramener l'explicitation à une suite d'abstractions et de représentations ? Application à la stratégie d'introduction de suites intermédiaires.

- Le concept de pré-condition n'est pas suffisamment précisé. Il semble, à posteriori, important d'associer, dans un S.I., une pré-condition à chaque symbole fonctionnel alors que dans notre étude cette association apparait comme une conséquence des $\mathcal{S}$-descriptions.

- Les objets formels considérés (S.I., énoncés etc...) ne sont pas structurés, la structuration n'étant exprimée qu'au niveau des descriptions. L'introduction de S.I. "structurés" devrait permettre une étude théorique plus fructueuse.

- Nous nous sommes volontairement limité au 1er ordre. L'introduction de S.I. d'ordre supérieur permettrait certainement de lever certains problèmes liés à la coexistence de symboles q-aires et de symboles o-aires de la sorte $[\overline{S}_1 x \ldots x \overline{S}_q \rightarrow \overline{S}_{q+1}]$.

- Enfin, au niveau du détail, signalons qu'il aurait fallu distinguer le S.I. V^∞ des suites sur V éventuellement infinies, de V^* (suites finies sur V).

2.3 Approfondissements méthodologiques

a) Au niveau de la spécification, il faudrait essayer de définir des stratégies locales (relatives à certains domaines) et des règles de spécification, ceci en liaison avec les travaux sur les langues naturelles.

Les principales difficultés du problème de la spécification est la vérification de l'adéquation au problème posé. Une voie intéressante pourrait consister à *modéliser l'univers du problème*, comme le suggère JACKSON [JAC, 78], puis de décrire une sorte d'évaluation symbolique permettant de valider, ou plutôt d'invalider en cas d'échec, une spécification formelle.

Enfin, il faut affiner le langage des s-descriptions pour en faire un véritable instrument de communication avec l'utilisateur.

b) Au niveau de l'explicitation enfin, nous avons vu au chapitre 2.4 que tout, ou presque, restait à faire :

- Définir un catalogue structuré de statégies d'explicitation, chacune d'elles devant être caractérisée par des conditions d'application.

- Décrire un système permettant de gérer les différents choix d'explicitation qui serait à la base d'un système d'aide à l'explicitation.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[ABR, 77a] J.R. ABRIAL, *Manuel du Langage Z (Z/13)*, Rapport non publié, 1977.

[ABR, 77b] J. R. ABRIAL, *Mécanismes de Transformations du Langage Z (Z/14)*, Rapport non publié, 1977.

[ABR, 77c] J.R. ABRIAL, *Utilisation du Langage Z pour l'Analyse d'une Petite Application de Gestion (Z/15)*, Rapport non publié, 1977.

[ABR, 78] J.R. ABRIAL, *Z : A Specification Language*, IFIP, Tokyo, 1978.

[ABR, 79] J.R. ABRIAL, S.A. SCHUMAN, *Non Deterministic System Specification, in* Semantics of Concurrent Computation, KAHN (ed.), Lectures Notes in Computer Science 70, Springer Verlag, 1979.

[AHO, 72] A.V. AHO, J.D. ULLMAN, *The Theory of Parsing, Translation, and Compiling*, vol. 1 et vol. 2, Prentice-Hall, 1972-1973.

[AHO, 74] A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[ARS, 77] J. ARSAC, *La Construction de Programmes Structurés*, Dunod, 1977.

[ARS, 78] J. ARSAC, EXELMANS, *Un Système Interactif de Manipulation de Programmes EXEL*, Rapport 78-43, LITP, Paris, 1978.

[ARS, 79] J. ARSAC, Y. KODRATOFF, *Some Methods for Transformation of Recursives Procedures into Iterative Ones*, Rapport 79-2, LITP, Paris, 1979.

[ASH, 76] E.A. ASHCROFT, W.W. WADGE, *LUCID : A Formal System for Writing and Proving Programs*, SIAM J. Comput., 5, 3, 1976, p. 336-354.

[ASH, 77] E.A. ASHCROFT, W.W.WADGE, *LUCID : A Non Procedural Language with Iterations*, Comm. A.C.M., 20, 1977, p. 519-526.

[BAL, 76] R. BALZER et al., *The Project SAFE*, SAFE annual Report, 1976.

[BAL, 79a] R. BALZER, N. GOLDMAN, *Principles of Good Software Specification and Their Implications for Specifications Language, in* Proceedings of Specifications of Reliable Software, I.E.E.E. Catalog 79-CH 1401-9C, 1979.

[BAL, 79b] R. BALZER, *Transformational Implementation : an Example*, Internal Report, U S C Information Sciences Institute, Marina del Rey, Californie, 1979.

[BAN, 78] M. BANATRE, A. COUVERT, D. HERMAN, M. RAYNAL, *Présentation et Evaluation du Projet SOC : Un Système d'Objets Conservés Typés*, Actes du Congrès AFCET : Théorie et Techniques de l'Informatique, éditions Hommes et Techniques, Tome 1, 1978, p. 446-456.

[BAU, 78a] F.L. BAUER, P. PEPPER, H. WÖSSNER, *Notes on the Project CIP : Outline of a Transformation System, in* Program Construction, F.L. BAUER et M. BROY (ed.), Lecture notes in Computer Science 69, Springer Verlag, 1979.

[BAU, 78b] F.L. BAUER, M. BROY, H. PARTSCH, P. PEPPER, H. WÖSSNER, *Systematics of Transformation Rules*, TUM-INT-BER-77-12-0350/1, Institut für Informatik, Technische Universität München, München, 1978.

[BAU, 79] F.L. BAUER, H. WÖSSNER, *Algorithmic Language and Program Development*, London, Prentice-Hall, 1979.

[BEL, 78a] F. BELLEGARDE, J.P. FINANCE, B. HUC, J. JARAY, P. LESCANNE, J. MAROLDT, C. PAIR, A. QUERE, J.L. REMY, *MEDEE: A Type of Language for the Deductive Programming Method*, Conference on Reliable Software, German ACM Chapter, Bonn, 1978.

[BEL, 78b] F. BELLEGARDE, J.P. FINANCE, B. HUC, J.M. PIERREL, A. QUERE, J.L. REMY, *Initiation à une Construction Méthodique de Programmes*, Rapport CRIN 78-E-81.

[BEL, 78c] F. BELLEGARDE, J.P. FINANCE, B. HUC, J. JARAY, *Quelques Prolongements de la Méthode de Programmation Déductive*, GROPLAN n° 6, Bulletin du Groupe Programmation et Langages de l'AFCET, 1978.

[BEL, 78d] F. BELLEGARDE, *Recherche Systématique d'Erreurs dans un Programme Construit Déductivement*, Journées d'Etudes sur la Fiabilité des Programmes dans les Applications Industrielles, IRIA-EDF-BIGRE-Chapitre français de l'ACM, Clamart, 1978.

[BEL, 79] F. BELLEGARDE, A. QUERE, J.L. REMY, *Construction et Transformation Systématiques de Programmes*, Rapport CRIN 79-R-045, 1979.

[BEN, 79] J.L. BENTLEY, M. SHAW, *An ALPHARD Specification of a Correct and Efficient Transformation on Data Structure, in* Proceedings of Specification on Reliable Software, I.E.E.E. Catalog 79-CH-1401-9C, 1979.

[BER, 71] C. BERTHET, *Le Langage PL/1*, Dunod, 1971.

[BER, 77] D. BERT, P. JACQUET, *Generic Abstract Data Types, in* Implementation and Design of Algorithmic Languages, Proceedings of the 5th I.I.I. Conference, IRIA, 1977, p. 71-82.

[BER, 78] D. BERT, P. JACQUET, *Some Validation Problems with Parametized Types and Generic Functions, in* 3ème Colloque International sur la Programmation, Dunod, 1978, p. 279-292.

[BER, 79] D. BERT, *La Programmation Générique, Construction de Logiciel, Spécification Algébrique et Vérification*, Thèse d'Etat, Univ. Scientifique et Médicale de Grenoble, 1979.

[BID, 79] M. BIDOIT, C. GRESSE, G. GUIHO, *A System which Synthetizes Array-Manipulating Programs from Specifications*, I.J.C.A.I., Tokyo, 1979.

[BOH, 66] C. BÖHM, *CUCH as a Formal and Description Language, in* Formal Languages Description Languages, Proceedings of I.F.I.P. Congress 1964, STEEL (ed.), North Holland, 1966, p. 179-197.

[BRO, 78] M. BROY, R. GNATZ, M. WIRSING, *Semantics of Nondeterministic and Noncontinuous Constructs, in* Program Construction, F.L. BAUER et M. BROY (ed.), Lectures Notes in Computer Science 69, Springer Verlag, 1979.

[BRO, 79a] M. BROY, J.P. FINANCE, A. QUERE, J.L. REMY, M. WIRSING, *Methodical Solution of the Problem of Ascending Subsequences of Maximum Length within a Given Sequence,* Inf. Proc. Letters, Vol. 8, N° 5, 1979, p. 224-229.

[BRO, 79b] M. BROY, W. DOSCH, H. PARTSCH, P. PEPPER, M. WIRSING, *Existential Quantifiers in Abstract Data Types,* Sixth International Colloquium on Automata, Languages and Programming, Graz, Autriche, 1979.

[BUR, 77a] R.M. BURSTALL, J.A. GOGUEN, *Putting Specifications Together,* I.J.C.A.I., M.I.T., Cambridge, 1979.

[BUR, 77b] R.M. BURSTALL, J. DARLINGTON, *A Transformation System for Developing Recursive Programs,* J.A.C.M., Vol. 24, N° 1, 1977, p.44-67.

[CAD, 73] J.M. CADIOU, *Recursive Definitions of Partial Functions and their Computations,* Ph. D. Thesis, Comput. Sc. Department Standford University, 1973.

[CAR, 62] J.Mc CARTHY, P.W. ABRAHAMS, D.J. EDWARDS, T.P. HART, M.I. LEVIN, *LISP 1.5 Programmer's Manual,* M.I.T. Press, Cambridge, 1962.

[CHA, 73] C. CHANG, R.C. LEE, *Symbolic Logic and Mechanical Theorem Proving,* Academic Press, 1973.

[CHA, 74] D.D. CHAMBERLIN, R.F. BOYCE, SEQUEL, *a Structured English Query Language,* A.C.M. SIGFIDET Workshop, 1974.

[CLI, 73] M. CLINT, *Program Proving : Corontines,* Acta Informatica 2, 1973, p. 50-63.

[COD, 74] E.F. CODD, *Récent Investigations in Relational Data Base Systems, in* Proceedings of I.F.I.P. Congress 1974, Stockholm, 1974, p. 1017-1021.

[COL, 75] A.COLMERAUER, *Les Grammaires de Métamorphose,* Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

[COU, 74] J. COURTIN, J. VOIRON, *Introduction à l'Algorithmique et aux Structures de Données,* I.U.T.B. Grenoble, 1974.

[COU, 77] G. COUSINEAU, *Les Arbres à Feuilles Indicées : un Cadre Algébrique pour l'étude des Structures de Contrôle,* Thèse d'Etat, Univ. ParisVII, 1977.

[COU, 78] B. COURCELLE, M. NIVAT, *The Algebraic Semantics of Recursive Program Schemes,* Rapport de Recherche IRIA-LABORIA N° 300, 1978.

[CUN, 79] P.Y. CUNIN, *Fiabilité et Sécurité des Programmes : Propositions Autour d'un Langage d'Essai,* Thèse, Institut National Polytechnique de Lorraine, CRIN, 1979.

[DAH, 72] O.J. DAHL, E.W. DIJKSTRA, C.A.R. HOARE, *Structured Programming,* Academic Press, 1972.

[DAR, 78] J. DARLINGTON, *A Synthesis of Several Sorting Algorithms,* Acta Informatica 11, 1978, p. 1-30.

[DAR, 79] J. DARLINGTON, M. FEATHER, *A Transformational Approach to Modification,* Internal Report, 1979.

[DAT, 75] C.J. DATE, *Introduction to Database Systems,* Addison-Wesley, 1975.

[DAV, 79] M. DAVIS, J.T. SCHWARTZ, *Correct-Program Technology/Extensibility of Verifiers, in* Ecole de la Recherche, Techniques de Transformations et d'Optimisation de Programmes, IRIA, 1979.

[DEM, 79] M. DEMUYNCK, B. MEYER, *Les Langages de Spécification,* Journées Génie Logiciel, IRIA-SESORI, Pont-à-Mousson, 1979.

[DER, 74] J.C. DERNIAME, *Le Projet Civa - Un Système de Programmation Modulaire,* Thèse d'Etat, Univ. Nancy I, 1974.

[DER, 79] J.C. DERNIAME, J.P. FINANCE, *Types Abstraits de Données : Spécification, Utilisation et Réalisation,* Cours de l'Ecole d'Eté de l'AFCET, Monastir, Rapport CRIN, 79-E-57, 1979.

[DIJ, 76] E.W. DIJKSTRA, *A Discipline of Programming,* Prentice Hall, 1976.

[DIJ, 79a] E.W. DIJKSTRA, *Some Beautiful Argument Using Mathamatical Induction,* E.W.D. 697, Soumis à publication dans Acta Informatica, 1979.

[DIJ, 79b] E.W. DIJKSTRA, *A Theorem About Odd Powers of Odd Integers, in* Program Construction, F.L. BAUER et M. BROY (ed.), Lectures Notes in Computer Science 69, Springer Verlag, 1979.

[DIJ, 79c] E.W. DIJKSTRA, *In Honour of Fibonacci, in* Program Construction, F.L. BAUER et M. BROY (ed.), Lectures Notes in Computer Science 69, Springer Verlag, 1979.

[FAR, 78] M. FARAH, *Correct Translation of LUCID Programs, in* Program Transformations, B. ROBINET (ed.), 3ème Colloque International sur la Programmation, Dunod, 1978.

[FEA, 78] M.S. FEATHER, *Program Transformation Applied to the Telegram Problem, in* Program Transformations, B. ROBINET (ed.), 3ème Colloque International sur la Programmation, Dunod, 1978.

[FEA, 79] M.S. FEATHER, *A System for Developing Programs by Transformation,* Ph. D. Thesis, University of Edinburgh, 1979.

[FIN, 74] J.P. FINANCE, *Contribution à la Formalisation d'un Langage de Programmation, Application à Algol 68,* Thèse de Spécialité, Univ. de Nancy I, 1974.

[FIN, 76a] J.P. FINANCE, *Une Formalisation de la Semantique des Langages de Programmation,* RAIRO Informatique Théorique, Vol 10, N° 8, p. 5-32 et Vol 10, N° 10, p. 5-21, 1976.

[FIN, 76b] J.P. FINANCE, *Data Structure as a Framework to Formalize the Semantics of a Programming Language, in* Programmation, B. ROBINET (ed.) 2ème Colloque International sur la Programmation, Dunod, 1976.

[FIN, 77] J.P. FINANCE, *Formalisation de la Sémantique d'un Langage de Type Déclaratif,* GROPLAN 3, Bulletin du Groupe Programmation et Langages de l'AFCET, 1977.

[FIN, 78a] J.P. FINANCE, A. QUERE, *Construction et Preuve Simultanée d'un Programme dans le Cadre d'un Langage de Type Déclaratif : Problème du Tri Topologique,* Journées IRIA-SESORI sur la Construction, la Manipulation et la Transformation des Programmes, St Remy de Provence, 1978.

[FIN, 78b] J.P. FINANCE, *De la Spécification Abstraite d'une Donnée à sa Représentation en Mémoire : Les Etats Successifs d'une Information,* Actes du Congrès AFCET Théories et Techniques de l'Informatique, Editions Hommes et Techniques, Tome 1, 1978.

[FIN, 78c] J.P. FINANCE, *Static and Computational Semantics of a Definitional Language, in* Formal Description of Programming Concepts, E.J. NEUHOLD (ed.), North-Holland, 1978.

[FIN, 78d] J.P. FINANCE, P. LESCANNE, *Trois Approches Sémantiques d'un Langage Permettant l'Ecriture des Programmes Analysés Déductivement,* Actes du Congrès AFCET Théorie et Technique de l'Informatique, Editions Hommes et Techniques, Tome 1, 1978.

[FIN, 79] J.P. FINANCE, B. HUC, P. LESCANNE, C. PAIR, M. QUERE, J.L. REMY, *Programmation Déductive et Structures de Données,* Rapport CRIN 79-E-051, 1979.

[FLO, 67] R.W. FLOYD, *Assigning Meanings to Programs,* Proceedings of the Symposium in Applied Mathematics, Mathematical Aspect of Computer Science 19, J.T. SCHWARZ (ed.), American Mathematical Society, 1967, p.19-32.

[GAL, 78] M. GALINIER, A. MATHIS, *Les Machines Abstraites : Unités de Conception de Logiciel Fiable,* Congrès sur la Fiabilité des Programmes dans les Applications Industrielles, Chapitre Français de l'ACM, EDF-IRIA-BIGRE, 1978;

[GAM, 77] GROUPE GAMMA, *L'Ensemble ARIANE d'Aides à l'Analyse-Programmation,* 9 bis rue de Vezelay, Paris, 1977.

[GAU, 76] M.C. GAUDEL, *Structures de Données et Traduction des Langages de Programmation*, Rapport IRIA-LABORIA N° 193, 1976.

[GAU, 77] M.C. GAUDEL, *A Formal Approach to Translator Specification*, Congrès I.F.I.P. 77, Toronto, 1977.

[GAU, 78a] M.C. GAUDEL, G. TERRINE, *Synthèse de la Représentation d'un Type Abstrait par des Types "Concrets"*, Actes du Congrès AFCET Panorama de l'Informatique, Editions Hommes et Techniques, Tome 1, 1978.

[GAU, 78b] M.C. GAUDEL, C. PAIR, *Structures de Données et Algorithmes Fondamentaux*, IRIA, 1978.

[GER, 77] A. GERBIER, *Mes premières Constructions de Programmes*, Lectures Notes in Computer Science 55, Springer Verlag , 1977.

[GOG, 77a] J.A. GOGUEN, J.W. THATCHER, E.G. WAGNER, *An Initial Approach to the Specification, Correctness and Implementation of Abstract Data Types, in* Current Trends in Programming Methodology, Vol 3 Data Structuring, R.T. YEH (ed.),Prentice Hall , 1977.

[GOG, 77b] J.A. GOGUEN, *Abstract Errors for Abstract Data Type, in* Formal Description of Programming Concepts, E.J. NEUHOLD (ed.), North-Holland, 1977.

[GOG, 79] J.A. GOGUEN, J.J. TARDO, *An Introduction to OBJ : A Language for Writing and Testing Formal Algebraic Program Specifications ,in* Proceedings of Specification of Reliable Software, I.E.E.E. Catalog N° 79 CH-1401-96, 1979.

[GOO, 74] G. GOOS, J. HARTMANIS (ed.), *Compiler Construction - An Advanced Course*, Lecture Notes in Computer Science 21, Springer Verlag, 1974.

[GRE, 79] P. GREUSSAY, *LISP, Langage de Spécification ou Langage d'Implémentation ?*, Journées GROPLAN : Panorama des Langages d'Aujourd'hui, Cargèse, 1979.

[GRI, 76] M. GRIFFITHS, *Program Production by Successive Transformation, in* Language Hierarchies and Interfaces, F.L. BAUER et K. SAMELSON (ed.), Lecture Notes in Computer Science 46, Springer Verlag, 1976.

[GRI, 79] M. GRIFFITHS, *Development of the Schorr-Waite Algorithm, in* Program Construction, F.L. BAUER et M. BROY (ed.),Lecture Notes in Computer Science 69, Springer Verlag, 1979.

[GUT, 75] J.V. GUTTAG, *The Specification and Application to Programming of Abstract Data Types*, Ph. D. Thesis, CSRG TR 59, University of Toronto, 1975.

[GUT, 78a] J.V. GUTTAG, E. HOROWITZ, D.R. MUSSER, *Abstract Data Types and Software Validation*, Comm. A.C.M., 21, 12,1978, p. 1048-1063.

[GUT, 78b] J.V. GUTTAG, J.J. HORNING, *The Algebraic Specification of Abstract Data Types*, Acta Informatica 10, 1978, p. 27-52.

[GUT, 78c] J.V. GUTTAG, J. STAUNSTRUP, *Algebraic Axioms, Classes and Program Verification*, U.S.C. Report, 1978.

[GUT, 79] J.V. GUTTAG, *Note on Type Abstraction, in* Proceedings of Specification of Reliable Software, I.E.E.E. Catalog 79 CH-1401-9 C, 1979.

[HAM, 77] M. HAMMER, W.G. HOWE, V.J. KRUSKAL, I. WLADAWSKY, *A Very High Level Programming Language for Data Processing Applications*, Comm. A.C.M., 20, 11, 1977, p. 832-840.

[HEW, 72] C. HEWITT, *Description and Theoritical Analysis (using Schemata) of PLANNER : A Language for Proving Theorems and Manipulating Models in a Robot*, A.I. Memo N° 251, M.I.T., Project MAC, 1972.

[HOA, 69] C.A.R. HOARE, *An Axiomatic Basis of Computer Programming*, Comm. A.C.M. 12, 10, 1969, p. 576-580 et 583.

[HOA, 72] C.A.R. HOARE, *Proofs of Correctness of Data Representations*, Acta Informatica, Vol 1, N° 1, 1972, p. 271-281.

[HOG, 78] C.J. HOGGER, *Program Synthesis in Predicate Logic, in* Proceedings of the AISB/61, Conference on Artificiel Intelligence, Hamburg, 1978.

[HUC, 77] B. HUC, *Mise en oeuvre de la Méthode de Programmation Déductive,* Thèse de Docteur Ingénieur, Univ. de Nancy I, 1977.

[HUE, 77] G. HUET, *Confluent Réductions : Abstract Properties and Applications to Term Rewriting Systems, in* Proceedings of the 18th Annual I.E.E.E. Symposium on Foundations of Computer Science, 1977, p. 30-45.

[HUG, 79] J.W. HUGHES, *A Formalization and Explication of the MICHAEL JACKSON Method of Program Design,* Software - Pratice and Experience, Vol 9, 1979, p. 191-202.

[IBM, 75] IBM, Education Commerciale Ordinateurs, *GAP II Principes de Programmation, Figures,* 1975.

[JAC, 75] M.A. JACKSON, *Principles of Program Design,* Academic Press, 1975.

[JAK, 78] M.A. JACKSON, *Information Systems : Modelling Sequencing and Transformations,* 3d Conference on Software Engeenering, 1978.

[JAC, 78] P. JACQUET, *La Généricité comme Outil d'Abstraction dans les Langages de Programmation,* Congrès AFCET Panorama de l'Informatique, Editions Hommes et Techniques, tome 1, 1978.

[JOU, 77] J.P. JOUANNAUD, *Sur l'Infèrence et la Synthèse Automatique de Fonctions LISP à partir d'Exemples,* Thèse d'Etat, Univ. de Paris VI, 1977.

[JOU, 78] J.P. JOUANNAUD, Y. KODRATOFF, *Synthèse Automatique de Programmes à Partir d'Exemples,* Journées IRIA-SESORI sur la Synthèse, la Manipulation et la Transformation de Programmes, ST Remy de Provence, 1978.

[KAH, 77] G. KAHN, G. HUET, P. MAURICE, *Environnement de Programmation PASCAL,* Manuel d'Utilisation sous SIRIS 7/8, IRIA, 1977.

[KEN, 75] K. KENNEDY, J. SCHWARTZ, *An Introduction to the Set Theoritic Language SETL,* Computers & Mathematics with Application, 1, 97, 1975.

[KIN, 76] J.C. KING, *Symbolic Execution and Program Testing,* Comm. A.C.M., 19, 7, 1976.

[KLE, 52] S.C. KLEENE, *Introduction to Metamathematics,* Van Nostrand Compagny INC, New-York, 1952.

[KNU, 68] D.E. KNUTH, *The Art of Programming,* Addison-Wesley, 1978.

[KNU, 70] D.E. KNUTH, P. B. BENDIX, *Simple Word Problems in Universal Algebras, in* Computational Problems in Abstracts Algebras, J. LEECH (ed.), Pergamon Press, 1970, p. 263-297.

[KOD, 78] Y. KODRATOFF, *Choix d'un Programme LISP Correspondant à des Exemples,* Congrès AFCET-IRIA Reconnaissance des Formes, 1978.

[KOW, 74] R. KOWALSKI, *Logic for Problem Solving,* Memo N° 75, University of Edinburgh, 1974.

[KRI, 72] J.L. KRIVINE, *Théorie Axiomatique des Ensembles,* Presses Universitaires de France, Collection SUP, 1972.

[LAC, 77] M. LACROIX, A. PIROTTE, *ILL : An English Structured Query Language for Relational Data Bases, in* Architecture and Models in Data Base Management Systems, G.M. NIJSSEN (ed.), North Holland, 1977.

[LAM, 76] B.W. LAMPSON et al., *EUCLID Report,* Xerox Research Center, Palo Alto et SIGPLAN Notices 12, 1977.

[LAM, 79] A. Van LAMSWEERDE, *Programming Systems for non-Programmers : Some Wishes and some Attempts,* Rapport R 390, M.B.L.E., Bruxelles, 1979.

[LAU, 68] P. LAUER, P. LUCAS, H. STIGLEITNER, *Method and Notation for the Formal Definition of Programming Language,* Technical Report TR 25. 087, IBM Laboratory Vienna, 1968.

[LAU, 76] J.L. LAURIERE, *Un Langage et un Programme pour Enoncer et Résoudre des Problèmes Combinatoires*, Thèse d'Etat, Univ. de Paris VI, 1976.

[LEH, 77] D.J. LEHMAN, M.B. SMYTH, *Data Types*, *in* Proceedings of 18th I.E.E.E. Symposium on Foundation of Computer Sciences, 1977, p. 7-12.

[LES, 78] P. LESCANNE, *Semantique d'un Langage Adapté à la Construction Déductive des Programmes*, CRIN Rapport 78-R-012.

[LES, 79] P. LESCANNE, *Etude Algèbrique et Relationnelle des types abstraits et de leur Représentation*, Thèse d'Etat, INPL, Nancy, 1979.

[LIS, 74] B.H. LISKOV, S.N. ZILLES, *Programming with Abstract Data Types*, SIGPLAN Notices 9, 4, 1974.

[LIS, 75] B.H. LISKOV, *An Introduction to C L U*, *in* New Directions in Algorithmic Languages, S.A. SCHUMAN (ed.), IRIA, 1975.

[LIS, 77] B.H. LISKOV, S.N. ZILLES, *An Introduction to Formal Specifications Data Abstractions*, *in* Current Trends in Programming Methodology, R.T. YEH (ed.), Vol 1, Prentice Hall, 1977.

[LIV, 78] C. LIVERCY, *Théorie des Programmes*, Dunod, 1978.

[LON, 76] R.L. LONDON, M. SHAW, W.A. WULF, *Abstraction and Verification in ALPHARD : A Symbol Table Exemple*, U.S.C. Report, 1976.

[LON, 78] R.L. LONDON, J.V. GUTTAG, J.J. HORNING, B.W. LAMPSON, J.G. MITCHELL, G.J. POPEK, *Proof Rules for the Programming Language EUCLID*, Acta Informatica 10, 1, 1978, p. 1-26.

[LOR, 74] B. LORHO, *De la Définition à la Traduction des Langages de Programmation : Méthode des Attributs Semantiques*, Thèse d'Etat, Univ. Paul Sabatier, Toulouse, 1974.

[LUC, 77] M. LUCAS, P.C. SCHOLL, J. VOIRON, *Apprentissage et Utilisation du Traitement Séquentiel pour la Construction de Programmes*, Rapport de Recherche RR 74, Laboratoire IMAG-USMG, Grenoble, 1977.

[MAJ, 77] M.E. MAJSTER, *Limits of the "Algebraic" Specification of Abstract Data Types*, SIGPLAN Notices, 10, 12, 1977.

[MAN, 71] Z. MANNA, R.J. WALDINGER, *Toward Automatic Program Synthesis*, Comm. A.C.M., 14, 3, 1971, p. 151-165.

[MAN, 74] Z. MANNA, *Mathematical Theory of Computation*, Mc Graw-Hill, 1974.

[MAN, 75] Z. MANNA, R.J. WALDINGER, *Knowledge Reasoning in Program Synthesis*, Artificial Intelligence, 1975.

[MAN, 78] Z. MANNA, R.J. WALDINGER, *A Deductive Approach to Program Synthesis*, Technical Notes 177, S.R.I. International, 1978.

[MAR, 77] J.J. MARTIN, *Critique of Mila E. MAJSTER'S Paper "Limits of the Algebraic Specification of Abstract Data Types"*, SIGPLAN Notices, 12, 12, 1977.

[MEY, 78] B. MEYER, C. BAUDOIN, *Méthodes de Programmation*, EYROLLES, Collection DER-EDF, 1978.

[MIL, 76] R. MILNER, *Program Semantics and Mechanized Proof*, *in* Foundations of Computer Science II, K.R. APT et J.W. de BAKKER (ed.) Mathematical Centre Tracts 82, Mathematisch Centrum, Amsterdam, 1976, p. 3-44.

[MIN, 79] R. MINOT, *ATM: Un Système de Fabrication de Programmes Basé sur Les Concepts de Modularité et de Type Abstrait*, Thèse de Troisième Cycle, Univ. de Nancy I, 1979.

[MUS, 78] D. MUSSER, *Convergent Sets of Rewrite Rules for Abstract Data Types*, U.S.C. Information Science Institute, 1978.

[MUS, 79] D. MUSSER, *Abstract Data Type Specification in the AFFIRM System, in* Proceedings of Specification of Reliable Software, I.E.E.E. Catalog 79 CH-1401-96, 1979.

[NAK, 78] R. NAKAJIMA, M. HONDA, H. NAKAHARA, *Describing and Verifying Programs with Abstract Data Types, in* Formal Description of Programming Concepts, E.J. NEUHOLD (ed.), North-Holland, 1978.

[NAK, 79] R. NAKAJIMA, H. NAKAHARA, M. HONDA, *Hierarchical Program Specification and Verification - A Many - Sorted Logical Approach,* RIMS Preprint 265, 1979.

[NIV, 75] M. NIVAT, *On the Interpretation of Polyadic Recursive Schemes,* Symp. Mathematica, 15, Academic Press, 1975.

[PAI, 71] C. PAIR, *Les Structures d'Information,* Cours de l'Ecole d'Eté de l'AFCET, Alès, 1971.

[PAI, 74] C. PAIR, *Formalization of the Notions of Data, Information and Information Structure, in* Data Base Management Systems, Klimbie-Hoffeman (ed.), North Holland, 1974, p. 149-167.

[PAI, 75] C. PAIR, *Du Problème à sa Solution,* Journées IRIA-SESORI : Logique et Programmation, Le Bischenberg, 1975.

[PAI, 76] C. PAIR, J. MAROLDT, *Introduction à une Méthode de Programmation Déductive,* INPL, Nancy, 1975.

[PAI, 77] C. PAIR, *Mise en Evidence de l'Ensemble de Départ dans les Itérations en Programmation Déductive,* A TOUT CRIN 6, Bulletin de Liaison du Centre de Recherche en Informatique de Nancy, 1977.

[PAI, 79] C. PAIR, *La Construction des Programmes,* RAIRO Informatique, Vol 13, 2, 1979, p. 113-118.

[PAG, 79] B. PAIGE, J.T. SCHWARTZ, *Expression Continuity and the Formal Differentiation of Algorithms,* Ecole de la Recherche, Techniques de Transformations et d'Optimisation de Programmes, IRIA, 1979.

[PAR, 77] D.L. PARNAS, *The Use of Precise Specifications in the Development of Software, in* Proceedings of I.F.I.P. Congress 77, Toronto, North-Holland, 1977.

[PIR, 76] A. PIROTTE, *Explicit Description of Entities and their Manipulation in Languages for the Relational Data Base Model,* Thèse, Univ. Libre de Bruxelles et Rapport R 336 M.B.L.E., Bruxelles, 1976.

[PIR, 77] A. PIROTTE, *High Level Data Base Query Languages,* International Seminar on Intelligent Question - Answering and Data Base Systems, Bonas, 1977.

[PRY, 77] N.S. PRYWES, *Automatic Generation of Computer Programs,* NCC1 1977, A.F.I.P.S. Press, Montvale, N.J., 1977, p.679-689.

[REM, 74] J.L. REMY, *Structure d'Information, Formalisation des Notions d'Accès et de Modification d'une Donnée,* Thèse de Specialité, Univ. Nancy I, 1974.

[REM, 78] J.L. REMY, *Un Algorithme de Calcul des Distances dans un Graphe, Utilisant la Décomposition en Composantes,* Rapport CRIN 78-P-043, 1978.

[REM, 79] J.L. REMY, *Construction, Evaluation et Amélioration Systématique des Structures de Données,* RAIRO Informatique Théorique 1979 (à Paraître).

[REY, 79] J.C. REYNOLDS, *Reasoning about Arrays,* Comm. A.C.M., 22, 5, 1979, p. 290-299.

[ROB, 65] J.A. ROBINSON, *A Machine Oriented Logic Based on the Resolution Principle,* J.A.C.M. 12, N°1, 1965, p. 23-41.

[ROU, 75] P. ROUSSEL, *PROLOG, Manuel de Référence et d'Utilisation,* Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

[SCH, 75] J.T. SCHWARTZ, *On Programming : an Interim Report on the SETL Project,* Courant Institute of Mathematical Sciences, New-York University, 1975.

[SCH, 79] P.C. SCHOLL, *Vers une Programmation Systématique : Etude de Quelques Méthodes, Techniques et Outils*, Thèse d'Etat, USMG et INPG, Grenoble, 1979.

[SCO, 71] D. SCOTT, C. STRACHEY, *Toward a Mathematical Semantics for Computer Languages, in* Proceedings of the Symposium on Computer and Automata, J. FOX (ed.), Polytechnic Institute of Brooklyn Press, 1971, p. 19-46.

[SCO, 72] D. SCOTT, *Data Types as Lattices*, Cours de l'Ecole d'Eté d'Amsterdam, 1972.

[SHO, 67] J.R. SHONFIELD, *Mathematical Logic*, Addison-Wesley, 1967.

[SIN, 79] M. SINTZOFF, *Composing and Specifying Program Design Decisions*, Working Notes (Communication Privée), 1979.

[SIS, 78] Société d'Informatique et de Systèmes, *Le Système d'Analyse PROTEE*, Tour Neptune,Paris La Défense, 1978.

[SIM, 79] M. SIMONET, *Transcription en PROLOG de la W-Grammaire d'un Langage Simple*, GROPLAN 7, Bulletin du Groupe Programmation et Langages de l'AFCET, 1979.

[SPI, 75] J. SPITZEN, B. WEGBREIT, *The Verification and Synthesis of Data Structures*, Acta Informatica 4, 1975, p. 127-144.

[TER, 79] G. TERRINE, *Types Abstraits Source et Objet → Implementation ?* GROPLAN 7, Bulletin du Groupe Programmation et Langages de l'AFCET, 1979.

[VUI, 74] J. VUILLEMIN, *Syntaxe, Semantique et Axiomatique d'un Langage de Programmation Simple*, Thèse d'Etat, Univ. de Paris VII, 1974.

[WAL, 69] R.J. WALDINGER, R.C.T. LEE, *PROW : A Step Toward Automatic Program Writing, in* Proceeding of the International Joint Conference on Artificial Intelligence, Washington, 1969, p. 241-252.

[WAR, 75] J.D. WARNIER, *Les Procédures de Traitement et leurs Données*, Editions d'Organisation, 1975.

[WAR, 77a] D.H.D. WARREN, L.M. PEREIRA, *PROLOG : The Language and its Implementation Compared with LISP*, Proceedings of the A.C.M. Symposium on A.I. and Programming Languages, Rochester, N.Y.,1977.

[WAR, 77b] D.H.D. WARREN, *Logic Programming and Compiler Writing*, D.A.I. Research Report N° 44, 1977.

[WAS, 79] A.I. WASSERMAN, S.K. STINSON, *A Specification Method for Interactive Information Systems, in* Proceedings of Specification of Reliable Software, I.E.E.E. Catalog N° 79 CH -1401-9 C, 1979.

[WEG, 75] B. WEGBREIT, *Goal-Directed Program Transformation*, Rapport CSL-75-8, Xérox Palo Alto Research Center, 1975.

[WEI, 67] C. WEISSMAN, *LIPS 1.5 Primer*, Dickenson Publishing Co. 1967.

[WIN, 75] A. Van WIJNGAARDEN et al, *Revised Report on the Algorithmic Language ALGOL 68*, Acta Informatica, 5, 1975.

[WIN, 78] T. WINOGRAD, *Why Programming Languages are Obsolete*, Unpublished Draft, 1978.

[WIR, 71] N. WIRTH, *The Programming Language PASCAL*, Acta Informatica, 1, 1, 1971, p. 35-63.

[WIR, 73] N. WIRTH, *Systematic Programming : an Introduction*, Prentice Hall Series in Automatic Computation, 1973.

[WIR, 76] N. WIRTH, *Algorithms + Data Structures = Programs*, Prentice Hall series in Automatic Computation, 1976.

[WUL, 75] W.A. WULF, R.L. LONDON, M. SHAW, *Abstraction and Verification in ALPHARD, in* New Directions in Algorithmic Languages, S.A. SCHUMAN (ed.) IRIA, 1975.

[ZIL, 75] S.N. ZILLES, *Abstract Specifications for Data Types*, IBM Research Laboratory, San José, 1975.

NOM DE L'ETUDIANT : FINANCE Jean-Pierre

NATURE DE LA THESE : Doctorat ès Sciences

VU, APPROUVE

et PERMIS D'IMPRIMER

NANCY LE 21 SEP 1979 79 5

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITE DE NANCY I

LE PRESIDENT

M. BOULANGE

UNIVERSITE DE NANCY I

UER DE MATHEMATIQUES

ETUDE DE LA CONSTRUCTION DES PROGRAMMES : méthodes et langages de Spécification et de résolution de problèmes



THESE

Présentée à l'UNIVERSITE DE NANCY I

Pour l'obtention du grade de

DOCTEUR ES SCIENCES MATHEMATIQUES

Par

Jean-Pierre FINANCE

Soutenue le 11 OCTOBRE 1979

Devant la commission d'examen

RAPPORTEURS	C. PAIR	PRESIDENT
	J.P. JOUANNAUD	
	M. SINTZOFF	
EXAMINATEURS	J. ARSAC	
	J.C. DERNIAME	
	M. GRIFFITHS	
	J.P. HATON	
	M. NIVAT	