BIBLIOTHEQUE INTERUNIVERSITAIRE
Section Sciences
54600 VILLERS-LES-NANCY

Institut National Polytechnique
de Lorraine

Centre de Recherche en Informatique
de Nancy

Sc N 84
B
378

THÈSE

présentée
à l' INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE
pour le titre de
DOCTEUR-INGENIEUR EN INFORMATIQUE
par

Loutfi SOUFI

BIBLIOTHEQUE-SCIENCES UNIVERSITE NANCY

Un système d'aide à la construction de programmes itératifs

Soutenue le 11 décembre 1984

Service Commun de la Documentation
INPL
Nancy-Brabois

Composition du jury :

Président : J.P. FINANCE

Examinateurs : D. COULON
A. VAN LAMSWEERDE
P. LESCANNE
A. QUERE
P.C. SCHOLL

D 136 037467 5

1360374670

[M] 1984 SOOFI, L.

Je remercie tous ceux qui ont contribué directement ou non à ce travail.

Je remercie Axel Van LAMSWEERDE, Pierre-Claude SCHOLL et Daniel COULON qui ont accepté de venir juger ce travail.

Je remercie vivement Alain QUÉRÉ pour les conseils qu'il a su me prodiguer et pour sa lecture attentive de certains papiers.

Pierre LESCANNE me fait le plaisir de siéger dans ce jury. Je l'en remercie ainsi que de ses critiques constructives, des fructueuses suggestions dont il m'a fait bénéficier et aussi de son apport au niveau de la documentation.

Quant à Jean-Pierre FINANCE "je lui dois tout". Il m'a initié au travail de recherche. Il m'a fait profité de sa compétence et d'un certain savoir faire scientifique.
Sa patience, son soutien, ses encouragements, ses critiques aux différentes étapes de développement de ce travail ont été inestimables.
Aujourd'hui je lui dis tout simplement : merci.

J'adresserai mes remerciements au Greco Programmation pour son apport technique ayant permis de réaliser le système.

Je remercie Nadine Gautier qui a assuré la frappe de cette thèse avec gentillesse et conscience, et qui a supporté avec patience les fastidieuses corrections.

SOMMAIRE

INTRODUCTION

I. Enoncé du problème et quelques idées générales 1

II. Spes et l'objectif de notre travail 4

II.1. Objectif de notre travail 6

III. Plan de la thèse 7

CHAPITRE I ROLES ET FONDEMENTS DU SYSTEME 9

I.1. Les idées à la base du Système Interactif d'Explicitation 9

I.1.1. Objectif de notre travail 9

I.1.2. Idées sur la solution 10

* Présentation de la solution 10

* Justification de la solution 11

I.2. Caractérisation de l'Explicitation 13

I.2.1. Notion d'explicitation 13

I.2.2. Définition des énoncés 14

* Exemples de problèmes 15

* Notion de problèmes 17

I.2.2.1. Les énoncés implicites : le langage Spes 21

I.2.2.2. Les énoncés récurrents : le langage Médée 24

I.2.3. Fondement du Système Interactif d'Explicitation 28

I.3. Analyse générale du problème de l'explicitation d'un énoncé 29

I.3.1. La méthode de programmation déductive 29

I.3.2. Analyse de la résolution 31

I.4. Structure générale du système 34

CHAPITRE II UNE RESOLUTION DU PROBLEME DE L'EXPLICITATION D'ENONCES 37

II.1. Les règles de preuves 38

II.1.1. Définition d'une règle de preuve 38

II.1.2. Exemples d'application des règles de preuves 42

II.2. Les règles de constructions 48
II.2.2. Correspondance règles de constructions-règles de preuves 49
II.2.3. La validité d'une règle de construction 51
II.3. Quelques classes de règles 52
II.3.1. La classe des règles non-terminales 53
II.3.2. La classe des règles logiques 54
II.3.3. La classe des règles terminales 56
II.3.4. La classe des règles valeurs 58
II.4. Mise en oeuvre des règles de constructions : la technique de remplacement 59
II.4.1. Une idée générale de la technique 59
* Définition de la transformation 60
* Un exemple de calculs de filtres 60
II.4.2. Les arbres 62
II.4.2.1. Définitions et arbres bien formés 62
a) Définition d'un arbre 63
b) Définition d'un arbre attribué 66
II.4.2.2. Sous-arbre et filtre 68
II.4.3. La technique de remplacement 70
II.4.3.1. Le mécanisme de la transformation : un exemple 70
II.4.3.2. Définitions des filtres σ et σ' 73
II.4.3.3. La règle générale de remplacement 77
II.4.3.4. Un exemple de calcul des filtres σ et σ' 77
II.4.4. Conditions d'application de transformations 80
II.5. Des règles aux stratégies 82
II.5.1. Une introduction aux stratégies d'explicitations 82
II.5.2. Les stratégies d'explicitations locales 82
II.5.3. Exemples de stratégies locales 84
II.5.4. Les stratégies d'explicitations globales 87
II.6. Le problème du choix dans l'explicitation 88
* Discussion sur les décisions 88
* L'aide à la décision 90
II.7. Un exemple technique de transformation 92
II.8. Conclusion 97

CHAPITRE III STRATEGIES ET META-ALGORITHME D'EXPLICITATIONS 99
III.1. Le problème de la découverte de stratégies 100
III.1.1. L'identification de stratégies 101
III.1.1.1. Les stratégies techniques et opérationnelles 101
III.1.1.2. Une vue sur la construction intuitive de stratégies 102
III.1.2. Quelques stratégies d'explicitations globales 109
III.1.2.1. Stratégie d'explicitation par affaiblissement 110
III.1.2.2. Stratégie d'explicitation d'initialisation 113
III.1.2.3. Stratégie d'analyse par cas 114
III.1.2.4. Stratégie d'explicitation de modulation 116
III.1.3. Pour un système de construction de stratégies 118
III.2. Méta-algorithme d'explicitation 123
III.3. Exemples d'explicitations 125
III.4. Conclusion 143

CHAPITRE IV LE SYSTEME D'EXPLICITATION
IV.1. Présentation générale 144
IV.2. Les connaissances du système 148
IV.2.1. Le catalogue des stratégies d'explicitations locales 148
IV.2.1.1. Les stratégies locales 148
IV.2.1.2. La bibliothèque. Les fonctions attendues 149
IV.2.2. Le catalogue des stratégies d'explicitations globales 153
IV.2.2.1. La mise en oeuvre des stratégies globales 153
a) une mise en oeuvre de <sea> 153
b) une mise en oeuvre de <sei> 155
c) une mise en oeuvre de <sac> 157
d) une mise en oeuvre de <sem> 161
IV.2.2.2. La bibliothèque des stratégies globales 163
IV.2.3. L'arbre des choix 163
IV.3. Les modules fonctionnels du système 166
IV.3.1. L'interface programmeur/SIE 166
IV.3.2. Le mode édition 168

IV.3.3. Le mode connaissance 170
IV.3.4. Le mode résolution 174
IV.3.4.1. Le module de sélection de l'identificateur 176
IV.3.4.2. Le module de sélection de la stratégie à appliquer 176
IV.3.4.3. Le module d'identification 177
IV.3.4.4. Le module de transformation 180
IV.4. Les développements ultérieurs du système 183

CHAPITRE V UNE APPLICATION : LE TYPE TABLEAU 185
V.1. Spécification du type abstrait TABLEAU 185
* Définition de la signature Σ 186
* Les axiomes du type TABLEAU 192
* Une spécification du type INTERVALLE D'ENTIER 195
V.2. Exemple d'énoncés sur le type TABLEAU 197
V.3. Fonctionnement du système 201

CHAPITRE VI SIE ET LES SYSTEMES TRANSFORMATIONNELS 209
VI.1. Travaux sur les systèmes de construction de programmes 209
VI.2. Présentation de quelques systèmes transformationnels 211
VI.2.1. Le systèmes PECOS 212
VI.2.2. Le système ZAP 216
VI.2.3. Le système CATY 221
VI.2.4. Le système DEDALUS 226
VI.2.5. Le projet CIP 229
VI.3. Conclusion 232

CONCLUSION 234
I. Résultats 234
II. Notes sur la réalisation 235
III. Prolongements : Vers un Système Automatique d'explicitation 236

BIBLIOGRAPHIE

ANNEXES

A. Un exemple d'application de règles de constructions
B. Grammaire Spes utilisée
C. Une illustration du mode connaissance
D. Visualisation de l'arbre des choix et des stratégies
E. Un exemple d'une session de travail
F. Algorithmes de filtrage et de transformation

INTRODUCTION

INTRODUCTION

I. ENONCE DU PROBLEME ET QUELQUES IDEES GENERALES

Les problèmes en programmation peuvent se situer d'une part,
au moment de la construction : ce sont les problèmes de corrections et d'efficacité de programmes,
et d'autre part,
dans la vie ultérieure de programmes : c'est le problème d'évolutivité ou de maintenance de programmes.
Pour y remédier il s'est avéré nécessaire de mieux comprendre, mieux maîtriser et bien définir le "phénomène programmation" [LAM-83, ZEL-79], le processus de développement de programmes [ARS-77, BAR-79, BAU-82, FLO-67, GUI-83, PAI-78, WIR-71, JON-80].

Et cela a conduit aux études sur :

1. La spécification de problèmes à résoudre, c'est-à-dire que l'on cherche à exprimer un problème de façon précise, compréhensible et non ambigue [WIN-84, LIS-75, GUT - 83, MEY-80, DUB-84, FIN-79],

2. La construction de programmes, c'est-à-dire que l'on cherche à définir le passage d'une spécification à un programme. [DIJ-76, LIV-78, MAN-78, WIR-74, FIN-79, GUI-80].
Notre travail appartient aux secondes études.

Ces études montrent que pour spécifier et pour construire il est nécessaire de définir des méthodes et des outils :
la programmation transformationnelle [BAU-76, BAU-79, DAR-83, BROY-83, FIN-79, LOV-77, MAN-79, MAN-80 , PAG-83, PAR-83], est une façon de préciser le processus de développement de programmes.

C'est l'approche que nous avons choisie.

L'approche transformationnelle consiste à obtenir un programme correct à partir d'une spécification d'un problème ou de son résultat par l'application répétée de règles de transformations garantissant que la version finale obtenue est sémantiquement correcte (dans le sens de BACK [BAK-80]) vis-à-vis de la spécification initiale.

Ainsi la programmation transformationnelle qui se définissait comme une méthode d'optimisation de programmes corrects mais inefficaces, est devenue depuis quelques années « une méthode de construction de programmes par applications successives de règles de transformations» [PAR-83].
Cette méthode est alors mise en oeuvre par un système de transformation.

Par exemple l'étude sur la programmation automatique proposée par E. CHEATHAM et WEGBREIT [CHE-72] consistait déjà en une méthode de développement de programme par transformation, guidée par le souci d'obtenir un programme correct et efficace.
Cette étude s'appuie sur les idées de la programmation descendante comme exprimée par E.W. DIJKSTRA [DIJ-76] et WIRTH [WIR-71, WIR-73], et sur une méthode de preuve [FLO-67, HOA-69] permettant d'assurer la correction de la transformation.
Ces deux méthodes sont mises en oeuvre par un système interactif doté d'un catalogue de règles de transformations et composé d'un identifieur de forme, d'un mécanisme d'exploration d'arbre avec retour en arrière, et d'un analyseur de mesures de performances d'un programme.
Ce système hypothétique a en entrée une spécification algorithmique et produit un programme correct et efficace vis-à-vis de cette spécification.

On peut caractériser une étude sur la programmation transformationnelle par les trois points suivants :

1. une étude théorique de l'approche transformationnelle, et notamment sur la notion de correction de la transformation [ASH-75, BAU-79b, HUE-77, KOT-78, SRI-84].

2. l'étude de règles de transformations [BAR-79, BRO-81, BUC-77, CHA-77, SCW-78, WAN-80, BEL-84, GER-83].

3. la définition de systèmes de transformations [BAR-79, BART-81, BID-80, SOU-82, GRE-84, MAN-79].

Le premier point aborde les aspects de définition d'un système formel permettant de préciser les propriétés des objets d'un problème, et de méthodes de preuve, de substitution permettant ainsi de prouver formellement dans un certain système formel la correction de la transformation.

Le second point signifie que l'on veut exprimer des outils permettant la construction progressive de programmes corrects. On veut décrire le passage d'une étape à une autre par des "règles de transformations". Ces règles sont supposées valides. Ainsi leur application a un énoncé d'un problème permet d'obtenir un nouvel énoncé correct sémantiquement vis-à-vis du précédent. Ces règles sont donc utilisées comme une méthode de preuve.
Elles sont représentées soit sous forme algorithmique soit sous forme de "schéma de transformation" (composé d'une partie "forme", d'une partie "résultat" et muni d'une condition d'applicabilité).

Et de nos jours le problème le plus important, en programmation transformationnelle, est la découverte de telles règles.
On peut les classer de la façon suivante :

a) les règles de programmation (introduction d'une itération, ...)
b) les règles propres à une classe de problème (propriétés d'un domaine)
c) les règles du calcul formel (le calcul propositionnel, ...)
d) les règles générales de transformation (pliage, dépliage, des techniques d'induction).

Le dernier point aborde le problème de la systématisation du passage d'une spécification initiale à un programme par application de règles de transformation. Il s'agit de définir un logiciel mettant en oeuvre une méthode formelle de transformation.

Les systèmes de transformation se différentient alors essentiellement par la forme de l'entrée (algorithme, programme, spécification non algoritmique, spécification d'un type abstrait) et le but poursuivi (modification de programmes, orienté vers l'efficacité, vers la correction, la translation de programmes).

Quels quesoient la forme de l'entrée et l'objectif du système, tous ces systèmes ont en commun deux choses :
- la première est que toute spécification de départ est "complète",
- la seconde est l'aspect "constructif" de la version finale.

Il faut noter alors qu'en plus des systèmes référencés plus haut, les études sur l'évolution de programmes [DERS-77], de réécriture [DERS-82, LES-83], de synthèses de programmes à partir d'une spécification [MAN -71] ou à partir d'exemples [BIE-76], les générateurs de programmes [PRY-77] et les éditeurs structurés [DON-80] concernent également la programmation transformationnelle.
On peut remarquer que le système de vérification de S. IGARASHI et Cie [IGA-73] procède également par transformation d'assertions.
Pour se faire une idée générale de ce que peut recouvrir la programmation transformationnelle on renvoie le lecteur à la comparaison de deux exemples d'application [FIN-77] et [BAL-81] le premier orienté vers la "correction" et le second vers "l'implémentation".

On s'intéresse uniquement, de façon pratique, aux deux derniers points cités précédemment, (les règles et les systèmes) en s'attachant plus particulièrement à la définition d'un système de transformation.

II. SPES ET L'OBJECTIF DE NOTRE TRAVAIL

Spes est un environnement de spécification en cours de développement à Nancy.
Les fondements de cet environnement sont l'objet de la thèse du professeur J.P. FINANCE [FIN-79]. Dans sa thèse, Finance, a précisé une méthodologie de développement de programme que l'on peut intuitivement décrire de la façon suivante :

en se plaçant dans le système formel du calcul du prédicat du premier ordre fortement typé, on cherche à spécifier des problèmes à résoudre.
Cette spécification se construit de manière "déductive" : elle exprime le résultat du problème en utilisant des intermédiaires.
La spécification d'un problème ainsi obtenue, n'est pas nécessairement calculable : elle est statique et implicite.
On cherche alors à la rendre calculable en la transformant par l'application de stratégies d'explicitation préservant la correction.
On obtient alors, et aussi de manière déductive, une spécification explicite et statique.
Cette dernière spécification est enfin traduite en une autre plus efficace et plus dynamique prête à être interprétée ou traduite facilement en un programme.
En outre le processus de transformation d'une spécification implicite peut se faire en séparant distinctivement le processus de transformation de types de celui des énoncés.

Issu de cette méthodologie, le projet SPES [FIN-84], animé par A. Quéré, a donc pour objectif la réalisation de l'environnement de spécification schématisé par la figure 0.1, ci-dessous, donnée dans [FIN-84].

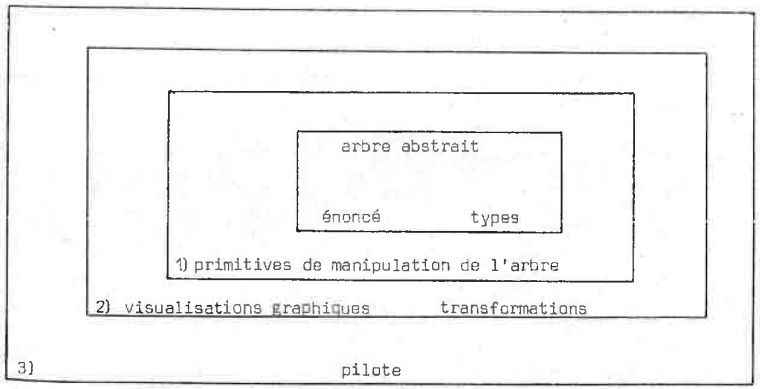


figure 0.1.

Cet environnement a pour but de permettre à l'utilisateur de spécifier et de résoudre des problèmes.
Pour spécifier et pour transformer, il est nécessaire de se doter d'abord de méthodes et de langages :

- les méthodes de spécification et de transformation développées dans [FIN-79] s'inspirent de la méthode déductive de Nancy,
- le système SPES dispose de son propre formalisme d'expressions de spécifications. Ce formalisme est fondé sur le calcul du prédicat du premier ordre,

et ensuite d'outils logiciels supportant la construction d'une spécification et le processus de transformation. Ces outils sont organisés en couche comme le montre la figure 0.1 précédente :

- la couche 1 joue le rôle d'éditeur structuré par la syntaxe,
- la couche 2 dotée d'outils de visualisation et de graphisme, est composée d'un système de transformation de types, et d'un système de transformation d'énoncés,
- la couche 3 est composée d'outils d'aides à la construction d'une spécification d'un problème.

Les travaux [DUB-84] et [LEV-84] sont des contributions au développement du projet SPES. Le premier est une étude sur des outils d'aides à une spécification, et le second une étude transformationnelle sur les types abstraits algébriques.

II.1. Objectif de notre travail

Notre travail est aussi une contribution au projet Spes. Il a pour but d'approfondir de façon pratique certaines idées avancées dans [FIN-79] et plus précisément les chapitres 2.3 et 2.4.
Les idées ont déjà été exprimées plus haut.

Cet approfondissement consiste alors à préciser le passage systématique d'un énoncé implicite (on ne sait pas comment le calculer) à un énoncé explicite dont le mode de calcul est la récurrence.
On considère que la systématisation du passage nécessite :

1. De mieux connaître les "règles" à la base de la construction de programme,
2. de réaliser un système de transformation pour assister la démarche du processus de construction.

De plus comme le domaine de la construction automatique de programme est encore peu exploré, il est aussi nécessaire de définir un cadre pour mener des expériences sur ce processus de construction. Pour cela on veut que ce système soit extensible, c'est-à-dire permettant l'adjonction de nouvelles règles et doté de tout autre outil d'aide à l'apprentissage comme par exemple le graphisme et l'historique des calculs et des décisions intervenus lors d'une construction de programme.

Ce système de transformation est un système interactif d'explicitation d'énoncés Spes, dénommé dans la suite S.I.E.

SIE est développé sous UNIX-VAX et écrit en LISP [WIS-81]. Il est appliqué à la construction de programmes manipulant des tableaux à une dimension.

III. PLAN DE LA THESE

Un système de transformation se caractérise essentiellement par :

- ses fondements (la notion de correction dans le système),
- ses caractéristiques propres (forme de l'énoncé à construire, forme du résultat, et but poursuivi),
- la méthode de passage d'un énoncé à un autre, on parlera de méthode d'explicitation,
- la définition de règles de transformations.

Le rôle et les fondements de notre système sont précisés dans le chapitre I mettant en évidence la méthode de preuve, les caractéristiques de celui-ci et sa structure générale.

Le chapitre II précise la méthode d'explicitation en mettant en évidence principalement :
- la notion de transformation,
- la définition des règles de transformations,
- le problème du choix des règles.

Cela nous conduit à définir au chapitre III un méta-algorithme général d'explicitation utilisant des stratégies d'explicitations particulières précisées dans ce même chapitre et à donner des exemples d'explicitations.
A ce stade du travail sont connus :
- les caractéristiques propres du système,
- la méthode, le meta-algorithme d'explicitation,
- les stratégies d'explicitations.

Cela suffit pour décrire complètement dans le chapitre IV notre système de transformation.

Ce système a la particularité de ne pas être propre à une classe de problème donnée. Pour mener alors notre expérience sur la construction de programme nous avons choisi tout d'abord d'expliciter des problèmes manipulant des tableaux à 1 dimension. Le chapitre V précise cette application et montre le fonctionnement du système.

Enfin, afin de situer SIE parmi d'autres travaux sur la programmation transformationnelle, le chapitre VI permet de comparer notre système avec d'autres systèmes connus et proches de SIE.

CHAPITRE I

I. ROLES ET FONDEMENTS DU SYSTEME

L'activité de programmation peut se résumer en disant que

"programmer c'est résoudre des problèmes".

Une forme de résolution d'un problème donné consiste à en construire une solution algorithmique. L'exécution par une machine de cette solution algorithmique pour une valeur des données produira une solution du problème.

L'objectif de ce chapitre est de présenter cette forme de résolution d'un problème étudiée dans les chapitres suivants.

Cette présentation est une analyse générale des mécanismes à la base de la construction d'un algorithme conduisant à une façon de les mettre en oeuvre par un système interactif d'aide à la construction d'algorithme.

Cette étude nécessite de préciser les points suivants :
- la notion de problème,
- la définition des langages d'expressions de problèmes et d'algorithmes,
- les mécanismes de construction utilisables par le système interactif.

Il sera alors possible de préciser la constitution d'un système d'explicitation. Plus précisément nous décrirons :
- la méthode utilisée par le système,
- sa structure.

I.1. Les idées à base du Système Interactif d'Explicitation

I.1.1. Objectif de notre travail

Réaliser un système permettant de construire interactivement de "bons" programmes, telle est la ligne générale dans laquelle se situe notre

travail. De là se dégage notre désir de systématiser, voire d'automatiser, le passage d'un énoncé à un programme. L'énoncé est l'expression formelle du problème à résoudre. Il décrit la relation entre données et résultats sans a priori sur une méthode particulière pour le résoudre. Il est bien clair qu'en général d'importantes transformations sont nécessaires pour obtenir un programme exécutable résolvant le problème décrit dans l'énoncé. Et ce passage prend généralement le nom de "construction de programmes".

I.1.2. Idées sur la solution

* Présentation de la solution

Extraire un programme d'un énoncé se fait rarement de manière directe. Comme mentionné précédemment des transformations sont le plus souvent nécessaires.
Par un raisonnement purement "transformationnel" il est possible de réécrire progressivement un énoncé initial en un programme. Ce raisonnement signifie intuitivement que l'on procède par application de règles de transformations, supposées valides, pour aboutir à un programme.
Ces règles expriment des mécanismes permettant par leur application de se rapprocher d'une solution algorithmique.
Le problème du programmeur est alors beaucoup plus la caractérisation de ces règles que leur application. En conséquence la voie vers l'écriture de systèmes transformationnels est ouverte :
écrire des logiciels de transformations dirigés par le programmeur et le libérant de toute "idée de calcul".
(il y a "idée de calcul" dès lors où l'on sait que certains aspects du processus de résolution d'un problème ne nécessitent pas "l'intelligence" ou "l'intuition" du programmeur.)

Nous avons donc entrepris, en suivant une telle démarche transformationnelle, l'écriture d'un système de construction de programmes où les règles de transformations (on dit aussi règles de constructions) sont des stratégies d'explicitations.

Le but de ces systèmes est de "faire faire des programmes". Il est reconnu que cela n'est pas une chose facile, en particulier parce que nous ne connaissons pas a priori la majorité des règles de constructions, mais aussi parce que l'implantation de système de règle du second ordre est très inefficace.
Nous avons alors décidé d'adopter un point de vue expérimental facilitant l'étude des mécanismes de construction. Nous avons donc retenu comme fonctionnalités d'un système d'aide à la construction de programmes :

1. fournir un ensemble d'aides permettant au programmeur de construire progressivement son programme,
2. fonder ces aides sur une méthode de programmation suggérant au programmeur de respecter certains principes durant le processus de la construction d'un programme. On peut voir cette méthode comme un moyen d'organiser le déroulement de la construction,
3. mettre en évidence l'aspect "mécanisme de construction", c'est-à-dire aider le programmeur à enrichir ses connaissances sur les "règles". Ce qui permet un enrichissement du système, on peut aussi parler d'expertise [LAU-82].

Le premier point conduit à concevoir un logiciel interactif. Cette interaction va permettre à l'utilisateur de faire des choix durant le processus de construction.

Le second point consiste à faire reposer la construction de programmes sur une méthode de programmation.

Le dernier point conduit à concevoir un logiciel extensible. Cela signifie de paramétrer le logiciel de façon à permettre l'adjonction de nouvelles règles.

* Justification de la solution

La solution présentée ci-dessus a deux volets.
Le premier est l'utilisation d'une méthode de programmation. L'importance de méthodes de programmation permettant de construire des pro-

grammes corrects et faciles à maintenir n'est plus à démontrer [ARS-75, DAH-72, DIJ-76, WIR-73,71, BAU-82- GUI-80, PAI-78].
De plus une méthode peut aider le programmeur à créer éventuellement des règles, voire même permettre au système de créer automatiquement des règles.

Le deuxième volet est donné par les deux caractéristiques du logiciel proposé, à savoir :
- interactif, (c'est-à-dire l'assistance au programmeur)
- extensible (c'est-à-dire la paramétrisation du logiciel).

Ces deux caractéristiques font que la solution proposée est ouverte.

Cette ouverture consiste à permettre au programmeur, à mieux connaître les règles à la base de la construction, de les découvrir. Ainsi l'assistance proposée, facilitant le passage d'un énoncé à un programme, permet au programmeur de voir le processus de la construction comme une suite de choix de règles définie par lui et laissant au système les tâches annexes qui apparaissent comme des calculs de programmes.

L'important pour le programmeur est alors d'essayer de répondre aux pourquoi de certains choix. (Ce point de vue est analogue à celui de SCOTT [SCT-83] où il distingue "inférence" et "dérivation"). Pour cela la paramétrisation a pour effet de permettre une étude expérimentale dans le domaine de la construction de programmes.

Pour être réaliste, le système est conçu d'une manière telle que cette expérience ne puisse se faire que pour des classes de problèmes spécifiques dans lesquels on s'intéresse à la construction de petits programmes.

Le système proposé est donc un logiciel :
- interactif,
- extensible,
- et reposant sur une méthode de programmation.

Voyons comment ces aspects interviennent dans ce système. En d'autres termes il s'agit maintenant de l'identifier en précisant :
- ce qu'il faut entendre par "construction",
- ce qu'est l'environnement propre de SIE.

I.2. Caractérisation de l'explicitation

Dans tous systèmes de construction de programmes interviennent :
- une définition de l'énoncé de départ,
- une définition de l'énoncé d'arrivée,
- une méthode ou une technique de passage du premier au second.

Ces trois éléments sont définis après avoir précisé le terme "explicitation".

I.2.1. Notion d'explicitation

On schématise la construction de programmes comme suit :

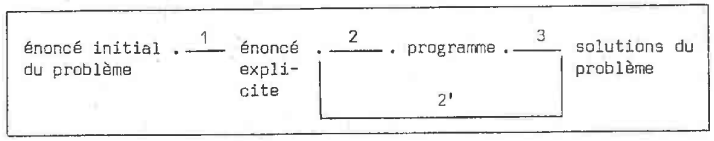


Schéma I.1

On appelle explicitation la phase 1 : le passage d'un énoncé initial à un énoncé dit explicite.

Un énoncé explicite exprime formellement un mode de calcul. C'est un algorithme présenté sous forme de déclarations. Il peut être ensuite soit interprété et dans ce cas on obtient directement les solutions du problème (phase 2'), soit traduit dans un langage impératif (de programmation, fonctionnel) et dans ce cas la phase 2 consiste à obtenir un programme et la phase 3 correspond à l'exécution de celui-ci pour une valeur des données.

L'énoncé initial est implicite dans le sens où il ne contient aucune information sur sa résolution. Exprimé formellement on veut qu'il soit concis, le plus proche possible du programmeur et surtout facile à écrire.

Comme on ne sait pas le définir proprement alors on dira que tout énoncé non explicite est implicite et on parlera d'explicitation d'énoncé implicite pour dire que l'on veut construire un énoncé explicite à partir d'un énoncé implicite.

Par exemple :

Soit l'énoncé $\mathcal{E}_1$ exprimé dans un pseudo-langage du calcul du prédicat du premier ordre (CP1) :

$$\mathcal{E}_1 = \underline{\text{trouver}}\ l\ \underline{\text{tel que}}\ \forall k\ \ d[l] \leqslant d[k]$$

$\mathcal{E}_1$ est un énoncé implicite ; il n'indique pas la manière de trouver l. Il exprime la recherche de l'indice d'un élément minimal d'un tableau d.

L'énoncé $\mathcal{E}_2$

$$\mathcal{E}_2 = j_o == 1$$
$$l == \underline{\text{der}}\ j_m\ \underline{\text{pour}}\ m\ \underline{\text{de}}\ 1\ \text{à}\ n$$
$$j_m == \underline{\text{si}}\ d[j_{m-1}] \leqslant d[m]\ \underline{\text{alors}}\ j_{m-1}\ \underline{\text{sinon}}\ m$$

est un énoncé explicite.

A condition d'en préciser le sens ce que nous faisons, ci-dessous, de façon intuitive :

une suite (j_m) d'indices du tableau d est définie

- "der" permet d'extraire le dernier élément de cette suite et "pour m" permet de la construire.
- d "[i]" retourne la valeur du tableau d en un indice i.
- [1..n] est l'intervalle de définition du tableau d.

L'explicitation de $\mathcal{E}_1$ consiste à construire $\mathcal{E}_2$. L'explicitation peut être vue comme la première activité de la construction de programmes (schéma I.1). On peut poursuivre la construction en traduisant $\mathcal{E}_2$ en Pascal par exemple. Il est également possible d'interpréter directement $\mathcal{E}_2$ pour une valeur effective de d et obtenir ainsi le résultat correspondant.

I.2.2. Définition des énoncés

Comme nous l'avons déjà dit, l'énoncé d'un problème est une relation entre données et résultats de ce problème. Plus précisément un problème est formé des deux composantes suivantes :

- son contexte : les informations nécessaires à sa compréhension,
- son énoncé.

Schématiquement un contexte est un ensemble de domaines munis d'opérations. L'énoncé s'exprime ainsi à l'aide de ces opérations. Afin de mieux préciser ces notions illustrons les par quelques exemples simples.

* Exemple de problèmes

a) Exemples mathématiques

1. Construire un programme qui imprime les racines de l'équation $ax^2 + bx + c = 0$ où a, b, c sont des constantes vérifiant : $b^2 - 4ac > 0$ et $a \neq 0$.

2. Trouver 2 nombres premiers entre eux et proportionnels à deux nombres donnés non nuls.

3. Représenter deux matrices creuses par des S-expressions lisp et écrire un algorithme de multiplication (Lisp).

b) Exemples de gestion :

1. Les produits d'un super marché constituent l'ensemble "produit à vendre" dont chaque élément est composé :
   - d'une chaîne de caractères (nom du produit)
   - d'un poids,
   - du prix au kg ou au litre,
   - d'une marque.

On veut savoir pour un produit et une quantité donnés la marque la plus économique.

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

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

c) Exemples généraux de programmation :

1. 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.

2. Dans un système d'exploitation en monoprogrammation avec "spooling" les travaux en attente d'être exécutés sont stockés sur disque dans leur ordre d'arrivée avec une priorité. Ecrire un algorithme choisissant le prochain travail à exécuter.

3. On dispose d'une mémoire principale, écrire un algorithme de gestion de la mémoire en supposant en outre que l'utilisateur a la possibilité de faire des demandes d'allocation/restitution d'emplacement mémoires.

d) Exemples sur les tableaux :

1. Soit d un tableau de booléens. On veut connaître les nombres de valeurs vrai associés à toutes séquences connexes de d de longueur 32, [BEN-84b],

2. Interclasser de 2 tableaux,

3. Trouver la première séquence connexe d'un tableau d'entiers dont la somme de ses éléments soit maximale. Une séquence est délimitée par 2 nombres négatifs [BEN-84 a].

e) Exemples sur les traitements de textes.

1. On cherche le nombre de mots de longueur 1, 2, 3, ... 20 d'un texte donné ; 20 est la longueur maximum d'un mot ; deux mots sont séparés par un blanc.

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

3. Etant donnée une famille de mots, trouver la première occurrence de l'un de ces mots dans le texte.

Ces exemples sont pour la plupart tirés de [FIN-79,82, PAI-80]. Ils montrent bien qu'un problème se caractérise par des domaines supposés connus, les files, les tableaux, les caractères, les ensembles, exprimant intuitivement un contexte, par un énoncé exprimant informellement l'objet du problème.

* La notion de problème

Pour préciser ces idées, donnons les définitions suivantes :

- un problème P est un couple $(\mathcal{S}, \mathcal{E})$ [FIN-79] où :

$\mathcal{S}$ est le contexte du problème spécifié en termes de types abstraits algébriques paramétrés,

$\mathcal{E}$ est l'énoncé du problème qui s'exprime à l'aide d'opérations et de prédicats du contexte.

Par exemple le problème de la fraction irréductible peut être donné par le couple :

$$P = (\text{ENTIER}^+, \mathcal{E}_1)$$

avec $\mathcal{E}_1$ = propor(x,y,a,b) et prem(x,y) ;

le contexte est donné par le type abstrait ENTIER^+ étendu par les prédicats :

"propor", indiquant que 2 entiers sont proportionnels à 2 autres,

et "prem", indiquant que 2 entiers sont premiers entre eux.

$\mathcal{E}_1$ est l'énoncé du problème.

Précisons les définitions précédentes de $\mathcal{S}$ et $\mathcal{E}$.

Une spécification d'un type abstrait algébrique [DER-79, LEV-84] est la donnée :

- d'une signature Σ,

et - d'un ensemble d'axiomes, Ax.

Une signature Σ est la donnée d'un triplet (T,F,P) où :

T est l'ensemble de noms de types (noms de domaines),

F est l'ensemble des symboles d'opérations,

P définit les profils des symboles de F.

Le profil d'un symbole est le couple formé de sa source et son but :

$$\forall f \in F \quad profil(f) = (w, T')$$

où w est une suite finie sur T et $T' \in T$

que l'on note : $f : T_1 \times T_2 \times \ldots \times T_j \rightarrow T'$,

$source(f) = w$,

$but(f) = T'$.

La longueur de w est l'arité de f. Un symbole d'arité nulle est une constante (c'est un symbole 0-aire appartenant à F).

On distingue dans T un sous ensemble $\mathcal{T} = \{T'1, T'2, \ldots, T'k\} \subset T$ appelés les types paramètres formels de $\mathcal{S}$.

$\mathcal{S}$ est paramétré si $\mathcal{T}$ est défini (cf. figure I.1)

Un type non paramétré est appelé un type de base.

Il existe un nom de type $T' \in T$ qui apparaît au moins une fois dans le profil de chaque f ; on l'appelle type d'intérêt de $\mathcal{S}$.

Une Σ-algèbre est la donnée

- d'une famille d'ensemble, indicée par T : $A = (A_{Ti})_{Ti \in T}$ appelée support de l'algèbre,

et

- d'une famille d'opérations dans A indicée par $F : F_A = (f^A)_{f \in F}$ telle que Si

$$f : T_1 \times T_2 \times \ldots \times T_j \rightarrow T'$$

alors

$$F^A : A_{T_1} \times \ldots \quad A_{T_j} \rightarrow A_{T'}$$

Soit X l'ensemble des variables de $\mathcal{S}$. Un identificateur x est un symbole 0-aire appartenant à X. Chaque x est d'un type unique :

$$profil(x) = (Ti, T'i) \text{ noté } x : T'i$$

L'algèbre libre des termes sur la signature Σ est notée $\mathcal{T}_\Sigma(X)$.

Son support noté aussi $\mathcal{T}_\Sigma(X)$ est défini par :

- pour tout $Ti \in T$ $\quad C_{Ti} \cup X_{Ti} \subset$ support $(\mathcal{T}_\Sigma(X))$

  C_{Ti} est l'ensemble des symboles 0-aires de but Ti,

  et X_{Ti} est l'ensemble des variables x de type Ti.

- $\forall f \in F \quad \forall t_1 \ldots t_j \in$ support $(\mathcal{T}_\Sigma(X))$

  tels que $but(t_n) = Ti_n$ pour $n = 1, \ldots, j$

  $f(t_1, \ldots, t_j) \in$ support $(\mathcal{T}_\Sigma(X))$

f est une opération de $\mathcal{T}_\Sigma(X)$ et $f(t_1, \ldots, t_j)$ est <u>le terme</u>.

Le profil d'un terme est le couple formé de la source du terme et de son but :

$$\forall t \in \mathcal{T}_\Sigma(X) \quad profil(t) = (source(t), but(t))$$

$$source(f(t_1, \ldots t_j)) = source(t_1) \ldots source(t_j)$$

$$but(f(t_1, \ldots t_j)) = but(f)$$

Un axiome $ax \in Ax$ est un couple de termes u,v de $\mathcal{T}_\Sigma(X)$ de même but, noté : $u = v$

Cette description rapide de $\mathcal{S}$ suppose que l'on est familier avec la notion de types abstrait algébriques [GOG-78, GUT-78]. Les figures V.1 sur le type TABLEAU et I.1 sur le type multiensemble, MENS, sont des exemples de spécification $\mathcal{S}$.

L'énoncé $\mathcal{E}$ du problème $(\mathcal{S},\mathcal{E})$ est une conjonction de formules. Ces formules sont obtenues en appliquant aux termes les symboles logiques $\Rightarrow$, $\Leftrightarrow$, <u>ou</u>, <u>non</u>, $\forall$, <u>et</u>. Dans un énoncé $\mathcal{E}$ certaines variables (au moins une) sont appelées le résultat r de $\mathcal{E}$.

Si dans tous les termes de $\mathcal{E}$ figure r alors $\mathcal{E}$ est appelé <u>énoncé strict</u>. (on parlera aussi de définition stricte). Sinon on dira que $\mathcal{E}$ est constitué d'une définition stricte caractérisant r et de définitions intermédiaires caractérisant des résultats intermédiaires nécessaires au calcul de r.

Une définition intermédiaire de résultat r' de $\mathcal{E}$ peut être, elle aussi, soit stricte, soit nécessitée pour calculer r' de résultats intermédiaires.

1.	type MENS (VAL)	MENS est le type d'intérêt.
2.	mvide : → MENS majt : MEN x VAL → MENS mapt : VAL x MENS → BOOL msup : MENS x VAL → MENS	VAL est le type des éléments, c'est le paramètre formel. La partie 2 définit la signature de MENS
3.	∀ v, V1, de type VAL, ∀ e de type MENS mapt (v, mvide) = faux mapt (v, majt(e,v1)) = si v = v1 alors vrai sinon mapt (v,e) msup (majt(e,v),v1) = si v = V1 alors e ' sinon majt (msup(e,v1), v) pré-condition sur : msup (e,v) : mapt (v,e) = vrai	La partie 3 définit les axiomes de MENS pré-condition exprime des restrictions sur certaines opérations

Figure I.1. Un type abstrait multiensemble : MENS

Par exemple en reprenant le problème de la fraction irréductible, citée plus haut :

$$P = (ENTIER^{+}, \mathcal{E}_1)$$

$\mathcal{E}_1$ = propor (x,y,a,b) <u>et</u> prem (x,y), alors

l'énoncé $\mathcal{E}_1$ est un énoncé strict, sans intermédiaire. Dans la formule $\mathcal{E}_1$ le couple (x,y) est le résultat de $\mathcal{E}_1$ et le couple (a,b) est la donnée de P.

Un autre énoncé de ce même problème peut être donné par la formule $\mathcal{E}_2$:

x = div (a,z) <u>et</u> y = div (b,z) <u>et</u> z = pgcd (a,b)

La fonction div est la division entière. Une opération supplémentaire du type $ENTIER^{+}$.

Dans $\mathcal{E}_2$ z est le résultat intermédiaire caractérisé par le sous énoncé :

z = pgcd (a,b). (pgcd est aussi une opération sur le type $ENTIER^{+}$)

De manière générale un problème est vu comme un système formel du type "calcul des prédicats du premier ordre égalitaire fortement typé" :

- composé de types abstraits algébriques,
- de formules précédemment définies (plus les égalités entre termes),
- et muni d'un ensemble de règles d'inférences.

L'énoncé du problème est une formule de ce système formel, que nous ne détaillons pas davantage ici. Pour la définition d'un tel système se rapporter aux travaux de J.P. Finance [FIN-79]. On peut noter que l'ensemble des règles peut être commun à tout problème. Par exemple l'utilisateur choisira comme règles d'inférences le modus-ponens, la substitution et d'autres règles "plus fortes". Dans notre cas le paragraphe II.3. précise les règles d'inférences sur lesquelles est fondé SIE.

Le lecteur est invité à consulter [CHAN-73, KLE-71, MAN-74] pour plus de détails sur la notion de système formel.

I.2.2.1. Les énoncés implicites : le langage Spes

Il a été déjà dit au § I.2.1. qu'un énoncé non explicite est dit être implicite. Nous confondrons dans la suite les termes "énoncé" et "énoncé implicite". Un énoncé implicite est exprimé en Spes.

Spes est un langage de très haut niveau développé au CRIN par J.P. Finance et A. Quéré [QUE-83].

Une première caractéristique importante de ce langage est de pouvoir exprimer facilement et clairement un problème de façon méthodique. Spes permet alors de construire formellement et progressivement une spécification d'un problème de manière "déductive" :

- pour caractériser le résultat r d'un énoncé, on introduit des objets "données intermédiaires". Ces intermédiaires sont considérés

comme autant de nouveaux résultats à caractériser. Et on réitère le processus jusqu'à ce que les objets soient des données primitives de bases.

Ainsi tout objet est d'abord défini de façon informelle, puis par la relation qui le lie aux objets utilisés.

La deuxième caractéristique de ce langage est qu'il est fondé sur l'expression de formules du CP1.

Montrons les aspects syntaxiques et sémantiques de Spes. (On s'intéresse à la transformation d'un énoncé et non d'un ensemble d'énoncés).

Aspects syntaxiques

Une spécification est constituée d'un ensemble d'énoncés et d'un ensemble de descriptions de types. Un énoncé est un ensemble de descriptions d'objets.

Toute description d'un objet est composée :

- d'un commentaire,
- d'un prédicat qui exprime la propriété caractéristique de l'objet,
- du profil de l'objet.

Le commentaire est la définition informelle d'un objet introduite avant les deux autres définitions. La seconde définition précise formellement la relation qui lie un objet à d'autres objets utilisés.

Un profil est écrit en termes de types de données abstraits.

Un tel type peut être prédéfini. Mais l'utilisateur peut aussi décrire ses propres types et ce au cours de la construction de la sépcification.

Soit $\mathcal{E}$ un énoncé et x un identificateur de $\mathcal{E}$ désignant un objet alors :

la propriété d'un objet x est donnée par une définition implicite, c'est-à-dire de la forme :

x <u>tq</u> $P(x,\bar{y})$ où $\bar{y}$ est la liste des identificateurs de la définition (distincts de x).

Cette définition indique que :

- x est un résultat, (figure en partie gauche et droite de "tq")
- les identificateurs figurant dans $\bar{y}$ sont considérés comme des données ou des constantes.

De plus :

- si x figure en partie droite d'une définition alors x est utilisé dans cette définition. Il est alors considéré comme une donnée de cette définition,
- x désigne un unique objet dans $\mathcal{E}$,
- si x est quantifié alors x n'admet aucune autre définition formelle, informelle, de profil dans l'énoncé :

exemple : l'énoncé suivant est incorrect :

```
....
x tq P(x,ȳ)
....
y tq qqx : ENTIER ...
...
```

x est défini deux fois ("qq" est le quantificateur universel).

Aspects sémantiques

a) A tout énoncé implicite est associé une formule du CP1.

Par exemple : à l'énoncé suivant, composé de 2 définitions implicites,

```
r tq r ≤ d[b] et c ≤ r
  d : TAB    r : ENT    b : 1..n    c : ENT    r : ENT
b tq b = a + c
  a : ENT
```

est associée la formule du CP1 : (n, d, a, c sont des données)

$\forall n : ENT \ \forall d : TAB \ \forall a : ENT \ \forall c : ENT \ \exists r : ENT \ \exists b : 1..n$
$(r \leq d[b] \ \underline{et} \ c \leq r) \ \underline{et} \ (b = a + c)$

L'algorithme d'association est simple :

1. toutes les variables résultats sont quantifiées existentiellement,
2. toutes les autres variables sont quantifiées universellement,
3. Conjonction de toutes les formules apparaissant en partie droite de tous les "tq".

b) Les identificateurs portent sur tout l'énoncé.

La figure I.2 donne un exemple d'énoncé spécifié en Spes.

La troisième et dernière caractéristique du langage Spes est de permettre des définitions algorithmiques, c'est-à-dire de la forme $x == f(\bar{t})$ (où $\bar{t}$ est une liste d'arguments, de termes). Il offre donc un cadre général pour définir des transformations d'énoncés en un ensemble de définitions algorithmiques. Ces dernières sont en fait des phrases du langage Medee vu comme un sous-ensemble de Spes. Ces phrases caractérisent une classe particulière d'énoncés explicites : les énoncés récurrents.

I.2.2.2. Les énoncés récurrents : le langage Medee

Les énoncés récurrents ou itératifs sont une classe importante d'énoncés explicites. Ces énoncés occupent une grande place en programmation comme en témoigne la littérature variée à leur sujet : [ARS-77, DIJ-76, ASH-77,76, BAS-80b, BER-82, BLI-77, MIS-77, REM-84, SCH-82, TUR-84, WEG-74, WAT-83]

Une définition d'un énoncé récurrent

Un énoncé récurrent est un multisystème d'équations récurrents d'ordre 1. Un multisystème est un système de systèmes d'équations récurrentes définies inductivement comme suit :

Soit Sy : $y_1 = f_1 (y_0, \bar{x})$
$y_2 = f_1 (y_0\ y_1, \bar{x})$
.....
$y_n = f_1 (y_{n-1}\ y_{n-2} \dots y_0, \bar{x})$

où $\bar{x}$ est une liste de variable,
pour chaque variable z de $\bar{x}$ est défini un système S_z.
Pour plus de précision sur la notion d'énoncé récurrent, le lecteur pourra consulter [ARS-77, FIN-79, TUR-84].

Ces énoncés sont exprimés en Medee, un langage développé au CRIN [FIN-79, 77, LES-78] adapté à la définition d'énoncés récurrents. Medee est un langage statique : au sens où il manipule des suites plutôt que des variables informatiques.

Ainsi un nom ne peut désigner qu'un seul objet. Le concept de module permet de structurer et hiérarchiser un énoncé Médee qui est intuitivement un ensemble de définitions d'objets. Une définition est l'un des types suivants :

1, itérative, 2, conditionnelle, 3, simple.

1. On distingue deux formes d'itérations ("der" et "prem") dérivées de la définition itérative suivante :

(ITERATION) r == prem u pour i de [k..∞[tel que a_i,

où : - u est une suite a priori infinie de valeurs, a une suite booléenne (infinie),
- r est le premier élément de u tel que a_i soit vrai.

(i : $ENTIER^+$ prend successivement ses valeurs dans l'intervalle d'entiers [k..∞[).

Et d'où les formes : (nous ne démontrons pas ici la correction de der, prem vis-à-vis de ITERATION)

a) (der) r == der [u_i pour i de k jqa a_i]

où : r est de type T1, u une suite de T1, k est la première valeur de i. L'itération der signifie :

si il existe un indice l tel que a_l = vrai
et pour tout i, $k \leqslant i < l$ a_i = faux alors la valeur de r est celle de u_l sinon r n'est pas défini.

Le constructeur [... pour de jqa] définit les éléments u_i (pour $i > k$) par récurrence à l'aide de u_{i-1}. (dans la suite les crochets "[" et "]" sont omis).

Et der extrait le dernier élément de la suite u qui est u_l.

b) (prem) r == prem i dans u depuis k tel que p_i

où : r,i,k : $ENTIER^+$ et p : SUITEdeBOOL.

L'itération prem signifie :

si il existe l tel que i = l et p_l et pour tout $k \leqslant i < l$ non p_l
alors r = l sinon r = 0 (r est le premier indice i tel que p_i)
(dans u signifie i prend ses valeurs dans [borne inf(u) .. borne sup(u)]).

2. La forme de la définition conditionnelle est :

r == si b_1 alors r_1 sinon r_2 $\quad b_1$: BOOL, r, r_i : T'.

Cette forme peut se généraliser à une conditionnelle à n branches.

3. La forme de la définition simple est :

$r == f(x_1, \ldots, x_n)$, f est une opération sur un type T' défini.

La figure I.3. donne un exemple type d'énoncé Médée explicitant la spécification de la figure I.2.

r TRI	l'entête de l'énoncé : énoncé de nom TRI et l'objet r est le résultat de TRI,
r ? tableau trié à partir de d	--> définition informelle de r,
r : TAB	--> définition du profil de r,
r tq equ(r,d) et croiss(r)	--> la définition formelle de r, 'tq' permet de distinguer la partie résultat (gauche) et la définitionnelle (droite),
d ? tableau donnée	
d : TAB	

figure I.2. : une spécification d'un problème de tri

equ et croiss sont définis dans la description du type TABLEAU (§ V).

u_0 == tvide

a_0 == faux

r == der u_i pour i de 1 jqa a_i

u_i == insert (u_{i-1}, d(i), l_i)

l_i == si i = 1 alors 1 sinon
si u_{i-1}(i-1) <= d(i) alors i sinon
si d(i) <= u_{i-1}(1) alors 1 sinon m_i

m_i == prem j dans u_{i-1} depuis 2 tel que d(i) <= u_{i-1}(j)

a_i == (i = bs(d))

-d(i) donne la valeur au point i, insert et bs sont aussi des opérations sur le type TABLEAU (cf. § V).

L'énoncé Médée ci-dessus est un transformé de celui de la figure I.2 auquel il faut y adjoindre le profil des objets :

u : SUITE-DE-TAB

a : SUITE-DE-BOOL

l : SUITE-DE-ENTIER i : ENTIER m : SUITE-DE-ENTIER

d : TAB r : TAB

Figure I.3. : un énoncé Médée : un tri par insertion

I.2.3. Fondement du Système Intéractif d'Explicitation

Avec les idées et définitions précédentes, nous pouvons maintenant préciser le rôle et l'objectif du système réalisé.
SIE construit un énoncé récurrent M à partir d'un énoncé initial E de telle manière que "M résoud E" :
toute solution de M pour une donnée d est solution de E pour d.

Dans le système formel associé à SIE, M est construit de façon que :

(I) r == M(d) => E(r,d)

soit un théorème ; r est le résultat du problème.
Si (I) est un théorème alors on dira que M est correct vis à vis de E.
Par exemple l'énoncé Médée de la figure I.3. est correct vis à vis de l'énoncé Spes de la figure I.2. Cette preuve nécessite d'utiliser les propriétés de $\leqslant$, croiss, "der", "prem", ...

Le schéma I.2. montre l'explicitation que doit donc mettre en oeuvre SIE.

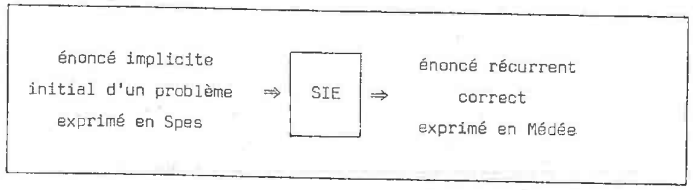


Schéma I.2.

Le problème alors est de savoir comment SIE construit M correct à partir de E, ou encore comment SIE prouve la validité de (I).
La résolution de ce problème est l'objet du chapitre suivant.
Donnons en cependant une idée intuitive dès maintenant.

I.3. Analyse générale du problème de l'explicitation d'un énoncé

Pour prouver que la formule (I) précédente est un théorème la méthode de transformation consiste à associer à l'explicitation une méthode de programmation et à utiliser les règles de preuves du système formel associé à SIE comme des règles de constructions.
L'explicitation repose sur l'utilisation de la méthode de programmation dite déductive développée à Nancy [FIN-82, PAI-74].
On pourra noter ici que Médée a été développée comme support linguistique à cette méthode.
Les règles de constructions, quant à elles, expriment des équations particulières :

S :: E,D mettant en évidence que la transformation d'un énoncé S à expliciter fait apparaître une définition D de Médée et un nouvel énoncé implicite, plus "simple" que S.
Ces règles sont appelées "stratégies d'explicitations" et elles peuvent se combiner entre elles pour donner lieu à des stratégies un peu plus globales.
Ces stratégies sont considérées comme valides. En conséquence leur application répétée produira une énoncé explicite M correct vis à vis de l'énoncé initial $\mathcal{E}_r$.

I.3.1. La méthode de programmation déductive

L'explicitation fait l'hypothèse que l'énoncé initial $\mathcal{E}_r$ a été spécifié d'une manière déductive comme indiquée en I.2.2.1. :

$\mathcal{E}_r$ exprime le résultat du problème utilisant éventuellement des intermédiaires. Toutes les opérations figurant dans l'énoncé sont des opérations sur des types pré-définis.
La méthode déductive peut se résumer ainsi :

Soit $\mathcal{E}_r$, on cherche :

- soit à construire une fonction f telle que $r == f(x_1, \ldots, x_n)$
- soit à faire une analyse par cas,
- soit à définir r par approximations successives.

Cela conduit à introduire des intermédiaires éventuellement de type suite. Ces intermédiaires sont eux-mêmes les résultats de nouveaux sous problèmes, et ainsi de suite jusqu'à ce que l'intermédiaire soit une donnée.

Illustrons cette manière de faire en considérant l'énoncé $\mathcal{E}_1$ du § I.2.1., exprimé en Spes, comme $\mathcal{E}_r$:

$\mathcal{E}_r$ est défini par : r <u>tq</u> $\forall k\ d[r] \leqslant d[k]$
$k : 1 .. bs(d)$
$r : 1 .. bs(d)$,

pour expliciter $\mathcal{E}_r$ on décide de le définir par approximations successives. On obtient

r == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i

Cette définition introduit les suites u et a :

$\mathcal{E}_{u_i}$: u_i <u>tq</u> $\forall k\ \ 1 \leqslant k \leqslant bs(d1_i) \Rightarrow d[u_i] \leqslant d[k]$

(bs rend la borne supérieure du tableau d, et $d1_i$ dénote l'opération "tr", tranche sur un tableau ; $d1_i$ = tr(d,1,i) le domaine de ce tableau est l'intervalle [1..i]).

$\mathcal{E}_{a_i}$: a_i == (i = bs(d))

L'intermédiaire u doit être également explicité.

<u>Explicitation de u</u> : On cherche une relation de récurrence définissant u_i.

Pour cela comparons les définitions $\mathcal{E}_{u_i}$ et $\mathcal{E}_{u_{i-1}}$:

$\mathcal{E}_{u_i}$ u_i <u>tq</u> $\forall k\ \ 1 \leqslant k \leqslant bs(d1_i) \Rightarrow d1[u_i] \leqslant d[k]$

$\mathcal{E}_{u_{i-1}}$ u_{i-1} <u>tq</u> $\forall k\ \ 1 \leqslant k \leqslant bs(d1_{i-1}) \Rightarrow d1[u_{i-1}] \leqslant d[k]$

or le tableau $d1_i$ sur le domaine [1..i-1] est égal au tableau $d1_{i-1}$ d'où une nouvelle forme de $\mathcal{E}_{u_i}$:

$\mathcal{E}_{u_i}$ u_i <u>tq</u> $\forall k\ \ 1 \leqslant k \leqslant bs(d1_{i-1}) \Rightarrow d[u_i] \leqslant d[k]$
et $k = i \Rightarrow d[u_i] \leqslant d[i]$

<u>Idée de récurrence</u> : la première partie de $\mathcal{E}_{u_i}$ est identique à $\mathcal{E}_{u_{i-1}}$ d'où l'idée d'une analyse par cas

<u>Cas 1</u> $d[u_{i-1}] \leqslant d[i]$ on choisit $u_i = u_{i-1}$

<u>Cas 2</u> $d[u_{i-1}] > d[i]$ on choisit $u_i = i$

Il faut aussi expliciter u_0, c'est-à-dire initialiser u_i :

$\mathcal{E}_{u_0}$: u_0 <u>tq</u> $\forall k\ \ 1 \leqslant k \leqslant bs(d1_0) \Rightarrow d[u_0] \leqslant d[k]$

On choisit comme initialisation :

u_0 == 1

l'énoncé Médée correct vis à vis de $\mathcal{E}_r$ est donc :

M_r : u_0 == 1
r == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i
u_i == <u>si</u> $d[u_{i-1}] \leqslant d[i]$ <u>alors</u> u_{i-1} <u>sinon</u> i
a_i == (i = bs(d)).

A présent mettons en évidence la preuve de M_r.

I.3.2. <u>Analyse de la résolution.</u>

La démarche suivie précédemment pour construire M_r correct vis à vis de $\mathcal{E}_r$ est intuitive.

Pour prouver la correction d'un énoncé Médée M vis-à-vis d'un énoncé de départ $\mathcal{E}$, il est nécessaire de formaliser et rationnaliser la construction des énoncés itératifs Médée. Cela conduit à

définir une technique de preuve assurant la correction de M vis à vis de $\mathcal{E}$. La technique choisie repose sur celle développée par Hoare [HOA-69, ASH-75]. Et nous l'appliquons à la construction de programmes.

Pour prouver la correction d'un énoncé Médée, on prouve :

1. La correction partielle

Il s'agit de démontrer que si M_r (précédent) s'arrête (i.e fournit un résultat) alors il produit un résultat correct ; plus précisément :

soit l une valeur i tq a_l = vrai

et $\forall i \in [k .. l-1]$ a_i = faux

le résultat $r = u_l$ est correct si l'on a :

$e_i(r,i)$ <u>et</u> a_i == vrai $\Rightarrow$ e_r.

Par ailleurs il faut prouver $e_u(i)$ pour tout i tel que a_i = faux. Ceci se fait par induction. On cherche ainsi à prouver les implications suivantes

$e_u(i-1)$ <u>et</u> définition de Médée de u_i $\Rightarrow$ $e_u(i)$

et définition de Médée de u_0 $\Rightarrow$ $e_u(0)$.

Cette preuve se fait de manière récursive et nécessite de prouver chaque définition de M_r correcte vis à vis de sous énoncés de $\mathcal{E}_r$ correspondants.

Par exemple montrons que l'énoncé M_r de l'exemple précédent est correct partiellement vis à vis de $\mathcal{E}_r$:

Supposons que pour i = l on ait : a_l = vrai

Prouvons :

(I) $(\forall k \quad 1 \leqslant k \leqslant bs(d1_{i-1}) \Rightarrow d[u_i] \leqslant d[k]$

<u>et</u> $k = i \Rightarrow d[u_i] \leqslant d[i])$ <u>et</u>

$(u_i$ == <u>si</u> $d[u_{i-1}] \leqslant d[i]$ <u>alors</u> u_{i-1} <u>sinon</u> $i) \Rightarrow \forall k \quad 1 \leqslant k \leqslant bs(d1_i)$

$\Rightarrow d[u_i] \leqslant d[k]$

du cas 1 (voir exemple précédent) on a :

$u_i = u_{i-1}$

(I) devient :

$(\forall k \quad 1 \leqslant k \leqslant bs(d1_{i-1}) \Rightarrow d[u_{i-1}] \leqslant d[k]$

<u>et</u> $k = i \Rightarrow d[u_{i-1}] \leqslant d[i])$ <u>et</u> vrai $\Rightarrow \mathcal{E}_u(i-1)$

l'implication :

$k = i \Rightarrow d[u_{i-1}] \leqslant d[i]$ est vraie (cas 1)

Il s'ensuit alors que (I) est valide (cela est immédiat)

du cas 2 (voir exemple précédent), on a : $u_i = i$

par analogie au cas précédent la formule (I) est valide.

2. La correction totale

Il s'agit de démontrer la terminaison du module itératif M_r (ou M) prouvant ainsi l'existence d'une solution. Plus précisément il faut prouver :

$\exists i$ <u>tq</u> a_i

M_r termine car le tableau est fini, donc i = bs(d) convient.

La démonstration de propriétés, assurant la correction de M_r vis à vis de $\mathcal{E}_r$, nécessite l'utilisation de règles de preuves et de la sémantique des définitions Médée.

Par exemple la preuve d'une définition itérative se fait à l'aide de la règle de preuve :

e_u <u>et</u> e_a <u>et</u> r == <u>der</u> u_i <u>pour</u> i <u>de</u> k <u>jqa</u> $a_i \Rightarrow e_r$

décrite au § II.2.1. L'application de cette règle à $\mathcal{E}_r$ conduit à démontrer que $\mathcal{E}_n$ est un invariant et que $\mathcal{E}_a$ est vrai pour i = bs(d).

De manière plus générale les règles de preuves utilisées sont de la forme :

(II) : E <u>et</u> D $\Rightarrow$ S où E et S sont des énoncés Spes et D une définition Médée.

Mais comme on veut construire plutôt que prouver a posteriori alors on cherche à associer aux formules (II) une forme plus constructive :

(III) : {pre} S :: E, D {post}.

Les règles de la forme (III) sont appelées règles de constructions. Ces règles sont telles que l'on sait des "choses" sur E, D, et S et elles permettent alors d'établir que :

E <u>et</u> D ⇒ S

On retrouve ainsi la preuve de la correction d'une définition Médée vis à vis d'un sous énoncé. En conséquence la construction et la preuve sont menées conjointement. Aux particularités de E, D et S d'une règle de construction peuvent être attachées des pré-post conditions désignées par pré et post. Les pré-post sont vus au § II.5.
Les règles de preuves sont présentées au § II.1, et les règles de constructions au § II.2. La méthode générale d'explicitation est analysée dans [FIN-79]. Elle est à l'origine de cette analyse de la résolution étudiée en détail dans le chap. II.

I.4. Structure générale du système

La figure I.4. décrit SIE comme étant composé :
- d'un ensemble de connaissances relatives à divers problèmes,
- d'un ensemble de modules aidant le programmeur à construire son programme.

a) les connaissances comprennent :

. des ensembles de stratégies propres à chaque problème : par exemple un ensemble pour les problèmes relatifs aux tableaux, un autre pour les problèmes sur les entiers, ...
. une bibliothèque de types abstraits de données pré définis,
. un arbre des choix, celui-ci est créé au cours de l'explicitation. Il rappelle au programmeur ses choix antérieurs et l'aide à les remettre en cause.

b) Les différents modules de service assurent :

. le dialogue programmeur-SIE (module interface),
. l'application d'une stratégie (module interpréteur),
. la représentation en arbre de tout énoncé ou toute règle de construction de la forme {pre} S :: E,D {post} et l'exécution de toutes les fonctions associées à cette représentation (module constructeur d'arbres),
. l'apprentissage de nouvelles règles et la gestion des stratégies (module bibliothécaire),
. la possibilité au programmeur de manipuler l'arbre des choix (module manipulateur d'arbre des choix).

Les liens entre tous ces modules sont assurés par un module spécial, le module noyau.

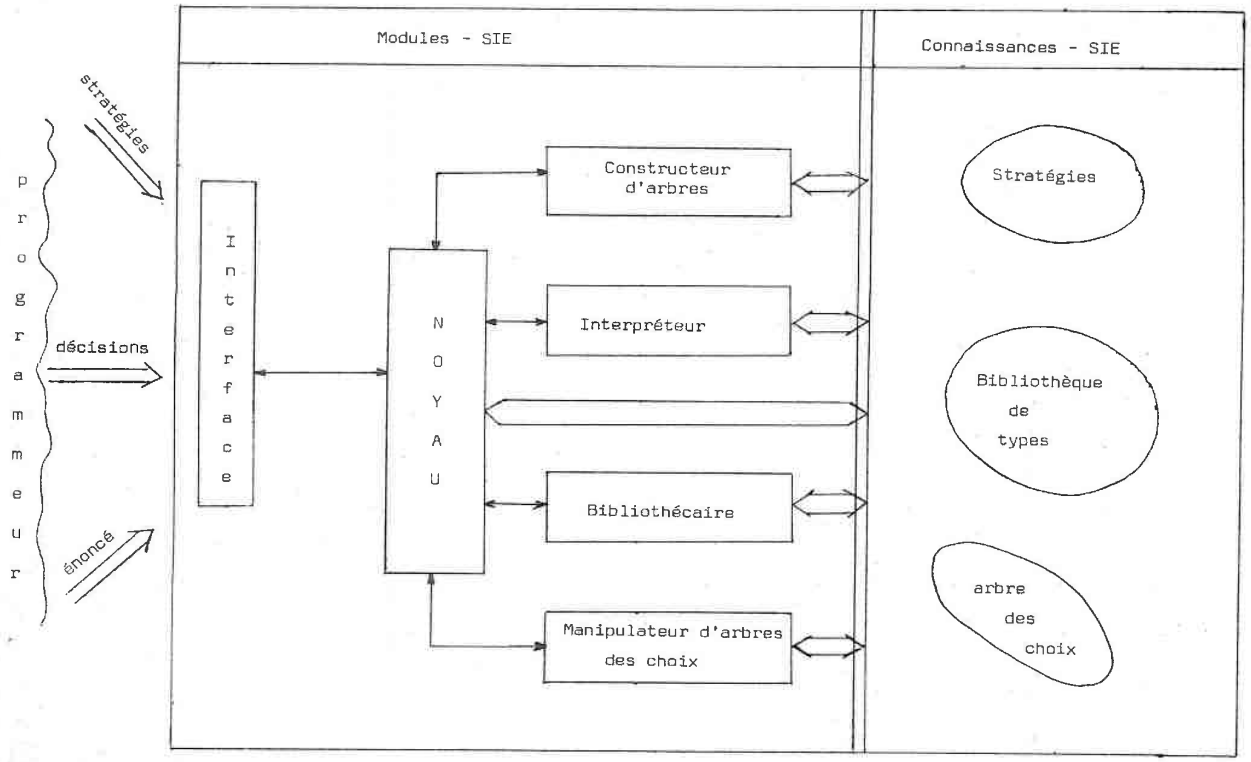


Figure I.4. Vue d'ensemble de SIE

CHAPITRE II

II. UNE RESOLUTION DU PROBLEME DE L'EXPLICITATION D'ENONCES

L'approche constructive mise en oeuvre par le système et décrite dans le chapitre I est fondée sur l'emploi d'une méthode de programmation et surtout sur l'utilisation de règles de constructions.

Cette façon (cf. I.1.2. et analyse § I.3.2.) de résoudre le problème de l'explicitation repose donc sur la notion de transformations d'énoncés.
Ce chapitre a pour but de l'étudier, c'est-à-dire que la solution évoquée au § I.1. est précisée à travers cette notion de transformation.
La transformation d'énoncés proposée, est fondée sur une technique de remplacement de constituants d'énoncés développée au § II.4. La mise en oeuvre de cette technique fait intervenir :
- la notion de règles de constructions,
- la notion d'arbre,

et pose le problème fondamental du choix de la stratégie mis en évidence au § II.6.
Cette technique a pour effet d'assurer le passage d'un énoncé à un autre par l'application d'une règle.
Ce passage s'appuie sur la représentation des énoncés et des règles sous forme d'arbres et amène l'utilisateur à prendre certaines décisions précisées. En fin de chapitre une illustration d'un tel passage est donnée sur un exemple simple.

Nos règles de transformations sont une forme constructive des règles de preuves présentées au § II.1. Ces règles sont définies par le programmeur. Et afin de permettre l'adjonction de nouvelles règles il est utile que le système soit extensible. Ce qui conduit à paramétrer le logiciel en mettant en évidence des classes de règles.
Le paragraphe II.3. présente une classification des règles de constructions. De telles règles traduisent en fait un certain raisonnement du programmeur lui permettant de progresser vers une solution algorithmique.

C'est ainsi que de la notion de règles on passera à celle de stratégies d'explicitations au § II.5.
Cette technique permet donc de voir l'explicitation comme une suite de transformations d'énoncés :

E, l'énoncé de départ, est transformé en E_1 par l'application d'une stratégie sur E,

E_1 en E_2, ..., E_n en E_{n+1}, ...,
pour aboutir à

E_k tel que E_k soit M qui est une solution algorithmique de E.

Ces stratégies sont supposées valides relativement au système formel de construction associé à SIE. En conséquence la transformation de E_i en E_{i+1} par la technique de remplacement prouve que E_{i+1} est un théorème dans ce système formel.
L'application récursive de ces stratégies assure simultanément de la correction de M vis à vis de E. (cf. § I.2.3.).

II.1. Les règles de preuves

II.1.1. Définition d'une règle de preuve

Pour qu'un énoncé Médée M soit une solution algorithmique d'un énoncé Spes E, il faut prouver que :

(Q1) : $\forall d\ \forall r$ Si (r est une solution de M pour d) alors (r est une solution de E pour d).

Plus formellement si SOL(A,d) est l'ensemble des solutions de l'énoncé A pour la donnée d, Q1 s'écrit : (en se plaçant dans un certain système formel SF)

$$SF \models SOL(M,d) \subset SOL(E,d)$$

Ceci est équivalent à prouver que M(d,r) ⇒ E(r,d) est un théorème dans SF :

$$SF \models M(r,d) \Rightarrow R(r,d)$$

Remarque : En pratique on ne veut pas que SOL(M,d) soit vide.

On décompose M de la manière suivante :

r = D(r') et M(r',d) où D est une définition Médée.

Dans ce cas (Q2) devient alors :

(Q3) r = D(r') et M(r',d) ⇒ E(r,d). (r' est un résultat intermédiaire)

Pour établir que (Q3) est un théorème dans SF, il est utile d'effectuer un raisonnement inductif par cas selon la forme de D.

SF contient donc les règles de preuves de la forme (Q3) :

REGP : E et D ⇒ S où

S est un énoncé implicite,

E un énoncé quelconque et D une définition Médée.

Ainsi à chaque définition Médée est associée une règle de preuve (REGP) :

a) La règle de preuve d'une définition itérative :

ITER e_u et $e_{u'}$ et r == der u_i pour i de k jqa u'_i ⇒ e_r

où

- u est une suite dont le dernier terme permet de définir r,
- u' est une suite de booléens dont le premier indice 1 tel que u'_1 = vrai caractérise l'indice du dernier terme de u.

La suite u (resp u') est caractérisée par l'énoncé e_u appelé l'invariant de l'itération (resp. par l'énoncé $e_{u'}$ appelé la partie variante de l'itération).
Par exemple :

e_r : r tq croiss(r) et equ(r,d)
e_{u_i} : u_i tq croiss(u_i) et equ(u_i,$d1_i$)
$e_{u'_i}$: u'_i tq u'_i == (i=n)

(croiss(r) : tableau r croissant, equ(r,d) r est une permutation de d, $d1_i$: la tranche [1..i] du tableau d).

On a bien intuitivement,

e_u <u>et</u> $e_{u'}$ <u>et</u> r == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> u'_i ⇒ e_r.

La justification de cette règle repose sur la sémantique d'une définition itérative. Le lecteur pourra consulter [FIN-77, FIN-78, LES-78] pour une telle définition.

Cependant cette règle si elle est associée étroitement à la sémantique d'une définition itérative, n'est pas directement utilisable puisqu'elle va nous conduire à manipuler une forme explicite de cette sémantique. Par exemple ici il faudrait remplacer

r == <u>der</u> u_i <u>pour</u> i <u>de</u> k <u>jqa</u> u'_i par : r == u_l avec
l <u>tq</u> u'_l == vrai et
$\forall i \in [k..l-1]$ u'_l = faux.

Nous substituerons donc à ITER l'implication suivante :

ITER' $e_u(r,i)$ <u>et</u> $e_{u'}$ <u>et</u> u'_i == vrai ⇒ e_r

L'existence d'une solution est formulée par $e_{u'}$: u'_i == vrai signifie
$\exists i$ <u>tq</u> u'_i = vrai

<u>Remarque</u> : Pour prouver e_u il est souvent nécessaire d'effectuer un raisonnement par récurrence. Une autre étape de la preuve nécessite alors de prouver :

- $e_u(i-1)$ <u>et</u> définition de u_i ⇒ $e_u(i)$,
- $e_u(k-1)$.

b) <u>La règle de preuve d'une définition simple</u>

SIMP (e_{x1} <u>et</u> e_{x2} <u>et</u> ... e_{xn}) <u>et</u> r == f(x1, ... x_n) ⇒ e_r

La justification de cette règle repose sur la sémantique de f.

c) <u>La règle de preuve de la définition conditionnelle</u>

COND e_a <u>et</u> ((a <u>et</u> e_{r1}) <u>ou</u> (¬a <u>et</u> e_{r2})) <u>et</u> r == <u>si</u> a <u>alors</u> r1
<u>sinon</u> r2 ⇒ e_r

où a est un booléen caractérisé par l'énoncé e_a.

par exemple :

e_r : r <u>tq</u> d[i] > 0 ⇒ egal (r,r1)
ou d[i] ⩽ 0 ⇒ egal (r,r2)

e_{r1} : r1 <u>tq</u> egal (r1,di_i)

e_{r2} : r2 <u>tq</u> egal (r2,tvide)

e_a : a <u>tq</u> (a <=> d[i] > 0) <u>ou</u> (¬a <=> d[i] ⩽ 0)

(d[i] : valeur au point i du tableau d, egal : égalité de 2 tableaux).

On a bien intuitivement,

e_a <u>et</u> ((a et e_{r1}) <u>ou</u> (¬a et e_{r2})) <u>et</u> r == <u>si</u> a <u>alors</u> r1
<u>sinon</u> r2 ⇒ e_r

La justification de cette règle repose sur la sémantique de la conditionnelle. Et par analogie à la règle (r1), pour être utilisable, elle nous conduit ici à remplacer :

r == <u>si</u> a <u>alors</u> r1 <u>sinon</u> r2 soit par : r == r1 avec a = vrai,
soit par : r == r2 avec a = faux.

De COND nous dérivons donc les deux implications suivantes :

COND' : a == faux <u>et</u> e_{r2} ⇒ e_r
a == vrai <u>et</u> e_{r1} ⇒ e_r.

d) <u>La règle de preuve du premier élément d'une suite vérifiant une propriété</u>

PREM e_q <u>et</u> (r == <u>prem</u> i <u>dans</u> d <u>depuis</u> n <u>tel que</u> q_i) ⇒ e_r

où

- n ⩽ i ⩽ n', n' étant la limite supérieure de d considéré comme une suite d'éléments,
- q une suite de booléens, caractérisé par l'énoncé e_q, dont le premier indice l tel que q_l = vrai permet de définir r : r == l.

Par exemple :

e_r : r <u>tq</u> k < r ⩽ m <u>et</u>
$\forall j$ k < j < r ⇒ d[j] = 0

e_q : q_i <u>tq</u> q_i <=> ¬d[k] = 0

On a bien, intuitivement,

e_q <u>et</u> r == <u>prem</u> i <u>dans</u> d <u>depuis</u> k + 1 <u>tel que</u> $q_i \Rightarrow e_r$

Il s'agit aussi d'une règle de preuve d'une définition itérative. Elle est une forme simplifiée de (r1).

Ainsi par analogie à (r1) on est conduit à manipuler une forme explicite de la sémantique de cette définition. Mais ici il faudrait remplacer

r == <u>prem</u> i <u>dans</u> d <u>depuis</u> n <u>tel que</u> q_i par r == l avec
l <u>tq</u> q_l = vrai <u>et</u>
$\forall$ j [n..l] q_l = faux

Nous substituerons donc à (r4) l'implication suivante :

PREM' e_q <u>et</u> q_i == vrai <u>et</u> r == i $\Rightarrow$ e_r.

L'existence de la solution est assurée par le fait que i parcourt une suite finie.

Et le cas où : $\forall$ j [n..n'] q_j = faux

C'est-à-dire $\exists$ l <u>tq</u> l $\in$ [n..n'] <u>et</u> q_l = vrai

alors on admettra que :

r == 0.

Une application de ces règles de preuves est illustrée par deux exemples donnés dans le paragraphe suivant.

II.1.2. <u>Exemples d'application des règles de preuves</u>

<u>Exemple 1</u>.

Soit l'énoncé e_t : construire un tableau t de même domaine qu'un tableau d'entiers d donné tels que :

1. à toute valeur "0" associée à un indice i de t correspond dans d à l'indice i, la valeur i, (<u>si</u> d[l] = l <u>alors</u> t[l] = 0)
2. à tout élément v associé à un indice i de t correspond dans d à l'indice i l'élément v. (<u>si</u> d[l] $\neq$ l <u>alors</u> t[l] = d[l])

Une spécification formelle de e_t est donnée par la colonne "énoncé implicite" de la figure II.1.

énoncé implicite	profil	énoncé explicite
t <u>tq</u> dom(t) = dom(d) <u>et</u> $\forall$ l $\in$ [1..bs(d)] d[l] = l $\Rightarrow$ t[l] = 0 <u>et</u> d[l] $\neq$ l $\Rightarrow$ t[l] = d[l]	t : TAB d : TAB [ ] : TAB x ENT $\rightarrow$ VAL dom : TAB $\rightarrow$ INT chval : TAB x ENT x val $\rightarrow$ TAB	u_0 == d t == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i u_i == chval(u_{i-1},i,v_i) v_i == <u>si</u> d[i] = i <u>alors</u> 0 <u>sinon</u> d[i] a_i == (i = bs(d)

Figure II.1.

Quelques notations utilisées dans la suite :

$e_x(\bar{y})$: permet de préciser la liste des paramètres, $\bar{y}$, sur lesquels on peut effectuer une sustitution dans l'énoncé e_x de résultat x.

dk_l désigne le tableau tr(d,k,l), une tranche du tableau d ; l peut être l'indice d'une suite. On emploiera l'une ou l'autre notation de la tranche en fonction des commodités d'écriture. Il en sera de même pour l'opération de concaténation (concat) et le symbole "$\sim$" et pour l'opération "valeur en un point d'un tableau" (acces) et le symbole ("[ ]").

VAL est le type des éléments associés aux indices d'un tableau.

INT est le type intervalle d'entiers.

L'opération "chval"(t,i,v) remplace la valeur associée à i par v.

L'opération "appartenance à un intervalle" (appit ([i..j], k)) est désignée par le symbole "$\in$" (appartenance à un ensemble, cf. chap. V § V.1., k $\in$ [i..j]).

Prouvons que l'énoncé explicite de la figure II.1. est correct vis-à-vis de l'énoncé e_t.

Etape 1 de la preuve

La preuve de la définition de t s'écrit alors :

$e_u(t,i)$ <u>et</u> $a_i == (i = bs(d))$ <u>et</u> a_i = vrai $\Rightarrow e_t$

e_{u_i} : u_i <u>tq</u> $\forall l \in [1..i]$

$d[l] = l \Rightarrow u_i[l] = 0$

et $d[l] \neq l \Rightarrow u_i[l] = d[l]$

et $dom(u_i) = dom(d)$

Cette preuve est immédiate.

Etape 2.

Il faut maintenant prouver $e_u(i)$ pour tout i entre 1 et bs(d).

(1) u_i est donné par une définition simple, on applique la règle correspondante : (pour $i \geqslant 1$).

$e_u(i-1)$ <u>et</u> $u_i == chval(u_{i-1},i,v_i) \Rightarrow e_u(i)$

Il faut remplacer dans e_u, u_i par sa définition :

$e_u(i-1)$ <u>et</u> $u_i == chval(u_{i-1}]i,v_i) \Rightarrow \forall l \in [1..i]$

$d[l] = l \Rightarrow chval(u_{i-1},i,v_i)[l] = 0$

<u>et</u> $d[l] \neq l \Rightarrow chval(u_{i-1},i,v_i)[l] = d[l]$

Ensuite en appliquant la propriété : $chval(t,i,v)[i] = v$ si $t \neq$ tvide, et en procédant par une analyse par cas des résultats de v_i ($v_i = 0$ ou $v_i = d[i]$), on déduit alors par récurrence que $e_u(i)$ est vrai.

(2) u_o est donné par $u_o == d$

et l'on a : $u_o == d \Rightarrow e_u(0)$

On prouve facilement que $e_u(o)$ est vrai :

- $dom(u_o) = dom(d)$ est vraie car $u_o = d$,
- $\forall l \in [1..0]$ $d[l] = l \Rightarrow u_o[l] = 0$ <u>et</u> $d[l] \neq l \Rightarrow u_o[l] = d[l]$ est vraie car $[1..0] = \phi$.

(3) La définition de v_i est à prouver :

v_i est donné par une définition conditionnelle. La preuve de la définition de v_i s'écrit alors :

$d[i] = i$ <u>et</u> $v_i = 0 \Rightarrow e_v$

<u>non</u> $d[i] = i$ <u>et</u> $v_i = d[i] \Rightarrow e_v$

e_{v_i} : v_i <u>tq</u> $(d[i] = i$ <u>et</u> $v_i = 0)$ <u>ou</u> $v_i = d[i]$.

La preuve est immédiate.

Exemple 2.

L'énoncé informel de e_t est : construire un tableau t constitué de tous les éléments positifs d'un tableau d donné.

La spécification formelle de e_t est donnée dans la figure II.2.

Enoncé implicite	Profil	Enoncé explicite
t tq ssuite(t,d) et $\forall k$ [1..bd(d)] t[k] > 0 et bs(t) = nbrintq(d,d[l]> 0)	t : TAB d : TAB ssuite : TAB x TAB → BOOL nbrintq : TAB → ENT [ ] : TAB x ENT → VAL	u_o == tvide t == der u_i pour i de 1 jqa a_i u_i == si d[i]>0 alors (concat(u_{i-1},di_i) sinon u_{i-1} a_i == (i = bs(d))

Figure II.2.

Prouvons que l'énoncé explicite de la figure II.2. est correct vis-à-vis de e_t (preuve analogue à l'exemple 1)

Etape 1.

La preuve de la définition de t s'écrit :

$e_u(t,i)$ <u>et</u> $a_i == (i = bs(d))$ <u>et</u> a_i = vrai $\Rightarrow e_t$

Ce qui est immédiat.

e_u : u_i <u>tq</u> ssuite(u_i,$d1_i$) <u>et</u> $u_i[i] > 0$

et $bs(u_i)$ = nbrintq($d1_i$, $d1_i[l] > 0$).

(nbrintq(d,P) rend le nombre d'indices de d dont tous les éléments vérifient P : Card({l <u>tel que</u> $l \in [1..bs(d)]$ <u>et</u> P(d[l])})

Etape 2.

A présent prouvons $e_u(i)$ pour tout i entre 1 et bs(d).

(1) u_i est donné par une définition conditionnelle, la preuve de u_i (pour $i \geqslant 1$) s'écrit alors :

$d[i] > 0$ et $u_i = concat(u_{i-1}, di_i) \Rightarrow e_u(i)$

non $d[i] > 0$ et $u_i = u_{i-1} \Rightarrow e_u(i)$.

* Cas $d[i] > 0$ on a :

$e_u(i-1)$ et $u_i == concat(u_{i-1}, di_i) \Rightarrow e_u(i)$

c'est-à-dire

$e_u(i-1)$ et $u_i == concat(u_{i-1}, di_i) \Rightarrow ssuite(u_{i-1} \sim di_i, d1_i)$

et $u_{i-1} \sim di_i[i] > 0$

et $bs(u_{i-1} \sim di_i) = nbrintq(d1_i, d1_i[1] > 0)$

En appliquant les propriétés :

- $d1_i = d1_{i-1} \sim di_i$
- $ssuite(t_1 \sim x, t2 \sim x) = ssuite(t_1, t_2)$
- $d_1 \sim d_2[i] = d_2[i]$ si $i = bs(d_1 \sim d_2)$
- $bs(t \sim di_i) = bs(t) + 1$
- $nbrintq(d1_{i-1} \sim di_i, P) = nbrintq(d1_{i-1}, P) + 1$ si $P(d[i])$

On déduit alors dans ce cas là par récurrence que $e_u(i)$ est vrai.

* Cas non $d[i] > 0$ on a :

$e_u(i-1)$ et $u_i == u_{i-1} \Rightarrow ssuite(u_{i-1}, d1_i)$ et

$u_{i-1}[i] > 0$

et $bs(u_{i-1}) = nbrintq(d1_i, d1_i[1] > 0)$

En plus des propriétés précédentes, on considère les suivantes :

- $ssuite(t, d \sim t') = ssuite(tvide, t')$ si $ssuite(t, d)$
- $ssuite(tvide, t) = vrai$
- $nbrintq(d1_{i-1} \sim di_i, P) = nbrintq(d1_{i-1}, P)$ si non $P(d[i])$

(le tableau u_i sur le domaine $[1..i]$ est égal au tableau u_{i-1} sur le domaine $[1..i-1]$) ;

ce qui permet alors de déduire par récurrence que dans ce cas là $e_u(i)$ est vrai.

(2) u_0 est donné $u_0 == tvide$

et l'on a :

$u_0 == tvide \Rightarrow e_u(0)$.

On prouve facilement avec les axiomes de ssuite et tranche (tr) que $e_u(0)$ est vrai :

- $tr(d,1,0) = tvide$
- $ssuite(tvide, tvide) = vrai$
- $bs(tvide) = 0$
- $nbrintq(tvide, P) = 0$.

Ces deux exemples permettent de montrer que l'utilisation des règles REGP pour construire des énoncés explicites nécessite de véritables démontrations.

Ces démontrations consistent principalement à justifier l'introduction d'une définition Médée en utilisant une famille de règles de preuve.

Il apparaît, ici, qu'une partie du travail effectué est inutile.

Par exemple :

Pour construire une définition itérative ITER est choisie. On applique alors la règle de preuve ITER' pour justifier l'introduction de cette définition. Et ensuite on cherche à prouver que cette application est autorisée, c'est-à-dire correcte, pour pouvoir enfin affirmer que la définition introduite est correcte vis-à-vis d'un énoncé de départ.

Au lieu d'essayer de prouver que l'application de REGP est correcte, il est plus raisonnable de chercher à utiliser cette preuve pour construire une définition correcte vis-à-vis d'un énoncé de départ.

Ce qui évitera alors de faire la preuve de cette définition Médée.

Par exemple, en prenant le cas simple de $u_0 == tvide$ (exemple précédent) nous avons à prouver $e_u(0)$; la règle appliquée est

$u_0 == tvide \Rightarrow e_u(0)$.

Notre but est d'utiliser la preuve de $e_u(0)$ pour construire $u_0 == tvide$.

Pour qu'il en soit ainsi, les règles de preuves vont être alors remplacées par des règles de constructions.

II.2. Les règles de constructions

II.2.1. La forme des règles

L'utilisation d'une règle de preuve REGP

E et D ⇒ S,

comme illustrée au § II.1.2, nécessite de faire la preuve de S étant donnée une définition D introduite. Mais comme notre objectif est d'utiliser cette preuve pour construire précisément D alors toute règle REGP peut être vue comme une "équation" à domaine dans les énoncés, où les inconnues sont E et D.

Le problème consiste alors à mettre en évidence des techniques de résolution systématiques de ces équations. On peut voir ces techniques comme des fonctions qui étant donné un énoncé implicite S rendent comme valeurs un couple (E,D).

Expliquons cette technique à travers un exemple simple sur le calcul algébrique.

Soit l'équation ax = b,

on appelle technique de résolution la méthode qui permet de passer de l'équation précédente à la définition $x = \frac{b}{a}$.

Cette méthode, ici, est liée à une propriété de la multiplication des réels à savoir l'existence d'un symétrique de tout nombre non nul.

Cet exemple illustre ce que l'on entend par la mise en évidence de techniques de résolutions et l'utilisation qu'on en fait.

L'utilisation de la propriété de la multiplication a permis d'écrire l'équation $x = \frac{b}{a}$.

Par analogie, nous voulons associer aux "équations" E et D ⇒ S une forme plus constructive :

S :: E,D. (il n'existe pas d'ordre en partie droite de "::")

On appelle une telle équation une règle de construction (RDC).

Une telle règle s'interprète de la façon suivante :

A une forme donnée d'un énoncé S on associe un sous énoncé D et un énoncé intermédiaire E tel que :

E et D ⇒ S soit un théorème dans un certain système formel SF, associé à SIE.

Ce système formel de construction peut être défini comme suit :

- les énoncés (formules) sont les énoncés Spes et les définitions simples de Médée,
- les axiomes : ceux du CP1, l'axiome SIMPC et les axiomes algébriques des types prédéfinis,
- les règles d'inférences :
  - . celles du CP1 (modus-ponens, substitution),
  - . les règles ITERC, CONDC, PREMC.

ITERC, SIMPC, CONDC, PREMC sont les règles de constructions des règles de preuves ITER, SIMP, COND, PREM respectivement.

II.2.2. Correspondance règles de constructions - règles de preuves.

Aux règles de preuves REGP (cf. § I.1.1.) vont correspondre alors les règles de constructions RDC :

a) La règle de construction d'une définition itérative (ITERC) :

e_r :: r == der u_i pour i de k jqa u'_i , $e_{u'}$ et e_u

Cette règle introduit une définition itérative correcte vis-à-vis de e_r, e_u (l'invariant), $e_{u'}$ (la condition d'arrêt) telle que

u'_i = vrai et n == μ(i) et e_u $[i \leftarrow n]$ ⇒ e_r $[r \leftarrow u_n]$ soit valide dans SF. ($[i \leftarrow n]$: le remplacement de i par n)

μ(i) rend la plus petite valeur de i pour laquelle u'_i est vrai.

Reprenons l'exemple illustrant la règle r1 § I.1.1. :

e_r : r tq croiss(r) et equ(r,d)

e_{u_i} : u_i tq croiss(u_i) et equ(u_i,$d1_i$)

$e_{u'_i}$: u'_i tq u'_i == (i = bs(d))

On a bien :

u'_i = vrai <u>et</u> n = μ(i) <u>et</u> e_u [i ← n] ⇒ e_r [r ← u_n]
(μ(i) = bs(d))

Cette implication est en fait ITER' (§ II.1.1).

b) <u>La règle de construction d'une définition simple</u>

e_r :: r == f(x_1, ..., x_n), e_{x1} <u>et</u> e_{x2} <u>et</u> ... e_{xn}.

Cette règle introduit une définition simple, caractérisée par un symbole fonctionnel f n-aires, correcte vis-à-vis des énoncés e_{x1}, e_{x2}, e_{xn} telle que

e_{x1} <u>et</u> e_{x2} ... <u>et</u> e_{xn} ⇒ e_r [r ← f(x1,...,xn)] soit valide dans SF.

Par exemple : considérons l'exemple 1 du § II.1.2.

e_{u_i} : u_i <u>tq</u> dom(u_i) = dom(d)
∀ l ∈ [1..i] d[l] = 1 ⇒ u_i[l] = 0
<u>et</u> d[l] ≠ 1 ⇒ u_i[l] = d[l]

e_{v_i} : v_i <u>tq</u> (d[i] = i <u>et</u> v_i = 0) <u>ou</u> v_i = d[i]

On a bien :

$e_{u_{i-1}}$ <u>et</u> e_{v_i} ⇒ e_u [u_i ← chval(u_{i-1},i,v_i)]

en employant la preuve donnée à l'étape 2 de cet exemple 1.
(analyse par cas, et par récurrence) permettant alors d'introduire $e_{u_{i-1}}$,e_{v_i} <u>et</u> u_i == chval(u_{i-1},i,v_i) correcte vis-à-vis de $e_{u_{i-1}}$ et e_{v_i}.

c) <u>La définition conditionnelle</u>

e_r :: r == <u>si</u> a <u>alors</u> r1 <u>sinon</u> r2, (a <u>et</u> e_{r1}) <u>ou</u> (non a <u>et</u> e_{r2})
<u>et</u> e_a

Cette règle peut être vue comme une définition simple particulière.
Elle introduit une conditionnelle correcte vis-à-vis des énoncés e_r, e_{r1} et e_{r2} telle que :

- a <u>et</u> e_{r1} ⇒ e_r [r ← r1] est valide dans SF,
- <u>non</u> a <u>et</u> e_{r2} ⇒ e_r [r ← r2] est valide dans SF.

Pour un exemple se rapporter à celui donné pour la règle de preuve COND.

d) <u>La règle de construction de la définition "prem" (PREMC)</u> :

PREMC e_r :: r == <u>prem</u> i <u>dans</u> d depuis n <u>tel que</u> q_i, eq

Cette règle introduit la définition itérative "prem" correcte vis-à-vis de e_r et e_{q_i} telle que

e_q <u>et</u> q_1 = vrai <u>et</u> r = 1 ⇒ e_r [r ← 1] qoit valide dans SF.

Comme pour REGP, ces RDC conduisent à prouver certaines propriétés pour pouvoir les appliquer. En fait ces règles ne sont pas fonctionnelles : à e_r on peut associer plusieurs (une infinité) explicitations. Mais la mise en évidence de techniques de résolution particulières va permettre de les rendre effectivement fonctionnelles.
L'idée consiste à particulariser ces RDC en particularisant la forme des énoncés. Les règles de constructions ainsi obtenues prennent le nom de "stratégies d'explicitations locales" (cf. II.5).
Cette particularisation des RDC implique la nécessité de les classifier dans le but de faciliter la paramétrisation du système SIE. (cf. § II.3). Un exemple d'application de règles de constructions est illustré dans l'annexe 1.

II.2.3. <u>La validité d'une règle de construction</u>

Une règle de construction S :: E,D est dite valide relativement à SF si et seulement si

E <u>et</u> D ⇒ S est valide.

La validité d'une règle est fondamentale pour la construction dans la mesure où elle assure qu'au cours d'une explicitation il n'y aura pas de transformations contradictoires.
Dans la suite nous supposerons que les RDC, voire les stratégies, d'explicitations locales sont valides, et nous ne développerons pas leur preuve dans ce travail.
En conséquence l'application récursive des règles RDC, ou plutôt des stratégies d'explicitations locales, produit un énoncé Médée M correct vis-à-vis de l'énoncé E initial.

II.3. Quelques classes de règles

Les techniques de résolution discutées au § II.2.1. sont abordées de façon intuitive. On ne sait pas créer systématiquement une règle. D'ailleurs ce problème de recherche de règles (de stratégies) est posé au § III.1.
Une idée simple consiste alors à classifier les règles de constructions afin d'aider au mieux le programmeur à découvrir des techniques de résolutions, c'est-à-dire à définir des règles.
Cette classification est donc liée à la notion d'extensibilité du système SIE ; elle structure les paramètres du logiciel.

On distingue pour le moment quatre classes de règles. Ces classes se différencient entre elles par la forme de la partie droite d'une règle. Ainsi les quatre formes générales de règles sont :

1. La forme la plus générale est donnée par :

S :: E,D.

Les règles de cette forme sont dites appartenir à la classe des règles non-terminales, appelées nt-règles,

2. Les règles sans définition explicite en partie droite, c'est-à-dire de la forme :

S :: E

définissent la classe des règles logiques, appelées l-règles ou p-règles suivant qu'il s'agisse d'une règle exprimant une transformation portant sur un connecteur logique ($\Rightarrow$, $\forall$, $\Leftrightarrow$, $\neg$, ou, et) ou sur une propriété d'un prédicat,

3. Les règles avec des définitions explicites en partie droite, c'est-à-dire de la forme :

S :: D

définissent la classe des règles terminales, appelées t-règles,

4. Les règles dont la partie droite est composée uniquement d'une opération 0-aire définissent la classe des règles valeurs, appelées les v-règles. La forme d'une v-règle est :

S :: D telle que D est une opération 0-aire.

II.3.1. La classe des règles non-terminales

Cette classe est constituée de toutes les règles de constructions particulières dont les formes générales, au nombre de quatre, ont été définies au II.2.2. La classe des nt-règles comprend donc :
- les nt-règles particulières introduisant une définition itérative ITERC,
- les nt-règles particulières introduisant une définition simple SIMPC,
- les nt-règles particulières introduisant une conditionnelle CONDC,
- les nt-règles particulières introduisant la définition itérative simplifiée PREMC.

Donnons quelques exemples de nt-règles

- une nt-règle ayant la forme d'une ITERC :

t <u>tq</u> $\forall$ k R(t,d) <u>et</u> P(t[k])	::	t == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i
t : TAB d : TAB		u_i <u>tq</u> R(u_i,$d1_i$) <u>et</u> P(u_i[i])
		a_i == i = bs(d)

- une nt-règle ayant la forme d'une SIMPC :

u_i <u>tq</u> R(u_i,$d1_i$)	::	u_i == concat(u_{i-1},m_i)
u : SUITE-DE-TAB d : TAB		m_i <u>tq</u> R(u_{i-1},m_i) <u>et</u> m_i = di_i

- Une nt-règle ayant la forme d'une CONDC :

r <u>tq</u> (P(r1) <u>et</u> r = r1) <u>ou</u> r = r2	::	r == <u>si</u> a <u>alors</u> r1 <u>sinon</u> r2
r : $T r1 : $T r2 : $T		a <u>tq</u> a <=> P(r1)

- une nt-règle ayant la forme d'une PREMC :

r <u>tq</u> k < r $\leqslant$ m <u>et</u>	::	r == <u>prem</u> i <u>dans</u> d <u>depuis</u> k + 1
$\forall$j k < j < r $\Rightarrow$ P(d,j)		<u>tel que</u> a_i
d : SUITE-DE-VAL		a_i <u>tq</u> P(d,j) $\Leftrightarrow$ a_i

Nous reviendrons sur cette classe de règles au II.5 où la notion de stratégies d'explicitations est introduite.

II.3.2. La classe des règles logiques

Deux sortes de règles logiques caractérisent cette classe.

a) Les l-règles dont le but est de donner a un énoncé e_r une autre forme logique en se guidant principalement par les propriétés des opérateurs logiques.

Par exemple la règle informelle suivante :

"un énoncé S de la forme "vrai et Q(u)" peut être transformé en un énoncé de la forme "Q(u)" ",

est valide car la sémantique de l'opérateur logique "et" permet une telle transformation de S en S' = Q(u).

Cette l-règle s'écrit :

vrai <u>et</u> Q(u) :: Q(u)

Donnons d'autres exemples formels de l-règles.

1. $P(u_{i-1})$ <u>et</u> $Q(u_{i-1}) \Rightarrow P(u_i)$ <u>et</u> $Q(u_i)$
   ::
   $P(u_{i-1})$ <u>et</u> $Q(u_{i-1}) \Rightarrow P(u_i)$
   <u>et</u> $P(u_{i-1})$ <u>et</u> $Q(u_{i-1}) \Rightarrow Q(u_i)$

2. P(u) <u>et</u> Q(u) ⇒ P(u) <u>et</u> Q'(u)
   ::
   Q(u) ⇒ Q'(u)

3. P <u>et</u> Q ⇒ c :: P ⇒ (Q ⇒ c)

4. P ⇒ Q <u>et</u> P' ⇒ Q :: P <u>ou</u> P' ⇒ Q

5. ∀k (P ⇒ Q(k)) :: P ⇒ ∀k Q(k)

6. P(f(x)) :: P(y) <u>et</u> y = f(x)

On peut remarquer que la plupart de ces règles sont inversibles.

Ainsi par exemple l'utilisateur devra introduire la règle 6' :

P(y) <u>et</u> y = (f(x)) ⇒ P (f(x))

(le système ne permet pas à l'utilisateur d'indiquer si une règle est inversible).

On ne dispose pas d'une liste exhaustive de ces l-règles.

On voudrait trouver un ensemble minimal de telles règles adapté à la transformation d'énoncés.

Ce problème n'a pas été étudié pour le moment.

b) Les p-règles dont le but est de transformer un énoncé implicite en appliquant une propriété particulière des prédicats.

Voici quelques exemples :

On peut définir croiss (par une relation d'ordre R) par :

croiss(tvide) = vrai

croiss(t) = vrai <u>si</u> lg(t) = 1 (lg rend la longueur d'un tableau)

croiss(affect(t,i,v)) = <u>si</u> R(t[i - 1], v) <u>alors</u> croiss(t)
<u>sinon</u> faux

D'où, par exemple la règle (propre au "TABLEAU")

1. croiss(t) :: croiss(tr(t,1,bs(t) - 1)) <u>et</u> t [bs(t) - 1] ⩽ t[bs(t)]

En effet, on peut réécrire la partie gauche de cette p-règle par sa partie droite, car on a :

croiss(tr(t,1,bs(t) -1)) <u>et</u> (t[bs(t) -1] ⩽ t[bs(t)]) ⇒ croiss(t).

2. croiss(tr(d,i,i)) ⇒ vrai

   croiss(tr(d,i,i)) :: vrai (nous verrons cette propriété comme une V-règle)

   ou encore

- $\boxed{\text{croiss(t) } \underline{\text{et}} \text{ ssuite(u,t)} \Rightarrow \text{croiss(u)}}$
- $\boxed{\text{egal(u1}_1\text{,t) } \underline{\text{et}} \text{ egal(u1+1}_{\text{bs(u)}}\text{,d) :: egal(u, concat(t,d)).}}$

Les deux premières p-règles sont des propriétés caractéristiques de croiss ; et on peut noter que le symbole "::" a la sémantique de "$\Leftrightarrow$" (l'équivalence).

Les deux dernières p-règles expriment des propriétés générales sur le type tableau. Et ainsi, par exemple, pour construire un tableau u "croissant' on peut utiliser la troisième p-règle :

croiss(u) :: croiss(t) et ssuite(u,t).

II.3.3. La classe des règles terminales

Une t-règle, S :: D, exprime que D est une solution algorithmique de S. Par conséquent l'application d'une telle règle n'introduit pas de nouvelles définitions implicites.

Les t-règles comme les précédentes sont définies par le programmeur. Cependant certaines t-règles particulières peuvent être créées systématiquement.

Ces t-règles sont telles que :

S est composé d'un seul prédicat p qui est une opération du type abstrait T caractérisant le contexte du problème que l'on cherche à expliciter,

et D est une version itérative de la définition algébrique de l'axiome de l'opération p.

Dans ce cas précis il est possible de systématiser le passage de la définition récursive de p à une définition itérative Médée. Le passage d'un énoncé récursif à un énoncé itératif est un problème connu et maîtrisé [ARS-82, BUR-77, DAR-77].

Cette classe de règles est actuellement ignorée du système.

Donnons quelques exemples de t-règles.

1. Les axiomes sur l'égalité de 2 tableaux : (cf. type TABLEAU chap. V)

```
si t ≠ tvide  egal(t,tvide) = faux      ; egal (tvide, tvide) = vrai
si t ≠ tvide  egal(tvide,t) = faux
egal(affect(t,i,v),affect(t',i',v')) = si v = v' et i = i'
                                        alors egal(t,t')
                                        sinon faux
```

permettent d'écrire la t-règle

```
egal(t,t') :: b == si bs(t) = bs(t') alors
                                     si bs(t) = 0 alors b == vrai
                                                  sinon
                                       b == der u_i pour i de 1 jqa a_i
                                       u_i == (t[i] = t'|i|)
                                       a_i == (i = bs(t) ou non u_i)
                                     sinon
                                       b == faux
```

Cette t-règle, que l'on doit pouvoir déduire des axiomes, peut s'appliquer dans le cas où l'on veut tester si 2 tableaux sont égaux.

2. $\forall j \quad i \leqslant j \leqslant k \leqslant bs(t) \Rightarrow (P(t[k] \ \underline{et} \ \text{non } P(t[j]))$

$::$

$k == \underline{prem}\ j\ \underline{dans}\ t\ \underline{depuis}\ i\ \underline{tel\ que}\ P(t[j])$

3. $\text{egal(u1}_1\text{,t) } \underline{\text{et}} \text{ egal(u1+1}_{\text{bs(u)}}\text{,d)}$

$::$

$u == \text{concat(t,d)}$

On peut remarquer que cette dernière t-règle a la même partie gauche que la quatrième P-règle (§ II.3.2.) produisant comme résultat : egal(u,t ∿ d).

Ce qui montre que les règles ne sont pas des fonctions.

II.3.4. La classe des règles "valeurs"

Les règles "valeurs" expriment que les transformations portent sur des objets et non sur des énoncés. L'application d'une telle règle montre que l'utilisateur s'intéresse à la sémantique (la valeur) d'un objet particulier.
Une v-règle peut être définie systématiquement à partir des axiomes des types. Le programmeur, bien sûr, est libre de définir des v-règles qui ne sont pas des axiomes particuliers d'un type.
Exemples de v-règles.

1. L'axiome : croiss(tvide) = vrai du type TABLEAU
   se réécrit en une v-règle sous la forme :
   `croiss(tvide) :: vrai`
   Il en est de même pour
2. L'axiome de l'opération concat :
   `concat(tvide,tvide) :: tvide`
3. L'axiome de l'opération "nombre d'indices vérifiant une propriété",
   `nbrintq(tvide,P) :: 0`
4. Une v-règle qui n'est pas un axiome du type TABLEAU :
   `tr(d,i,i) :: d[i]` (rappelons que "[]" donne la valeur au point i)
5. Encore une v-règle qui n'est pas un axiome
   `P et Q ⇒ Q :: vrai`

Pour le moment notre attention s'est portée uniquement sur les v-règles issues d'axiomes de types abstraits.
Une utilisation des v-règles consiste à calculer les valeurs initiales de variables. Et dans certains cas ce calcul peut se faire systématiquement. Nous reviendrons sur ce dernier point dans le chapitre III.

Comme certains axiomes sont munis de pré-conditions, il faudrait alors pouvoir les exprimer dans les v-règles. Par exemple, l'axiome sur les tranches d'un tableau :

si i > j tr(d,i,j) = tvide est muni de la pré-condition "i > j" ; on désire intégrer cette pré-condition dans la v-règle associée à cet axiome.
De manière plus générale pour toute règle appartenant à l'une quelconque des classes précédemment vues on voudrait, si nécessaire, exprimer dans celle-ci des pré-conditions.

Le § II.5. montre comment une pré-condition s'insère dans la forme générale d'une règle de construction. Ainsi l'axiome précédent s'écrit sous la forme de la v-règle suivante :

{i > j} tr(d,i,j) :: tvide

A ce stade de nos connaissances suffisantes sur les règles et les énoncés implicites Spes, la question est de savoir comment appliquer concrètement ces règles à de tels énoncés.
En effet, il faut maintenant préciser une technique de transformation d'énoncé par applications de règles de constructions.

II.4. Mise en oeuvre des règles de construction : La technique de remplacement

II.4.1. Une idée générale de la technique

Soient E l'énoncé explicite de la figure II.2, et S l'énoncé implicite de cette même figure.
E est déduit de S en remplaçant progressivement certains de ses constituants par d'autres définitions.
La technique de réalisation de ces remplacements consiste à voir ces énoncés comme des arbres :
on retrouve ainsi la notion bien connue d'arbre abstrait ("structure profonde" des linguistes).

Par exemple S est représenté par un arbre dont le constituant "ssuite(t,d)" est un sous arbre de S. De même l'objet k peut être vu comme un sous arbre de "t[k] > 0".

La notion de remplacement de sous arbres tirée de [ROS-73] va permettre de définir celle de transformation.

* Définition de la transformation

Soit la règle r_j : S_j :: B_j où S_j et B_j sont des arbres.
Une transformation d'un arbre S en un arbre S' par la règle r_j consiste à remplacer dans S, un sous arbre E de S par un arbre B.
La définition de B nécessite :

1. d'identifier S_j à un sous arbre E de S ;
on parlera alors de l'existence d'un filtre d'identification, σ, entre S et S_j,

2. de remplacer dans B_j les paramètres formels.
On parlera alors du calcul du filtre σ',

et de la sorte l'arbre B sera défini par :

$$B = \sigma'(\sigma B_j)$$

Illustrons tout de suite les calculs de σ et σ' par un exemple.

* Un exemple de calculs de filtres

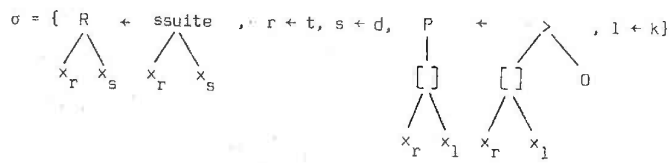
Soit l'énoncé de la figure II.2. simplifié :

t tq ssuite(t,d) et ∀k t[k] > 0, (d est considéré comme une suite de tableau),

et étant donnée la règle :

r_j : R(r,s) et ∀l P(r[l]) :: r == der u_i pour i de 1 jqa a_i
u_i tq R(u_i,s_i) et P(u_i[i])
a_i == (i = bs(s)),

alors le filtre σ est donné par :

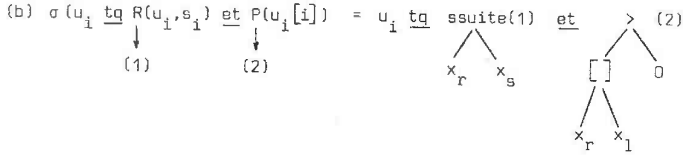


x_r,x_s,x_l sont des paramètres formels que l'on va retrouver dans σ B_j comme suit :

appliquons σ à B_j :

(a) σ (r == der u_i pour i de 1 jqa a_i) = t == der u_i pour i de 1 jqa a_i

(b) σ (u_i tq R(u_i,s_i) et P(u_i[i]) = u_i tq ssuite(1) et > (2)



(c) σ (a_i == (i = bs(s)) = a_i == (i = bs(d))

Le filtre σ' est donné par :
en considérant σ, σB_j (plus particulièrement (b) puisque c'est là qu'apparaissent "R" et "P" des variables fonctions de B_j), et les couples paramètres formels - paramètres effectifs à savoir
{(x_r,u_i), (x_s,s_i), (x_l,i)}

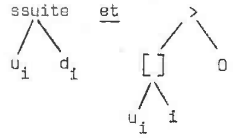
On obtient :

$\sigma_{(1)}$ = {$x_{r(1)}$ ← σ(u_i), $x_{s(1)}$ ← σ(s_i)} c'est-à-dire en composant σ :

$\sigma_{(1)}$ = {$x_{r(1)}$ ← u_i, $x_{s(1)}$ ← d_i}, et

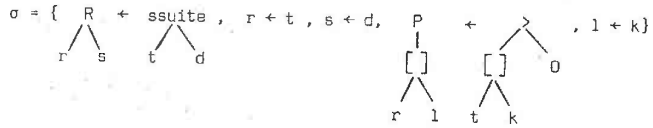
$\sigma_{(2)}$ = {$x_{r(2)}$ ← u_i, $x_{l(2)}$ ← i}.

(b) devient alors : u_i tq ssuite et >



Cet exemple soulève un certain nombre de questions, étudiées et résolues dans le prochain paragraphe (§ II.4.2.) :

1. Comment les paramètres formels x_r, x_s, x_l sont-ils introduits dans σ? En effet, sans l'introduction de ces paramètres on aurait eu :



2. Quelle est la définition et quel est le rôle des paramètres formels et effectifs ?

3. Quel est le rôle et comment se calcule l'ensemble des substitutions $\sigma_{(n)}$ désigné par σ' ?

II.4.2. Les arbres

La section précédente a permis de présenter les points suivants :

- un énoncé est représenté sous forme arborescente,
- la transformation consiste à remplacer une structure de E par une autre également arborescente, en appliquant une règle.

Le mécanisme de remplacement de structures arborescentes est décrit plus précisément dans cette section.

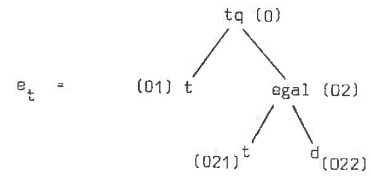
II.4.2.1. Définitions et arbres bien formés

On cherche à décrire "l'arbre" d'un énoncé. Exprimons intuitivement une telle description par un exemple simple :

- Soit l'énoncé e_t :

t tq egal(t,d),

l'arbre e_t est donné par :



tel que l'ensemble $dom(e_t) = \{0, 01, 02, 021, 022\}$ caractérise le squelette de l'arbre t,

et les éléments de l'ensemble $V' = \{tq, t, egal, t, d\}$ soient des étiquettes associées à chaque élément de $dom(e_t)$.

a) Définition d'un arbre

L'exemple ci-dessus définit intuitivement un arbre comme un squelette étiqueté par un ensemble de symboles. Ce squelette se trouve être un sous ensemble de $\mathbb{N}_+^*$ dont les éléments sont des chaînes finies sur $\mathbb{N}_+^*$.

* Définitions d'un squelette d'arbre

Soit $\mathbb{N}_+^*$ l'ensemble des chaînes sur $\mathbb{N}_+^*$ muni des opérations suivantes :

- ε : la chaîne vide,
- $*$: l'opération de concaténation sur $\mathbb{N}_+^*$,
- $|\ |$: la longueur d'une chaîne $\omega \in \mathbb{N}_+^*$, $(|\omega|)$
- ders : une fonction qui rend le dernier élément d'une chaîne $x \in \mathbb{N}_+^*$ $(ders(x))$,
- pour toute chaîne ω et tout $k \in \mathbb{N}$:
  . $k : \omega$ produit la chaîne formée des k premiers caractères de ω.

(ignorons le $k > |\omega|$, cela se justifiera par la définition d'un squelette d'arbre).

A présent définissons les fonctions p (père) et fg (frère gauche) sur $\mathbb{N}_+^*$:

1. $p(\alpha) = (|\alpha| - 1) : \alpha$ pour $\alpha \neq \varepsilon$,
2. $fg(\alpha) = p(\alpha) * (ders(\alpha) - 1)$ pour $\alpha \neq \varepsilon$ et $ders(\alpha) \neq 0$.

- Définition

Un squelette d'arbre est un sous-ensemble fini de $\mathbb{N}_+^*$ stable pour les opérations père et frère gauche.

- Définition

Soit un ensemble de symboles V, muni d'une arité a :

$$a : V \to \mathbb{N}_+ ,$$

Un squelette étiqueté E que l'on appellera arbre est :
une fonction partielle

$$E : \mathbb{N}_+^* \to V$$

telle que le domaine de E, noté dom(E), soit un squelette.

Un énoncé arborescent E est un squelette étiqueté par les symboles de l'énoncé E. On a alors :

$$\forall \alpha \in dom(E) \quad p(\alpha) \in dom(E) \quad \text{et} \quad fg(\alpha) \in dom(E).$$

L'arbre E (l'énoncé arborescent E) est dit être bien formé si et seulement si :

$$\forall \alpha \in dom(E) \quad \forall i \ [1..a(E(\alpha))] \quad \alpha * i \in dom(E).$$

(E(α) désigne le symbole étiquetant l'occurrence $\alpha \in dom(E)$).

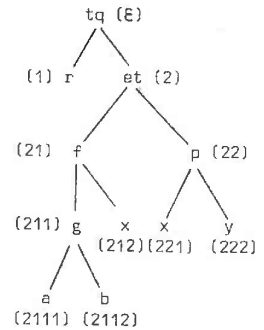
On définit le prédicat feuille et une fonction produisant tous les fils d'une occurrence "père" d'un squelette d'arbre :
Soit E un arbre :

1. $feuille(\alpha, E) = \forall \beta \in dom(E) \quad \neg (P(\beta) = \alpha)$
2. $p^{-1}(\beta) = \{\alpha \in dom(E) \mid p(\alpha) = \beta\}$ ($p^{-1}(\beta)$ est l'ensemble des fils de β)

L'exemple ci-après résume ce qui vient d'être dit sur les arbres :

Soit l'énoncé E : r tq f(g(a,b),x) et p(x,y)

L'arbre E est



- le domaine de E est :

dom(E) = {ε,1,2,21,22,211,212, 2111,2112,221,222}

- les éléments de dom(E) sont appelés occurrences.

- L'occurrence de "et" est 2 ; 2 est en quelque sorte l'adresse de "et" dans E : E(2) = "et" , E (222) = "y" ...

- Les opérations p et fg appliquées à l'occurrence 22 rendent comme résultat :

  $p(22) = (|22|) - 1) : 22 = 2$,

  $fg(22) = p(22) * (ders(22) - 1) = 21$,

  fg(21) n'existe pas car l'occurrence $20 \notin dom(E)$ ce qui signifie qu'aucune occurrence n'est adressée par 20

- Les opérations feuille et p^{-1} appliquées respectivement à 212 et 21 :

  feuille(212,E) = vrai car $\forall \beta \in dom(E)$ on a bien $\neg p(\beta) = 212$ ou encore "x" = E(212) n'a pas de fils.

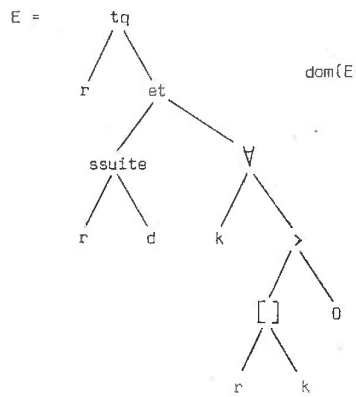
  $p^{-1}(21) = \{211,212\}$

  C'est-à-dire les fils directs (les descendants immédiats) de "f" à l'occurrence 21 sont "g" à l'occurrence 211 et "x" à l'occurrence 212.

b) Définition d'un arbre "attribué"

La transformation opère sur des arbres bien formés présentant une particularité ; illustrons la par un exemple. Pour cela l'énoncé de la figure II.2 est choisie : (voir § II.4.1.)

L'arbre de l'énoncé E : r tq ssuite(r,d) et $\forall$ k r[k] > 0 est :



dom(E) = {ε,1,2,21,22,211,212,221, 222,2221,2222,22211,22212}

On dit que l'arbre E est "attribué" pour dire que toute feuille de E possède un attribut :

. toute occurrence $\alpha \in$ dom(E) telle que E(α) = r possède l'attribut : (TAB, résultat, libre), "TAB" (type TABLEAU), "résultat", et "libre" sont appelés les caractéristiques du symbole r. On écrira :

att-r = (TAB, résultat, libre).

Un attribut est une liste de "caractéristiques".

On définit une caractéristique d'un arbre bien formé comme une propriété particulière pouvant être attachée aux feuilles de cet arbre.

Les propriétés choisies sont :

Soit E un arbre et $\alpha \in$ dom(E) une feuille de E,

1. Une feuilles est une occurrence "libre" dans E,
2. Une feuille est une occurrence "liée" dans E,
3. Une feuille est une occurrence "résultat" dans E,
4. Une feuille est une occurrence "constante" dans E,
5. Une feuille est de type T'.

Une occurrence libre α est telle que :

$\alpha \in$ dom(E) feuille (α,E) et $\forall$ $\alpha' \in$ dom(E) E(α') = E(α) $\Rightarrow$ E(p(α')) $\neq$ "$\forall$".

("$\forall$" est le seul quantificateur accepté dans un énoncé).

Une occurrence non libre est liée.

Une occurrence "résultat" α est telle que :

feuille (α,E) et E(p(α)) = "tq"

(par construction de l'arbre E on ne peut pas avoir :

(E(p(α)) = "tq" et E(α') = E(α) et E(p(α')) = "$\forall$")

Une occurrence "constante" α est telle que :

E(α) est soit un symbole défini comme une constante soit une constante proprement dite d'un type.

Un arbre attribué est un arbre bien formé dont, en particulier, les feuilles possèdent certaines propriétés.

Reprenons l'exemple précédent et donnons les attributs des feuilles d, k et 0 :

- att-d = (TAB, libre),
- att-k = (ENTIER, liée),
- att-0 = (ENTIER, constante).

Il faut souligner qu'à toute feuille d'un arbre est attaché son type.

La règle de remplacement énoncée plus bas va donc s'appliquer sur l'arbre E d'un énoncé E tel que E soit :

- bien formé

et - attribué.

Remarque : Dans la transformation les propriétés pouvant être attachées aux différents symboles d'opérations sont ignorées. Par exemple on ignore que l'opérateur "et" est commutatif, associatif, ou les deux.

II.4.2.2. Sous-arbre et filtre

Soit E un arbre.

Le sous-arbre de E à l'occurrence α est l'arbre E' tel que :

$$\forall \beta \in \mathbb{N}_+^* \quad E'(\beta) = E(\alpha * \beta).$$

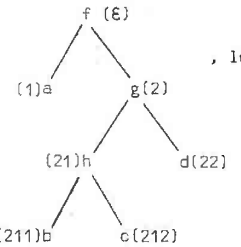
On le note E/α.

(Si $\alpha \notin dom(E)$ alors $E/\alpha = \mathcal{E}vide$. On note par $\mathcal{E}vide$ l'arbre vide, c'est-à-dire $dom(\mathcal{E}vide) = \phi$. $\mathcal{E}vide$ est un symbole particulier tel que $\mathcal{E}vide \notin V$).

Exemple de sous-arbres :

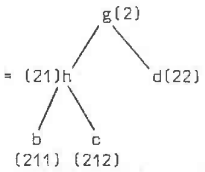
soit

E =



, le sous-arbre E/2 est donné par :

E/2 =



On note par $E\,[\alpha \leftarrow E_1] = E_2$ tel que $\alpha \in dom(E)$ pour dire que l'on substitue l'arbre E_1 à E/α :

$$E_2/\alpha = E_1$$

et $\quad dom(E_2) = \{\beta \mid \beta \in dom(E) \text{ et } \beta \neq \alpha * \omega\} \cup \{\alpha * \omega' \mid \omega' \in dom(E_1)\}$

$$E_2(\beta) = E(\beta) \quad \text{et} \quad E_2(\alpha * \omega') = E_1(\omega')$$

(On suppose que $\varepsilon \in dom(E_1)$).

Le couple $\alpha \leftarrow E_1$ est appelé une substitution de E.

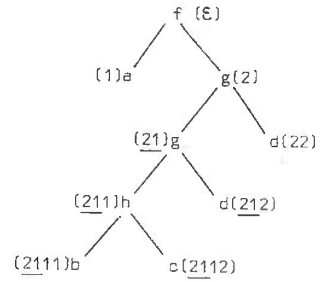
Exemple de substitution

Reprenons l'exemple précédent et posons

$E_1 = E/2$ tel que $dom(E_1) = \{\varepsilon, 1, 2, 11, 12\}$

de la substitution $21 \leftarrow E_1$ dans E : $E\,[21 \leftarrow E_1]$ on obtient l'arbre

E_2 =



Un filtre est un ensemble de substitutions $\alpha_i \leftarrow A_i$ tel que $\alpha_i \in \mathbb{N}_+^*$ et A_i soit un arbre.

On dit qu'un arbre S filtre un arbre E s'il existe un ensemble de substitutions σ tel que :

$$E = S\,[\alpha_1 \leftarrow A_1, \ldots, \alpha_n \leftarrow A_n]$$

pour $\sigma = \{\alpha_1 \leftarrow A_1, \ldots, \alpha_n \leftarrow A_n\}$

signifiant que l'arbre E est égal à l'arbre obtenu en substituant dans S

A_1 à S/α_1, ..., A_n à S/α_n

que l'on notera aussi par :

$E = S\,[\sigma]$ ou encore $E = \sigma S$.

Nous retiendrons cette dernière notation.

On dit aussi que E est une instance de S.

Et de façon générale :

Soit γ un ensemble de substitutions et E' un arbre alors l'arbre $\gamma E'$ est l'arbre obtenu en effectuant les remplacements donnés par les substitutions $\alpha_i \leftarrow A_i \in \gamma$.

II.4.3. La technique de remplacements

Cette technique est définie par le mécanisme de transformation lié à l'application de la règle générale de remplacement. Cette règle est essentiellement caractérisée par les filtres σ' et σ introduits au § II.4.1.

Expliquons tout d'abord à travers un exemple le mécanisme général de la transformation permettant ainsi de donner les définitions de σ' et σ pour ensuite donner celle de la règle générale de remplacement.

II.4.3.1. Le mécanisme de la transformation : un exemple

Le mécanisme de la transformation se base sur les notions de substitution et de filtre définies précédemment.
Prenons l'exemple donné au § II.4.1. et étudions alors les questions qui y sont soulevées.
Appelons E l'énoncé effectif :

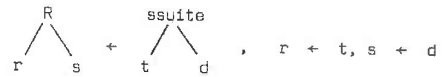
E : t tq ssuite(t,d) et $\forall$ k t[k] > 0

(l'arbre de E a déjà été donné au § II.4.2.1).

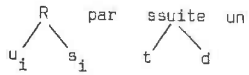
Soit r une règle de la forme S::B.
Trouver E', le transformé de E, revient à définir une instantiation de B.
Instantier B signifie :

1. Remplacer certaines feuilles de B (les variables de premier ordre) par les variables de premier ordre de l'énoncé effectif E,
2. Remplacer certains sous arbres de B par des sous arbres de E. (un sous arbre de B, à l'occurrence α, à instantier est caractérisé par le fait que α adresse une étiquette particulière. Cette étiquette est symbolisée par les lettres "R" et "P").

Le calcul d'une instance de B est guidé par l'identification entre S et E. Les sous-arbres et les variables de premier ordre de B (les feuilles de B) à remplacer apparaissent alors dans S. Analysons d'abord le remplacement d'un sous-arbre de B par un autre effectif. Prenons l'identification σ (§ II.4.1) et considérons les remplacements suivants :

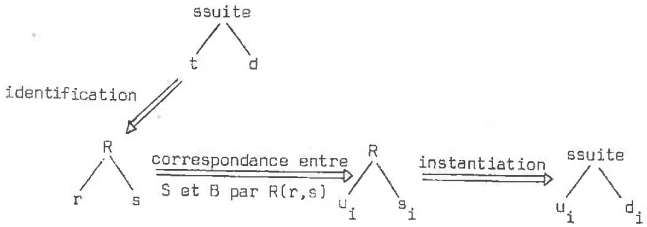


Cette identification signifie que l'on veut instantier un sous-arbre de B (un constituant particulier de B),



constituant de E.

On a alors le schéma suivant :



Un tel schéma est appelé schéma de transformation pour un constituant de B à savoir "$R(u_i, s_i)$".
Ce schéma montre que les remplacements de variables de premier ordre, r ← t et s ← d, jouent deux rôles :

1. le premier consiste à appliquer les remplacements dans

correspondant à

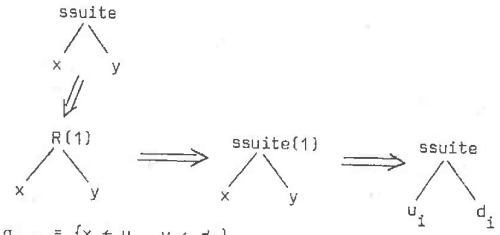
,

   s dans ce cas est considéré comme formelle.

2. Le second consiste à préciser la place de u_i et s_i dans la structure effective de "R", c'est-à-dire dans ssuite où t et d sont considérées comme formelles.

Afin de faire apparaître les 2 rôles de tout remplacement de premier ordre figurant dans σ, des variables intermédiaires sont introduites. Et toute correspondance entre S et B d'un schéma de transformation est donnée par des substitutions particulières précisant le remplacement de tout paramètre formel (les intermédiaires) par un paramètre effectif.

Pour être plus précis le schéma de transformation précédent devient :



avec $\sigma_{(1)} = \{x \leftarrow u_i,\ y \leftarrow d_i\}$

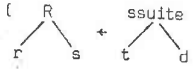
Le premier rôle des substitutions $r \leftarrow t$ et $s \leftarrow d$ permet de faire les compositions σu_i et σs_i

Le second détermine σ_1.

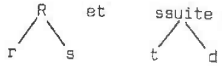
Dans cet exemple il n'existe qu'un seul "remplacement de structure".

Par conséquent il n'existe qu'un seul calcul de filtre du genre $\sigma_{(1)}$.

Cette substitution



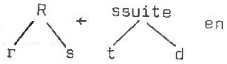
) est modifiée en renommant les variables de



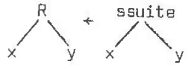
et ssuite permettant ainsi d'introduire les variables intermédiaires x et y.

On supposera alors que cette modification est réalisée par une fonction de calcul d'intermédiaires s'appliquant à tout remplacement de structure figurant dans σ.

Ainsi l'application de cette fonction sur



en considérant $\{r \leftarrow t,\ s \leftarrow d\}$ produit comme résultat la substitution



.

Définissons à présent σ et σ' :

1. en mettant en évidence le calcul de la fonction d'introduction des intermédiaires,
2. en précisant la définition des filtres du genre $\sigma_{(1)}$.

Il est à souligner que ce mécanisme de transformation est situé dans un certain contexte précisé au § II.4.4.

II.4.3.2. Définitions des filtres σ et σ'.

Soient E l'énoncé à transformer (un arbre) et r : S::B la règle à appliquer sur E.

S'il existe un filtre σ et une occurrence α tels que $E/\alpha = \sigma S$ alors la règle est applicable.

On appelle σ un filtre d'identification de E pour r.

On distingue dans σ les 2 sortes de remplacement suivants :

1. Un remplacement de variables de premier ordre, (premier ordre) c'est-à-dire toute substitution $[\alpha \leftarrow \beta] \in \sigma$ telle que : feuille (α, S) et $\beta \in dom(E)$.
2. Un remplacement de variables fonctions, on dira aussi de structure, c'est-à-dire toute substitution $[\alpha \leftarrow \beta] \in \sigma$ telle que : non feuille (α, S) et $\beta \in dom(E)$ (que l'on note aussi par : $\triangle \leftarrow \triangle$)

Afin d'appliquer σ à B et de calculer σ', des variables intermédiaires sont introduites de la manière suivante :

On applique la fonction de calcul d'intermédiaires à tout remplacement de structure : $S/\sigma \leftarrow E/\alpha' \in \sigma$, comme indiqué au § II.4.3.1. précédent.

Cette fonction, appelons la "inter", procède au calcul suivant :

inter(σ) est défini par ($\underline{\Delta}$) :

inter(σ) $\underline{\Delta}$ σ_f == {$S/\alpha \leftarrow E/\alpha' \in \sigma$ des remplacements de structures (pas de premier ordre)

<u>tant que</u> <u>non</u> $\sigma_f = \phi$ <u>faire</u>

σ_v == l'ensemble des substitutions de premier ordre de σ

choisir un remplacement $S/\alpha \leftarrow E/\alpha' \in \sigma_f$

<u>tant que</u> <u>non</u> $\sigma_v = \phi$ <u>faire</u>

choisir un remplacement $\omega \leftarrow \omega' \in \sigma_v$

<u>Si</u> $\omega \in$ dom(S/α) <u>et</u> resp. $\omega' \in$ dom(E/α')

<u>alors</u> renommage par une même étiquette des étiquettes

$S/\alpha(\omega)$ et $E/\alpha'(\omega')$

Supprimer $\omega \leftarrow \omega'$ de σ_v

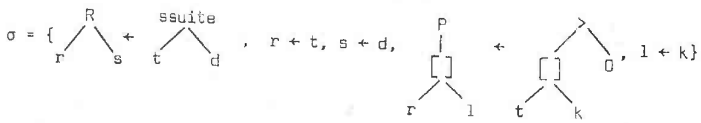
<u>ftq</u>

supprimer $S/\alpha \leftarrow E/\alpha'$ de σ_f

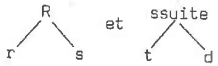
<u>ftq</u>

résultat : σ modifié par les changements d'étiquettes

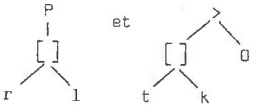
Par exemple de l'identification : (§ II.4.1)



On déduit que :

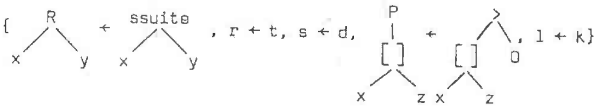
- Dans 

r et t sont changées en x puisqu'il existe $r \leftarrow t$,
et s et d sont changées en y puisqu'il existe $t \leftarrow d$,

- Dans 

r et t sont changées en x puisqu'il existe $r \leftarrow t$
(on préfère garder la même étiquette x donnée dans le cas précédent, mais on peut tout aussi bien dans le cas présent changer r et t en une autre étiquette. Cela se justifie par le calcul de σ'),
et l et k en z puisqu'il existe $l \leftarrow k$.

On a alors :



Le nouveau σ ainsi obtenu s'appellera également σ.

<u>Définition de σ'</u>

Le filtre σ' a pour but alors d'instantier l'arbre B. (B est le membre droit de la règle).

Cette instantiation se fait en appliquant successivement un certain nombre de filtres :

$$\sigma' = \sigma_{\beta i} \circ \alpha_{\beta i-1} \circ \ldots \circ \alpha_{\beta 1}$$

Un filtre $\alpha_{\beta n}$ est caractérisé par :

- un remplacement de structure appartenant à σ,
- σB et B.

Il permet de remplacer les variables intermédiaires figurant dans un sous arbre de σB par les variables effectives correspondantes figurant dans B. Ce sous arbre est défini grâce à un remplacement de structure.

Les filtres $\alpha_{\beta n}$ se définissent ainsi :

On pose $B' = \sigma B$.

Pour tout $\beta \in dom(B)$ tel que $B(\beta)$ soit une variable fonction et pour $\alpha \leftarrow E/\alpha' \in \sigma$ un remplacment de structure tel que l'étiquette $S(\alpha)$ soit égale à l'étiquette $B(\beta)$ (on considère le même symbole de fonction),

$$\sigma_\beta = \{\gamma \leftarrow \sigma\,(B/\delta) \ \underline{\text{tel que}}\ \gamma \in p^{-1}(\beta, B')$$

$$\underline{\text{et}}\ \ \delta \in p^{-1}(\beta, B)$$

$$\underline{\text{et}}\ \ \forall\, r, \delta\ \exists\, \omega \in dom(S/\alpha)\ \underline{\text{tel que}}$$

$$\text{param}\ (B'/\gamma,\ B/\delta,\ S/\omega)\}$$

* les fils de β dans B' = ssuite *
  - $\underline{x}$
  - $\underline{y}$

* les fils de β dans B = R *
  - $\underline{u_i}$
  - $\underline{s_i}$

* $x \leftarrow u_i$, $y \leftarrow s_i$ *

- le prédicat param indique que :
  1. l'étiquette $B'(\gamma)$ est égale à l'étiquette $S(\omega)$
  2. l'étiquette $B(\gamma')$ est le paramètre effectif de l'étiquette $S(\omega)$ en considérant les sous-arbres B/β et S/α.

(R (u_i, s_i) , R (x, y))

Maintenant donnons la définition de la règle générale de remplacement.

II.4.3.3. La règle générale de remplacement

Cette règle s'énonce de la manière suivante :

Soit E et E' les arbres des énoncés Spes E et E'.

E' est le transformé de E si et seulement si

il existe une règle S:: B, il existe $\alpha \in dom(E)$,

il existe un filtre σ tels que

$$E/\alpha = \sigma S \quad \text{et} \quad E' = E\,[\alpha \leftarrow \sigma'(\sigma B)]$$

et $\sigma' = \sigma_{\beta i} \circ \sigma_{\beta i-1} \cdots \circ \sigma_{\beta 1}$

L'exemple donné au § II.4.1. illustre une application de cette règle :

$$E' = E\,[\alpha \leftarrow \sigma'(\sigma B)]$$

B est le membre droit de la règle r_j,

$\sigma' = \sigma_{(2)} \circ \sigma_{(1)}$ c'est-à-dire

$$E' = E\,[\alpha \leftarrow \sigma_{(2)}\,(\sigma_{(1)}\,(\sigma B)]\ ,$$

E est l'arbre donné par l'exemple du § II.4.2.1.

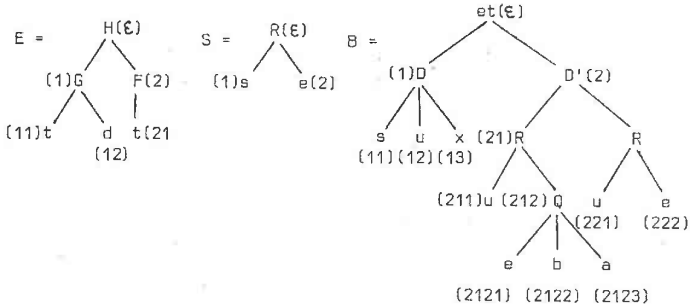
II.4.3.4. Un exemple de calcul des filtres σ et σ'

On se donne E et S::B une règle applicable sur E.

Cet exemple montre les quatre stades successifs d'une transformation, c'est-à-dire la manière dont la règle de remplacement s'applique sur E :

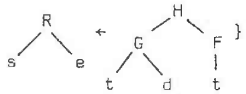
- stade 1 : identification,
- stade 2 : application de σ sur B,
- stade 3 : définition de $\sigma_{\beta n}$,
- stade 4 : instantiation de σB.

Soit :



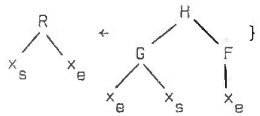
- stade 1 : identification de E à S et introduction de variables intermédiaires

a) $\sigma = \{s \leftarrow d,\ e \leftarrow t,$



$\}$

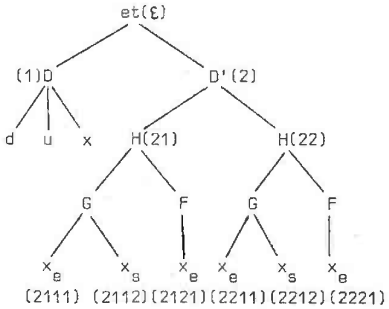
b) $\sigma = \{s \leftarrow d,\ e \leftarrow t,$



$\}$

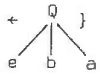
- stade 2 : application de σ sur B.

σB :



- stade 3 : définition des substitutions à appliquer pour instantier σB

. pour $R_{(21)}$ de B : $\sigma_{(21)} = \{x_{s(2112)} \leftarrow u,\ x_{e(2111,2121)} \leftarrow$



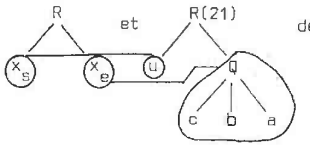
$\}$

. $\sigma_{(21)}$ composé avec σ :

$\sigma_{(21)} = \{x_{s(2112)} \leftarrow u,\ x_{e(2111,2121)} \leftarrow$

$\}$

$\sigma_{(21)}$ est calculé à partir de la correspondance entre :



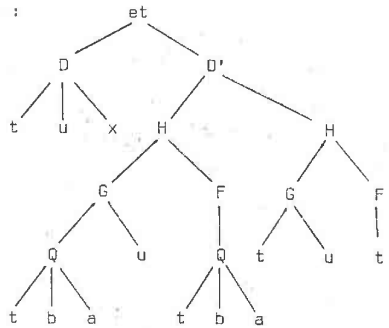
de B.

. pour $R_{(22)}$ de B, en le composant tout de suite avec σ :

$\sigma_{(22)} = \{x_{s(2212)} \leftarrow u,\ x_{e(2211,2221)} \leftarrow t\}$

- stade 4 : définition de l'instantiation de B, c'est-à-dire le résultat de la transformation :

$\sigma_{(22)}\ (\sigma_{(21)}\ (\sigma B))$:



Remarque :

Avant de remplacer E par l'arbre construit, il est procédé au renommage des variables effectives de B (variables figurant dans B mais pas dans S). On suppose que l'on dispose d'une fonction de renommage (les variables u,b, et a sont renommées).

II.4.4. Conditions d'application de transformations

La transformation d'un énoncé E par l'application d'une règle r, S::B, s'effectue dans un certain contexte et sous certaines conditions.

Le contexte est des hypothèses faites sur r et E.

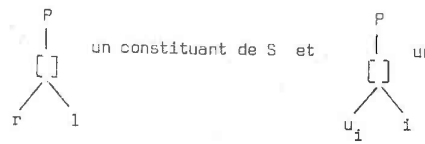
Ces hypothèses sont :

- une hypothèse sur E,
  l'hypothèse H1 : E est une conjonction de définitions Spes,
- des hypothèses sur r :
  . H2 : toute forme $R(\bar{x})$ de S et B est elle que la liste d'argument $\bar{x}$ ne comporte aucune variable fonction.

Par exemple on n'a pas de règles de la forme : R(R'(x),y) :: B, (ordre > 2) ou encore : R(x,y) :: R(R'(u),i)..., (profondeur du 2ème ordre limitée)

. H3 : Tout symbole de fonction F d'un type abstrait figurant dans S comme fils d'un symbole de variable de fonction, R, apparaît nécessairement dans B comme fils de ce même symbole R.

Par exemple, en reprenant encore l'exemple de la règle r_j donnée au § II.4.1. on a bien :



un constituant de S et

un constituant de B,

la fonction accès "[ ]" du type TABLEAU apparaît aussi bien dans S que dans B et comme fils du même symbole de variables de fonction "P".

. H4 : tout symbole de variable de fonction figurant dans S figure nécessairement au moins une fois dans B.

On notera, à propos de H4 que l'on peut avoir des règles ayant dans leur membre droit (B) des variables fonctions n'apparaissant pas dans leur membre gauche (S).

Par exemple l'utilisateur peut définir la règle suivante :

r : $R(x,y) :: R(u_i,y_i)$ et $R1(u_i)$ et $x == F(u_i,y_i)$

L'application d'une telle règle conduira l'utilisateur à proposer une définition pour "R1" n'apparaissant pas dans le membre gauche de r. Et en conséquence il sera nécessaire de procéder à la vérification d'une certaine post-condition. Nous reviendrons sur ce dernier point au § II.5.

Sous ces hypothèses la transformation suppose que les conditions suivantes soient satisfaites :

cond1 : E et S sont tels que le filtre d'identification σ est composé seulement de substitutions de premier ordre et de substitutions de sous arbres qui ne soient pas des feuilles.

Par exemple :

dans σ n'apparaîtra pas une substitution comme : $x \leftarrow F(a,b)$ ou encore de la forme $G(x,y) \leftarrow z$,

cond2 : Pour toute substitution de structure figurant dans σ le nombre de fils de la variable fonction de S est inférieur ou égal à celui de la fonction définie de E.

Par exemple pour R(s,e) ← ssuite(t,d) le nombre de fils de "R" est inférieur ou égal à celui de "ssuite",

cond3 : Pour toute substitution de premier ordre, $x \leftarrow y \in \sigma$ on a :

att-x = att-y signifiant que x et y ont même attribut.

Ces hypothèses et ces conditions ont été prises afin de faciliter au système la mise en oeuvre du mécanisme de transformation.

Cependant nous prévoyons de ne pas limiter la profondeur d'une forme d'une règle, c'est-à-dire permettre des règles du second ordre comme par exemple : $S :: R(R1(R2(\bar{x}),\bar{z}), R3(R4(\bar{y})))$

où R, R1, R2, R3, R4 sont des variables fonctions.

II.5. Des règles aux stratégies

II.5.1. Une introduction aux stratégies d'explicitations

Intuitivement une stratégie exprime une manière de transformer un énoncé donné ou encore une façon de progresser dans l'explicitation. Ces stratégies ont deux caractéristiques essentielles.

1. elles peuvent être dépendantes les unes des autres, par exemple l'application d'une stratégie s peut nécessiter l'application préalable de certaines stratégies, on parlera alors de pré-requis associés à s,

2. d'une manière générale elles sont indéterministes. Cette seconde caractéristique soulève le problème du choix de la stratégie au cours d'une explicitation. Ce problème est abordé dans le paragraphe suivant.

En outre si l'application d'une stratégie s à un énoncé infère une seule transformation, c'est-à-dire une seule application de la règle générale de remplacement, alors on dira que s est une stratégie d'explicitation locale, nommée SEL, mais si elle infère une suite de transformations alors on dira que s est une stratégie d'explicitation globale, notée SEG.

Le chapitre III est consacré à cette notion en insistant particulièrement sur les SEG et en posant le problème de la créativité des stratégies.
Le présent chapitre s'attache alors surtout à la définition de SEL.

II.5.2. Les stratégies d'explicitations locales

La nécessité de particulariser les règles de formes générales :

S::E,D ,

a été précisée au § II.3. et au chapitre I.

Cela est rendu possible en particularisant S en un énoncé S'. A partir de S', on définit E' et D' d'une manière telle que :

E' et D' ⇒ S' soit un théorème dans SF (cf. § II.2.)

La preuve de la validité de rdc' peut conduire à vérifier certaines conditions sur S' et (E' et D'). De telles conditions nécessaires à la preuve de rdc' sont alors précisées dans toutes règles S'::E', D' en termes de pré-post conditions sur les règles particulières. On appelle de telles règles des SEL.

La forme donnée à une SEL est (">—<" notation schématique de "::")

s : pre(S) >—— S / E,D ——< post(S,E,D)

Une SEL est aussi notée par : {pre} S :: B{post}.
Les prédicats pré et post sont les pre-post conditions à vérifier.
Une règle de la forme de s se lit :

si (pre(S) et post(S,E,D) alors (E et D ⇒ S est valide)

Une pré-condition a pour rôle de mieux préciser des particularités de S. C'est une propriété exprimée en Spes ou Medee portant essentiellement sur les objets de S. On désire que les pré-conditions soient facilement vérifiables pour permettre une vérification automatique de ces propriétés.

Une post-condition a pour rôle essentiel d'établir la propriété à vérifier pour toutes définitions de variables fonctions apparaissant dans B mais pas dans S.
Ces définitions sont proposées par le programmeur.
Une post-condition est un énoncé implicite à prouver (cf. stratégie S7 du § II.5.2.1.).

L'absence de pré-post conditions à vérifier signifie que pre(S) et post(S,E,D) ont la valeur vraie.

Dans sa version actuelle SIE ne sait pas vérifier les pré-post.

Remarque : On emploie indifféremment dans la suite "règles de constructions" (règles), "stratégies d'explicitations" (stratégies). Mais il faut bien savoir qu'une SEL est une instance particulière d'une des règles de construction : ITERC, SIMPC, CONDC, PREMC.

II.5.3. Exemples de stratégies locales

Quelques SEL relatives à la classe des programmes itératifs manipulant des tableaux à 1 dimension sont présentées ci-dessous.

S1 .

t tq R(t,d)
t:TAB d:TAB

---

t==der u_i pour i de 1 jqa a_i
u_i tq R(u_i, tr(d,1,i))
u : SUITE de TAB i : ENT d : TAB
a_i == (i = bs(d))

Commentaire :
- cette stratégie ne possède ni de post ni de pré conditions.
- S1 propose de calculer le résultat par approximations successives par induction sur la donnée d.

S2 .

r tq R(r,s) et ∀l P(r[l])
r : TAB s : TAB l : 1..bs(r)

Symbol (P) et P ∉ sign (TAB)

---

r == der u_i pour i de 1 jqa a_i
u_i tq R(u_i, s_{1i}) et P(u_i[i])
a_i == (i = bs(s))

Commentaire :
- S2, comme S1, propose de calculer r par approximations successives par induction sur s et en plus elle procède à l'élimination du "∀".
- La pré-condition de S2 précise une particularité de la forme du résultat par :
  . symbol (P) indiquant que P symbolise uniquement un symbole de prédicat et non une structure plus complexe, et que le prédicat est tel qu'il n'appartient pas à la signature du tableau (P ∉ sign(TAB)).

S3 .

r tq R(r,k)
r : TAB k : ENT

---

r==der u_i pour i de 1 jqa a_i
u_i tq R(u_i,i)
a_i == (i = k)

Commentaire :
- S3 propose de calculer r, comme S1, mais l'affaiblissement de l'énoncé R(r,k) est caractérisé ici par une décomposition : r(u_i,i) un invariant et i = k la condition d'arrêt.

S4 .

v_i tq R(v_i, $q1_i$) et P(v_i[i])
v : SUITE de TAB q : TAB i : ENT

pré-requis(S2)

---

v_i == Concat(v_{i-1},m_i)
m_i tq R(v_{i-1}, q_{1i-1}) et P(v_{i-1}[i-1]) ⇒
R(concat(v_{i-1},m_i), concat($q1_{i-1}$,qi_i)) et P(m_i[i])
v_0 tq R(v_0, q_{10}) et P(v_0[0])

Commentaire :

- S4 permet de définir une récurrence sur v_i :
la précondition signifie que cette relation de récurrence permet alors de définir u_i (de S2), on peut écrire : u_i == concat(u_{i-1}, m_i) .
S4 est en quelque sorte le résultat d'une analyse récurrente qui consiste à chercher la manière de passer de u_{i-1} à u_i.

S5 .

$$\frac{t == tr(d,i,j)}{t == tvide} \quad i > j$$

Commentaire :

- Ceci est un exemple d'une règle de construction propre à cette classe de problèmes qui n'est pas considérée comme une stratégie.
De façon générale toutes les v-règles, t-règles, l-règles et p-règles sont appelées règles de construction et les nt-règles stratégies d'explicitations locales.
Un autre exemple de règle de construction qui n'est pas vu comme une stratégie est donné par S6.

S6 .

$$\frac{ssuite(concat(d1_i,v),\ concat(d1_i,di_i))}{v == di_i} \quad v : TAB \quad d : TAB$$

S7 .

$$\frac{t = \underline{der}\ u_i\ \underline{pour}\ i\ \underline{de}\ 1\ \underline{jqa}\ a_i}{t\ \underline{tq}\ R(t,d)} \quad \begin{array}{l} t : TAB \quad d : TAB \\ R(V_i,V_i)\ \underline{et}\ R1(V_i,d)\ \underline{et}\ a_i \Rightarrow R(u_i,d) \end{array}$$

u_i $\underline{tq}$ $R(u_i, V_i)$ V : SUITE de TAB
V_i $\underline{tq}$ $R1(V_i, d)$ d : TAB
a_i == i = bs(d) a : SUITE de BOOL

Commentaire :

- S7 propose de calculer le résultat t par induction sur suite intermédiaire et non par induction sur la donnée d comme le propose S1.
- R1 et R2 sont alors inventés par le programmeur. Ainsi la post-condition établit la propriété P à valider :
Si P alors (E et D => S) (pré est vraie)
(S est le membre supérieur de S7, D la définition itérative et E les définitions implicites de R, R1 et R2).

Faisons dès à présent, la distinction entre SEG et SEL.

II.5.4. Les stratégies d'explicitations globales

Les stratégies d'explicitations globales (discutées au chapitre III) décrivent une manière d'enchaîner les SEL.

Ainsi pour exprimer globalement qu'une suite de transformations est applicable à un énoncé, on cherche à décrire des schémas de stratégies.
Ces schémas sont appelés SEG.
La notion de schémas de stratégies est comparable à celle des règles dérivées en logique.
Par conséquent une SEG peut être conçue à partir d'une explicitation, mais aussi, comme pour les SEL, elle peut être définie intuitivement.

Une SEG est alors un algorithme particulier de résolution fondé sur des stratégies. Cet algorithme décrit donc un enchaînement particulier de stratégies (enchaînement de rdc, SEL, et SEG).

II.6. Le problème du choix dans l'explicitation

L'explicitation peut être vue comme une suite de calculs et de décisions. On dit que le système "calcule" pour signifier que l'intelligence du programmeur n'intervient pas dans le traitement d'une action par SIE. Par exemple si l'action est la recherche d'un filtre d'identification il y a un calcul du filtre puisque le programmeur n'intervient pas dans cette action.

Mais il appartient au programmeur de décider des calculs à faire. Néanmoins le système peut l'aider dans sa décision. Ce dernier point est abordé juste après une discussion sur les décisions.

* Discussion sur les décisions

Une des caractéristiques des stratégies d'explicitations est leur indéterminisme. Ainsi la décision fondamentale que prend le programmeur est le choix de la stratégie à appliquer et ce de manière interactive au cours de l'explicitation. Ce choix signifie une manière pour le programmeur de progresser vers la construction d'une solution algorithmique.

Les différents types de décisions qui constituent une explicitation sont successivement :

1. Décision de la définition implicite à expliciter,
2. Décision du choix de la stratégie,
3. Décision du filtre d'identification,
4. Décision d'introduction d'intermédiaires.

Ces décisions sont en rapport avec la règle générale de remplacement puisqu'elle est à la base de l'explicitation. En la prenant sous la forme :

$$\exists s : S :: B \quad \exists \alpha \in dom(E) \quad \exists \sigma \ \underline{\text{tel que}} \ E' = E\,[\alpha \leftarrow \sigma'(\sigma B)]$$

Montrons comment interviennent ces décisions.

Il ressort de cette règle trois sortes de recherches :

1. la recherche d'une stratégie s,
2. la recherche d'un identificateur ,
3. la recherche d'une définition implicite à expliciter (E/α).

Si $\mathcal{S}$ est l'ensemble des stratégies disponibles, l'application d'une recherche d'une stratégie s va conduire à $\{s \in \mathcal{S} \quad E/\alpha = \sigma_s S\}$.
Cet ensemble peut être soit vide, soit composé d'une seule stratégie, soit de plusieurs. Les décisions du programmeur sont alors respectivement :

- cas vide : décision d'introduire une nouvelle stratégie (dans la bibliothèque) et éventuellement de l'appliquer (modification ou ajout d'une stratégie),

- le deuxième cas exprime une unique possibilité, décision de l'accepter ou de la rejeter (le rejet nous ramène au cas précédent),

- le dernier cas correspond à la décision de choisir une stratégie dans un ensemble (le rejet de cet ensemble nous ramène au premier cas).

Comme les stratégie locales et les stratégies globales sont de formes différentes on peut convenir que le programmeur choisi en priorité les SEG et qu'au cours de la mise en oeuvre de l'une d'entre elles il est amené à choisir parmi les SEL et les règles de constructions.

Le choix entre les stratégies globales est décidé par le programmeur sans qu'aucun calcul associé à ce problème de choix ne soit effectué par le système.
L'application de la recherche d'un filtre σ va conduire pour chaque variable de premier ordre x de S à $\{x \leftarrow y$ tel que pour tout y de E/α l'on ait att-x = att-y.
Cet ensemble, appelons le X_x, indique toutes les substitutions $(x \leftarrow y)$ possibles d'une même variable x de S. C'est au programmeur de choisir

pour chaque ensemble X_x un remplacement de x, tout en évitant des conflits de choix, c'est-à-dire en choisissant $x \leftarrow y \in X_x$ et $x' \leftarrow y' \in X_x'$, alors nécessairement $y \neq y'$.

Pour les variables fonctions, le système ne calcule pas toutes les substitutions possibles d'une même variable fonction de S. Il propose une et une seule substitution possible et ce pour chaque variable fonction de S. Et c'est au programmeur d'accepter cette proposition ou d'en formuler une autre.

L'application de la recherche d'une définition implicite à transformer (E/α : un énoncé SPES E est un ensemble de définitions) pose le problème du choix de l'identificateur résultat à expliciter. Ce choix est laissé totalement au programmeur. Aucun calcul n'est proposé par le système pour faciliter son choix.
On a noté dans le § II.4.4., à propos de l'hypothèse H4, l'existence possible de variables fonctions dans B n'apparaissent pas dans S. Lors de l'instantiation de B ces variables devront être définies. C'est le problème du choix des intermédiaires. Il est entièrement résolu par le programmeur.
Il est vu plus loin (chapitre V), la manière dont l'interaction entre le système et le programmeur prend en compte ces décisions.

La constatation que l'on peut faire est que plus les calculs du système porteront sur les décisions citées plus haut et plus l'explicitation deviendra automatique. Mais comme il ne sait pas encore calculer ces décisions il est doté d'outils d'aide à la décision.

* L'aide à la décision

La première aide apparente est que le système propose au programmeur un sous-ensemble restreint de stratégies. Cela est rendu possible du fait que S (de la règle S::B) est un énoncé particulier, ce qui limite le nombre possible des instances de S.

On veut dire que :
soit E un énoncé à transformer,
la probabilité pour que E soit une instance possible d'un certain S n'est pas forte du fait de la spécificité de ce S.

L'outil principal que le système offre au programmeur pour l'assister dans le processus de l'explicitation est un arbre des choix.
Cet arbre comporte toutes les informations concernant une transformation. (en particulier l'instantiation de B). Le programmeur dispose de fonctions de déplacements dans l'arbre. Cet arbre va donc lui permettre de remettre en cause des transformations, ou des choix et de revenir sur des choix antérieurs. Et donc de reprendre l'explicitation d'un énoncé déjà explicité.

Un second outil important est le graphisme.
Le système utilise pour le moment le graphisme uniquement pour la visualisation synthétique de l'arbre des choix et des énoncés.
L'intérêt du graphisme consiste surtout à aider le programmeur à construire un programme par la visualisation de certains dessins : dans le cas, par exemple, de la construction de programmes manipulant des tableaux à 1 dimension, il serait intéressant d'instaurer un dialogue entre le système et l'utilisateur dont le but est de permettre la visualisation de tableaux et d'intervalles tels que le programmeur les imagine dans son esprit.

Ainsi la possibilité de dessiner des tableaux à l'image du programmeur peut lui apporter une aide appréciable.

II.7. Un exemple technique de transformation

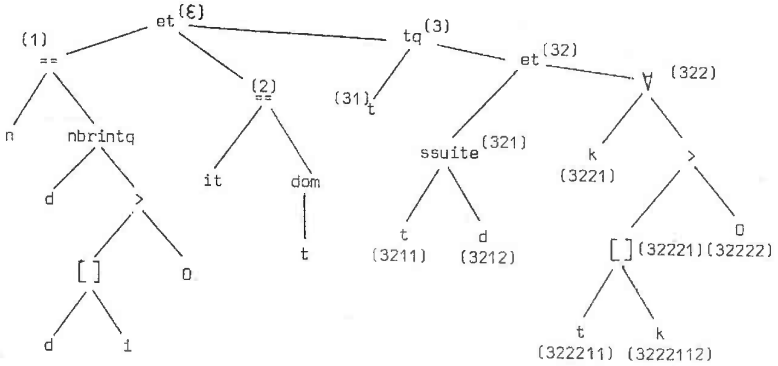
Cet exemple met en oeuvre la technique de remplacement telle qu'elle a été présentée au § II.4. Reprenons encore l'exemple de la figure II.2. et mettons en évidence les calculs de σ et σ'.
Soit E l'énoncé de départ donné par la colonne de droite de la figure II.3 ci-dessous.

Lexique	Profil	Enoncé formel
t est un tableau d'entiers de n éléments positifs de d d est le tableau donné ssuite est un prédicat indiquant que t est une sous suite de d [ ] donne la valeur en un point du tableau	t : TAB d : TAB ssuite : TAB x TAB → BOOL k : [1..n] n : ENT [ ] : TAB x ENT → VAL it : [1..n] dom : TAB → INT	résultat t t tq ssuite(t,d) et ∀k t[k] > 0 n == nbrintq(d,d[i] > 0) it == dom(t) donnée : d

figure II.3.

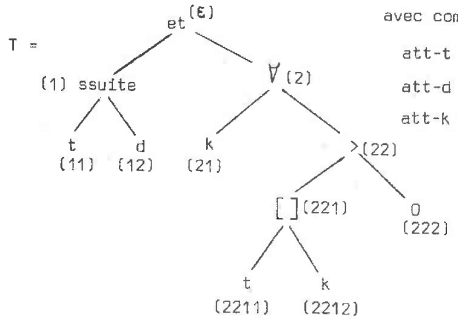
(it : [1..n] et it == dom(t) signifient que dom(t) = [1..n], c'est-à-dire que le résultat de la fonction dom(t) (it == dom(t)) ne peut être que l'intervalle [1..n] car sinon il se produirait une erreur de type).

E est représenté par l'arbre E suivant :



L'identificateur à expliciter est t. Il est défini par l'arbre T = E/32.

T =

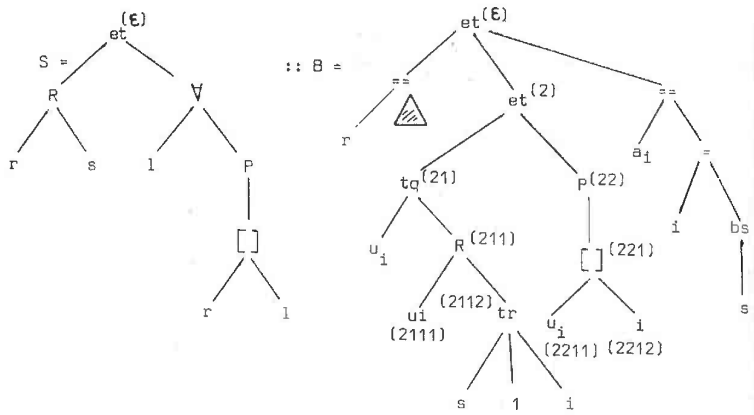


avec comme attribut de variables :

att-t = (TAB,libre, résultat),
att-d = (TAB,libre),
att-k = (ENT,liée).

Soit la stratégie s de la forme S::B (les pré-post sont vraies) choisie par le programmeur :

s :



s : R(r,s) <u>et</u> ∀l P(r[l]) :: r == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i
u_i <u>tq</u> R(u_i, tr(s,1,i)) <u>et</u> P(u_i[i])
a_i == (i = bs(s))

Les attributs des différentes variables de S sont :

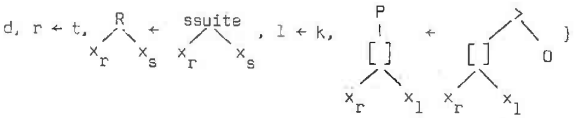
att-r = (TAB,libre, résultat),

att-s = (TAB,libres),

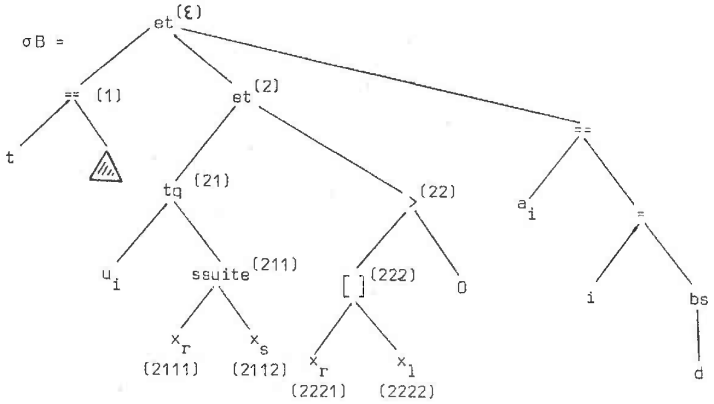
att-l = (ENT,libre).

Appliquons s à T :

- <u>Stade 1.</u> Identification de T à S.

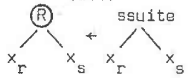


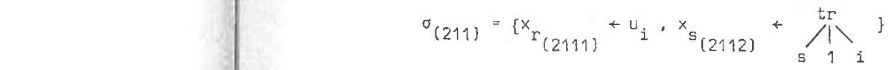
- <u>stade 2.</u> Application de σ à B.



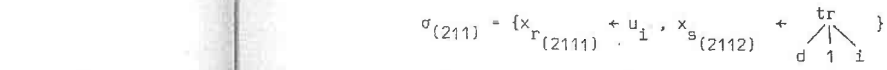
- <u>stade 3.</u> Définition des substitutions à appliquer pour instancier σB.

Pour $R_{(211)}$ de B on définit $\sigma_{(211)}$ puisque dans σ apparaît





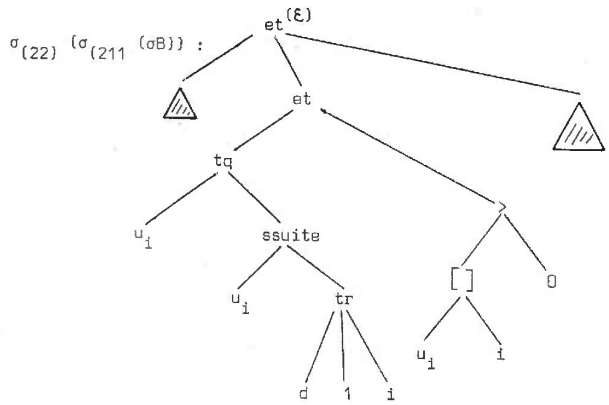
$\sigma_{(211)}$ composée avec σ :



Pour $P_{(22)}$ en composant tout de suite avec σ :

$\sigma_{(22)} = \{x_{r_{(2221)}} \leftarrow u_i \ , \ x_{l_{(2222)}} \leftarrow i \ \}$

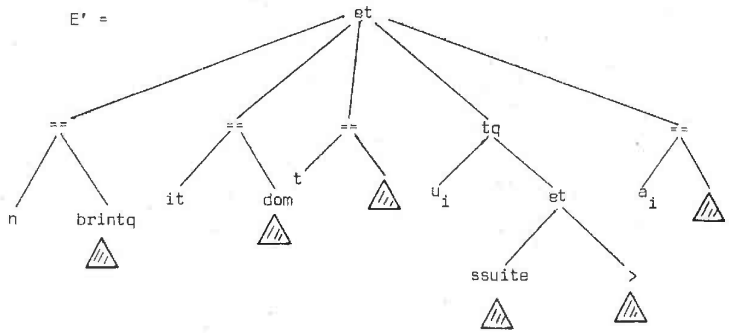
- stade 4. Définition de l'instantiation de σ B.



E' le transformé de E par s est donc :

E' : t == der u_i pour i de 1 jqa a_i
u_i tq ssuite(u_i, tr(d,1,i)) et u_i[i] > 0
a_i == (i = bs(d))
n == nbrintq(d, d[i] > 0)

Dans l'arbre E on remplace le sous arbre E/3 par l'arbre précédent (stade 4) pour obtenir finalement l'arbre E' :



II.8. Conclusion

La démarche d'explicitation proposée ici est fondée sur les notions de stratégies d'explicitations et de transformation.

Dans cette démarche les preuves de programmes sont utilisées comme un ensemble de règles de constructions. Nous avons analysé ces règles en précisant leur forme, leur rôle et en proposant une certaine classification. Ces règles indiquent une façon particulière de progresser vers une solution algorithmique. Pour bien marquer cette idée là la notion de stratégies d'explicitations a été introduite. Et nous avons alors distingué les stratégies d'explicitations locales (SEL) de la forme {pre} S :: B {post}, des stratégies d'explicitations globales (SEG) de forme algorithmique. Et nous avons illustré par des exemples de SEL commentés, l'idée précédente de progression. Cette notion englobe donc tout ce qui a été dit sur les règles de constructions. (On peut parler ainsi de classification de stratégies).

Une transformation se définit par l'application de la règle générale de remplacement et par la technique de remplacements d'arbres. L'application de cette règle pose essentiellement le problème du choix dans l'explicitation. Et elle est mise en oeuvre par la technique de remplacements d'arbres : un énoncé E et une stratégie locale (ou une règle de construction) r sont des arbres bien formés et attribués. Le calcul du filtre d'identification permet d'appliquer r sur E, et le calcul du filtre c' permet de produire le résultat de cette application.

Une transformation consiste alors :

- à choisir parmi les identificateurs d'un énoncé formel E celui dont on veut expliciter la définition,
- puis de choisir parmi toutes les règles de constructions, applicables sur E, celle qui paraît la mieux s'appliquer compte tenu du contexte du travail de l'explicitation,
- et enfin à appliquer cette règle, ce qui peut avoir pour conséquence l'introduction de nouvelles définitions à expliciter ultérieurement.

Poursuivons à présent la discussion sur les stratégies d'explicitations, notamment sur les stratégies globales.
Et sur ce qui vient d'être dit sur une transformation, décrivons aussi un méta-algorithme d'explicitation pour montrer la manière dont est construite une suite de transformations.

CHAPITRE III

III. STRATEGIES ET META-ALGORITHME D'EXPLICITATIONS

Le problème de l'explicitation d'un énoncé E (en Spes) en un énoncé M (en Médée) correct vis-à-vis de E a été essentiellement caractérisé par (cf. chap. II) :

- l'utilisation de stratégies d'explicitations (les stratégies locales, globales, et les règles de constructions),
- une technique de remplacement.

On peut donc dire que l'explicitation est une suite de transformations d'énoncés dont le premier terme est un énoncé E de départ et le dernier terme un énoncé M explicite.

La manière de construire une telle suite est donnée par un méta-algorithme d'explicitations au § III.2. lequel est suivi d'exemples d'explicitations (§ III.3.).

Cette construction s'appuyant sur l'utilisation de stratégies, il se pose alors le problème de leur découverte.
Une analyse de ce problème est donnée au § III.1. où des stratégies globales sont définies.

III.1. Le problème de la découverte de stratégies

Les stratégies expriment formellement des mécanismes de résolution. Elles traduisent certains raisonnements du programmeur d'une manière telle que leur application répétée conduit à une solution algorithmique d'un énoncé.
Nous nous intéressons ici à l'activité de recherche de ces stratégies. Pour mettre en évidence des mécanismes de constructions nous proposons de travailler par apprentissage, en ce sens SIE est un outil expérimental.
Par ces expériences sur la construction de programme on vise deux objectifs :
1. l'identification des mécanismes,
2. la construction d'un mécanisme d'apprentissage.

L'identification de mécanisme signifie définir des stratégies (c'est-à-dire préciser une forme, cela a déjà été fait au chap. II) et préciser leur origine.
Pour identifier des mécanismes, une démarche consiste à se restreindre à une classe d'énoncés explicites (cf. chap. I), à décrire formellement des stratégies (cf. Exemples de SEL § II.3, § II.5.3.) et surtout à se placer dans un certain contexte de mécanismes d'explicitation caractérisé par une vue sur la construction intuitive de stratégies. (Cette démarche se retrouve dans certains travaux sur la programmation transformationnelle [BAK-80, BAS-80, BRO-83, BUR-77, DAR-83, PAG-83]).
C'est ainsi que pour identifier des stratégies on cherche à les classer et surtout on s'appuie sur la notion d'induction sur une suite et sur la technique des invariants et de la récurrence. Analysons le premier objectif sous ce point de vue.

Le second objectif n'a pas encore fait l'objet d'une véritable étude. Cependant, nous montrons au § III.1.3. que sur la base de l'analyse précédente il est peut être possible de tendre vers une systématisation de la découverte de stratégies.

III.1.1. L'identification de stratégies

Une nouvelle classification de stratégies en familles, une technique et l'autre opérationnelle, est proposée. Et une discussion est engagée sur la construction intuitive de stratégies.

III.1.1.1. Les stratégies techniques et opérationnelles

La forme d'un énoncé E à expliciter peut être vue par le programmeur soit comme étant complexe, soit au contraire maîtrisable.
Dans le premier cas on veut alors "simplifier" la structure de E par l'application de stratégies de simplification. De telles stratégies sont appelées "stratégies techniques".
Dans le second cas, on désire plutôt appliquer des stratégies utilisant la forme de E pour progresser vers une solution algorithmique. On les appelle "stratégies opérationnelles".
Concrétisons ces idées par quelques exemples :

1) l'application de la 1-règle,

vrai et Q ⇒ P :: Q ⇒ P

a un énoncé permet de simplifier sa structure. Cette règle est une stratégie technique.
De façon générale toutes les règles dites logiques (les 1-règles § II.2) sont considérées comme des stratégies techniques,

2) la stratégie transformant un énoncé

E de la forme : f(r,g(t,h(z))) et h'(p(a),b) et x = h'(p(b),a)

en un énoncé E' de la forme

f(r,u) et u = g(t,v) et v = h(z) et h'(w,b) et w = p(a)
et x = h'(w',a) et w' = p(b)

est une autre stratégie technique. Elle remplace dans E tous les symboles d'opérations figurant comme paramètre d'un autre symbole par des variables. (le symbole "=" n'étant pas considéré dans ce processus de remplacement),

3) Donnons un dernier exemple de stratégie technique :

soit l'énoncé E :

s,r tq R(r,s) et R'(s)

où dans le sous énoncé R'(s) la variable r ne figure pas ; la stratégie transformant E en :

r tq R(r,s)

s tq R'(s)

est aussi une stratégie technique.

Quant aux stratégies opérationnelles, toutes les SEL (à l'exception des 1-règles) sont considérées comme appartenant à cette seconde famille. Trois des quatre stratégies globales définies par la suite appartiennent également à cette famille. La quatrième est une stratégie technique.

Le but d'une stratégie technique est de réduire la complexité de la construction d'une solution algorithmique. Et elles peuvent ainsi aider le programmeur à mettre en évidence des stratégies opérationnelles. En effet le résultat de l'application d'une stratégie technique est un énoncé "simple", voire connu du programmeur lui-même.

Les stratégies opérationnelles sont donc à la base de toute explicitation. On s'intéresse alors à cette famille de stratégies en étudiant la relation qui existe entre les SEL opérationnelles et les SEG opérationnelles.

III.1.1.2. Une vue sur la construction intuitive de stratégies

Le raisonnement d'un programmeur lors de la résolution d'un énoncé est généralement de haut niveau comparativement aux SEL (cf. § II.5.2.3). Ces dernières apparaissent plutôt comme des décisions prises par le programmeur pour appliquer une partie de son raisonnement.
Par exemple :

Soit E : t tq ssuite(t,d) et $\forall$k t[k] > 0,

un raisonnement peut être constitué des points suivants :

. affaiblir E,

. trouver un invariant,

. trouver une relation de récurrence,

. déterminer une condition d'arrêt ;

la stratégie opérationnelle

$$\frac{R(t,d) \text{ et } \forall i\ P(t[i])}{\begin{array}{l} t == \text{der } u_i \text{ pour } i \text{ de } 1 \text{ jqa } a_i \\ u_i \text{ tq } R(u_i, d1_i) \text{ et } P(u_i[i]) \\ a_i == (i = bs(d)) \end{array}}$$

peut être vue comme la décision prise par le programmeur pour appliquer en partie ce raisonnement. En effet d'autres décisions sont également à prendre afin de construire une relation de récurrence.
On cherche alors à définir des stratégies globales permettant au programmeur d'appliquer entièrement un certain raisonnement. C'est ainsi que de telles stratégies sont des algorithmes particuliers de résolution fondés sur les SEL.
Et le résultat de l'application d'une SEG est soit un énoncé explicite, soit un énoncé non totalement explicite.
Une stratégie globale est donc liée à la manière dont les stratégies locales sont construites intuitivement (une SEG est une manière d'enchaîner les SEL).

La construction d'une SEL repose d'une part sur la notion d'induction [BAS-80, BUR-69, KIN-80, MOR-78, SCW-78, WEG-74, MAN-74] et d'autre part sur la technique des invariants et la récurrence [AMY-78, ARS-77, BAK-80, BAS-77, BAU-76, BROY-80, REM-84, DIJ-76, HOA-69]. Plus précisément on cherche soit une induction sur une suite, soit une induction sur le résultat. Dans le premier cas on est conduit à exprimer le résultats r par une définition itérative, dans l'autre cas à utiliser une opération d'un type abstrait (en particulier le type TABLEAU) ou la conditionnelle.

Une induction sur le résultat r d'un énoncé implicite e_r est une preuve (cf. § I) vérifiant que l'explicitation de r par une définition simple ou une conditionnelle est correcte vis-à-vis de e_r.
Une induction sur r peut être aussi caractérisée par une stratégie particulière d'introduction d'une définition explicite. Cette stratégie est supposée valide.
Une induction sur r est donc un raisonnement sur la structure du résultat.
Par exemple :

Soit e_r : r <u>tq</u> bs(r) = j-i+1
<u>et</u> $\forall k \in [1..bs(r)]$ $r[k] = d[i+k-1]$

avec e_i : i <u>tq</u> ...
e_j : j <u>tq</u> ...

On cherche à exprimer r à l'aide d'opérations sur le type d'intérêt (cela conduit à introduire des intermédiaires). Ainsi le tableau r peut être donné par :

r == tr(d,i,j).

L'induction sur r consiste à prouver :

e_i <u>et</u> e_j <u>et</u> r == tr(d,i,j) ⇒ e_r

L'induction sur une suite v consiste à calculer le résultat r (r : T') d'un énoncé e_r comme le dernier terme d'une suite d'objets u_i (u : suite de T') construite étape par étape telle que l'on ait à chaque étape :

$e_u(u_{i-1}, v_i)$ ⇒ $e_u(u_i, v_i)$ vraie
et $e_u(u_i)$ ⇒ $e_r(u_i)$ vraie,

et à la dernière étape caractérisée par u_l le dernier terme de la suite :

$e_u(u_{l-1}, v_l)$ ⇒ $e_u(u_l, v_l)$ est vraie
et $e_u(u_l)$ ⇒ $e_r(u_l)$ avec r == u_l est vraie.

C'est ainsi que r est explicité par une définition itérative.
Cette induction est un raisonnement d'induction structurelle.
Par exemple la stratégie opérationnelle donnée précédemment exprime un calcul de t par approximations successives qui utilise une induction sur le tableau d. (on a alors une suite de tranches de d).

Un autre exemple d'induction sur une suite est donné par la stratégie suivante :

t <u>tq</u> R(t,k)
t : TAB k : ENT

t == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i
u_i <u>tq</u> R(u_i,i)
a_i == (i = k)

Cet exemple diffère du précédent dans la mesure où l'induction porte sur la variable entière k.
Pour calculer r (r : T') par approximations successives il est donc nécessaire de définir une certaine suite u (voir les 2 exemples précédents). Et afin de construire effectivement cette suite u telle que e_{u_i} soit correct vis-à-vis de e_r, la méthode d'induction sur une suite est complétée par la technique des invariants et de récurrences.

On sait déjà [§ II.1.1.a)] que dans une définition itérative de la forme :

r == <u>der</u> u_i <u>pour</u> i <u>de</u> k <u>jqa</u> a_i,

u est une suite caractérisée par un invariant e_u,
a est une suite de booléans caractérisée par une condition d'arrêt e_a,
et que e_a est liée à l'invariant par :

e_{a_i} <u>et</u> <u>non</u> a_i <u>et</u> $e_u(u_{i-1})$ ⇒ $e_u(u_i)$
e_{a_i} <u>et</u> a_i <u>et</u> $e_u(u_i)$ ⇒ $e_r(u_i)$

On se pose alors le problème de la découverte des invariants et par conséquent celui du choix d'une fonction f devant construire la suite u. Cette fonction permet d'établir la relation de récurrence de premier ordre de la forme : u_i == $f(u_{i-1}, v_i)$.

Pour découvrir e_u on peut avoir dans certains cas une intuition de l'invariant. Dans ces cas il faut alors faire des "preuves". Et dans d'autres cas, on cherche à <u>calculer</u> l'invariant à partir de l'énoncé du résultat. C'est alors que les règles nous servent.

Illustrons cette idée par les deux exemples suivants :

1) Soit l'énoncé :

e_r r <u>tq</u> equ(r, concat(t_1,t_2)) <u>et</u> croiss(r)

t_1 : TAB, t_2 : TAB, r : TAB (des tableaux d'entiers)

avec dom(t_1) = [1..k_1] <u>et</u> dom(t_2) = [1..k_2]

<u>et</u> croiss(t_1) <u>et</u> croiss(t_2)

pour lequel r est exprimé par une définition itérative. Alors le programmeur peut être amené à choisir entre les invariants suivants :

a) e_{u_i} : u_i <u>tq</u> equ(u_i, concat(t_1, tr(t_2,1,i)) <u>et</u> croiss(u_i)
avec comme condition d'arrêt liée à e_u : e_a : a_i == (i = k_2)

b) e'_{u_i} : u_i <u>tq</u> equ(u_i,t_1 ∿ t_2) <u>et</u> croiss(u_i) <u>et</u> v_i = t_1 ∿ t_2
avec e'_a : a_i == (i = k_1 + k_2)

c) e''_{u_i} : u_i <u>tq</u> equ(u_i,tr(t_1,1,l) ∿ tr(t_2,1,q_i)) <u>et</u> croiss(u_i)
avec e'_a : a_i == (i = k_1 + k_2)

l_i,q_i <u>tq</u> 1 ⩽ l_i ⩽ k_1 $t_1[l_i]$ ⩽ $t_2[q_i]$ ⇒ l_i = l_{i-1}+1
et q_i = 0

ou 1 ⩽ q_i ⩽ k_2 $t_2[q_i]$ < $t_1[l_i]$ ⇒ q_i = q_{i-1}+1
et l_i = 0

<u>Commentaire</u>

L'invariant e_u est obtenu en appliquant la stratégie opérationnelle :

R (t,d)

t == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i
u_i <u>tq</u> R(u_i,$d1_i$)
a_i == i = bs(d)

, c'est-à-dire par un calul. Notons qu'il y a un choix de la variable à induire).

L'invariant e'_{u_i} est également obtenu par calcul : application d'une stratégie technique sur e_r (voir § III.1.1.1 2ème exemple de stratégie technique), et ensuite application de la stratégie précédente.

Quant à l'invariant e''_u on peut supposer qu'il a été plutôt "inventé" par le programmeur. Il est donc nécessaire de faire une certaine preuve dans le but de démontrer que e''_u est effectivement un invariant (§ II.1, II.2).

2) Reprenons à nouveau l'énoncé :

e_t : t <u>tq</u> ssuite(t,d) <u>et</u> ∀ k t[k] > 0

t : TAB d : TAB k : 1..n

avec n == nbrintq(d,d[m] > 0)

pour lequel t est exprimé par une définition itérative.

Comme précédemment le programmeur peut être amené à choisir entre :

a) e_{w_i} : w_i <u>tq</u> ssuite(w_i, tr(d,1,i)) <u>et</u> $w_i[i]$ > 0,
avec e_a : a_i == (i = bs(d))

b) e'_{w_i} = w_i <u>tq</u> ssuite(v_i,d) <u>et</u> tr(w_i,i,i) = v_i
et e'_a : a_i == (i = n)
avec v_i <u>tq</u> ssuite(v_i, tr(d,l_i,l_i)) <u>et</u> $v_i[i]$ > 0
l_i <u>tq</u> ∀ k, l_{i-1} < k < l_i ⇒ non d[k] > 0.

<u>Commentaire</u>

L'invariant e_{w_i} est obtenu par un calcul précisé par la stratégie opérationnelle donnée au début de ce paragraphe (§ II.1.1.2).
On peut considérer que e'_{w_i} a été "inventé" par le programmeur. Il faut donc faire la preuve de e'_{w_i}.

Ces deux exemples montrent que la découverte des invariants e''_{u_i} et e'_{w_i} n'est pas évidente. Alors que celle de e'_{u_i}, e_{u_i} et e_{w_i} est plus évidente. Et dans ce cas précis on pourrait même systématiser de tels invariants en s'appuyant sur deux conditions simples d'existence d'invariants à savoir :

- <u>Condition 1</u> :

Si le résultat r d'un énoncé e_r à expliciter est un argument d'un prédicat n-aire (n ⩾ 2) où figure au moins une variable entière k

libre, alors un invariant est obtenu en remplaçant r (respectivement k) par un terme u_i (resp. k_i) :

$$e_u(u_i,i) = e_r\ [r \leftarrow u_i,\ k \leftarrow k_i].$$

- Condition 2

Si le résultat r (de type tableau) d'un énoncé e_r à expliciter est un argument d'un prédicat n-aire ($n \geqslant 2$) où figure au moins une variable de type tableau d libre alors un invariant est obtenu en remplaçant r (respectivement d) par un terme u_i (resp. $d1_i$) :

$$e_u(u_i,i) = e_r\ [r \leftarrow u_i,\ d \leftarrow d1_i].$$

Donnons quelques exemples :

1. dans l'énoncé e_t précédent, t dépend de d ce qui est traduit par le prédicat binaire ssuite(t,d). Alors un invariant possible peut être donné en remplaçant systématiquement dans e_t :

$$[t \leftarrow u_i,\ d \leftarrow d1_i,\ k \leftarrow k_i]$$

e_{u_i} : u_i tq ssuite(u_i,$d1_i$) et $u_i[k_i] > 0$ (cet invariant est différent de celui donné précédemment).

2. dans l'énoncé e_r précédent, r dépend de t_1, t_2 mais aussi de concat(t_1,t_2). La condition 2 ne précise pas ce dernier cas. Il appartient alors au programmeur de choisir t_1 et/ou t_2. Si t_1 est choisi comme variable inductive on obtient :

e_{u_i} : u_i tq equ(u_i, tr(t_1,1,i) $\sim t_2$) et croiss(u_i)
(cet invariant n'a pas été proposé dans l'exemple 1).

3. un exemple général :

e_x : x tq P(x,y,z) et P'(x,t)
x : TAB y : TAB z : ENT t : VAL

dans cet énoncé x dépend de y mais aussi de z et t, au programmeur de faire son choix par exemple :

e_{w_i} : w_i tq P(w_i,$y1_i$,z_i) et P'(w_i,t)
le programmeur a décidé de ne pas faire porter la récurrence sur t.

Les deux conditions d'existence d'invariants, énoncées plus haut, sont suffisantes pour une systématisation de la détection d'invariants laissant le choix des variables à indicer au programmeur. (Les variables indicées sont : d, k, t_1, x, y dans les exemples précédents).

Les stratégies locales ne permettent de réaliser qu'une partie d'une démarche d'explicitation comme par exemples l'introduction d'une définition itérative, la caractérisation d'une relation de récurrence. Au contraire, le rôle des stratégies globales est de décrire l'application de démarches telles que l'induction.
On désire alors que les stratégies globales soient conçues d'une manière telle qu'elles permettent une application totale d'une induction ou de la technique d'invariants et de récurrence. Illustrons ces propos en présentant quelques stratégies globales.

III.1.2. Quelques stratégies d'explicitations globales

La forme des stratégies globales a déjà été précisée au § II.5.3. : puisqu'un calcul de programmes précise un enchaînement de stratégies, une stratégie globale apparaît donc comme un algorithme. Un tel calcul peut éventuellement lier des stratégies globales. Il n'existe pas de relations particulières entre les SEG présentées ci-dessous.

Quatre stratégies principales ont été reconnues :
- la stratégie d'explicitation d'affaiblissement (<sea>),
- la stratégie d'explicitation d'initialisation (<sei>),
- la stratégie d'analyse par cas (<sac>),
- la stratégie d'explicitation de modulation (<sem>).

Chacune des présentations suit le plan suivant :
1. But général de la stratégie,
2. Description générale de sa mise en oeuvre (cf. § IV.2.2.)
3. Etude d'un exemple.

Remarque :

Ces stratégies sont les seules stratégies globales disponibles dans la bibliothèque actuelle du système. En outre le système ne dispose pas de stratégies appliquant une induction sur le résultat permettant d'exprimer directement le résultat r d'un énoncé par une définition simple (qui ne soit pas une relation de récurrence).

III.1.2.1. Stratégie d'explicitation par affaiblissement

Le but de <sea> est de résoudre un énoncé donné par approximations successives du résultat. L'application répétée de <sea> permet d'aboutir à un énoncé itératif. Son mécanisme est interactif.

Description générale de <sea>

Soit e_r l'énoncé à expliciter :

1. Sélection de toutes les SEL applicables à e_r et caractérisées par une définition itérative,
2. Choix par le programmeur d'une SEL ou proposition d'une stratégie à appliquer, l'invariant est alors l'énoncé e_u, et la condition d'arrêt e_a,
3. Sélection de toutes les SEL applicables à e_u et introduisant une relation de récurrence,
   choix entre : a) une SEL sélectionnée,
   b) une induction sur le résultat (proposition du programmeur),
   c) une induction sur une suite et application de <sea> pour e_u,
4. Définition de la condition d'arrêt (comme 3 si e_a n'est pas explicite),
5. Initialisation des variables récurrentes :
   choix entre : - proposition du programmeur,
   - <sei>.

Une illustration de <sea> :

l'énoncé de la figure III.1. ci-dessous sera également utilisé pour illustrer les autres stratégies. Explicitons ainsi le problème suivant :

Construire la première séquence constante de longueur n à partir d'un tableau d'entiers d.

Lexique	Profil	Enoncé formel
t tableau résultat de longueur n constitué d'éléments de même valeur d tableau d'entiers donné n la longueur de la séquence constante à trouver a la borne supérieure de d m la borne supérieure de t	t : TAB d : TAB k : [1..a] l : [1..a] n : ENT m : ENT extens : TAB x P → BOOL egal : TAB x TAB → BOOL	résultat : t t,k,l tq egal(t,tr(d,k,l)) et extens(t,d[k] = t[1]) et n = lg(t) et a = bs(d) données : d,r

Figure III.1.

Le prédicat extens sur le type TABLEAU (§ V.1.) indique qu'un élément du tableau vérifie une propriété P et qu'alors tous les autres éléments la vérifient aussi.

De façon formelle, dans le cas présent,

extens(t,d[k] = t[1]) signifie : $\forall$ l [1..bs(t)]
d[k] = t[1] et d[k] = t[l].

Application du mécanisme

1. sélection de toutes les SEL applicables à e_r et introduisant une définition itérative,

2. le programmeur décide d'appliquer une stratégie locale sélectionnée en 1. (le lecteur la reconnaîtra facilement) on obtient alors :

$$t == \underline{der}\ u_i\ \underline{pour}\ i\ \underline{de}\ 1\ \underline{jqa}\ a_i$$
$$u_i, k_i, l_i\ \underline{tq}\ egal(u_i, tr(d, k_i, l_i))$$
$$\underline{et}\ extens(u_i, d[k_i] = u_i[1])\ \underline{et}\ a = bs(d)$$
$$\underline{et}\ n = lg(u_i)$$
$$a_i == (n = lg(u_i))\ \underline{ou}\ (l_{i-1}+1 = bs(d))$$

3. introduction d'une relation de récurrence : proposition de <sac> décrite ci-dessous, on obtient :

$$u_i == \underline{si}\ n = lg(u_{i-1})\ \underline{alors}\ u_{i-1}$$
$$\underline{sinon}\ tr(d, k_i, l_i)$$
$$k_i, l_i\ \underline{tq}\ l_{i-1} < k_i \leqslant l_i$$
$$\underline{et}\ \forall p\ l_{i-1} < p < k_i \Rightarrow d[p]\ \underline{non} = d[k_i]$$
$$\underline{et}\ (k_i = b_i(d)\ \underline{ou}\ d[k_i] = d[l_i])$$
$$\underline{et}\ l_i+1 = bs(d)\ \underline{ou}\ d[l_i+1] = d[k_i]$$

$$u_o\ \underline{tq}\ egal(u_o, tr(d, k_o, l_o))\ \underline{et}\ extens(u_o, d[k_o] = u_o[1])$$
$$\underline{et}\ a = bs(d)\ \underline{et}\ n = lg(u_o)$$

4. la condition d'arrêt est explicite

5. initialisation de la variable u_o : application de <sei> décrite ci-dessous : u_o == tvide.

Ensuite <sea> s'applique sur k_i, l_i :

1. Sélection de SEL,

2. Choix du programmeur : recherche associative (principe de la règle rdc4 § II.3.1.)
   on obtient :

$$k_i == \underline{prem}\ p\ \underline{dans}\ d\ \underline{depuis}\ l_{i-1}+1\ \underline{tel\ que}\ d[l_{i-1}+1]\ \underline{non} = d[k]$$
$$l_i = k_{i-1}$$

3. Initialisation de l_i : choix du programmeur (l_i est une variable indicée et "non récurrente"). $l_o = 0$

Le mécanisme <sea> s'arrête ici puisqu'il n'existe plus d'identificateurs à expliciter.

III.1.2.2. Stratégies d'explicitation d'initialisation

Le but de cette stratégie est de construire les initialisations de certaines variables ; en particulier celles des variables récurrentes introduites par une définition itérative.

Description générale de <sei>.

Elle est donnée par un mécanisme interactif, non récursif : (pour l'initialisation des variables récurrentes).

Soit u_o la variable récurrente à initialiser d'énoncé e_o.

(1) Choix, par le programmeur, d'un prédicat $P(u_o, \bar{x})$ ou $\bar{x}$ est une liste d'arguments.

(2) Choix entre :
- recherche d'une règle r1 de la forme
  r1 : {pre} $P(c, \bar{x}')$ :: vrai {post}
  (si r1 existe alors on pose : u_o == c où c est une constante)
- proposition du programmeur d'une initialisation : u_o == c

(3) Vérification de la définition explicite de u_o :
il s'agit soit de prouver les définitions
$\bar{x} == \bar{x}'$ (correspondance entre les arguments de $P(u_o, \bar{x})$ et $P(c, \bar{x}')$),
soit dans le cas d'une proposition de prouver un prédicat dans $e_o\ [u_o \leftarrow c]$.

Remarque :

Il suffit d'un prédicat $P(c, \bar{x})$, un constituant de $e_{u_o}\ [u_o \leftarrow c]$, soit vrai pour que u_o == c soit correcte vis-à-vis de e_{u_o}.

Pour illustrer ce mécanisme, considérons la définition de u_o de l'exemple précédent :

$$u_o\ \underline{tq}\ egal(u_o, tr(d, k_o, l_o))\ \underline{et}\ extens(u_o, d[k_o] = u_o[1])$$
$$\underline{et}\ a = bs(d)$$

1. Choix de egal(u_o, tr(d,k_o,l_o))
2. Choix du programmeur : u_o == tvide
3. Vérification

Soit la règle r1 : $\frac{\text{egal(tvide,tvide}}{\text{vrai}}$

On doit vérifier que tr(d,k_o,l_o) = tvide, le programmeur admet que $l_o < k_o$.
Ce qui permet d'appliquer la règle tr(d,k_o,l_o) = tvide si $l_o < k_o$.

III.1.2.3. Stratégie d'analyse par cas

L'idée est d'étudier des formes particulières d'énoncé à partir desquelles on sait reconnaître des "schémas" conditionnels en exhibant un ensemble d'axiomes préconditionnés d'un type abstrait choisi. C'est-à-dire que l'on cherche à mettre en évidence à partir de tels axiomes des formules de la forme $P \Rightarrow Q(r)$ (appelée F), où

P exprime une condition déduite des pré-conditions

et Q(r) une définition implicite déduite à partir de l'énoncé e_r et des pré-conditions précédentes.

Un schéma conditionnel est un ensemble de formules (F). Cet ensemble est donc à construire à partir de l'énoncé à expliciter e_r et de règles munies de pré-conditions.
Dans ce but, l'analyse par cas consiste à faire apparaître les différents cas soit à partir de la variable résultat de l'énoncé, soit à partir de variables considérées comme données. A l'ensemble ainsi construit, la règle de construction de définition d'une conditionnelle est alors appliquée, c'est-à-dire :

de $P_1 \Rightarrow Q_1(r_1)$ ou <u>non</u> $P_1 \Rightarrow Q_2(r_2)$ on passe à

r == <u>si</u> P_1 <u>alors</u> r_1 <u>sinon</u> r_2

avec r_1 <u>tq</u> $r_1 = r \Rightarrow Q_1(r_1)$, r_2 <u>tq</u> $r = r_2 \Rightarrow Q_2(r_2)$
(cela peut se généraliser à plusieurs branches).

<u>Description générale</u>

Le mécanisme de <sac> est également interactif :

Soit e_r la définition implicite à expliciter.

1. Transformer e_r afin de mettre en évidence des prédicats qui serviront à exprimer les différents cas d'analyse,
2. Choix entre :
   a) - analyse par le résultat,
   b) - analyse par les données,
   Ce choix permet de rechercher des règles pré-conditionnées propres à a) ou b),
3. Application des règles pré-conditionnées choisies et déduction en tenant compte du résultat de l'étape 1 :
   - soit des conditions P_i (cas b) de l'étape 2) (on veut construire $P_i \Rightarrow Q_i(r_i)$)
   - soit des définitions des résultats $Q_i(r_i)$,
4. Déduire à partir de P_i (ou Q_i) et e_r : Q_i(ou P_i) ; on construit le schéma conditionnel,
5. Transformation du schéma obtenu en 4. en une définition conditionnelle.

<u>Illustration du mécanisme.</u>

Appliquons <sac> pour définir la relation de récurrence de l'énoncé :

e_{u_i} : u_i,k_i,l_i <u>tq</u> egal(u_i, tr(d,k_i,l_i)
<u>et</u> extens(u_i, d[k_i] = u_i[1])
<u>et</u> r = lg(u_i) <u>et</u> a = bs(d)

(voir exemple de <sea>),

1. Le programmeur décide de transformer e_{u_i} par l'application d'une stratégie technique (admettons l'existence d'une telle stratégie). Ce qui donne :

u_i tq egal(u_i, tr(d,k_i,l_i)
et r = lg(u_i)
k_i,l_i tq extens(u_i,d[k_i] = u_i[1]) (voir définition précise dans l'exemple de <sea>)
et a = bs(d)

2. Choix analyse par le résultat : on veut exprimer le passage de u_{i-1} à u_i par une définition conditionnelle, pour cela comparons u_i et u_{i-1} tq egal(u_{i-1}, tr(d,k_{i-1},l_{i-1})
et n = lg(u_{i-1}),
le programmeur décide de passer à l'étape suivante pour exprimer la comparaison entre u_i et u_{i-1} par des préconditions (pas de recherche de règles).

3. 4. Recherche des résultats de u_i et des pré-conditions :
n = lg(u_{i-1}) ⇒ u_i = u_{i-1}
non n = lg(u_{i-1}) ⇒ egal(u_i, tr(d,k_i,l_i))

5. Le schéma construit est :
n = lg(u_{i-1}) ⇒ u_i = u_{i-1}
non n = lg(u_{i-1}) ⇒ u_i = r_1
r_1 tq u_i = r_1 et egal (r_1, tr(d,k_i, l_i))

la conditionnelle est alors :
u_i == si n = lg(u_{i-1}) alors u_{i-1}
sinon r_1
en appliquant la règle egal(r_1, tr(d,k_i,l_i)) :: r_1 == tr(d,k_i,l_i)
On retrouve : u_i == si n = lg(u_{i-1}) alors u_{i-1}
sinon tr(d,k_i,l_i).

III.1.2.4. Stratégie d'explicitation de modulation

Les précédentes stratégies sont des stratégies opérationnelles. <sem> est un exemple de stratégie technique.
Le but de <sem> consiste à mettre un énoncé à expliciter sous une forme connue du programmeur et simple à traiter par le SIE. On peut parler alors de formes canoniques d'énoncés.

Et comme il a été déjà dit l'intérêt de <sem> est de faciliter la description d'une recherche de stratégies opérationnelles et éventuellement de contribuer alors à une systématisation de la découverte de ces stratégies.

Description générale.
Le mécanisme de <sem> est interactif et itératif :

Soit e_r l'énoncé à simplifier.

① Choix entre :
a) proposition de simplification : introduction d'une stratégie locale technique et application à celle-ci sur e_r, (retour en 1 ou sortir),
b) remplacement de toute fonction, figurant comme argument d'une fonction ou d'un prédicat, par une variable. (Voir 2ème exemple de stratégie technique § III.1.1.1.).

Illustration du mécanisme.
Un exemple du cas a) du mécanisme de <sem> peut être donné par la transformation effectuée dans l'illustration précédente à l'étape 1.
A cette étape 1 on est passé de
u_i tq egal(u_i, tr(d,k_i,l_i))
et extens(u_i,d[k_i] = u_i[1]) et n = lg(u_i) et a = bs(d)
à u_i tq egal(u_i, tr(d,k_i,l_i)) et n = lg(u_i)
et k_i,l_i tq extens(u_i,d[k_i] = u_i[1]) et a = bs(d)
(voir définition dans l'exemple de <sea>)
par une "décomposition descendante" : dans la première définition u_i est un résultat, dans l'autre définition u_i est considérée comme une donnée.
Cette décomposition a facilité l'application de <sac>.

Un exemple du cas b) :
e_{u_i} : u_i tq egal(u_i, tr(d,k_i,l_i))
et extens(u_i,d[k_i] = u_i[1])

se réécrit en :

$$e'_{u_i} : u_i \ \underline{tq}\ egal(u_i, r') \ \underline{et}\ extens(u_i, d[k_i] = u_i[1])$$
$$\underline{et}\ r_1 = tr(d, k_i, l_i)$$

par l'application de l'axiome :

$$P(x,y) \Leftrightarrow P(x,z) \ \underline{et}\ z = y$$

en conséquence e_{u_i} est équivalent a e'_{u_i}.

e'_{u_i} est alors caractérisé uniquement par des constituants de la forme $P(\bar{x})$ où dans $\bar{x}$ ne figure aucun symbole d'opération (exception faite des prédicats).

III.1.3. Pour un système de construction de stratégies

La tâche fondamentale du programmeur est de découvrir les stratégies à appliquer au cours d'une application.
On veut alors fournir un mécanisme d'apprentissage. Ce mécanisme aidera le programmeur dans sa tâche de recherche de stratégies à appliquer. Cependant le système peut fournir des stratégies pré-définies.
Analysons notre objectif.

Le problème de la découverte de stratégies est lié à la définition de mécanismes intervenant dans le processus intellectuel de la construction.
Ce processus est continu. Et par l'analyse intuitive des stratégies, présentée au § III.1.1., on a cherché à discrétiser ce processus par :

1. Une classification des stratégies par leur forme et leur rôle,
2. Une analyse particulière de la construction intuitive de stratégies.

Le but de la classification est de mieux situer un mécanisme dans le processus intuitif de construction. (Ainsi à partir de la forme d'une SEL {pre} S :: E,D {post} des classes de mécanismes, de stratégies ont pu être définies. (cf. § II.3., II.5.)).

La particularisation d'un mécanisme est liée à la façon de voir intuitivement la transformation d'énoncés. Dans le cas de la construction d'énoncé Médée à partir d'énoncé Spes cette transformation repose sur des notions précises :
- l'induction,
- la technique d'introduction d'invariants et de récurrence.

L'application de ces notions a pour effet de définir des stratégies opérationnelles (locales ou globales).

Par conséquent, cette discrétisation se fonde sur des choses que nous avons su définir, à savoir, répétons-le : la définition de classes de mécanismes et les notions d'induction et d'invariants. Elle montre ainsi que l'on arrive à préciser, et à détecter les mécanismes à la base de la construction.

On peut donc dire :
puisque l'on est capable de reconnaître des mécanismes de construction et que ceux-ci sont précis, on devrait être capable de systématiser la découverte de stratégies.

Illustrons cette idée de systématisation. Pour cela essayons de définir succintement un système de construction de stratégies.

On désire que ce système repose sur la classification de stratégies (SEG, SEL, règles générales de constructions - p-règles, l-règles -, stratégies opérationnelles, techniques) et les notions d'induction et d'invariants.

Ce système hypothétique est donné par le schéma III.1. suivant :

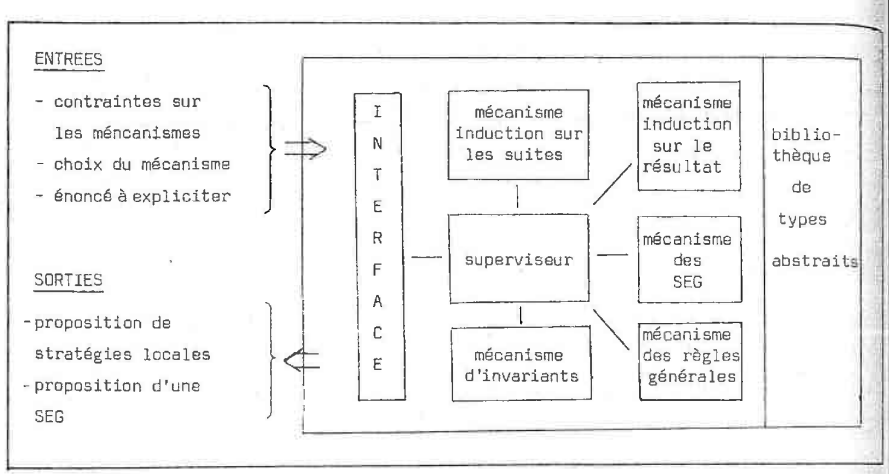


Schéma III.1. Un système de construction de stratégies

L'objectif de ce système interactif est la construction de stratégies adaptées à une classe d'énoncés à expliciter. Pour cela l'utilisateur fournit des contraintes sur le mécanisme choisi.

Les mécanismes devraient être alors paramétrés et interactifs.
Par exemple le mécanisme des SEG signifie que l'on voudrait avoir la possibilité de construire des heuristiques de <sea>, <sei>, <sac>, et <sem> à partir de l'énoncé et de contraintes sur le mécanisme.
Un tel système n'a pas encore fait l'objet d'une étude approfondie.

Donnons alors un exemple de construction systématique de stratégies.
Cet exemple illustre un mécanisme d'invariants.
On veut construire des stratégies locales adaptées à l'énoncé :

e_r : r <u>tq</u> equ(r,d) <u>et</u> croiss(r).

L'utilisateur choisit les mécanismes d'induction sur une suite et d'invariants.
Il formule comme contraintes particulières :

. Contruction de SEL sans pre-post, énoncé sur le type TABLEAU.

Le mécanisme d'induction sur une suite produit la définition itérative suivante :

d_x : x = <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i.

A ce state sont connus D (d_x) et les pré-post (valeurs vrais)
Par conséquent le mécanisme d'invariants doit produire des SEL de la forme :

S :: E, d_x.

Il reste donc à calculer S et E.
Pour calculer S le mécanisme d'invariants généralise la forme de e_r de sorte que e_r soit une instance d'une forme de S. Et de cette forme générale un invariant et une condition d'arrêt sont alors proposés.
Montrons ce calcul de SEL particulières.
A partir de la forme de e_r on peut calculer les SEL suivantes :

- pour S de la forme S_1 : R(x,y) <u>et</u> R1(x) on obtient comme SEL :

(1) S_1 :: u_i <u>tq</u> R(u_i,$y1_i$) <u>et</u> R1(u_i)
a_i == i = bs(y)
d_x

(2) S_1 :: u_i <u>tq</u> R(u_i,$y1_i$)
a_i <u>tq</u> R1(u_i) ⟺ a_i
d_x

(3) S_1 :: u_i <u>tq</u> R1(u_i)
a_i <u>tq</u> R(u_i,$y1_i$) ⟺ a_i
d_x

- pour S de la forme S_2 : R(x,y) on obtient :

$$S_2 :: u_i \ \underline{tq}\ R(u_i, y1_i)$$
$$a_i == i = bs(y)$$
$$d_x$$

- pour S de la forme S_3 : R(x) on obtient :

$$S_3 :: u_i \ \underline{tq}\ R(u_i)$$
$$a_i == u_i = tvide$$
$$d_x$$

On a admis au cours de ce calcul de S et E que le programmeur a accepté les trois formes de S (S_1, S_2, S_3) proposées par le mécanisme.
Cet exemple montre qu'il est possible de développer un système de construction assistée de stratégies. Un tel système peut être considéré comme un pas vers la résolution du problème de la découverte de stratégies.

En conclusion

Toutes les stratégies mentionnées jusqu'à présent sont des exemples de stratégies qu'aurait pu découvrir un programmeur. Et si la construction automatique de telles stratégies n'est pas réalisée, on peut quand même en décrire "à la main" et c'est une capacité importante de SIE.

Consacrons nous maintenant à l'explicitation mise en oeuvre par SIE. Pour cela on présente un méta-algorithme (§ III.2) et des exemples d'explicitation d'énoncés (§ III.3.). Cela permet de préciser l'utilisation et la manière dont les stratégies interviennent dans une explicitation.

III.2. Meta-algorithme d'explicitation

La spécification du méta-algorithme, donnée par la figure III.2., reprend celle proposée par [FIN-79] en utilisant les stratégies antérieurement définies et l'arbre des choix.

Ell est décrite en Médée (un programme Médée se présente sous la forme d'une suite de tableaux à 3 colonnes -(énoncé formel, profil, énoncé explicite)- chacun d'eux décrivant un module).

On suppose dans ce méta-algorithme que :

1. Il appartient à l'utilisateur de choisir la stratégie favorable parmi celles sélectionnées par SIE (ou d'en proposer), et l'identificateur à expliciter,
2. l'utilisateur n'a pas à choisir <sea> (stratégie d'affaiblissement). Celle-ci est la stratégie de base (comme le montre la figure III.1.),
3. lors du choix d'une stratégie, SIE calcule un ensemble de stratégies propres à chacune des classes :
   p-règles (<rdp>), l-règles (<rdl>), t-règles (<rdt>), ...
   indiqué dans la figure par :
   s == choix entre {<rdl>, <rdt>, <sac>, <rdp>, ...}.

En outre il faut bien voir que lors de ce calcul on peut avoir par exemple <rdl> = Ø (une classe peut être vide).

Lexique	Profil	ENONCE
E énoncé Spes comme décrit au chapitre I. P est le problème M énoncé Médés comme décrit au chapitre I ARBRE-CHOIX est l'arbre des résultats des applications des stratégies construit par SIE $idexpl_i$: booléen indiquant si tous les identificateurs ont une définition explicite. R est l'ensemble des stratégies disponibles	E : Enoncé M : Enonce explicite idexpl : Bool* Em : Enoncé*	résultat : M M == der Em_i pour i de 1 jqa $idexpl_i$ données : P = (S, E), R, ARBRE-CHOIX
* Em		
init crée un tableau à 3 colonnes en plaçant dans la colonne gauche E dans la colonne centrale les profils de tous les objets figurant dans E et dans la colonne de droite "résultat r" (r est le résultat de E). <expl> : fonction générale d'explicitation	init : E → E <expl> : E x bool → E	Em_i == <expl> (Em_{i-1}, $idexpl_{i-1}$) Em_o == init(E) $idexpl_o$ == faux
* <expl> (E_1, b) = E_2		
<seg> : stratégie générale appliquée à un énoncé caractérisant un identificateur dans une suite de modules explid : prédicat vrai si un identificateur est défini explicitement	E_1 : Enoncé E_2 : Enoncé <seg> : Module* x Enoncé x Symbole → Module* Explid : Ident → Bool	E_2 = <seg> (E_1, E_X, X) X = choix dans {y ∈ colonne centrale (E_1) tq ¬ explid(y)}
<seg> (E_1, E_X, X) = E_2		
<sea> : stratégie d'affaiblissement comme expliquée au § IV.1. retour : permet de remonter dans l'arbre des choix, et retourne un énoncé		E_2 == choix entre - retour(n) - <sea> (E_1, E_X, X) n == choix entre {m tq m ∈ dom (arbre-choix)}
retour(n) = E_2		
arbre-choix calcule à partir du point de reprise n le nouvel énoncé à expliciter	arbre-choix : ENT x ARBRE-CHOIX → ENONCE	E_2 == arbre-choix(n, $ARBRE\text{-}CHOIX_{i-1}$) ARBRE-CHOIX == E_0
<sea> (E_1, E_X, X) = E_2		
transfo : retourne l'énoncé E_1 transformé par l'application d'une stratégie s sur E_X maj : met à jour l'arbre des choix en ajoutant le résultat de la trasformation de l'application de s sur E_X <rdi> désigne toute stratégie de R (ou proposée par le programmeur) introduisant une définition itérative ("der" ou "premier tel que"). <rds> stratégie d'introduction d'une définition simple <sac> stratégie d'analyse par cas <sei> stratégie d'initialisation <rdt> stratégie d'introduction d'une définition explicite <rdr> stratégie d'introduction de la récurrence <sem> stratégie de modulation	transfo : ENONCE x ENONCE x STRATEGIE → ENONCE maj : ARBRE-CHOIX x ENONCE x STRATEGIE x ENONCE → ARBRE-CHOIX	E_2 == transfo(E_1, E_X, s) $ARBRE\text{-}CHOIX_i$ == maj($ARBRE\text{-}CHOIX_{i-1}$, E_X, s, E_2) s == choix entre {<rdi>, <rds>, <sac>, <sei>, <sem>, <rdt>, <rdr>, <rds>, <rdi>}

Figure III.2. : Méta-algorithme d'explicitation

III.3. Exemples d'explicitation

L'objectif des exemples présentés est d'illustrer le processus d'explicitation, basé sur notre méta-algorithme, et l'application de stratégies. Pour cela on structure le déroulement du méta-algorithme autour des mots réservés suivants :

but : pour indiquer l'énoncé initial à expliciter,

action : pour décrire un tableau Médée,

choix de l'identificateur à expliciter parmi : donne la liste des symboles résultats à expliciter

choix de indique le choix de l'utilisateur,

parce que donne une justification informelle du choix,

choix de la stratégie pour désigner une liste de stratégies favorables,

à faire : pour indiquer ce qu'implique le choix d'une stratégie,

décisions : précise la manière "d'éxécuter à faire".

(ici on peut également décider d'appliquer d'autres stratégies nécessaires à l'exécution de "à faire").

Deux exemples d'explicitations de problèmes sur la construction de programmes manipulant des tableaux à 1 dimension sont traités :

Exemple 1 : Trouver la première séquence connexe d'un tableau d d'entiers dont la somme soit maximale. Une séquence est délimitée par 2 nombres négatifs. La somme de toute séquence composée de nombres négatifs est nulle.

Exemple 2 : Construire un tableau d'entiers t triés par ordre croissant à partir d'un tableau d donné.

Exemple 1

a) Spécification du problème (figure III.3)

Lexique	Profil	Enoncé formel : e_t
t tableau d'entiers dont la somme est maximale d tableau donnée seem(t,d) = t,p,q tq t = tr(d,p,q) et j[p..q] d[j] < 0 ⇒ som(t) = 0 et (p = 1 ou d[p-1] < 0) et (q = bs(d) ou d[q+1] < 0)	t : TAB d : TAB l : [1..bs(d)] m : [1..bs(d)] [ ] : ENT x TAB → VAL seem : TAB x TAB → TAB som : TAB → ENT	résultat t t tq seem(t,d) et ∀l,m et (l = 1 ou d[l-1] < 0) et (m = bs(d) ou d[m+1] < 0) et j[l..m] d[j] < 0 ⇒ som(tr(d,l,m) = 0) et som(t) ⩾ som(tr(d,l,m)) donnée d

Figure III.3.

(* on se donne l'opération "som" sur les tableaux d'entiers :
som(tvide) = 0, som(affect(t,v)) = som(t) + 1,

* l'opération "seem" est définie par la propriété P_3 caractérisant t, à partir d'un tableau d, envoyée en paramètre dans le type de t :
TAB[VAL, ENT, ..., P_3] (cf. le type TABLEAU)). P_3 est la propriété figurant au lexique.

b) Explicitation du problème

Initialisation

but : expliciter l'énoncé formel de la figure III.2.
action : création du tableau initial Médée suivant

Enoncé formel	profil'	énoncé explicite
e_t figure III.3	t : TAB d : TAB figure III.3	résultat t .... donnée d

$idexpl_e$ == faux (il existe des définitions implicites)

Etape 1, $idexpl_1$ == faux
transformation d'un objet (des cadres "sans choix" comme celui ci-dessous ne sront plus mis en évidence)

choix de l'ident parmi : t
choix de : t

choix de la stratégie : <rdi>, <rds>, <sac>, <sem>, <rdt>, <rdp>, <rdl>

choix de : member(x,k̄)
x : [1..bs(d)] d: TAB
k : LISTE d'IDENTIFICATEURS

s tq ∀k̄ R(s,d,k̄)
s : TAB d : TAB k̄ : ENT
s == der u_i pour i de 1 jqa a_i
u_i tq R(u_i,l_i,d)
a_i == x_i = bs(d)

à faire : choix d'un invariant et d'une condition d'arrêt

décisions : u symbole de suite, e_u invariant, a symbole de suite booléenne, e_a condition d'arrêt, et élimination du "∀".
t == der u_i pour i de 1 jqa a_i
e_{u_i} : u_i,l_i,m_i tq seem(u_i,d)
et (l_i = 1 ou d[l_{i-1}] < 0)
et (m_i = bs(d) ou d[m_i + 1] < 0)
et ∀j [l_i..m_i] d[j] < 0 ⇒ som(tr(d,l_i,m_i)) = 0
et som(u_i) ⩾ som(tr(d,l_i,m_i))
e_{a_i} : a_i == m_i = bs(d)

Résultat de la transformation

action : mise à jour du tableau précédent (e_u,e_a,"t", profils) et création d'un nouveau module u (un nouveau tableau).

Etape 2, $idexpl_2$ == faux

choix de l'identificateur parmi : u_i, l_i, k_i
choix de : u_i parce que c'est la suite de "l'affaiblissement"

choix de la stratégie : <rdr>, <rdi>, <rds>, <sac>, <seem>, <rdt>, <rdp>, <rdl>
choix de : <rdr> (analyse récurrente) parce que définir u_i à l'aide de u_{i-1}

à faire : trouver une relation de récurrence : u_i == f (u_{i-1})

décisions : * comparaison de e_{u_i} et e_{u_i} $[i \leftarrow i-1]$

* nouvelle définition entre e_{u_i} et introduction de l'intermédiaire n_i

e_{u_i} : u_i tq (seem(u_{i-1}, d)
et som(u_{i-1}) $\geqslant$ som(tr(d,l_{i-1},m_{i-1})))
$\Rightarrow$ (seem(u_i,d) et som(u_i) $\geqslant n_i$)

et (($l_i = l_{i-1}$ et $m_i = m_{i-1}$) $\Rightarrow$ seem(u_i,d)
et som(u_{i-1}) $\geqslant n_i$)

avec

e_{l_i,m_i} : l_i, m_i tq $m_{i-1} < l_i \leqslant m_i$ ou $l_i = 1$
et $\forall k \; l_i \leqslant k \leqslant m_i \Rightarrow (d[m_i+1] < 0$ ou $m_i = bs(d))$

e_{n_i} : n_i tq $\forall j \; [l_i..m_i] \; d[j] < 0 \Rightarrow$ som(tr(d,l_i,m_i)) = 0
et n_i = som(tr(d,l_i,m_i))

e_{u_o} : e_{u_i} $[i \leftarrow 0]$

* application d'une analyse par cas sur le nouveau e_{u_i}
cas som(u_{i-1}) $\geqslant n_i$ on choisit $u_i = u_{i-1}$
cas som(u_{i-1}) $< n_i$ on choisit u_i = tr(d,l_i,m_i)
u_i == si som(u_{i-1}) $\geqslant n_i$ alors u_{i-1}
sinon tr(d,l_i,m_i)

(Nous ne donnerons plus le "résultat de la transformation")

Etape 3, $idexpl_3$ == faux

choix de l'identificateur parmi : <l_i,m_i>, n_i, u_o
choix de : u_o parce que suite de la définition de u_i

choix de la stratégie : <rds>, <rdt>, <sei>, <rdi>, <rdp>, <rdl>
choix de : <sei> parce que variable récurrente à initialiser

à faire : trouver une valeur c telle que e_{u_o} $[u_o \leftarrow c]$

décisions : calcul de c en considérant "som(u_o) $\geqslant$ som(tr(d,l_o,m_o))
et vérification

e_{u_o} : u_o tq seem(u_o,d)
et (l_o = 1 ou d[l_o-1] < 0)
et (m_o = bs(d) ou d[m_o+1] < 0
et $\forall j \; [l_o..m_o]$ d[j] < 0 $\Rightarrow$ som(tr(d,l_o,m_o)) = 0
et som(u_o) $\geqslant$ som(tr(d,l_o,m_o))

. en considérant

som(u_o) $\geqslant$ som(tr(d,l_o,m_o)) = vrai on choisit u_o = tvide
som(tvide) = 0 et som(tr(d,l_o,m_o)) = 0 $\Rightarrow$ tr(d,l_o,m_o) et m_o = tvide
on a bien = e_{u_o} $[u_o \leftarrow$ tvide$]$

Etape 4, $idexpl_4$ == faux

transformation d'un objet

choix de l'identificateur parmi : <l_i,m_i>, n_i
choix de : <l_i,m_i> parce que au hasard

choix de la stratégie : <rdi>, <rdt>, <rds>, <rdp>, <rdl>, <sac>, <sem>

choix de : <rdi> parce que recherche associative

$P_{i-1} < q_i \leqslant P_i$ et $\forall k\ (p_{i-1} < k < q_i) \Rightarrow P(d[k])$
et $\forall k\ q_i \leqslant k \leqslant P_i \Rightarrow P'(d[k])$

P_i == prem k depuis $q_{i-1}+1$ dans d tel que $P(d[k])$
q_i == prem k depuis P_i dans d tel que $P'(d[k])$
q_o == 0

à faire : instantiation de la stratégie <rdi> ci-dessus.

décisions :

e_{l_i,m_i} l_i,m_i tq $m_{i-1} < l_i \leqslant m_i$ ou $l_i = 1$
et $\forall k\ l_i \leqslant k \leqslant m_i \Rightarrow d[m_i+1] < 0$ ou
$m_i + 1 = bs(d)$

on obtient :

l_i == prem k depuis $m_{i-1}+1$ dans d tel que $d[k] > 0$
m_i == prem k depuis l_i dans d tel que $d[k+1] < 0$
m_i == 0

Etape 5, $idexpl_5$ == faux

Transformation d'un objet : choix de n_i

choix de la stratégie : <rdi>, <rds>, <rdt>, <rdp>, <rdl>, <sac>, <sem>
choix de : <sac> parce que n_i dépend d'une condition sur un domaine

à faire : mettre en évidence les pré-conditions dont dépend n_i

décisions : analyse par les résultats et introduction de "moins"

pour $n_i = 0$ cas $moins_i$ = vrai
pour n_i = som(tr(d,l_i,m_i)) cas non $moins_i$

avec

e_{moins_i} : $moins_i$ tq $\forall j\ [l_i..m_i]\ d[j] < 0 \Leftrightarrow moins_i$
moins : SUITE-DE-BOOL

. définition de la conditionnelle

n_i == si $moins_i$ alors 0 sinon som(tr(d,l_i,m_i))

Etape 6, $idexpl_6$ == faux

Transformation d'un objet : choix de $moins_i$

choix de la stratégie : <rdi>, <sac>, <rds>, <sem>, <rdp>, <rdt>, <rdl>

Choix de : <rdi> : r tq $\forall k\ [x..y]\ R(k,t)$
r : $T

---

r = der u_i pour i de x jqa f_i
u_i tq $x_i \leqslant i$ et $R(i,u_i)$
f_i == $i = m_i$

à faire : choix d'un invariant et d'une condition d'arrêt

décisions : V symbole de suite, e_V, invariant, stop symbole de suite booléenne et e_{stop} condition d'arrêt

e_{V_k} : V_k tq $l_i \leqslant k$ et $d[k] < 0 \Leftrightarrow V_k$
e_{stop_k} : $stop_k$ == $k = m_i$
$moins_i$ == der V_k pour 1 de l_i jqa $stop_k$

Etape 7, $idexpl_7$ == faux

Transformation d'un objet :

choix de l'identificateur parmi : V_k
choix de : V_k

choix de la stratégie : <rdr>, <rdi>, <sac>, <rdt>, <rds>, <sem>, <rdp>, <rdl>, <sem>

choix de : <rdr> : u_k tq $R(k) \Leftrightarrow u_k$
u : SUITE-DE-BOOL k : ENT

---

u_k == $R(k)$ et u_{k-1}
u_o == vrai

parce que V_k est un invariant.

à faire : instantiation de <rdr> ci-dessus

décisions : V_k == d[k] < 0 et V_{k-1}
V_o == vrai

Etape 10, $idexpl_{10}$ == vrai

<Fin explicitation>

c) Résultat de l'explicitation

Enoncé formel	Profil	Enoncé explicite
e_{u_i} e_t	t : TAB u : SUITE-DE-TAB a : SUITE-DE-BOOL	résultat t t == der u_i pour i de 1 jqa a_i donnée d
module u		
nouvelle définition de e_{u_i} e_{l_i}, e_{m_i} e_{n_i} e_{moins_i} e_{V_k}, e_{V_o}	n : SUITE-DE-ENTIER moins : SUITE-DE-BOOL V : SUITE-DE-BOOL Stop : SUITE-DE-BOOL k : ENT	u_i == som(u_{i-1}) ⩾ n_i alors u_{i-1} sinon tr(d,l_i,m_i) u_o == tvide l_i == prem k depuis m_{i-1}+1 dans d tel que d[k] > 0 m_i == prem k depuis l_i dans d tel que d[k + 1] < 0 ou k = bs(d) m_o == 0 a_i == m_i = bs(d) n_i == si $moins_i$ alors 0 sinon som(tr(d,l_i,m_i)) $moins_i$ == der V_k pour k de l_i jqa $stop_k$
module V		
		V_k == d[k] < 0 et V_{k-1} V_o == vrai $stop_k$ == k = m_i

Exemple 2

a) Spécification du problème

Une spécification d'un problème de tri est donnée par la figure III.4.

Lexique	Profil	Enoncé formel e_t
t est le tableau résultat d est le tableau donné equ indique l'équivalence entre 2 tableaux (à une permutation près) croiss indique qu'un tableau est trié	t : TAB d : TAB	résultat t t tq equ(t,d) et croiss(t) donnée d

Figure III.4.

b) Explicitation du problème

Initialisation

but : expliciter l'énoncé formel de la figure III.4.
action : création du tableau initial suivant :

Enoncé formel	Profil	Enoncé explicite
résultat t t tq equ(t,d) et croiss(t) donnée d	t : TAB d : TAB	résultat t .... donnée d

tableau III.0

$idexpl_0$ == faux

Comme pour l'exemple précédent les cadres "sans choix" et "résultats de la transformation" ne sont pas mentionnés.

Etape 1, $idexpl_1$ == faux

Transformation d'un objet : choix de t

choix de la stratégie : <sac>, <rdp>, <rdl>, <rds>, <rdt>, <sem>, <rdi>

choix de : <rdi> : t tq R(t,d) parce que t dépend de d

t : TAB d : TAB

---

t == der u_i pour i de 1 jqa a_i

u_i tq R(u_i,$d1_i$)

a_i == i = bs(d)

à faire : choix d'un invariant e_u et d'une condition d'arrêt e_a
(instantiation de la stratégie ci-dessus)

décisions : u symbole de suite, a symbole de suite booléenne.

t == der u_i pour i de 1 jqa a_i

e_{u_i} : u_i tq equ(u_i,$d1_i$) et croiss(u_i)

e_{a_i} : d_i == (i = bs(d))

Etape 2, $idexpl_2$ == faux

Transformation d'un objet : choix de u_i

choix de la stratégie : <sac>, <sea>, <rdl>, <rdp>, <rdr>, <rds>, <sem>

choix de : <rdr> (récurrence) parce que e_{u_i} est un invariant

u_i tq R(u_i, $d1_i$

---

u_i == insert(u_{i-1}, d[i], l_i)

l_i tq R(u_{i-1}, $d1_{i-1}$) ⇒

R(tr(u_{i-1},1,l_i-1) ∿ di_i ∿ tr(u_{i-1},l_i,i-1),$d1_i$)

u_o tq R(u_o, $d1_o$)

l : SUITE-DE-ENT u : SUITE-DE-TAB

(* "∿" est la notation infixe de "concat" (concaténation de 2 tableaux)

à faire : la relation de récurrence choisie est "insert",
instantiation de stratégies ci-dessus.

décisions : u_i == insert(u_{i-1},d[i],l_i)

l_i tq equ(u_{i-1},$d1_{i-1}$) et croiss(u_{i-1}) ⇒

equ(tr(u_{i-1},1,l_i-1) ∿ di_i ∿ tr(u_{i-1},l_i,i-1),$d1_i$)

et croiss(tr(u_{i-1},1,l_i-1) ∿ di_i ∿ tr(u_{i-1},l_i,i-1))

u_o tq equ(u_o,$d1_o$) et croiss(u_o)

Etape 3, $idexpl_3$ == faux

Transformation d'un objet

choix de l'identificateur parmi : u_o,l_i

choix de : u_o parce que suite de la définition de u_i

choix de la stratégie : <sei>, <rdt>, <sea>, <sac>, <rdp>, <rdl>, <sem>

choix de : <sei> parce que u_o est la valeur initiale de u_i

à faire : trouver une valeur particulière v telle que e_{u_o} [u_o ← c]

décisions : - calcul de "c" en choisissant : equ(u_o,$d1_o$)

- application des V-règles :

i > j t tq t = di_j / t == tvide equ(tvide, tvide) / vrai

- valeur de u_o déduite : u_o == tvide

- vérification pour u_o == tvide on a bien croiss(tvide = vrai)
(propriété)

Etape 4, $idexpl_4$ == faux

Transformation d'un objet : choix de l_i

choix de la stratégie : <sac>, <rdr>, <rdt>, <rdi>, <rdl>, <rdp>, <rds>, <sem>

choix de : <rdl> (logique)

$$\frac{P(i-1) \text{ et } Q(i-1) \Rightarrow Q(i) \text{ et } Q(i)}{P(i-1) \text{ et } Q(i-1) \Rightarrow P(i) \quad \text{et } P(i-1) \text{ et } Q(i-1) \Rightarrow Q(i)}$$

à faire : instantiation de la règle logique ci-dessus.

décisions :

$$l_i \text{ tq } equ(u_{i-1}, d1_{i-1}) \text{ et } croiss(u_{i-1}) \Rightarrow equ(tr(u_{i-1}, 1, l_i-1) \sim di_i \sim tr(u_{i-1}, l_i, i-1), n_i)$$

appelons ce constituant "F2"

$$\left[\begin{array}{l} \text{et } equ(u_{i-1}, d1_{i-1}) \text{ et } croiss(u_{i-1}) \Rightarrow \\ \quad croiss(tr(u_{i-1}, 1, l_{i-1}) \sim di_i \sim tr(u_{i-1}, l_i, i-1)) \end{array}\right.$$

Etape 5, $idexpl_5$ == faux

Transformation d'un objet : choix de la nouvelle définition de l_i

choix de la stratégie : <sac>, <rdr>, <rdt>, <rdi>, <rdl>, <rdp>, <rds>, <sem>

Choix de : <sem> (simplicifation de e_{l_i}) : application de la <rdl> :

$$r(t) \frac{t \text{ tq } R(t) \text{ et } R'(t)}{t \text{ tq } R'(t)}$$

à faire : simplification de e_{l_i} en prouvant

(F1) $\left[\begin{array}{l} equ(u_{i-1}, d1_{i-1}) \text{ et } croiss(u_{i-1}) \Rightarrow \\ \quad equ(tr(u_{i-1}, 1, l_i-1) \sim di_i \sim tr(u_{i-1}, l_i, i-1), d1_{i-1}, \sim di_i) \end{array}\right.$

(le constituant F1 est une instance de R)

décisions : analyse par cas :

Cas : $u_{i-1} \neq$ vide

(F1) se réécrit en (F'1) par application de :

a) $equ(t \sim d) :: equ(d \sim t)$ (":" notation textuelle de "$\rightarrowtail$")

b) $u1_i \sim ui+1_n :: u1_n$

(F'1) $equ(u_{i-1}, d1_{i-1}) \text{ et } croiss(u_{i-1}) \Rightarrow equ(tr(u_{i-1}, 1, i-1) \sim di_i, d1_{i-1} \sim di_i)$

(F'1) devient par :

a) $equ(t \sim d, t' \sim d) :: equ(t, t')$

b) $i \neq 1 \; (u_{i-1}, 1, i-1) :: u_{i-1}$

(F"1) $equ(u_{i-1}, d1_{i-1}) \text{ et } croiss(u_{i-1}) \Rightarrow equ(u_{i-1}, d1_{i-1})$

et par $P \text{ et } Q \Rightarrow P \text{ et } Q' :: Q \Rightarrow Q'$ on obtient :

$croiss(u_{i-1}) \Rightarrow$ vrai

et par : $P \Rightarrow$ vrai :: vrai on peut conclure alors que :

(F1) est vrai

Cas : u_{i-1} = tvide (c'est-à-dire i = 1)

$equ(tvide, d1_0) \text{ et } croiss(tvide) \Rightarrow equ(tvide \sim d1_1 \sim tvide, d1_0 \sim d1_1)$

vrai et vrai $\Rightarrow equ(d1_1, d1_1)$

$\Rightarrow$ vrai

Etape 6 , $idexpl_6$ == faux

Transformation d'un objet : nouvelle définition de l_i

choix de la stratégie : <sea>, <rdr>, <rdp>, <rdl>, <sac>, <rds>, <sem>
choix de : <sac> parce que forme $P(u_{i-1}) \Rightarrow P(u_i)$

à faire : recherche des différents cas : analyse par la donnée et des calculs associés

décisions : analyse par cas des tranches du tableau u_i

Cas 1 : $tr(u_{i-1},1,l_i-1) \neq tvide$
et $tr(u_{i-1},l_i,i-1) \neq tvide$

- application de la <rdp> de croiss :

$$\frac{croiss(tr(t,1,l-1) \sim di_i \sim tr(t,l,n))}{croiss(tr(t,1,l-1) \text{ et } t[l-1] \leqslant d[i] \text{ et } d[i] \leqslant t[l] \text{ et } croiss(tr(t,l,n))}$$

sur l_i, on obtient :

l_i tq $equ(u_{i-1},d1_{i-1})$ et $croiss(tr(u_{i-1},1,l_i-1)$ et $u_{i-1}[l_i-1] \leqslant u_{i-1}[l_i]$ et $croiss(tr(u_{i-1})\, l_i, i-1)) \Rightarrow$
$croiss(tr(u_{i-1},1,l_i-1)$ et $u_{i-1}[l_i-1] \leqslant d[i]$
et $d[i] \leqslant u_{i-1}[l_i]$ et $croiss(tr(u_{i-1},l_i,i-1))$

- application de la <rdl> :

P et Q $\Rightarrow$ P et Q' :: Q $\Rightarrow$ Q' à ce dernier l_i

on obtient :

l_i tq $equ(u_{i-1},d1_{i-1})$ et $u_{i-1}[l_i-1] \leqslant u_{i-1}[l_i] \Rightarrow$
$u_{i-1}[l_i-1] \leqslant d[i]$ et $d[i] \leqslant u_{i-1}[l_i]$

- application de la propriété de récurrence (invariant au rang i - 1 est vrai)

on obtient :

l_i tq $u_{i-1}[l_i-1] \leqslant d[i]$ et $d[i] \leqslant u_{i-1}[l_i]$

- introduction d'un intermédiaire m et résultat de la solution algorithmique de ce dernier l_i : On a alors

l_i tq "cas 1" $\Rightarrow l_i = m_i$
avec m_i tq $u_{i-1}[m_i-1] \leqslant d[i]$ et $d[i] \leqslant u_{i-1}[m_i]$

Cas 2 : $tr(u_{i-1},1,l_i-1) = tvide$
et $tr(u_{i-1},l_i,i-1) \neq tvide$

on a :

l_i tq $equ(u_{i-1},d1_{i-1})$ et $croiss(u_{i-1}) \Rightarrow$
$croiss(tvide \sim di_i \sim tr(u_{i-1},l_i,i-1)$

- application de la <rdp> précédente on obtient :

l_i tq $equ(u_{i-1},d1_{i-1})$ et $croiss(tr(u_{i-1},1,l_i-1)$
et $croiss(tr(u_{i-1},l_i,i-1)) \Rightarrow$
$d[i] \leqslant u_{i-1}[l_i]$ et $croiss(tr(u_{i-1},l_i,i-1))$

- application de la <rdl> précédente :

l_i tq $equ(u_{i-1},d1_{i-1})$ et $croiss(tr(u_{i-1},1,l_i-1) \Rightarrow$
$d[i] \leqslant u_{i-1}[l_i]$

- application de la propriété de récurrence :

l_i tq $d[i] \leqslant u_{i-1}[l_i]$

- simplification des conditions du cas 2 :

$tr(u_{i-1},1,l_i-1) = tvide \Leftrightarrow l_i = 1$
$tr(u_{i-1},1,i-1) \neq tvide \Leftrightarrow i > 1$

- finalement on a :

l_i tq $i > 1$ et $d[i] \leqslant u_{i-1}[1] \Rightarrow l_i = 1$

Cas 3 : $tr(u_{i-1},1,l_{i-1}) \neq tvide$
et $tr(u_{i-1},l_i,i-1) = tvide$
$tr(u_{i-1},1,l_i-1) \neq tvide \Leftrightarrow l_i > 1$
$tr(u_{i-1},l_i,i-1) = tvide \Leftrightarrow l_i = i$ ou $(i = 1$ et $l_i = 1)$
ce qui se simplifie en : $l_i = i$ et $i \neq 1$.

par analogie au cas 2 on obtient :

l_i tq equ(u_{i-1},$d1_{i-1}$) ⇒ u_{i-1} [l_i-1] ⩽ d[i]

et pour l_i = i on a :

l_i tq u_{i-1} [i - 1] ⩽ d[i] ⇒ l_i = i

Cas 4 : tr(u_{i-1},1,l_i-1) = tvide
et tr(u_{i-1},l_i,i-1) = tvide

Cela revient à avoir :

l_i = 1 et l_i = i ⇒ i = 1
l_i tq equ(u_{i-1},$d1_{i-1}$) et croiss(tvide) ⇒ croiss(tvide ∿ di_i ∿ tvide)
l_i tq equ(u_o,$d1_o$) ⇒ croiss($d1_1$)
l_i tq vrai

la définition de la structure conditionnelle est :

l_i tq d[i] ⩽ u_{i-1} [1] et i > 2 ⇒ l_i = 1
ou i = 1 ⇒ l_i = 1
ou u_{i-1} [i - 1] ⩽ d[i] ⇒ l_i = i
ou l_i = m_i

la conditionnelle s'écrit alors :

l_i == si i = 1 alors 1
sinon
si i > 1 et d[i] ⩽ u_{i-1} [1] alors 1
sinon
si u_{i-1} [i-1] < d[i] et i ≠ 1 alors i
sinon m_i

Etape 7, $idexpl_7$ == faux

Transformation d'un objet : choix de m_i

choix de la stratégie : <sac>, <rdr>, <rds>, <rdt>, <rdi>, <rdp>, <rdl>, <sem>

choix de : <rdi> parce que recherche associative

l ⩽ r ⩽ bs(t) et P(t[r])

r == prem k depuis l dans t tel que P(t[r])

à faire : instantiation de la <rdi> choisie

décisions : m_i tq u_{i-1} [m_i-1] ⩽ d[i] et d[i] ⩽ u_{i-1} [m_i]

- définition de l'intervalle de m_i :
  pour m_i = 1 u_{i-1} [0] n'est pas défini
  choix de m_i [2.. i-1]
- m_i == prem k depuis 2 dans u_{i-1} tel que
  u_{i-1} [k - 1] ⩽ d[i] et
  d[i] ⩽ u_{i-1} [k]

Etape 8, $idexpl_8$ == vrai

<Fin explicitation>

Résultat de l'explicitation

Enoncé formel	Profil	Enonce explicite
t tq equ(t,d) et croiss(t) u_i tq equ(u_i,$d1_i$) et croiss(u_i) a_i == (i = bs(d)	t : TAB d : TAB u : SUITE DE TAB a : SUITE DE BOOL	résultat t t == der u_i pour i de 1 jqa a_i donnée d
module u		
u_o tq equ(u_o,$d1_o$) et croiss(u_o) l_i tq comme donné par "F2" etape 5 m_i tq comme donné à l'étape 7 (Cas 1)	l : SUITE DE ENT m : SUITE DE ENT i : ENT k : ENT	u_i == insert(u_{i-1},d[i],l_i) l_i == si i = 1 alors 1 sinon si i > 1 et d[i] ⩽ u_{i-1}[1] alors 1 sinon si u_{i-1}[i-1] ⩽ d[i] et i ≠ 1 alors i sinon m_i m_i == prem k depuis 2 dans u_{i-1} tel que u_{i-1}[k-1] ⩽ d[i] et d[i] ⩽ u_{i-1}[k] a_i == i = bs(d) u_o == tvide

Remarque : le lecteur constatera par lui-même que ce résultat de l'explicitation est "équivalent" à celui donné par la figure I.3. du chapitre I.

III.4. Conclusion

Nous venons de voir d'une part, que le problème de la découverte de stratégies est fondamental et d'autre part, un méta-algorithme dont la mise en oeuvre est illustrée par deux exemples.

L'aspect apprentissage est important pour la découverte de stratégies. Ainsi la réalisation d'expériences sur la construction de programmes vise deux objectifs :

1. l'identification de mécanismes de constructions.
2. la construction de mécanismes d'apprentissage.

Le méta-algorithme a permis de mettre en relief le calcul du passage de E_0 à M :

$$E_0 \xrightarrow{s^*} M$$

Les exemples précédents montrent que le chemin est encore long pour aboutir à une véritable automatisation du processus de l'explicitation.

CHAPITRE IV

IV. LE SYSTEME D'EXPLICITATION

IV.1. Présentation générale

L'objectif du système d'explicitation, SIE, est de construire un énoncé itératif correct vis-à-vis d'un énoncé implicite initial (cf. schéma I.2. chapitre I).
Pour atteindre cet objectif nous imposerons (cf. § I.1.) au système de revêtir les aspects suivants :
- interactivité,
- extensibilité
- assistance à l'explicitation.

Tous ces aspects ont été développés dans les chapitres précédents. Rappelons brièvement ce qui a été dit ou montré à leur sujet.

a) "l'interactivité"

L'interaction apparaît à tous les niveaux de l'explicitation (cf. les exemples § III.2., problème du choix de la stratégie II.6., l'introduction de stratégies § III.1.).

b) "l'extensibilité"

L'extensibilité exprime l'idée de construction de stratégies durant ou en dehors d'une explicitation. (cf. le problème de définition de stratégies (de règles) § II.3., § II.5., et celui de leur découverte § III.1.).

c) "l'assistance"

L'assistance à l'explicitation intervient principalement au moment de l'utilisation des stratégies. Elle est marquée à la fois par le problème des calculs à faire (cf. la notion de transformation § II.4., l'exemple technique § II.7., et aussi § III.1.2. précisant les algorithmes de certaines SEG) et celui de l'aide à la décision (§ II.6.).

Les conséquences que l'on tire de ce qui est développé, à propos de ces aspects, sont :

1. l'importance de la notion de transformation (cf. définition de la transformation § II.4.1.) et d'enchaînement de transformations que nous avons précisée par un méta-algorithme d'explicitation (§ III.2.). Ce dernier montre la manière dont les stratégies sont utilisées.
2. l'importance de la notion d'explicitation paramétrée qui apparaît comme un moyen de permettre l'extensibilité.

Ces deux derniers points nous ont conduits à structurer le système en trois modes distincts comme indiqué dans le schéma IV.1. ci-dessous.

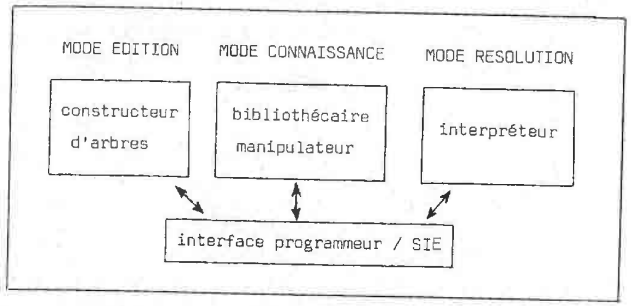


Schéma IV.1. : structure du système en modes.

Ce schéma montre que les trois modes sont gérés par un module d'interface assurant l'interaction entre le programmeur et le système.
La structure du système présentée dans le schéma précédent intègre l'ensemble des connaissances du système et l'ensemble de ses modules fonctionnels.
Ces deux ensembles sont illustrés par la figure IV.1. ci-après.

Les connaissances sont définies par le programmeur. Il s'agit des informations utilisées et mises à jour par le système. Les connaissances étudiées au § IV.2. sont :
- les stratégies d'explicitation,
- l'arbre des choix (qui est construit par le système).

Les types abstraits font l'objet d'une autre étude du groupe "programmation" du CRIN [LEV-84, FIN-79]. On ne s'intéressera alors pas à ce type de connaissance.
Chaque mode est caractérisé par certaines grandes fonctions que doit réaliser le système : il s'agit des modules fonctionnels mentionnés dans le schéma IV.1. et la figure IV.1.

Et c'est ainsi que les modules fonctionnels des modes "connaissance" et "résolution" assurent respectivement la paramétrisation de l'explicitation et l'enchaînement des transformations.
Tous les modules fonctionnels du système sont étudiés au § IV.3.

En outre nous montrons des extensions possibles de ces modules au § IV.4.

Par ailleurs l'annexe E présente un exemple d'une session de travail.

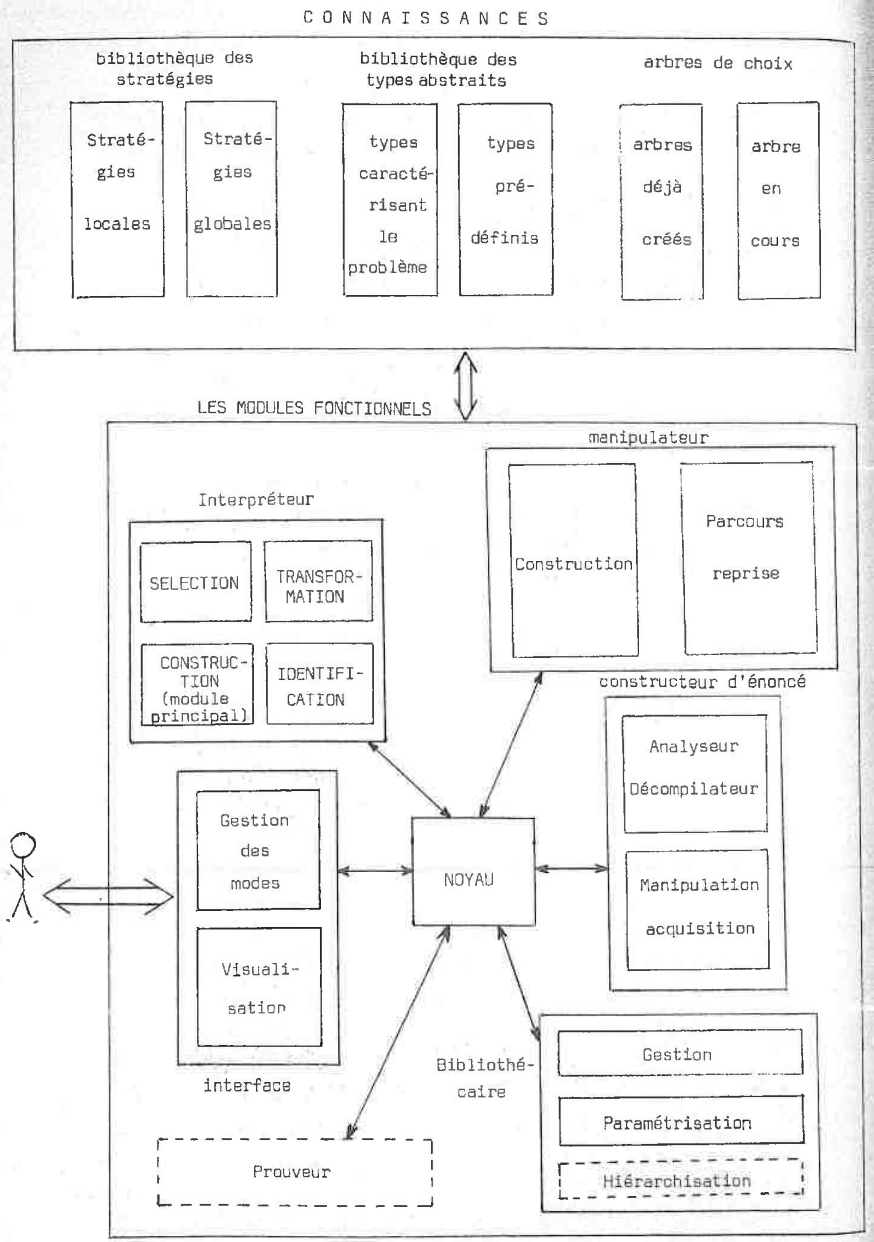


Figure IV.1.

IV.2. Les connaissances du système

Comme nous l'avons déjà dit les connaissances du système sont constituées :

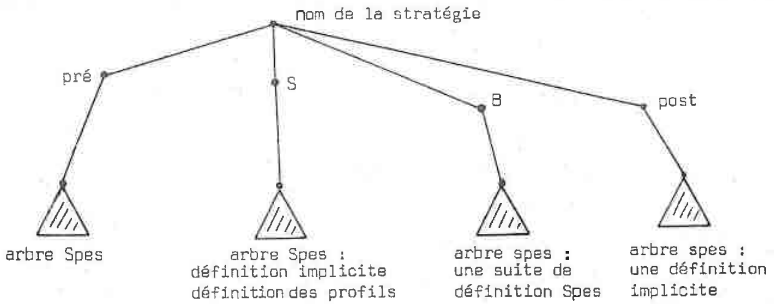
- des stratégies d'explicitations rangées dans la bibliothèque de stratégies,
- des arbres des choix rangés dans la bibliothèque des arbres.

La bibliothèque de stratégies est composée du catalogue (bibliothèque) des stratégies d'explicitations locales, et de celui des stratégies d'explicitations globales.
Ces deux catalogues sont décrits séparément (respectivement au § IV.2.1., et au § IV.2.2.).
La bibliothèque des arbres est présentée au § IV.2.3.

IV.2.1. Le catalogue des stratégies d'explicitations locales

IV.2.1.1. Les stratégies locales

La plupart des stratégies d'explicitations locales sont construites par le programmeur. Cependant nous avons vu que le système pourrait construire certaines SEL notamment celles appartenant à la classe V-règles (le schéma résultat de ces règles "valeurs" est donné par une constante). Une SEL construite est un arbre ayant la forme suivante :

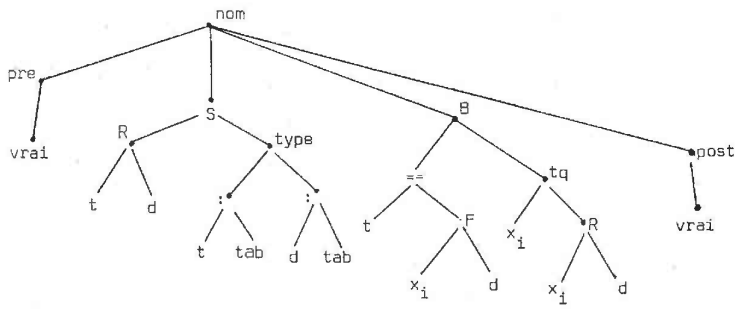


Par exemple l'arbre correspondant à la SEL

R(t,d)
t : tab d : tab

$t == F(x_i,d)$
x_i <u>tq</u> $R(x_i,d)$
d : tab x : suite de tab

est :



Un arbre d'une SEL est représenté en Lisp comme une S-expression.
Par exemple l'arbre précédent est donné par l'expression :

((nom(pre S B post)) (pre(vrai)) (S(R type)) (R(t d)) (type (: :))
(: (t tab)) (: (d tab)) (B (== tq)) .... (post(vrai))).

IV.2.1.2. <u>La bibliothèque : les fonctions attendues</u>

Le catalogue des SEL se présente actuellement comme une table d'arbres de SEL. C'est ce qu'illustre le schéma IV.2. suivant :

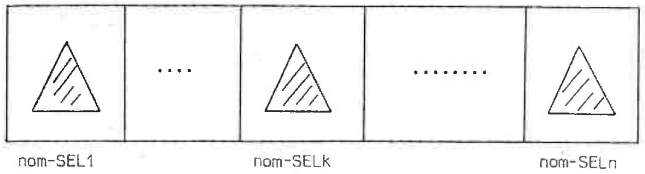


Schéma IV.2. : structure de la bibliothèque de SEL.

Cette table est également représentée par une S-expression :

((biblio((nom-SEL1 nom-SEL .... nom-seln))
(nom-SEL1 (S-expression de cette SEL))
(nom-SEL (S-expression de cette SEL))
⋮
(nom-selk (S-expression de cette SEL)).

Ce catalogue doit permettre deux types d'opérations :
- des opérations sur les SEL,
- des opérations sur l'ensemble des SEL (le catalogue lui-même).

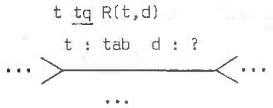
Les opérations sur les SEL et le catalogue sont identiques. Il s'agit des fonctions classiques d'accès (à une SEL du catalogue), de suppression (d'une SEL du catalogue ou d'un sous arbre d'une SEL), de modification (d'une SEL), et d'ajout.
Ces opérations appliquées à la structure du catalogue donnée par le schéma IV.2. ci-dessus sont alors traduites en Lisp :
- l'accès à une SEL s'exprime directement par l'opération "assoc" :
  (assoc nom de la SEL bibliothèque),
- la suppression d'une SEL du catalogue n'est pas obtenue directement :
  (append (reste la liste des SEL la liste d'une SEL)
  (cdr (member l-SEL SEL)))
  (reste retourne la liste constituée de toutes les listes SEL placées avant une certaine liste SEL),

- l'ajout est traduit par "cons",
- la modification par des applications successives de "subst".

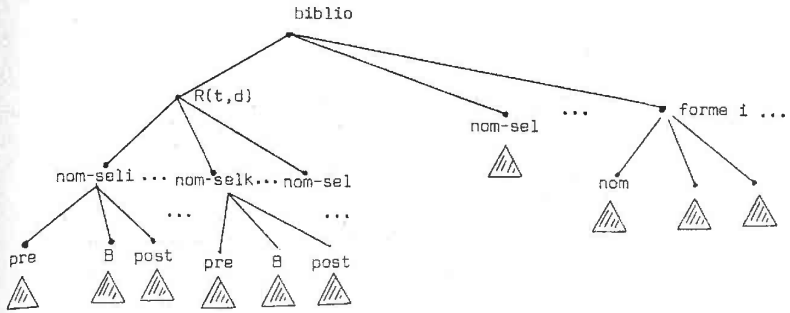
Cette traduction a été donnée à titre indicatif. Elle n'a pas d'incidence sur la suite du travail. L'important est d'étudier le temps de sélection d'un sous ensemble de SEL. On veut aussi que le catalogue permette une sélection rapide de SEL.
La structure de la bibliothèque (schéma IV.2.) fait apparaître qu'une SEL est rangée de manière séquentielle. Il est clair que ce rangement ne facilite pas l'exploitation de la bibliothèque lors d'une transformation. C'est-à-dire que le temps de sélection d'un sous ensemble de stratégies applicables à un énoncé est inefficace. Il est donc nécessaire de revoir une autre organisation du catalogue des SEL permettant une sélection plus rapide. Pour cela on peut penser à des organisations du genre proposé dans [LEI-83] où l'on peut voir que la base de connaissances du système expert ELI est conceptuellement représentée par un modèle tri-dimensionnel. Ce modèle exprime une structuration hiérarchique des règles de productions.
Discutons cette idée de hiérarchisation appliquée à la forme de nos stratégies locales.

Une hiérachisation possible des stratégies locales peut être faite par une étude sur la forme du membre gauche des SEL (c'est-à-dire la forme de S dans {pre} S::B {post}). Par exemple on peut regrouper les SEL ayant même partie gauche. Illustrons cela par ce qui suit :

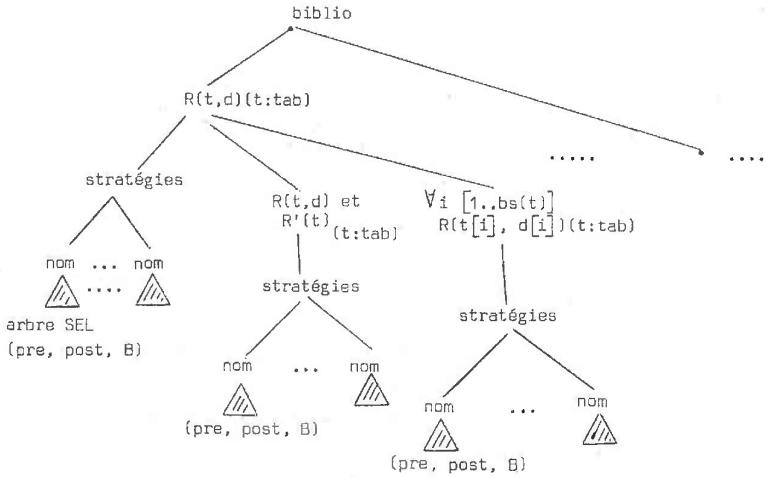
les stratégies ayant comme partie gauche R(t,d),



sont regroupées de la façon suivante :



Ce dernier arbre montre que la sélection de stratégies applicables à un énoncé de la forme R(t,d) est immédiate.
On peut aussi hiérarchiser, en plus, les parties gauches en disant qu'une partie gauche est moins générale qu'une autre. Par exemple en ajoutant à l'arbre précédent d'autres regroupements de parties de gauches, on obtient alors l'arbre suivant



Avec ce dernier arbre on peut dire que si un énoncé effectif n'est pas une instance d'une partie gauche d'une SEL (R(t,d) alors il n'est pas une instance de toutes les parties gauches de SEL de niveau inférieur ("R(t,d) et R'(t)", "∀i [1..bs(t)] R(t[i], d[i])").

On vient de voir par des illustrations ce que l'on entend par hiérarchiser la bibliothèque des SEL. Une telle hiérarchisation nécessite une étude plus approfondie avant de passer à toute implémentation ; ce serait un prolongement possible de ce travail (la figure IV.1. précise cette extension par le module hiérarchisation dans un cadre en pointillé).

IV.2.2. Le catalogue des stratégies d'explicitations globales

IV.2.2.1. La mise en oeuvre des stratégies globales

Une SEG est un algorithme caractérisant une heuristique de transformation. Quatre SEG générales ont été définies au § III.1.2. Donnons pour chacune d'elles un algorithme précisant la manière dont elles sont mises en oeuvre. Tous les algorithmes sont exprimés dans un pseudo-Médée facilement compréhensible.

a) une mise en oeuvre de <sea>.

Dans le méta-algorithme d'explicitation (cf. figure III.1. chap. III) <sea> est à la base de l'explicitation. Elle est un guide pour l'explicitation.
Affaiblir un énoncé c'est "oublier" certaines conditions le caractérisant (cf. III.1.1. pour des exemples d'affaiblissement - induction sur une suite). Donnons un exemple :
si un énoncé e_r contient une variable libre entière i alors e est décomposable en :

$e' = e_r [i \leftarrow k]$ et $e'' = (k = i)$ (on reconnaît l'application de la condition d'existance d'un invariant - condition 1. § III.1.1.).

L'algorithme de <sea> commence par sélectionner toutes les règles introduisant une définition itérative et applicables à un énoncé. Le choix d'une de ces règles précise la forme de décomposition de cet énoncé. A partir de ce choix d'autres choix sont appliqués pour expliciter les intermédiaires issus de cette décomposition.

* Algorithme de <sea>

La description générale de cet algorithme a été donnée au § III.1.2.1.
Soit r le résultat de l'énoncé e_r à expliciter,
$\mathcal{S}$ l'ensemble des stratégies disponibles,
et s la stratégie choisie (elle est vue comme une fonction, on notera par s(e) le transformé de e par s).

```
<sea> (𝒮, e_r) Δ
    si e_r est explicite alors retourne : e_r
      sinon
    cas  r est une variable récurrente : s = choix entre
            (ou indicée)                 {<rdr>, <rds>, <sac>, <rdp>,
                                          <rdc>, <rdl>}
         r est une variable d'initialisation (r_0) :
                                       s = choix entre
                                          {<sei>, <rds>, <rdr>, <rdp>,
                                           <rdl>}
      sinon s = choix entre {<rdc>, <sac>, <rdp>, <rdl>, <rds>}
  retourne : s(e_r)
```

L'illustration de <sea> donnée au § III.1.2.1. est un exemple d'application de cet algorithme. Dans les exemples traités au § III.3. le lecteur reconnaîtra d'autres applications de l'algorithme de <sea>.

b) Une mise en oeuvre de <sei>.

L'algorithme présenté ci-dessous est une forme particulière de la description générale de <sei> donnée au § III.1.2.2.

- Présentation de <sei> :

  donnée : e_r un énoncé de résultat r de la forme

  $$r_n \ \underline{tq}\ R(r_n)$$

  où toutes les opérations, de symboles différents de "=" et des connecteurs logiques figurant dans e_r ont comme arguments une liste de variables simples (aucun symbole d'opérations).

  résultat : l'équation $r_n == \alpha$

  où α est une constante (la valeur de la variable r_n initialisée).

Dans certains cas l'algorithme n'est pas capable de produire systématiquement l'équation $r_n == \alpha$, alors le programmeur la définit et en assure la correction.

* Algorithme de <sei>

```
<sei> (e_r) Δ  sortir == faux
   tant que non sortir faire
    1. choix d'un constituant de e_r, O(r_n,x̄) (O est un prédicat)
    2. Si il existe une v-règle de la forme
            vr : {pre} O(r',x̄') :: vrai {post} et telle que les
            pré-post soit vérifiées
                  alors si <sei-α> (x̄,x̄',E,vr)
                              alors r_n == r'  sortir == vrai
                        sinon proposition :
                  1. une v-règle {pre} O(r',x̄') :: vrai {post}
                  2. retourne r_n == r' ; sortir == vrai
          sinon choix entre
                  1. proposition : sortir == vrai
                  2. sortir == vrai
                  3. sortir == faux
   ftq
```


Commentaire

- <sei-α> est un prédicat qui indique que l'on peut associer aux valeurs de $\bar{x}'$ les variables correspondantes dans $\bar{x}$.
- $\bar{x}$ (respectivement $\bar{x}'$) est une liste $(x_1, x_2, \ldots x_i)$ où tous les x_i sont soit des variables, soit des constantes.
- E est l'énoncé complet ; e_r est une définition de E.
- Dans l'algorithme ci-dessous de <sei-α> x_1 (x'_1) désigne le premier élément de $\bar{x}$ (resp. $\bar{x}'$) et $\bar{x}_d$ (resp. $\bar{x}'_d$) désigne la liste $\bar{x}$ (resp. $\bar{x}'$) privée de son premier élément.

Donnons l'algorithme de <sei-α>.

```
<sei-α>  (x̄,x̄',E,vr) Δ
   Cas 1. x̄ est vide : retourne vrai
       2. x_1 est un identificateur
          et x'_1 est une constante : si il existe x_1 == x'_1 dans E
                                      alors <sei-α> (x̄_d,x̄'_d,E,vr)
                                    sinon
                                    si il existe x_1 == f(ȳ) dans E
                                      et une règle f(ȳ) :: x'_1
                                      telle que les pre-post soient
                                      vérifiées
                                      alors <sei-α> (x̄_d,x̄'_d,E,vr)
                                    sinon retourne faux
       3. x_1 est une constante
          et x'_1 est un identificateur : si pour x'_1 = x_1 les pre-post
                                      de vr sont vérifiées alors
                                          <sei-α> (x̄_d,x̄'_d,E,vr)
                                        sinon retourne faux
                 sinon <sei-α> (x̄_d,x̄'_d,E,vr)
```

Une illustration de l'algorithme de <sei> a été donnée au § III.1.2.2. Donnons un autre exemple en considérant la définition de u_o de l'exemple 2 § II.2.1. figure II.2. :

u_o <u>tq</u> ssuite $(u_o, d1_o)$ <u>et</u> $u_o(o) > 0$

Pour être conforme à la présentation de <sei>, ci-dessus, il est nécessaire de réécrire la définition de u_o en :

e_{u_o} : u_o <u>tq</u> ssuite (u_o, e) <u>et</u> $u_o(0) > 0$
<u>et</u> e == tr(d,1,0)

Déroulement de l'algorithme en considérant cette dernière définition :

1. choix d'un constituant de e_{u_o} : ssuite(u_o,e),
2. il existe la v-règle : {vrai} ssuite (tvide,tvide) : vrai {vrai}
3. appel : <sei-α> ((e), (tvide),E,vr) :

   on se retourve dans le cas 2, alors :

   l'équation x_1 == tr(d,1,0) est considérée.

Pour cette équation on trouve la règle :

{i > j} tr(d,i,j) :: tvide {vrai}

la partie droite de cette règle est égale à x'_1 alors on réappelle <sei-α>(()),()),E,vr) qui produit la valeur <u>vrai</u>. Par conséquent <sei> fournit comme résultat : u_o == tvide.

c) <u>une mise en oeuvre de <sac></u>.

L'heuristique proposée pour cette stratégie est aussi exprimée par un algorithme particularisant la description générale de cette <sac> donnée au § III.1.2.3. Cet algorithme est primaire. Il ne calcule pas une analyse par cas. Il aide seulement d'une certaine façon le programmeur à établir une telle analyse.

Cet algorithme assiste le programmeur dans la construction d'un schéma conditionnel (§ III.1.2.3.). A partir de ce schéma une structure conditionnelle est dérivée.

Un schéma conditionnel est un énoncé de la forme :

r <u>tq</u> $P_1 \Rightarrow Q_1(r)$
<u>et</u> $P_2 \Rightarrow Q_2(r)$
.....
<u>et</u> $P_i \Rightarrow Q_i(r)$

où

- r est le résultat de l'énoncé e_r sur lequel s'applique <sac>,
- chaque P_k (k = 1,..,i) exprime une condition sous forme d'une conjonction de prédicats n-aire tels que tous ces prédicats ne soient pas des opérations sur le type TABLEAU (on a choisi cette stratégie <sac> dans cette classe de problèmes),
- chaque Q_k (k = 1,...,i) est une propriété caractérisant une forme particulière du résultat r.

Le passage d'un schéma conditionnel à la structure conditionnelle correspondante repose sur l'application successive de la SEL suivante :

$P_1 \Rightarrow Q_1(r)$ <u>et</u> non $P_1 \Rightarrow Q_2(r)$
———————————————— sur le schéma
r == <u>si</u> P_1 <u>alors</u> r' <u>sinon</u> r"
r' <u>tq</u> $Q_1(r')$
r" <u>tq</u> $Q_2(r")$

De la sorte la structure conditionnelle construite est :

r == <u>si</u> P_1 <u>alors</u> r_1 avec r_1 <u>tq</u> $Q_1(r_1)$
<u>sinon</u>
<u>si</u> P_2 <u>alors</u> r_2 avec r_2 <u>tq</u> $Q_1(r_2)$
<u>sinon</u>
.....
<u>sinon</u> r_i avec r_i <u>tq</u> $Q_i(r_i)$

(c'est-à-dire <u>si</u> $P_1 = P_2 = \ldots P_{i-1}$ = faux <u>alors</u> P_i = vrai)

- Présentation de <sac> :

<u>donnée</u> : e_r un énoncé du résultat r

<u>résultat</u> : r == <u>si</u> P_1 <u>alors</u> r_1 <u>sinon</u>

.....

<u>si</u> P_{i-1} <u>alors</u> r_{i-1} <u>sinon</u> r_i

avec $1 \leqslant k \leqslant i$ r_k <u>tq</u> $Q_k(r_k)$

* <u>Algorithme de <sac></u>

<u>Lexique</u> :

schema désigne l'ensemble des formules conditionnelles,

P_k est la condition définie précédemment,

applic est une fonction permettant au programmeur de transformer une définition implicite par l'application de certaines SEL,

struc-cond est la fonction assurant le passage du schéma à la structure conditionnelle.

Algorithme :

```
<sac> (e_r) Δ schema == ϕ , sortir == faux
  tant que non sortir faire
    1. choix entre
         - sortir == vrai
         - transformer e_r : e'_r == applic(e_r)
         - proposition d'une condition P_k
    2. choix entre
         - transformer P_k : cond == applic(P_k)
         - cond == P_k
         - sortir == vrai
    3. choix entre
         - proposition de Q_k(r)
         - sortir == vrai
```



```
    4. choix entre
         - sortir == vrai
         - transformer Q_k(r) : Q'_k(r) == applic(Q_k(r))
         - maintenir Q_k(r) : Q'_k(r) = Q_n(r)
    5. schéma == schéma ∪ {P_k ⇒ Q'_k(r))
  ftq
    si schéma = ϕ ou schéma incomplet alors retourne ϕ
                                   sinon retourne (struc-cond
                                                   schéma))
```

<u>Commentaire</u>

Le programmeur peut interrompre le déroulement de l'algorithme grâce à "sortir".

- Définition des fonctions struc-cond et applic.

La fonction struc-conc est simple :

1. struc-cond demande au programmeur d'ordonner l'ensemble "schéma" de sorte que pour la dernière formule de "schéma" à savoir $P_i \Rightarrow Q_i(r)$, struc-cond procède au remplacement de r par r_i (voir présentation de <sac> précédente) avec r_i <u>tq</u> $Q_i(r_i)$,
2. struc-cond construit la conditionnelle comme formulée dans la présentation de <sac>.

La fonction applic est :

```
applic(e) Δ si non stop alors
                s == choix entre
                     {<rdp>, <rdl>, <rdr>, <rdt>}
                applic (s(e))
                     sinon retourne e
```

Lexique : stop veut dire que le programmeur décide de ne plus appliquer de propriétés particulières,

s(e) signifie l'application de la SEL s sur e.

Remarque :

L'application de <sac> nécessite une vérification du résultat à posteriori. Pour obtenir un résultat correct il faudrait rendre l'algorithme plus systématique. Une façon de systématiser l'algorithme serait de le doter d'un moyen aidant le programmeur à construire un schéma conditionnel.

Des exemples d'application peuvent être vus au § III.1.2. et § III.3.

d) une mise en oeuvre de <sem>

Par analogie avec les précédentes stratégies une heuristique de <sem> est donnée par un algorithme dérivé de sa description générale (§ III.1.2.4.). Le but de cet algorithme est de transformer un énoncé e_r en e'_r dans lequel il n'existe aucun symbole de fonction figurant comme argument d'un symbole d'opération (exception faite du symbole "=").

Pour cela cette heuristique de transformation repose sur l'application de l'axiome :

(ax) $P(x,y) \Leftrightarrow P(x,z)$ et $z = y$.

- Présentation de <sei>

donnée : un énoncé e_r

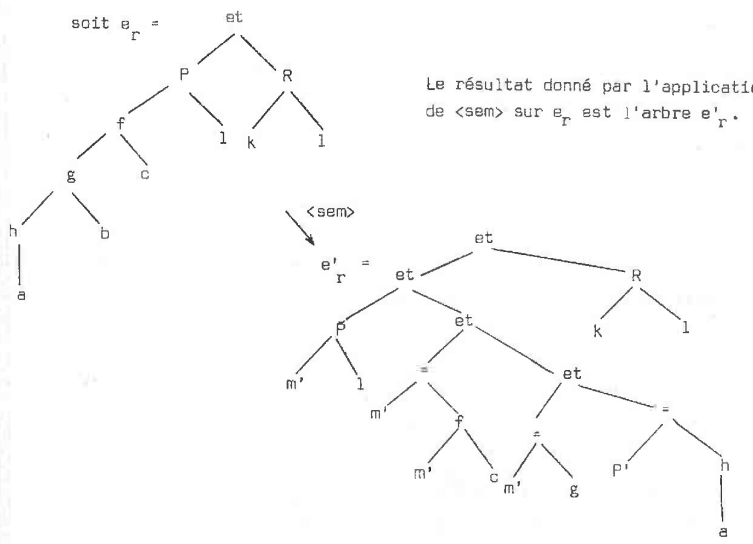
résultat : r caractérisé par e'_r transformé de e_r par applications successives de ax.

On peut dire qu'en vertu de l'axiome ax, e'_r obtenu est équivalent à e_r. C'est-à-dire toute solution de e_r pour une donnée d est solution de e'_r pour d et réciproquement.

* Algorithme de <sem>

```
<sem> (e_r) Δ symb == {f tel que f est un symbole de fonction
                       figurant comme argument d'un autre symbole de
                       fonction différent de "="
    tant que non symb = ∅ faire
        . prendre f ∈ symb
        . appliquer ax à e_r en considérant le remplacement
          de f(x̄) pour y et en ajoutant y = f(x̄)
        . supprimer f de symb
    ftq
        retourne e_r
```

Le paragraphe III.1.2.3. donne un exemple d'application de <sem>. L'exemple suivant montre la manière dont l'arbre d'un énoncé peut être "modulé" :



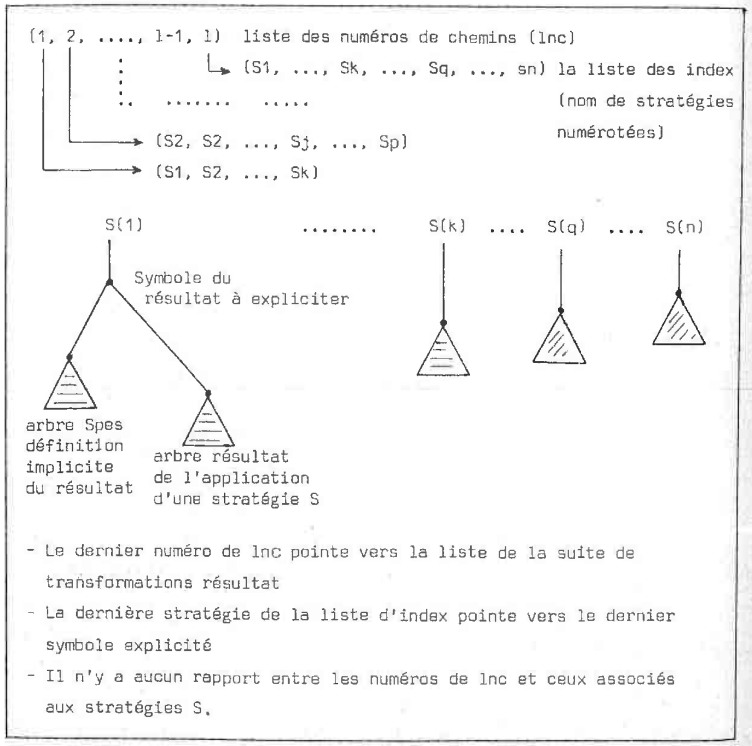
Le résultat donné par l'application de <sem> sur e_r est l'arbre e'_r.

IV.2.2.2. La bibliothèque des stratégies globales

La bibliothèque est structurée de la manière suivante :
- une liste de noms des différents algorithmes des SEG,
- un fichier regroupant ces algorithmes.

Une SEG est alors appelée comme on appelle une procédure paramétrée.
Le système met à jour cette liste et le fichier. Mais il ne sait pas modifier une SEG.
Le programmeur peut soit supprimer une stratégie globale, soit introduire un nouvel algorithme dans cette bibliothèque.

IV.2.3. L'arbre des choix



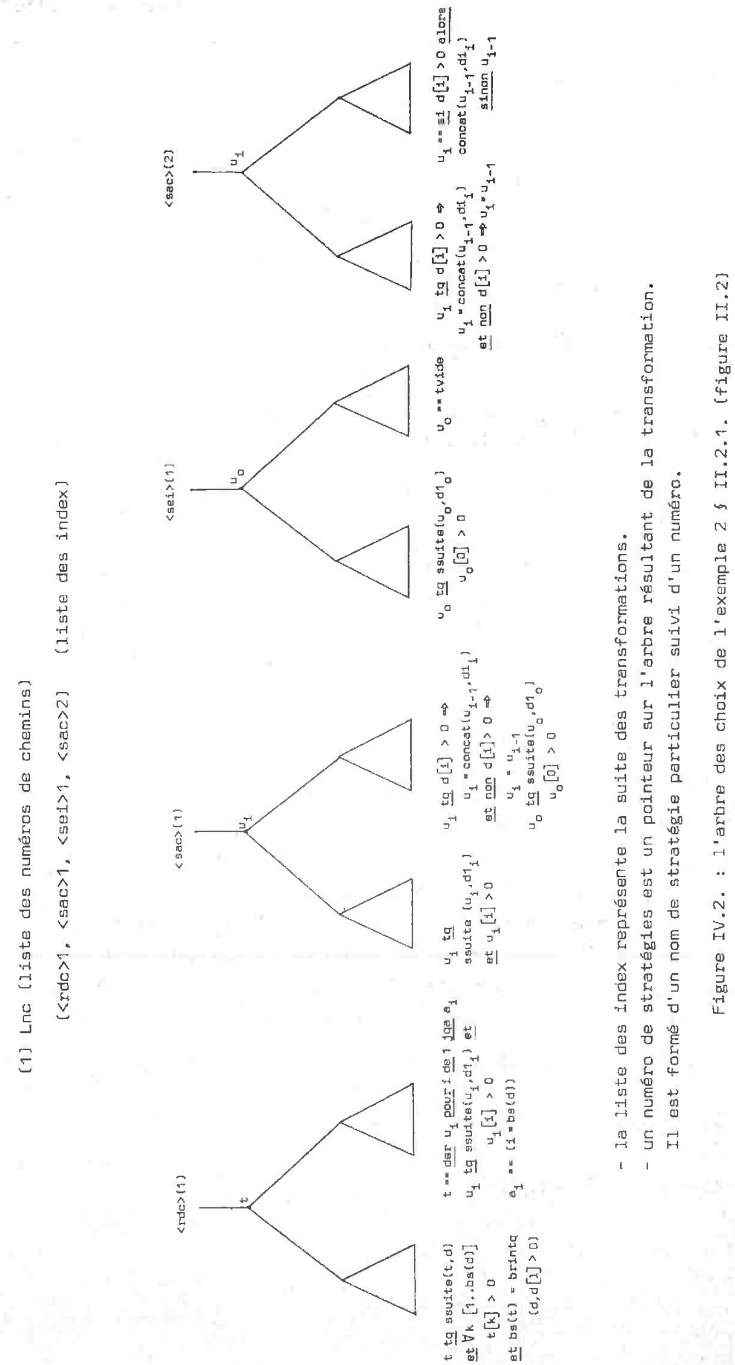
- Le dernier numéro de lnc pointe vers la liste de la suite de transformations résultat
- La dernière stratégie de la liste d'index pointe vers le dernier symbole explicité
- Il n'y a aucun rapport entre les numéros de lnc et ceux associés aux stratégies S.

schéma IV.3.

L'arbre des choix est construit au cours de l'explicitation. Il permet à l'utilisateur d'avoir une image de la suite des transformations de l'énoncé initial. Le programmeur peut s'appuyer sur cet arbre pour trouver éventuellement de nouvelles stratégies.
On rappelle que l'intérêt essentiel de cet arbre est d'aider le programmeur à faire des choix (d'identificateurs à expliciter, de stratégies) ou de revenir sur des choix antérieurs à partir desquels l'explicitation doit reprendre.
La structure de l'arbre des choix est donnée par le schéma IV.3.
Ce schéma ne fait pas apparaître explicitement ni l'énoncé complet initial, ni la solution algorithmique. Ces deux énoncés peuvent être facilement retrouvés en considérant la liste résultat des index associée au dernier numéro de la liste des numéros de chemins.
Un numéro d'une liste de chemins et la liste d'index associée à celui-ci caractérisent une reprise de l'explicitation.
Ainsi un arbre de choix dont la liste des numéros de chemins est composée d'un seul numéro (et par conséquent d'une seule liste d'index) précise que le programmeur n'est pas revenu sur des choix antérieurs.
La figure IV.2. ci-dessous présente un tel arbre.

En outre on peut remarquer une certaine redondance d'informations entre les différentes listes d'index dans la structure de l'arbre (schéma précédent). Cette redondance ne constitue pas en soi un inconvénient majeur, puisque le système explicite des petits énoncés. Par ailleurs cette redondance facilite le travail du manipulateur de l'arbre (§ IV.3.) et celui de la visualisation de cet arbre.

Tout arbre des choix construit lors d'une explicitation peut être gardé dans un fichier historique des explicitations : un fichier des arbres des choix.
L'intérêt d'un tel fichier est d'aider le programmeur à mettre en évidence des stratégies.



- la liste des index représente la suite des transformations.
- un numéro de stratégies est un pointeur sur l'arbre résultant de la transformation.

Il est formé d'un nom de stratégie particulier suivi d'un numéro.

Figure IV.2. : l'arbre des choix de l'exemple 2 § II.2.1. (figure II.2)

IV.3. Les modules fonctionels du système

Nous avons structuré le système en trois modes distincts gérés par le module d'interface programmeur/SIE (cf. schéma IV.1).
Chaque mode est caractérisé par une famille de fonctions.

Le mode édition (§ IV.3.2.) est caractérisé par la famille "constructeur d'arbres". Cette famille permet d'introduire des énoncés à expliciter. Il peut être vu comme un mini-éditeur syntaxique dont le rôle essentiel est de construire des arbres Spes bien formés.

Le mode connaissance (§ IV.3.3.) est caractérisé par la famille "bibliothécaire - manipulateur". Cette famille gère les stratégies et l'arbre des choix.

Et le mode résolution (§ IV.3.4.) est caractérisé par la famille "interpréteur". Cette famille permet l'enchaînement des différentes étapes de l'explicitation.

Avant de détailler ces trois familles de fonctions, donnons succintement le rôle de l'interface programmeur/SIE.

IV.3.1. L'interface programmeur/SIE

L'interaction entre le programmeur et le SIE permet à l'utilisateur :
- de faire les choix,
- d'introduire l'énoncé initial,
- de parcourir l'arbre des choix,

et au SIE de visualiser :
- l'arbre des choix de façon synthétique,
- les stratégies.

L'interface assure cette interaction en situant le dialogue dans un des trois modes, c'est-à-dire que trois types de dialogues sont à distinguer :

- le dialogue mode édition/programmeur,
- le dialogue mode connaissance/programmeur, et
- le dialogue mode résolution/programmeur.

Une illustration des différents dialogues et plus particulièrement les dialogues relatifs au mode connaissance et au mode résolution est donnée dans l'annexe E. Le dialogue mode édition/programmeur est ignoré car le mode édition n'est pas l'essentiel du travail.

Voyons maintenant la visualisation de l'arbre des choix et des stratégies illustrée dans l'annexe D.

a) La visualisation de l'arbre des choix se présente à l'écran comme suit :

```
          le père :     I (index)
                        I (nom de l'identificateur résultat, nom
                          de la stratégie)
  les fils :    I (index)                I (index)
                I (comme précédemment)   I (  "  )      ....

      "P" = remonter au père
      "Z" = zoom sur un noeud
"index fils" = descendre dans l'arbre
```

On dira que l'affichage de l'arbre est de profondeur 1, pour signifier que seuls les descendants directs d'un index "père" sont visualisés. "P" permet au programmeur de remonter dans l'arbre (avec un affichage de profondeur 1, c'est-à-dire que l'index "père" en cours devient un index "fils").
"index fils" permet au programmeur d'afficher progressivement l'arbre. Et enfin "Z" permet de visualiser les informations sur le résultat, l'application de la stratégie sur ce résultat et la stratégie elle-même. (pour cela on doit préciser "Z" et un index).

b) La présentation à l'écran des stratégies ne concerne que les stratégies d'explicitations locales. Une SEL est visualisée "par morceau". Pour cela le système affiche à l'écran les informations suivantes :

```
REGLE    nom de la règle

  "P" = visualisation de la pré-condition,
  "F" = visualisation du membre gauche (la forme) de la règle,
  "R" = visualisation du membre droit (partie résultat),
  "C" = visualisation de la post-condition
  "Q" = quitte
```

c) Le programmeur a aussi la possibilité d'avoir un affichage graphique d'un énoncé, de l'arbre des choix, et d'une stratégie locale.

d) La visualisation de l'arbre des choix, et des stratégies est faite soit systématiquement, soit à la demande de l'utilisateur dans les modes connaissances et résolution.

IV.3.2. Le mode édition

La famille "constructeur d'arbre d'énoncé" caractérise ce mode. Cette famille a pour tâche de reconnaître des énoncés Spes textuels et de les mettre sous forme d'arbres abstraits.
Un système tel que Mentor [DON-80, 83] peut être employé par notre système puisqu'il permet la représentation et la manipulation d'arbres abstraits. Mais comme SIE manipule des petits programmes Spes, on a préféré, dans un premier temps, définir un éditeur syntaxique très simple et adapté à nos propres besoins qui est appelé constructeur d'arbres d'énoncés.

Ce constructeur est schématisé par la figure IV.3. ci-dessous. Cette figure montre les quatre membres de la famille "constructeur d'arbres"

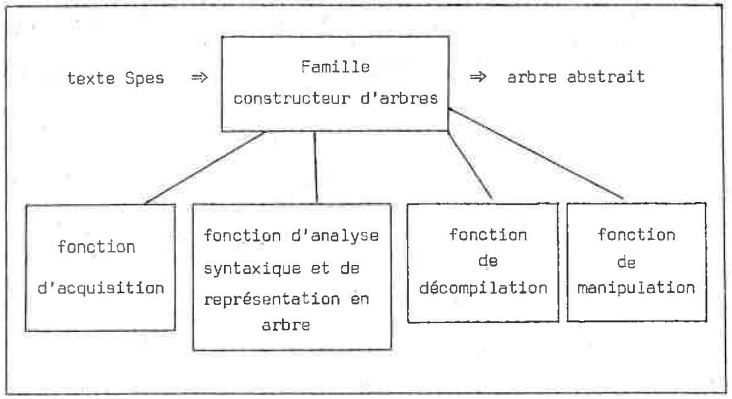


Figure IV.3. : la famille constructeur et ses membres.

qui sont les modules fonctionnels suivants :

- le module acquisition du texte Spes,
- le module d'analyse syntaxique et de représentation en arbre,
- le module de décompilation,
- le module de manipulation d'arbre Spes.

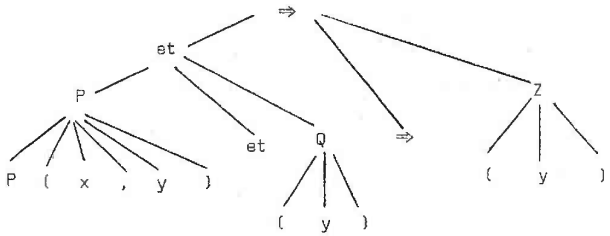
Pour la raison évoquée au § III.3.1. ces modules ne sont pas décrits.

Remarques :

1. Pour l'instant le contrôle de types n'est pas assuré.
2. Un texte Spes est représenté par un arbre abstrait étendu.

Par exemple :

le texte P(x,y) <u>et</u> Q(y) ⇒ Z(y) est représenté par l'arbre bien formé :



Cet arbre est dit "étendu" car la représentation en Lisp choisie est :

((Spes(⇒)) (⇒ (et ⇒ Z)) (et(P et Q)) (P(pm1))
(pm1(x,y)) (x(x)) (y(y)) (Q((pm2))) (pm2(y)) (y(y))
(Z(Z(pm3))) (pm3(y)) (y(y))).

Tous les arbres Spes manipulés par le système sont des arbres étendus. La syntaxe utilisée est décrite en annexe B.

IV.3.3. Le mode connaissance

La famille "bibliothécaire - manipulateur" caractérise ce mode. La figure IV.4. (cf. en fin de ce § IV.3.3.) schématise cette famille en distinguant les sous familles "bibliothécaire" et "manipulateur des choix" avec leurs membres respectifs.

a) La figure IV.4. indique que la famille "bibliothécaire" est composée des modules fonctionnels suivants :
   - le module de paramétrisation du SIE,
   - le module de gestion de la bibliothèque.

La paramétrisation est une tâche permettant à l'utilisateur d'exprimer des classes de stratégies et d'introduire au cours d'une explicitation de nouvelles stratégies. Actuellement SIE distingue 4 classes de stra-

tégies locales (cf. chap. II), et 4 sortes de stratégies globales (cf. § III.1.2.). Ainsi, grâce à la paramétrisation, le programmeur peut inventer de nouvelles classes de stratégies locales qui seront en fait des sous classes de celles dejà définies. Il peut également introduire de nouveaux algorithmes.

La gestion de la bibliothèque est réduite à :
- introduire une stratégie,
- supprimer une stratégie.

La tâche d'introduction d'une stratégie est la seule tâche de gestion qui nous intéresse pour le moment. (il n'est pas utile de discuter des fonctions de suppression et de modification).
L'introduction d'une stratégie locale se fait par "morceaux" (de la même manière qu'on la visualise par "morceaux" cf. § IV.3.1.b)). Chaque partie introduite est alors contrôlée syntaxiquement (il s'agit d'énoncés Spes).
Les stratégies globales sont représentées par des fonctions Lisp qui sont introduites directement dans la bibliothèque par le programmeur. En outre aucun contrôle de validité de ces stratégies n'est actuellement effectué (cf. II.4.4. sur la notion de validité). Elles sont supposées valides.

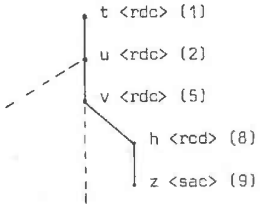
b) La famille "manipulateur des choix" est composée des modules fonctionnels suivants : (figure IV.4. ci-dessous)
- le module de construction,
- le module de parcours,
- le module de reprise.

Pendant la phase d'explicitation la construction de l'arbre se fait automatiquement à la fin de chaque transformation. La figure IV.2. et le schéma IV.1. (§ IV.2.3.) suffisent à comprendre ce processus de construction de l'arbre des choix.
Le parcours de l'arbre a déjà été abordé au § IV.3.2. a) lors de la présentation de la visualisation de l'arbre des choix.

La fonction de reprise (déjà discutée au § IV.3.2.) a pour but de reprendre l'explicitation à partir d'un noeud de l'arbre. Elle procède ainsi :
1. demande à l'utilisateur de l'index de reprise (§ IV.3.2.a)),
2. reconstitution du nouvel énoncé "complet" de départ.

Par exemple :
soit la structure suivante d'un arbre des choix :

Liste des numéros de chemins = (1, 2, 3)
liste des index → (<rdc>1, <rdc>2, <rdc>5 , <rdc>8, <sac>9)



le trait plein précise la dernière suite des transformations, les pointillés les suites antérieures (numéros de chemins 1, 2) ; supposons que le programmeur précise l'index <rdc> (5), alors la nouvelle structure de l'arbre est :

lnc = (1, 2, 3, 4)
→ (<rdc>1, <rdc>2, <u>rdc>5</u>)

et à partir de cette dernière liste d'index le module de reprise constitue le nouvel énoncé (un ensemble de définitions) à expliciter.

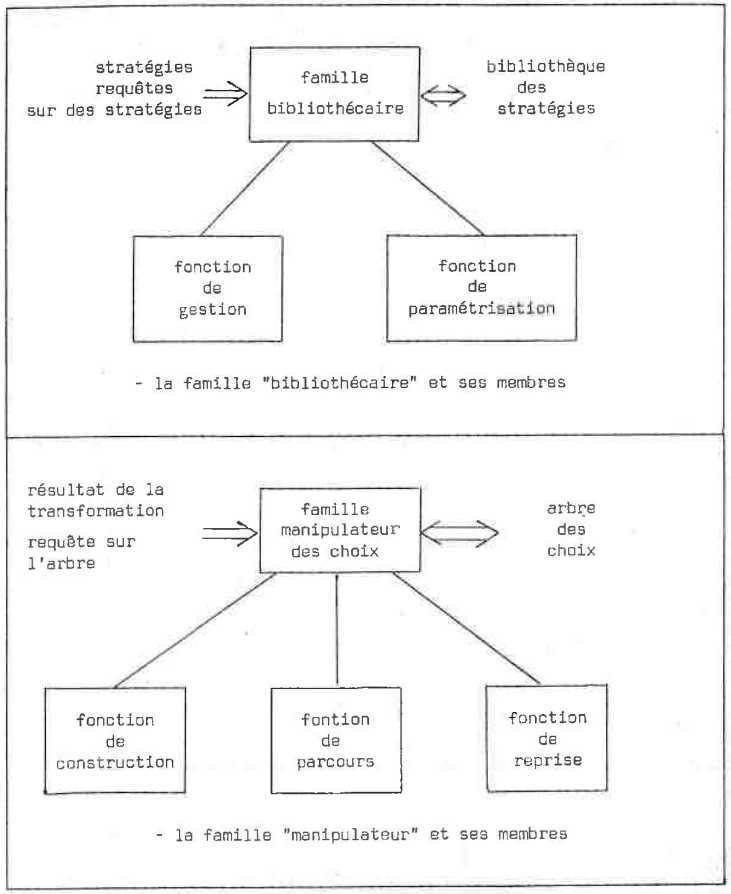


Figure IV.4. : La famille "bibliothécaire-manipulateur" en sous-familles.

IV.3.4. Le mode résolution

La famille "interpréteur" caractérise ce mode. Son rôle est de supporter la démarche d'explicitation telle qu'elle est présentée dans le chapitre II (notamment ses § II.4, II.5, II.6), et illustrée par des exemples au § III.3.

Et nous avons dit que cette démarche est vue comme une suite de transformations caractérisée par l'application successive de stratégies. Ce point de vue est exprimé par le méta-algorithme décrit au § III.2.1. Ainsi la tâche centrale de la famille "interpréteur" est la construction d'une telle suite, elle est accomplie par le module principal de l'interpréteur qui est décrit ci-dessous.

Notation : E est un énoncé transformé composé d'un ensemble de définition Spes e_x.

```
M_r == der E_k pour k de 1 jqa a_k
Choix entre
    - Explicitation d'un objet
      e_x == SELECT-I (E_k-1)
       s == SELECT-S (e_x)
      E_k == si est-globale(s) alors s(e_x, E_k-1)
                               sinon
             si s non = { } alors TRANSFORMATEUR (e_x, s, E_k-1)
                               sinon E_k-1
    - Remonter dans l'arbre
      E_k == MANIPULATEUR (arbre-des-choix)

E_o == énoncé de départ
a_k == est-explicite (E_k)
```

Le module principal réalise donc l'enchaînement des autres modules de l'interpréteur mentionnés dans le schéma IV.4. ci-dessous et qui sont :

1. le module de sélection de la définition à expliciter (SELECT-I),
2. le module de sélection de la stratégie à appliquer (SELECT-S),
3. le module d'identification (IDENTIFIEUR), utilisé par SELECT-S et TRANSFORMATEUR,
4. le module de transformation (TRANSFORMATEUR).

Remarques sur le module principal.

1. Par souci de simplification la suite E est composée à la fois d'énoncés transformés mais aussi d'énoncés issus d'une reprise (du MANIPULATEUR : fonction de reprise),
2. si s est une stratégie globale alors dans l'algorithme la décrivant figure implicitement des appels à SELECT-S, SELECT-I et TRANSFORMATEUR. l'algorithme "s" étant un enchaînement de stratégies locales, cf. § III.1.2.).
3. Le test "s <u>non</u> = { }" est expliqué dans le module SELCT-S.

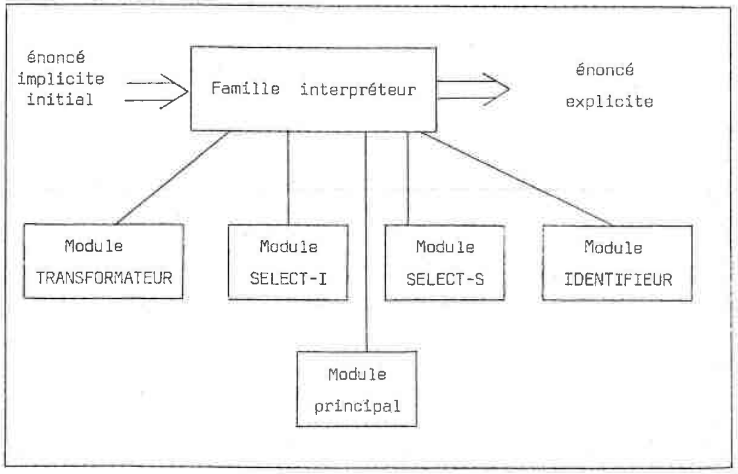


Schéma IV.4. La famille interpréteur

Maintenant décrivons les quatre modules de l'interpréteur cités précédemment.

IV.3.4.1. Le module de sélection de l'identificateur

Le sélectionneur SELECT-I schématisé par la figure IV.5. permet :

- de sélectionner l'ensemble des identificateurs non encore explicités,
- de permettre à l'utilisateur de choisir l'identificateur à expliciter et de saisir la définition implicite de l'identificateur choisi.

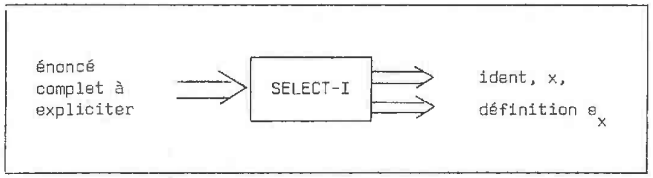


Figure IV.5.

L'algorithme de SELECT-I est très simple :

```
SELECT-I(E) Δ
            . choix de e_x tel que impl(E, e_x)
            . retourner e_x
```

<u>Commentaire</u> :

"impl" est un prédicat indiquant que la définition e_x est implicite.

IV.3.4.2. Le module de sélection de la stratégie à appliquer

Le sélectionneur SELECT-S donné par la figure IV.6. est chargé :

- de sélectionner un sous ensemble de stratégies applicables,
- de permettre à l'utilisateur de choisir une stratégie.

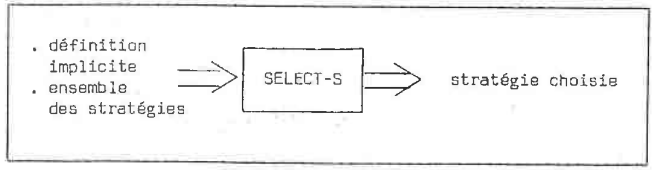


Figure IV.6.

L'algorithme de ce module est également très simple :

```
SELECT-S(e_x) Δ  choix entre
                 . choix de s ∈ {<sea>, <sei>, <sac>} ∪ {s ∈ 𝒮 tel que
                        pre(e_x) et IDENTIFIEUR (s, e_x) non = ϕ} ∪ { }
                 . retourne s
```

Commentaire

Le cas s = { } exprime une situation de blocage. L'intention alors du programmeur est de remonter dans l'arbre des choix ou de choisir une autre définition à expliciter.

IV.3.4.3. Le module d'identification

L'identifieur (figure IV.7.) a pour fonction :
de proposer une identification possible (si elle existe) entre la forme S d'une stratégie locale, {pre} S :: B {post}, et la forme d'une définition implicite e_x.

Le module "IDENTIFIEUR" a donc pour but de calculer un filtre d'identification σ (cf. définition § II.4.3.2.).

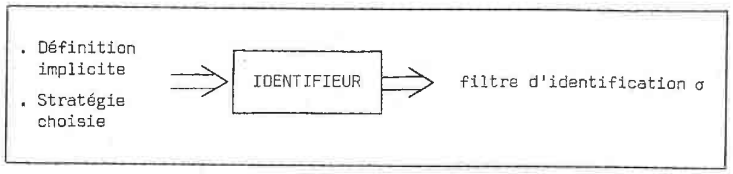


Figure IV.7.

Pour faciliter la lisibilité de l'algorithme du module IDENTIFIEUR on adopte les notations suivantes :

- $S/\alpha\ (e_x/\alpha')$ désigne le sous arbre de S (e_x) à l'occurrence $\alpha\ (\alpha')$ et $\alpha\ (\alpha')$ désigne aussi l'étiquette associée à cette occurrence (c'est-à-dire $S(\alpha)$, $e_x\ (\alpha')$), (S est l'arbre du membre gauche d'une stratégie locale),
- α_i désigne le $i^{ème}$ fils de α si $a(\alpha) > 0$,
- on supprime l'indice de récurrence et on note par $\bar{x}$ la précédente valeur de x.

L'algorithme est :

```
IDENTIFIEUR (S/α, e_x/α') Δ σ == ϕ  ;  σ_s == vrai
                          retourne (inter (sigma(S/α, e_x/α')))
```

Commentaires :

- Lors de l'appel de IDENTIFIEUR par SELECT-S, le premier peut rendre comme résultat "ϕ",
- inter est la fonction définie au § II.4.3.2. Elle introduit les paramètres intermédiaires,
- sigma calcule le filtre d'identification σ (un ensemble de substitutions),
- σ_s indique si σ existe (σ_s = vrai).

Décrivons sigma.

```
sigma(S/α, e_x/α') Δ
    si α = α' alors pour i de 1 à a(α) faire
                    si non σ_s alors σ == ϕ ; retourne σ
                                sinon
                    si feuille (α_i, S/α) alors  σ == σ̄ ∪ {α_i ← α'_i}
                                          sinon
                                  σ == σ̄ ∪ sigma (S/α_i, e_x/α'_i)
                 fpour
                 retourne σ̄
            sinon
    si est-variable-fonction(α) et a(α) ≤ a(α')
                                alors σ_f == σ - feuille(α,α')
                                    si σ_f = ϕ alors σ == ϕ ; σ_s == faux
                                                retourne σ
                                            sinon
                                      retourne σ_f ∪ {S/α ← e_x/α'}
                                sinon
                                   σ == ϕ ; σ_s == faux
                                   retourne σ
```

Commentaire :

σ_f est l'ensemble des remplacements de toutes les feuilles de la variable fonction désignée par α par des sous arbres de e_x/α'.

```
σ - feuille (α,α') Δ  σ_f == ϕ
                 pour i de 1 à a(α) faire
                   si a(α_i) > 0 alors si est-variable-fonction (α_i)
                                        ou α_i ≠ α'_i
                                        alors retourne ϕ
                                        sinon ;
                                sinon
```



```
                 si il existe k tel que
                          attribut de α_i = attribut de α'_k
                       et α'_k ∉ σ̄_f
                 alors
                       σ_f == σ̄_f ∪ {α_i ← e_x/α'_k}
                 sinon
                       retourne ϕ
        fpour
            retourne σ̄_f
```

Commentaire

La fonction σ- feuille contrôle l'hypothèse H2 et les conditions cond1 et cond2 mentionnées au § II.4.3.4. De plus σ- feuille calcule des substitutions de la forme $z \leftarrow F(a,b)$ où "z" est une feuille S (de la règle S::B) et "F" une fonction définie figurant dans e_x (voir condition 1 § II.4.4.) à condition, bien sûr, que le type de "z" soit égal au but de "F". Ce calcul fait partie de la recherche d'une feuille α'_k de e_x exprimée par : "il existe k tel que ... ".

IV.3.4.4. Le module de transformation

Ce module est chargé de l'instantiation du membre droit d'une stratégie locale (c'est-à-dire B de {pré} S::B {post}). Il a donc pour but de produire le résultat de l'application d'une stratégie locale à un énoncé. Par conséquent la tâche essentielle de TRANSFORMATEUR est le calcul de σ' (cf. définition de σ' § II.4.3.2.) : $\sigma' = \sigma_{\beta i} \circ \sigma_{\beta i-1} \ldots \circ \sigma_{\beta 1}$.
Ce calcul est effectué par la fonction transfo.

La figure IV.8. ci-après schématise le module TRANSFORMATEUR.

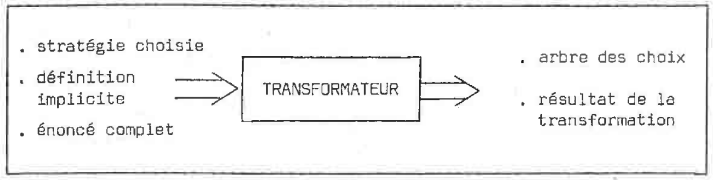


Figure IV.8.

Décrivons l'algorithme de TRANSFORMATEUR. Cette description utilise les notations de l'algorithme précédent.

```
TRANSFORMATEUR (e_x, S::B, E_k) Δ
        σ == IDENTIFIEUR (S/α, e_x/α')
        B' == σ B
        σ' == transfo (σ, B, B')
        pour β dans dom (B') tel que est-variable-fonction(β) et
                                 β non-étiquette de S        faire
            . proposition de substitution σ_β
            . σ' == σ̄' ∪ σ_β
        fpour
        si post-condition alors MANIPULATEUR (arbre des choix)
                               retourne E_k [e_x ← σ' B']
                         sinon
                               retourne E_k
```

Commentaires :

- non-étiquette indique que le symbole de la variable fonction désignée par β n'est pas aussi dans S.
- proposition indique que le programmeur propose un remplacement de la variable fonction β ne figurant pas dans S.
- l'appel à la famille MANIPULATEUR a pour but d'inclure le résultat de la transformation dans l'arbre des choix.

Donnons la description de la fonction transfo.

```
transfo (σ, B, B') Δ   σ' == ϕ
    pour β dans dom(B) tel que est-variable-fonction β faire
        . soit {S/α ← e_x/α'} ∈ σ (un remplacement de structure)
                    tel que α' = β  (même symbole de fonction)
        . σ_β == ϕ
    pour i de 1 à a(β) faire
        . soit β' = B'/β_i tel que α_i = β' (même symbole intermé-
                                             diaire)
        . σ_β == σ̄_β ∪ {β' ← σ(B/β_i)}
    fpour
      σ' == σ̄' ∪ σ_β
 fpour
    retourne σ̄'
```

Commentaire :

Il faut bien voir que β désigne l'adresse où a été effectué le remplacement de B/β par e_x/α lors de l'application de σ sur B, ainsi $\beta \in dom(B')$ mais aussi $\beta \in dom(B)$.

Remarque générale :

On ne s'est pas intéressé à l'aspect complexité de tous les algorithmes décrits précédemment.

IV.4. Les développements ultérieurs du système

Trois axes de développement du système semblent particulièrement intéressants :

- les modes,
- la représentation des connaissances,
- l'interpréteur.

a) Les modes

Faire évoluer le mode édition consiste à choisir un éditeur vidéo existant comme CEYX ou MENTOR vidéo [DON-83, HUL-83].
Une façon d'étendre le mode connaissance est de développer un système d'interrogation de la bibliothèque, dans le but de mieux assister le programmeur dans ses expériences.
Une évolution importante du mode résolution consiste à le doter d'un prouveur permettant de valider les stratégies locales et les pré-post conditions.
De plus il serait utile d'aider le programmeur à exprimer ces pré-posts d'une manière telle que leur expression permette de tendre vers une vérification systèmatique.

b) La représentation des connaissances

Comme nous l'avons évoqué plus haut (§ IV.2.1.2.) il est nécessaire de "bien" structurer la bibliothèque, notamment pour faciliter la sélection de stratégies locales, et ce par une étude approfondie de la hiérarchisation de ces stratégies.

c) L'interpréteur

Il s'agit essentiellement de développer les modules TRANSFORMATEUR et IDENTIFIEUR :

- en permettant dans la partie droite d'une stratégie (S::B) des formes telles que : R(f(R'(a,b)),y) où R et R' sont des variables fonctions, (en généralisant à plusieurs variables fonctions),

- de voir dans quelles mesures on peut autoriser que S soit de la forme R(f(R'($\bar{x}$)) ($\bar{x}$ ne contient aucune variable fonction)
- et en tenant compte, lors du calcul de l'identification, de certaines propriétés des connecteurs logiques : comme par exemple la commutativité du "et" et du "ou".

En conclusion

Nous venons de décrire un système intéractif de transformations d'énoncés.
Les tâches de "paramétrisation" et de "construction" constituent le rôle essentiel de ce système.
La paramétrisation s'exprime sous la forme d'un mécanisme d'enrichissement de la bibliothèque de stratégies. Ce mécanisme peut intervenir dans le processus de l'autre tâche.
La construction assure l'assistance à l'explicitation. Elle est fondée sur l'exploitation de stratégies d'explicitations, supposées valides.
L'assistance est principalement caractérisée par un arbre des choix, la sélection d'un sous ensemble de stratégies à chaque transformation, et la possibilité de construire "à la main".
Notre système diffère de ceux proposés dans [BUR-77], [BID-80], [PAG-83], [GRE-84]. Il se rapproche plutôt d'un système comme [BAR-79] mais encore plus du système automatique Dedalus [MAN-79] bien que ce dernier se veut très général alors que le notre est beaucoup plus spécifique : il construit des énoncés itératifs en se plaçant dans une classe de problèmes donnée. Par exemple une telle application peut être la construction de programmes manipulant des tableaux unidimensionnels, c'est ce que nous étudions au chapitre suivant.

CHAPITRE V

V. UNE APPLICATION : LE TYPE TABLEAU

Dans le chapitre I, il est dit que le système décrit précédement devrait permettre au programmeur de mieux connaître les règles à l'origine de la construction de programmes.
Pour cela, nous avons choisi, tout d'abord, d'utiliser le système pour construire des programmes manipulant des tableaux à 1 dimension.

Cette classe de problèmes est importante en programmation [DAH-72, DIJ-76]. Et comme dit HOARE dans [DAH-72] le tableau est pour beaucoup de programmeur la structure de donnée la plus familière, et dans de nombreux langages de programmation c'est la seule structure disponible explicitement.

Par ailleurs on peut citer certains travaux comme [ELL-81, GUI-79, BRO-83 JAF-82, JEF-80, JOH-73, REY-75] consacrés spécialement aux programmes sur les tableaux.

Le but de ce chapitre est d'étudier une classe d'algorithmes portant sur les tableaux. Pour cela il faut tout d'abord décrire le type TABLEAU, c'est-à-dire préciser le contexte dans lequel on exprime les énoncés à construire. Ainsi quelques exemples de ces énoncés seront donnés pour montrer ensuite le fonctionnement du système dans la construction d'un tel énoncé.

V.1. Spécification du type abstrait TABLEAU

La définition que l'on donne à un tableau (noté TAB) est une fonction exprimée par Hoare [DAH-72] et Dijkstra [DIJ-76] :

- un objet de type TABLEAU (noté TAB) est une fonction définie sur un ensemble d'éléments d'un type donné (les indices) à valeur dans un ensemble d'éléments d'un autre type (appelé VAL) associant à chaque indice une valeur.

Mais en plus une caractéristique particulière de notre type TABLEAU est :
le domaine d'un objet de type TAB est défini comme un intervalle d'entiers.

Par exemple :
Soit d une variable de type TAB (d : TAB)
alors dom(d) = [1..4], (dom indique le domaine de d qui est un intervalle), l'intervalle [1..4] est équivalent à (≡) l'ensemble de départ {1,2,3,4} de d
On notera que les indices sont ordonnés. Un type INTERVALLE est défini plus bas.
En outre 1 est appelé la borne inférieure de d (bi(d) = 1) et 4 sa borne supérieure (bs(d) = 4).

* Définition de la signature Σ

Les opérations de la signature Σ de TAB sont : [LEV-84]
- les constructeurs : les opérations internes (rendant un résultat du type d'intérêt) suffisantes pour générer tous les objets du type TAB [LES- 82],
- les modificateurs : des opérations internes qui sont des extensions [LES- 82],
- les observateurs : les opérations paramétrés du type TAB [LEV-84]. Le type TAB est paramétré par des opérations associées à ses types paramètres.

La construction d'un objet de type TAB se fait à partir du tableau "tvide" et en ajoutant par la "droite" les éléments associés à des indices grâce à l'opération "affect".

Exemple : soit d : TAB

d = affect(affect(affect(affect(tvide,v_1),v_2),v_3),v_4)

Remarque :
Il n'est pas possible d'ajouter un élément indéfini à un tableau affect (d, indéfini) "n'est pas permis", c'est-à-dire est indéfini.

Les modifications possibles d'un objet de type TAB sont présentées ci-dessous et illustrées chacune par un exemple.

- La suppression d'un élément repéré par son indice (supr),
  supr(d,2) = affect(affect(affect(tvide,v_1),v_3),v_4)
  On obtient un tableau sans "trou".

- le changement d'une valeur associée à un indice (chval)
  chval(d,v',3) = affect(affect(affect(affect(tvide,v_1),v_2),v')v_4).

- l'insertion d'un élément (insert)
  insert(d,v',2) = affect(affect(affect(affect(affect(tvide,v_1),v'), v_2),v_3),v_4).

- la transposition de 2 éléments repérés par leur indice,
  trsp(d,3,4) = affect(affect(affect(affect(tvide,v_1),v_2),v_4),v_3)

- la concaténation de 2 tableaux
  concat(d,affect(tvide,v'_5)) = affect(affect(affect(affect(affect (tvide,v_1)...),v'_5)

- l'inversion d'un tableau
  inverse(d) = affect(affect(affect(affect(tvide,v_4),v_3),v_2),v_1)

- la tranche d'un tableau
  tr(d,1,3) = affect(affect(affect(tvide,v_1),v_2),v_3).

Les observations permettent :

- l'accès aux éléments d'un tableau, par l'opération "acces" :
  acces(d,3) = v_3

- de déterminer les bornes d'un tableau (bs)

  bs(d) = 4 ; l'opération borne inférieure (bi) citée précédemment est inutile car la borne inférieure d'un tableau est toujours égale à 1,

  bi(tr(d,2,3)) = 1 , bs(tr(d,2,4)) = 2 ,

- de connaître l'intervalle d'un tableau grâce à l'opération "dom" :

  dom(d) = [1..4]

- d'avoir la longueur d'un tableau :

  lg(d) = 4

- de connaître le plus grand indice d'un élément :

  assoc(d,v_3) = 3

  assoc(d,v') = 0

- de savoir si un tableau est vide, ainsi que d'autres relations entre les tableaux :

  estvide(d) = faux

  stab(affect(affect(tvide,v_3),v_2),d) = vrai.

. Le prédicat stab indique si les éléments associés aux indices d'un tableau sont aussi des éléments associés à certains indices d'un autre tableau.

. Le prédicat ssuite indique que les éléments associés aux indices d'un tableau sont une séquence des éléments associés aux indices d'un autre tableau :

  ssuite(affect(affect(tvide,v_2),v_4),d) = vrai

  ssuite(affect(affect(tvide,v_3),v_2),d) = faux

. Le prédicat égal indique que 2 tableaux ont même domaine et que leurs éléments de même indice sont égaux.

. Le prédicat equ indique que 2 tableaux sont équivalents à une permutation près.

Les opérateurs paramétrés que comprend le type TAB sont :

- nbrintq : (se lit "le nombre d'indices tels que") rend le nombre des indices dont les éléments associés vérifient une propriété P1,
- croiss : indique que les éléments d'un tableau sont classés dans un ordre défini par une relation d'ordre R,
- extens : indique que si un élément d'un tableau vérifie une propriété P_2 alors tous les autres éléments de ce tableau vérifient P_2,
- seem i indique qu'un tableau, défini à partir d'un autre tableau, est caractérisé par une propriété P_3.
  seem est en quelque sorte un nom que l'on donne à P_3.

Ces définitions supposent que le type TAB est paramétré par les propriétés : P_1, P_2, R, et P_3 et un prédicat d'égalité "egal" et par le type des éléments, VAL :

- VAL précise le domaine des éléments, (le type formel),
- R : VAL x VAL → BOOL
- P_1 : VAL → BOOL
- P_2 : VAL → BOOL
- P_3 : TAB x TAB → BOOL

type TAB [VAL, egal, P_1, R, P_2, P_3]

les opérations de TAB [VAL, egal, P_1, R, P_2, P_3]

Constructeurs

affect : TAB x VAL x ENT → TAB		ajout d'un élément par la droite
tvide : → TAB		tableau vide

Modificateurs

supr : TAB x ENT → TAB		suppression de l'élément indicé par i : ENT
chval : TAB x VAL x ENT → TAB		changement d'une valeur en un point i du tableau
insert : TAB x VAL x ENT → TAB		insertion d'un élément à un point du tableau
trsp : TAB x ENT x ENT → TAB		échange de 2 éléments repérés par leur indice
concat : TAB x TAB → TAB		concaténation bout à bout de 2 tableaux
inverse : TAB → TAB		l'image miroir d'un tableau
tr : TAB x ENT x ENT → TAB		une séquence connexe d'un tableau (tranche)

Figure V.1. Signature de TAB

(la suite de cette signature est donnée à la page suivante par la figure V.2.).

Les opérations de TAB [VAL, egal, P1, R, P_2, P3] (suite de la figure V.1. précédente)

Observateurs

accès : TAB x ENT → VAL		la valeur en un point du tableau (dans les chapitres précédents "acces" était noté "[]").
bs : TAB → ENT		borne supérieure du tableau
dom : TAB → INT		domaine d'un tableau est un intervalle
lg : TAB → ENT		la longueur du tableau
assoc : TAB x VAL → ENT		l'indice associé à une valeur (la première)
appt : TAB x ENT → BOOL		appartenance d'un indice donné à un tableau
estvide : TAB → BOOL		le tableau est-il vide ?
stab : TAB x TAB → BOOL		un tableau est constitué d'éléments d'un autre tableau
ssuite : TAB x TAB → BOOL		un tableau est une séquence, non nécessairement connexe, d'un autre tableau
egal : TAB x TAB → BOOL		égalité entre 2 tableaux

Opérateurs

nbrintq : TAB → ENT		nombre d'indices tels que P1 soit vraie
croiss : TAB → BOOL		la relation d'ordre R vérifie tous les éléments d'un tableau
extens : TAB x VAL → BOOL		si P_2 vérifie une valeur repérée par i alors P_2 est vraie pour tous les autres éléments
seem : TAB x TAB → BOOL		le tableau d est défini à partir de d' tel que P_3(d,d')

Figure V.2. Les observateurs et les opérateurs du type TAB

* Les axiomes du type TABLEAU

```
i,j,k : ENT          v,v',v1, ... : VAL
t,d : TAB
```

invariant :

```
bs(affect(d,v)) = bs(d) + 1
```

(pour des commodités d'écriture on désignera parfois bs(d) + 1 par k).

Les opérations :

- Définition des modificateurs

```
supr(tvide,i) = tvide
supr(affect(t,v),i) = si i = bs(t) + 1 alors t
                                          sinon
                                      affect(supr(t,i),v)

chval(tvide,v',i) = tvide
chval(affect(t,v),v',i) = si i = bs(t) + 1 alors affect(t,v')
                                          sinon
                                      affect(chval(t,v'),v)

insert(tvide,v,i) = si i = 1 alors affect(tvide,v)
                             sinon tvide
insert(affect(t,v),v',i) = si i = bs(t) + 1 alors
                                      affect(chval(affect(t,v),v',i),v)
                                          sinon
                                      affect(insert(t,v',i),v)

trsp(tvide,i,j) = tvide
trsp(affect(t,v),i,j) = si (i = k et appt(t,j)) ou (j = k et appt(t,i))
                        ou (i = k et j = k)
                          alors si i = k alors
                            chval(chval(affect(t,v), acces(t,j),k),v,j)
                                          sinon
                            chval(chval(affect(t,v), acces(t,i),k),v,i)
                          sinon affect(trsp(t,i,j),v)
```



```
appt(tvide,i) = faux
appt(affect(t,v),i) = si i = k  alors vrai
                                sinon appt(t,i)

estvide(tvide) = vrai
estvide(affect(t,v)) = faux

stab(tvide,tvide) = vrai
stab(tvide,t) = vrai
stab(t,tvide) = faux
stab(affect(t,v),affect(t',v')) = si v = v' alors stab(t,t')
                                          sinon
                                  si assoc(t',v) = 0 alors faux
                                          sinon
                                  stat(t,supr(t',assoc(t',v)))

ssuite(tvide,t) = vrai
ssuite(t,tvide) = estvide(t)
ssuite(affect(t,v), affect(t',v')) = si v = v' alors ssuite(t,t')
                                          sinon
                                    ssuite(affect (t,v),t')

equ(tvide,t) = estvide(t)
equ(affect(t,v), affect(t',v')) = lg(t) = lg(t') et stab(affect(t,v),
                                          affect(t',v'))

assoc(tvide,v') = 0
assoc(affect(t,v),v') = si v = v' alors bs(t) + 1
                                sinon
                                    assoc(t,v')

concat(tvide,tvide) = tvide
concat(t,tvide) = t
concat(tvide,t) = t
concat(affect(t,v), affect(t',v')) = affect(concat(affect(t,v),t'),v')
```

```
inverse(tvide) = tvide
inverse(affect(t,v)) = insert(inverse(t),v,1)

tr(tvide,i,j) = tvide
tr(affect(t,v),i,j) = si j < i alors tvide
                                sinon
                          si k ≠ j  alors tr(t,i,j)
                                    sinon
                                affect(tr(t,i,j-1),v)
```

- Définition des observateurs

```
acces(tvide,i) = indéfini
acces(affect(t,v),i) = si  i = k alors  v  sinon acces(t,i)

bs(tvide) = 0
bs(affect(d,v)) = bs(d) + 1

lg(tvide) = 0
lg(affect(t,v)) = lg(t) + 1

egal(tvide,t) = estvide(t)
egal(affect(t,v), affect(d,v1)) = si v = v1 et bs(t) + 1 = bs(d) + 1
                                  alors egal (t,t')
                                  sinon faux

dom(tvide) = [1..0]  (l'intervalle vide)
dom(t) = [1..bs(t)]
```

- Définition des opérateurs

```
P1 : VAL → BOOL
nbrintq(tvide,P1) = 0
nbrintq(affect(t,v),P1(v)) = si  P1(v) alors nbrintq(t,P1) + 1
                             sinon nbrintq(t,P1)
```



```
R : VAL x VAL → BOOL
croiss(tvide) = vrai
croiss(affect(t,v)) = si  R(v,acces(t,bs(t)) et croiss(t)

P2 : VAL → BOOL
extens(tvide,P2(v')) = vrai
extens(affect(t,v), P2(v')) = si  P2(v) alors extens(t,v)
pré-condition : P2(v')

P3 : TAB x TAB → BOOL
seem(tvide,d) = vrai,   seem(tvide,tvide) = vrai
seem(affect(t,v), affect(t',v')) = P3(affect(t',v')),
                                   (affect(t,v))
```

* Une spécification du type INTERVALLE D'ENTIER

Un intervalle d'entier est noté $[m..n]$:

$[m..n] \equiv \{i \text{ tel que } m \leqslant i \leqslant n\}$;

si $n > m$ alors $[m..n]$ est l'intervalle vide que l'on peut noter par "$[..]$" :

$[..] \equiv \phi$.

La construction d'un objet de type intervalle d'entier se fait à partir de l'intervalle vide $[..]$ en introduisant successivement des entiers de façon à obtenir un intervalle $[m_0 .. m_i]$ tel que :

1. $(m_i - m_0) \geqslant 0$,
2. $m_0 \leqslant m_1 \leqslant \ldots \leqslant m_{i-1} \leqslant m_i$.

La notation $[m_0 .. m_i]$ désigne alors un ensemble ordonné :

$[m_0 .. m_i] \equiv \{m_0, m_1, \ldots, m_i\}$

Dans le cas $[m_0 .. m_i]$ tel que $m_0 = m_i$ alors $[m_0 .. m_i] \equiv \{m_0\}$

Type INTERVALLE d'ENTIER (noté INT)

* Opérations

Constructeurs

[..] : → INT	intervalle d'entiers (plus brièvement intervalle) vide
[ ] : INT x ENT → INT	construction d'un intervalle

Observateurs

ls : INT → ENT	limite supérieure d'un intervalle
li : INT → ENT	limite inférieure d'un intervalle
appit : INT x ENT → BOOL	appartenance à un intervalle

* Les axiomes

li ([..]) = 1
li ([m..n]) = m

ls ([..]) = 0
ls ([m..n]) = n

appit([m..n]), i) = si m ⩽ i ⩽ n alors vrai sinon faux

V.2. Exemples d'énoncés sur le type TABLEAU

Nous allons présenter quatre exemples d'énoncés formels sur le type TAB différents de ceux introduits dans le chapitre III qui sont :

- l'interclassement de 2 tableaux triés,
- le drapeau tricolore,
- un problème de traitement de texte,
- la plus longue sous-suite croissante.

a) Interclassement de 2 tableaux triés

On cherche à exprimer que l'on veut construire un tableau trié t d'entiers à partir de 2 tableaux d'entiers t_1 et t_2 triés.
La figure V.3. ci-dessous donne une expression formelle du résultat à expliciter :

- equ(t,concat(t_1,t_2)) spécifie que les éléments associés aux indices de t sont à la fois des éléments associés aux indices de t_1 et t_2,
- croiss(t) précise que la relation ⩽ sur l'ensemble des éléments de t doit être vérifiée : $\forall i \in [1..bs(t)-1]\ \ t[i] \leqslant t[i+1]$

Lexique	Profil	Enoncé formel
t est le tableau résultat t trié constitué des éléments de t_1 et t_2 t_1, t_2 tableaux d'entiers triés donnés	t : TAB [ENT,egal,,⩽,,] t_1 : TAB [ENT,egal,,⩽,,] t_2 : TAB [ENT,egal,,⩽,,]	résultat : t t tq equ(t, concat(t_1,t_2)) et croiss(t) données : t_1,t_2 tq croiss(t_1) et croiss(t_2)

Figure V.3.

Remarque : Dans le langage Spes tel qu'il est utilisé par le système on ne peut pas exprimer des propriétés particulières dans la partie "donnée" de l'énoncé formel.

b) Un énoncé du drapeau tricolore

Soit un tableau d constitué de plusieurs lettres "r", "b" et "v".
On veut exprimer qu'un tableau t à construire est un réarrangement du tableau d donné, c'est-à-dire :

soit dom(t) = [1..bs(t)] alors
$\forall$ l $\in$ [1..p] t[l] = " b"
et $\forall$ m $\in$ [p+1..q] t[m] = "r"
et $\forall$ n $\in$ [q+1..bs(t)] t[n] = "v" ;

c'est ce qu'exprime la figure V.4. ci-dessous.

Lexique	Profil	Enoncé formel
t tableau résultat un réarrangement des lettres figurant dans d d tableau donné constitué de lettres "r", "b", et "v".	t : TAB [CAR,egal,,, t[x] = "y",,] d : TAB [CAR,egal,,, t[x] = "y",,] i,j : [1..n] n : ENTIER	résultat : t t,i,j tq equ(t,d) et extens(tr(t,1,i),t[1] = "b") et extens(tr(t,i+1,j),t[i+1] = "r") et extens(tr(t,j+1,n),t[j+1] = "v") et n == bs(d) donnée : d

Figure V.4.

c) La recherche des mots de longueur 1,2,... 20 d'un texte donné

Pour préciser l'énoncé formel de ce problème (voir chapitre I, e-1), on cherche à exprimer que tout élément associé aux indices du tableau d'entiers t à construire à partir d'un tableau d de caractères, est donné par le nombre d'indices de d qui vérifient une certaine propriété caractérisant un mot de longueur i (appit([1..bs(t)],i)).

Une telle propriété, comme pour les précédentes, est passée en paramètre. Elle est définie par le programmeur, et peut être :

P_1 : lg(tr(d,k,j)) = i et (k = 1 ou d[h + 1] = "␣") et (j = bs(d)
ou d[j + 1] = "␣") et assoc(tr(d,k,j),"␣") = 0.

Le symbole "␣" désigne le caractère blanc. Un mot ne contient pas de blancs et il est compris entre 2 caractères " ". C'est ce qu'exprime la figure V.5. ci-dessous.

Lexique	Profil	Enoncé formel
t tableau résultat dont chaque élément à un indice i est le nombre de mots de longueur i d est un tableau de caractères, le texte donné k et j délimitent un mot (les mots sont une suite de caractères	t : TAB [ENT,egal,,,,,] d : TAB [CAR,egal,P_1,,,,] (P_1 voir précédemment) k : [1..bs(d)] j : [1..bs(d)]	résultat : t t tq dom(t) = [1..20] $\forall$i : [1..20] t[i] = nbrintq(d, lg(tr(d,k,j)) = i et (k = 1 ou d[k + 1] = "␣") et (j = bs(d) ou d[j + 1] = "␣") et (assoc(tr(d,k,j), "␣") = 0)) donnée : d

Figure V.5.

d) Un énoncé du problème de la recherche de la plus longue sous-suite croissante extraite d'une suite donnée

La figure V.6. ci-après montre que pour construire un tableau t celui-ci est considéré comme la sous-suite la plus longue appartenant à l'ensemble des sous-suites croissantes (non nécessairement connexes) d'un tableau d.

La formule P_3 : ssuite(t',d) <u>et</u> croiss(t')
précise la propriété caractéristique de cet ensemble,
et lg(t) $\geqslant$ lg(t') permet d'indiquer qu'il existe un tableau t
dans cet ensemble de longueur maximale.

Lexique	Profil	Enoncé formel
t est la plus longue sous-suite croissante de d d un tableau d'entiers	t : TAB [ENT,egal,,$\leqslant$,, P_3] (P_3: ssuite (t,d) <u>et</u> croiss(t)) d : TAB [ENT,egal,,$\leqslant$,]	<u>résultat</u> : t t <u>tq</u> seem(t,d) <u>et</u> $\forall$k,l croiss(tr(d,k,l)$\Rightarrow$ lg(t) $\geqslant$ lg(tr(d,k,l)) <u>donnée</u> : d

Figure V.6.

Montrons maintenant à travers un dialogue programmeur / SIE le fonctionnement du SIE.
Ce dialogue est structuré par une technique de menus.

V.3. Fonctionnement du système

Pour illustrer la mise en oeuvre du système ineractif d'explicitations, utilisons l'exemple 2 du § III.3 :

$$ t TRI t <u>tq</u> equ(t,d) <u>et</u> croiss(t) (TRI est le nom de l'énoncé de résultat t)

Au démarrage le système affiche à l'écran un menu de modes :

```
MENU-MODES

esc1 = MODE EDITION
esc2 = MODE CONNAISSANCE
esc3 = MODE RESOLUTION
esc4 = QUIT
```

Supposons que le programmeur ait choisi le mode "RESOLUTION".
Sous ce mode le système se déroule comme ci-dessous.

1)

```
? NOM DE L'ENONCE A EXPLICITER
* TRI
? NOM DU FICHIER OU SE TROUVE TRI
* essai1

MODE RESOLUTION    initialisation de l'explicitation
```

(? question du système, prompt * = réponse de l'utilisateur)

2)

```
! VISUALISATION DE L'ENONCE TRI
    $$ t  TRI  t  tq  equ(t,d)
                  et  croiss(t)

MODE RESOLUTION   Recherche de l'ident à expliciter
```

("!" décision du système)

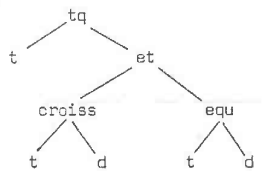
3)

```
! UN SEUL IDENT A EXPLICITER : t
! VISUALISATION DE LA DEFINITION DE L'IDENTIFICATEUR
! SQUELETTE ou TEXTE
* SQUELETTE

MODE RESOLUTION   Recherche des stratégies applicables
```

4)

ZENITH - LOCAL -
Graphique



MODE GRAPHISME

5)

```
! REGLES RETENUES : STRATX STRATY
? VISUALISATION  ARBRE-DES-CHOIX / REGLES
* non

MODE RESOLUTION    Choix de la stratégie à appliquer
```

6)

```
? ACCEPTES-TU CE SOUS ENSEMBLE DE REGLES : STRATX  STRATY
* Oui
? QUELLE STRATEGIE CHOISIS-TU
* STRATX

MODE RESOLUTION    Choix de la stratégie à appliquer
```

7)

! CHOIX de : STRATX
! RENOMMAGE DES VARIABLES FIGURANT DANS LA PARTIE RESULTAT
! INSTANTIATION DU MEMBRE DROIT DE LA REGLE

r == <u>der</u> u_i <u>pour</u> i <u>de</u> 1 <u>jqa</u> a_i
u_i <u>tq</u> R(u_i, tr(s,1,i))
a_i == (i = bs(s)

MODE RESOLUTION Application de la stratégie choisie

8)

```
! A PROPOS DU RESULTAT DE L'INSTANTIATION

      A = TRANSFORMATION ACCEPTEE
      B = TRANSFORMATION REJETEE
      V = VISUALISATION DU RESULTAT
      D = VISUALISATION DE L'IDENT EN COURS

* A
```

On suppose que le programmeur a tout d'abord tapé "V" ensuite par exemple "D" et enfin "A". Ce dernier ordre termine une transformation.

En tapant "V" le système affiche

```
t == der u_i pour i de 1 jqa a_i
t : TAB      a : SUITE-de-BOOL

u_i tq equ(u_i,tr(d,1,i)) et croiss(u_i)
u_i : SUITE-de-TAB     d : TAB     i : ENT

a_i == i = bs(d)

MODE RESOLUTION ... résultat de l'application de la stratégie
                    STRATX
```


En tapant "D" :

```
t tq  equ(t,d)  et croiss(t)
     t : TAB     d : TAB

MODE RESOLUTION      définition de l'identificateur t
```

Et en tapant "A" le système affiche :

```
! NOUVELLE ETAPE de L'EXPLICITATION
! VISUALISATION de l'ENONCE OBTENU
    $$  t  TRI
        t == der u_i pour i de 1 jqa a_i
        u_i tq eq(u_i,tr(d,1,i)) et croiss(u_i)
        a_i == i = bs(d)

MODE RESOLUTION  ETAPE 2   recherche de l'ident à expliciter
```

Supposons qu'à une étape 8) l'utilisateur décide de rejeter l'instantiation .
Alors en tapant "R" le système affiche :

```
                MENU DES POSSIBILITES

R = REMONTER DANS L'ARBRE
S = PROPOSER UNE REGLE POUR APPLICATION
M = MODIFIER UNE REGLE POUR APPLICATION
I = INTRODUIRE UNE REGLE ET L'APPLIQUER
V = VISUALISATION DU RESULTAT
O = VISUALISATION DE L'IDENT en COURS
L = VISUALISATION D'UNE REGLE
E = VISUALISATION DE L'ENONCE
```

A présent montrons le déroulement du système au cours d'une instantiation du membre droit d'une règle :

1)

```
! IDENTIFICATION du SECOND ORDRE RETENUE

        R(r,s)  ←
        equ(t,d)  et  croiss(t)

MODE RESOLUTION     instantiation des variables fonctions
```

2)

```
! IDENTIFICATION DES VARIABLES du PREMIER ORDRE

        r ← t
        s ← d

MODE RESOLUTION     instantiation des variables du premier
                    ordre
```


Dans le cas où l'utilisateur désire remettre en cause l'identification du second ordre proposée par le système, affichage 1) précédent, alors il doit attendre l'affichage 8) présenté plus haut, c'est-à-dire que le système présente cette identification comme un choix et l'applique.
Et dans le cas où il existe plusieurs manières de remplacer une variable de premier ordre par des variables effectives d'une énoncé à expliciter alors le système présente à l'écran par exemple l'image suivante :

```
! CHOIX DU REMPLACEMENT A EFFECTUER

1...        x ← t
2...        x ← d
3...        x ← s

MODE RESOLUTION     remplacement de la variable x
```

et l'utilisateur doit répondre par t, d, ou s.

Il faut noter que pour le moment le système ne procède à aucun contrôle de choix du "second ordre" ou du "premier ordre" proposé par le programmeur.

Enfin quant aux modes EDITION et CONNAISSANCE :
- le premier comme déjà dit au précédent chapitre ne nous intéresse pas,
- et le second permet à la fois la gestion de la bibliothèque et l'apprentissage des règles (cf. chapitre IV). Mais pour le moment il est plutôt orienté vers l'apprentissage :
  . introduction de nouvelles règles
  . création de classes règles.

Ce dernier mode est illustré dans l'annexe C.

En conclusion

Par l'illustration du fonctionnement du système faite précédemment on vient de montrer que celui-ci n'est pas propre à un contexte de problèmes particuliers :
Il peut s'adapter à différentes explicitations d'énoncés de problèmes distincts.
L'important étant alors :

1. de préciser le contexte dans lequel on exprime ces énoncés (comme on l'a fait avec le type TABLEAU pour lequel quelques exemples d'énoncés ont été donnés),

2. de définir les stratégies propres à la classe de problèmes choisie.

On peut remarquer qu'il existe des stratégies communes à différents problèmes à expliciter. Et là apparaît un aspect de l'apprentissage :

l'étude de stratégies globales communes à différentes classes de problèmes.

CHAPITRE VI

VI. SIE ET LES SYSTEMES TRANSFORMATIONNELS

Ce chapitre n'est pas une synthèse des différents systèmes transformationnels. Le lecteur trouvera une telle synthèse dans [AOM-81, GRE-81, GRE-84] et surtout dans [PAR-83]. Le but de ce chapitre est de comparer rapidement SIE et quelques systèmes transformationnels qui lui sont proches. Cependant, nous donnons brièvement les diverses approches sur la construction de programmes.

VI.1. Travaux sur les systèmes de construction de programmes

Les travaux sur les systèmes de transformations (ou de constructions) se différencient principalement par leur point de départ et leur mécanisme de construction. Par exemple l'approche suivie par SIE consiste à construire progressivement un programme à partir d'une spécification complète exprimée dans un langage fondé sur le CP1. Le passage de cette spécification, non algorithmique, au programme final est assuré par un mécanisme de transformations par applications de règles.

On trouve ainsi différents groupes de travaux concernant :

a) la synthèse de programmes à partir d'exemples ou de traces [JOU-84, BIE-76].
   Le point de départ est soit des exemples de couples données/résultats, soit une suite de calculs exprimés dans le langage du programme de départ. Et le mécanisme est une technique d'inférence inductive.

b) la vérification de programmes [IGA-73, TAM-80, WEG-76] où en fait l'aspect construction est ignoré. Le but est de prouver un programme par des assertions, ce processus de vérification est également un processus particulier de transformation.

c) les progiciels [LAM-83] où l'on a déjà un algorithme qu'il s'agit d'identifier. Ces logiciels se comportent comme des générateurs de programmes.

d) les systèmes fondés sur la logique des prédicats [CLA-81, KOW-79] où la spécification initiale est une formule du CP1 que le système transforme en un ensemble équivalent de clauses.

e) l'évolution de programmes [DERS-83, DERS-81] où un programme est construit par modifications de programmes déjà existants.

f) les environnements de programmations comme MENTOR-PASCAL [DON-80], MAIDAY [GUY-84], et en particulier AP [WAT-83, WAT-78], permettent d'aider le programmeur à exprimer des constructions algorithmiques.

g) la synthèse de programmes, à partir d'une spécification, fondée sur des règles et la démonstration de théorèmes [BID-80, 84, MAN-80, BRO-80] : Le point de départ est un algorithme précis, décrit sous une certaine forme, que l'on transforme de manière déductive en un programme.

h) la construction de programmes, à partir d'une spécification, fondée essentiellement sur des règles de transformations [ARS-82, BAR-79, BART-81, BAU-79b, BUR-77, BAK-80, FEA-82, GRE-84, GUI-79, PAG-83, SHA-80] :
La spécification initiale est décrite sous forme soit algorithmique ou non algorithmique. Et le mécanisme consiste essentiellement en l'application de règles devant conduire au programme final.

En outre on peut voir tous ces systèmes sur la construction de programmes comme appartenant à des systèmes généraux de développements de logiciels [LAM-83, ZEL-79]. C'est le cas par exemple de SIE puisque celui-ci est un composant du système Spes.

Enfin notons la naissance d'un nouveau groupe de systèmes de transformations : les systèmes fondés sur les systèmes de réécriture.
Cette optique est proposée par Dershowitz [DERS-84]. La méthode de transformation décrite par Dershowitz consiste à spécifier un énoncé initial d'un problème, $P(\bar{x},\bar{y})$, sous forme d'une règle de réécriture et de l'inclure dans un système de réécriture E caractérisant le domaine

du problème. On se donne aussi un ensemble d'équations H spécifiant les propriétés de P. La procédure de complétion est appliquée à E et H, ayant choisi un ordre bien fondé, jusqu'à ce qu'un algorithme A soit éventuellement généré. P est sous forme d'un système de réécriture constitué de termes primitifs. Il reste à transformer A en un programe. (le passage n'est pas évident).

Dans cette direction on peut citer [KOD-83] et REVE [LES-83]. Le premier apporte des compléments sur la méthode de complétion notamment sur la mise en oeuvre de stratégies devant permettre l'application de la "règle initiale" et le second peut être vu comme un outil de base pour cette approche dans la mesure où il repose sur l'algorithme de Knuth-Bendix.

VI.2. Présentation de quelques systèmes transformationnels

Nous avons choisi de présenter les systèmes suivants :
- PECOS développé à l'Université de Stanford par l'équipe de Barstow [BAR-79],
- ZAP développé à l'Université d'Edimburgh par l'équipe de Feather [FEA-82],
- CATY développé à l'Université d'Orsay par l'équipe de Guiho [GRE-84, GUI-83, GUI-80],
- DEDALUS développé par l'équipe Manna-Waldinger à Stanford [MAN-79],
- CIP développé à l'université de Munich par l'équipe de Bauer [BAU-81, BAU-79a, BAU-79b],

Ce choix est représentatif des différentes idées sur la construction d'une première version d'un programme par applications successives de règles de transformations.
La présentation de ces différents systèmes est telle qu'elle permet aussi de les comparer à SIE. Ainsi la comparaison s'appuie sur les critères suivants :

1. Le critère "règles-mécanismes" définit par
   - la forme de l'énoncé initial,
   - la forme de l'énoncé final,
   - la forme et la sélection des règles,
   - la méthode de transformation;
2. le critère "spécificité du système" définit par certaines particularités du système.

   Pour mieux comprendre ces critères appliquons les sur SIE, ce qui donne :
   - la spécification initiale est un énoncé non algorithmique,
   - la spécification finale est un énoncé itératif (applicatif),
   - les règles sont des stratégies d'explicitations. Certaines sont globales et jouent un rôle sémantique (les SEG), d'autres sont locales et sont plutôt syntaxiques (les SEL). La sélection de règles se fait par identification de la forme et la vérification d'une condition d'applicabilité ; parfois l'utilisateur aide le système à calculer une telle identification,
   - la méthode de transformation repose d'une part sur une méthode de programmation descendante à partir du résultat et d'autre part sur une technique de remplacements d'arbres abstraits. De plus, à chaque étape le choix des règles décidé par l'utilisateur porte sur un sous ensemble restreint.

   Et comme spécificités du système on retient que :
   - le système est ouvert à l'apprentissage de stratégies (possibilité par exemple d'introduire de nouvelles stratégies lors d'une explicitation),
   - le système est conçu pour s'appliquer à une classe de problèmes caractérisée par ses propres stratégies.

VI.2.1. Le systèmes PECOS

PECOS se comporte comme un expert du système PSI. Ce dernier permet essentiellement de construire la spécification initiale à partir de phrases en Anglais, et d'aider le programmeur à choisir le meilleur programme parmi tous les programmes Lisp produit par l'expert PECOS.

Par rapport à nos critères d'étude, PECOS présente les caractéristiques suivantes :

a) l'énoncé initial est une description algorithmique abstraite,

b) l'énoncé final est un programme Lisp.

c) On trouve 2 types de règles :
   - les règles liées au langage Lisp (construction de CAR, MEMBER, LONS ...),
   - les règles générales sur la connaissance des concepts de programmation (représentation d'une collection d'objets (une collection est soit un ensemble, soit un multiensemble), propriétés sur les collections, introduction d'une boucle while ...).

   Ces règles sont structurées en 3 classes :
   - règles de raffinement permettant de tendre vers une implémentation d'une structure de donnée ou d'une opération abstraite, (refine)
   - règles de propriété permettant de préciser la valeur d'un objet, (property)
   - règles requêtes liées à l'applicabilité des autres règles, (query).

Les règles sont locales et écrites en Lisp (sous forme de fonctions). Elles sont supposées correctes. Il n'existe pas de règles globales.

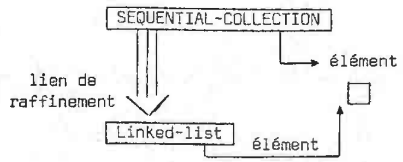
Donnons un exemple de règle :

```
(REFINE (SEQUENTIAL-COLLECTION                ]  nom
        (GET-PROP  element (BIND x)))         ]  premisse
        (NEW-NODE LINKED-LIST                 ]
        (SET-PROP  element x)))               ]  action
```

Cette règle dit qu'une collection séquentielle peut être raffinée en une liste chaînée.

SEQUENTIAL-COLLECTION est le nom du concept à raffiner. GET-PROP précise que la valeur de la variable x est liée à la propriété "élément". NEW MODE exprime la transformation de COLLECTION en LINKED-LIST. SET-PROP précise la propriété de NEW-NODE, c'est-à-dire "élément", et la valeur de cette propriété c'est-à-dire "x".

Cette règle est représentée par la structure suivante :



L'application de règles donne lieu à un réseau sémantique.
La sélection des règles est faite par PECOS et le choix par l'utilisateur sur un sous ensemble.

d) La méthode de transformation est basée sur un raffinement successif sur les données et sur les opérations abstraites.
Le raffinement est dirigé par 3 tâches :

-1- (refine n) : indique que la donnée ou la fonction au noeud n est à raffiner. Une règle de raffinement est alors appliquée.

-2- (property p n) : indique la propriété que l'on peut attacher au noeud. Elle intervient dans la sélection des règles de raffinement où sont précisées les propriétés (élément) attachées au noeud de départ (SEQUENTIAL-COLLECTION) et au noeud créé (LINKED-LIST),

-3- (query rel aig1 ...) : utilisée dans le cas de règles de raffinement ou de propriétés munies de pré-condition pour décider de leur applicabilité.

Donnons un exemple de transformation :

* Description abstraite initiale

Structure de données

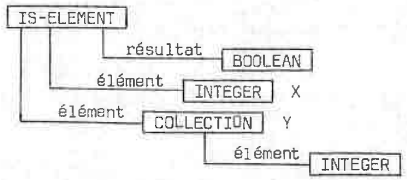
X un entier

Y une collection d'entiers

Algorithme

IS X un élément de Y

Cette description est représentée par le réseau : (elle est produite par PSI)



Etape 1. On précise si la collection est connue (explicite) ou "implicite" c'est-à-dire si elle est le résultat d'une opération.
Dans notre cas nous appliquons les règles :

une collection (resp. une opréation) peut être représentée explicitement. Ce qui donne :

IS-ELEMENT-EXPLICITE	X (ENTIER)	Y (COLLECTION EXPLICITE D'ENTIERS)

Etape 2. On choisit la manière dont doit être structurée une collection explicite. Par exemple le "rangement" (opposé à "distribuer") peut être considéré. On obtient alors :

IS-ELEMENT-RANGE	x (ENTIER)	Y (COLLECTION RANGEE D'ENTIERS)

Etape 3. Il s'agit de proposer une structure logique du rangement. On choisit le rangement séquentiel. On obtient :

IS-ELEMENT SEQUENCE	X (ENTIER)	Y (COLLECTION SEQUENTIELLE D'ENTIERS)

Etape 4. Il s'agit de donner une représentation physique d'une collection séquentielle. Elle peut être représentée par une liste chaînée :

IS-ELEMENT-LISTE	X (ENTIER)	Y (LISTE CHAINEE D'ENTIERS)

Etape 5. Il s'agit d'implémenter la représentation physique choisie précédemment. On choisit de représenter une liste chaînée en utilisant des cellules chaînées :

IS-ELEMENT-CELLULES	X (ENTIER)	Y (CELLULES CHAINEES D'ENTIERS)

Etape 6. Il s'agit maintenant de proposer une implémentation en Lisp. Des cellules chaînées sont représentées en une liste Lisp.

IS-ELEMENT-LISP	X (ENTIER)	Y (LISTE LISP D'ENTIERS)

Etape 7. PECOS propose une représentation de l'opération IS-ELEMENT-LISP par une fonction LISP, et des ENTIERS par des ENTIERS LISP :

MEMBER	X (ENTIER LISP)	Y (LIST LISP D'ENTIERS LISP)

De cette dernière proposition PECOS produit le programme Lisp : (MEMBER X Y). A chaque étape PECOS génère les tâches principales (refine n) et (property p n) en fonction de l'état du réseau sémantique. Ces tâches vont conduire à la sélection de règles par une simple identification des noms de noeuds figurant dans le réseau (IS-ELEMENT, COLLECTION) et des noms de propriétés (élément, résultat).
PECOS applique toutes les règles sélectionnées au cours d'une étape et l'utilisateur choisit avec l'aide de SPI la meilleure application. L'utilisateur peut également intervenir dans la sélection de règles pour lever d'éventuelles ambiguités.

Quant à l'aspect spécificité de PECOS on notera simplement :
- PECOS n'est pas conçu pour l'apprentissage de règles,
- PECOS peut traiter de grands programmes,
- il ne peut construire que des programmes itératifs.

Finalement PECOS a une base de connaissance très riche constituée de règles liées aux notions de programmation et du langage LISP. De plus toute strucutre de donnée est vue comme une "collection d'éléments" et toute opération comme une opération sur cette collection.
PECOS ne dispose pas d'outils de filtrage, d'induction ... ; cependant [BAR-84] propose un système PECOS doté de mécanismes déductifs.

VI.2.2. Le système ZAP

ZAP est fondé sur le système de Burstall-Darlington [BUR-77]. IL s'en différencie par les caractéristiques suivantes :
1. il accepte de transformer de gros programmes,
2. il propose une structuration du mécanisme de transformation,
3. il permet à l'utilisateur de diriger la transformation.

a) l'énoncé initial est une forme à instantier. Cette forme exprime des équations récursives en NPL [DAR-77].
Ces équations sont précisées dans un méta-programme servant à diriger la transformation (une séquence de commandes associée à l'énoncé initial).

b) L'énoncé est un programme NPL : l'énoncé initial instantié.

Un exemple de programme NPL :
la fonction Squares(l), "les carrés d'une liste", définie par :

```
Squares(l) = si l = nil alors nil
                         sinon cons(car(l) * car(l), Squares(cdr(l)))
```

(exprimée dans un pseudo-Lisp)
s'écrit en NPL sous forme de 2 équations récursives :

```
Squares(nil) ⇐ nil
Squares(cons(N,NUMLIST)) ⇐  cons(N * N, Squares(NUMLIST))
```

où cons est le constructeur du type List donné par la définition suivante :

```
list α  <= nil ++ cons(α, list α).
```

("++" sépare les différentes définitions de "list" précisant chacune un constructeur de type de la donnée). On peut noter alors que le langage NPL est fortement typé.

c) ZAP distingue les règles de transformations suivantes :
- une stratégie de transformation dite "ascendante" (bottom-up) : les fonctions de niveau le plus bas sont d'abord transformées, ensuite celles qui les utilisent et ainsi de suite,
- des règles dites "tactiques" qui sont :
  . la composition fonctionnelle ("combining"),
  . la combinaison fonctionnelle ("tupling"),
  . la généralisation ("embedding"),
- les règles de base à savoir :
  le pliage et le dépliage (Fold/unfold), l'instantiation, la définition, ainsi que des axiomes.

Les règles de base sont données sous forme d'équations récursives à instantier, et les stratégies et les tactiques sont exprimées sous forme algorithmique.

d) ZAP propose une hiérarchisation de la transformation de la manière suivante :

niveau le plus élevé → application d'une stratégie de transformation ("bottom-up")

passage manuel → ⇓ les tactiques de la transformation (combining, tupling ...)

passage assisté → ⇓ les transformation dirigées par la forme (filtage, application du méta-programme)

passage automatique → ⇓ les séquences de manipulation (instantiation, pliage, dépliage)

Le rôle d'une stratégie de transformation est de décomposer un problème sous forme d'un arbre d'appels de fonctions à transformer.(Chaque noeud est le nom d'une fonction). Ensuite on applique des tactiques sur cette structure dans le but de mettre en évidence les équations récursives à transformer. Ce qui nous conduit à la définition d'un méta-programme. A partir de ce dernier des règles de bases sont alors appliquées aux différentes définitions figurant dans ce méta-programme pour prouver leur correction.

Donnons un exemple de transformation.
On veut écrire un programme récursif calculant la somme des carrés d'une liste.
On divise notre problème en 2 sous tâches : Sum et Squares.
La définition de notre problème est caractérisée par : Sum(Squares(L)).
Supposons que les tâches Sum et Squares (voir exemple précédent) sont définies :

```
Sum(nil) <= 0
Sum(cons(N, NUMLIST)) <= N + Sum(NUMLIST).
```

La tactique "combining" suggère la définition d'une nouvelle fonction :

```
SumSquares(L) <= Sum(Squares(L)).
```

Le problème consiste à calculer une fonction, Foo, définie par Sum Squares (L).
L'utilisateur précise cela par le méta-programme suivant :


```
CONTEXT
  UNFOLD  Foo
  USING RESTRICTED Sum Squares
  TRANSFOR
      GOAL  Foo(L) <= $$(&&Sum Squares(L))
  END
END
```

CONTEXT ... END détermine le contexte de la transformation. (On peut avoir plusieurs contextes cela dépend de la manière dont le problème est décomposé).
UNFOLD Foo indique : application du dépliage sur l'équation Foo.
USING RESTRICTED Sum Squares dit que ces fonctions peuvent apparaître dans une nouvelle définition de fonction.
TRANSFOR ... END contient les transformations à effectuer. Chaque transformation est donnée par GOAL. Le membre gauche de "<=" est l'expression à transformer, le membre droit est appelé une "forme".
"&&" préfixe le nom que l'on veut donner à notre fonction, et "$$" peut être vu comme une variable fonction.
ZAP transforme ce méta-programme en un autre en s'appuyant sur les définitions de : "&&", Foo : list → Entier, et celle du type list.
On obtient :

```
CONTEXT
  UNFOLD SumSquares  Sum  Squares
  USING  SumSquares  (utilisation possible de SumSquares pour la
  TRANSFORM                                  transformation)
      GOAL SumSquares(nil) <= $$()
      GOAL SumSquares(cons(N,L)) <= $$(N,SumSquares(L))
  END
END
```

La partie gauche de "<=" ayant été déterminée il s'agit maintenant de calculer la partie droite.

Intéressons nous à la transformation du second but (GOAL). Elle se fait automatiquement comme suit :

membre gauche		forme
Sum Squares (cons(N,L))		$$ (N,SumSquares(L))
↓	dépliage SumSquares	
Sum(Squares(cons(N,L))		
↓	dépliage Squares	dépliage
Sum(cons(N*N), Squares(L))		
↓	dépliage Sum	
(N*N) + sum(Squares(L))		$$ (N,Sum(Squares(L)))

On vient de transformer le membre gauche

SumSquares(cons(N,L)) en (N * N) + Sum(Squares(L))

Pour exprimer cette transformation on instantie alors la forme

$$ (N,sum(Squares(L))) de façon à la rendre égale à

(N * N) + Sum(Squares(L)).

Ici la variable "$$" est donnée par la forme λ a b . (a * b) + b, et l'instantiation de cette fonction lambda donne (N * N) + Sum(Squares(L)). On a donc :

membre gauche		forme instantiée
(N * N) + Sum(Squares(L))	=	(N * N) + Sum(Squares(L))
↓	pliage	↓ pliage
(N * N) + Sum Squares (L)	=	(N * N) + Sum Squares (L)

Le résultat de la transformation de SumSquares est alors :

SumSquares(nil) <= $$()

SumSquares(cons(N,L)) <= (N * N) + SumSquares(L))

ZAP procède de la même manière pour transformer SumSquares(nil) :

SumSquares(nil) $\xrightarrow{\text{dépliage}}$ Sum(Squares(nil)) $\xrightarrow{\text{dépliage}}$ sum(nil) $\xrightarrow{\text{dépliage}}$ 0

La forme $$() est intantiée par 0, ce qui donne :

SumSquares(nil) <= 0

On peut remarquer que le filtrage est "faible" c'est-à-dire que l'identification d'une variable "$$" est directement donnée par le résultat de la transformation du membre gauche d'un GOAL.

Concluons par l'aspect spécificité de ZAP en notant que :

- ce système n'est pas orienté vers l'apprentissage de règles,
- les règles de bases et les tactiques sont des mécanismes de transformations paramétrés et on peut dire que les règles de ZAP sont des règles génériques,
- ZAP dispose d'une bibliothèque des définitions de fonctions et de types de données,
- ZAP peut transformer un programme NPL en un autre plus efficace, par exemple un programme récursif en un autre itératif,
- la hiérarchisation de la transformation élimine la notion de choix de règles, à chaque niveau l'utilisateur précise les règles à appliquer.

Finalement, relevons que ZAP transforme un programme "presque transformé". En effet, l'utilisateur définit son problème sous forme d'équations récursives et surtout il écrit le méta-programme qui caractérise en fait toute la transformation. Précisons cependant que l'on peut voir dans [DAR-81, 83] un pas vers la définition d'un système chargé de formuler le programme à transformer et aussi d'écrire les méta-programmes.

V.2.3. Le Système CATY

CATY a été conçu pour être intégré dans un environnement de programmation interactif. C'est un outil basé sur les types abstraits algébriques.

a) L'énoncé initial est une équation de la forme :

$$r \leftarrow E(x,y,z,...)$$

où E est le nom du programme, r le résultat de type R, et x,y,z,... appartiennent à des types X,Y,Z, ...

Tous ces types sont spécifiés dans une bibliothèque extensible de types abstraits.

b) L'énoncé final est un programme exprimé dans un langage proche de la syntaxe d'ADA. (essentiellement des procédures récursives et la structure conditionnelle).

c) Les règles de transformations sont données par des "schémas de décomposition" de types. Un schéma exprime "la structure algorithmique" d'un type.

CATY distingue 2 types de schémas :

- les schémas relatifs à un seul type abstrait,
- les schémas relatifs à un n-uple de types.

Un schéma de décomposition à un seul type est de la forme

```
cas P1(t) : t = C1(S1(t),S2(t),...)
    P2(t) : t = C2(s'1(t),S'2(t),...)
    ...           .....
    sinon : t = Cn(Sα(t),Sβ(t),...)
Fcas
```

où les Ci sont les constructeurs du type T, les S_j les sélecteurs et les P_k des prédicats. Si l'un des P_k est vrai alors tous les autres P_q sont faux et si aucun des P_k n'est vrai alors l'objet t est obtenu à partir du constructeur C_n.

Chaque type abstrait de la bibliothèque peut avoir 0, 1, ou plusieurs schémas.

Par exemple on peut avoir 2 schémas de décomposition pour le type séquence [entier] :

```
Schéma de décomposition CONS

Cas  videp(s) : s = vide
     sinon    : s = cons(deb(s),dern(s))
```

```
Schéma de décomposition APPEND

Cas  videp(s) : s = vide
     seul(s)  : s = unit(deb(s))
     sinon    : s = append(prem(s), deux(s))
Fcas
```

s est un objet du type séquence [entier]

La construction de ces schémas à partir d'un type est automatisable.

Un schéma à n-uple (complexe) fait intervenir plusieurs structures de données. Par exemple un schéma de décomposition complexe de la structure séquence [entier] - séquence [entier] est :

```
Cas  vidouble(x,y)    :  vide, vide
     videp(x)         :  vide, y
     videp(y)         :  x, vide
  membre(tête(x),y)   :  queue(x), supr(tête(x),y)
     sinon            :  x, y
Fcas
```

Ce schéma fait ressortir que 2 séquences peuvent être vides, que l'une des 2 peut être vide ou bien que lorsque aucune n'est vide le premier élément de x peut ou non appartenir à y. Dans le cas où l'on a "membre(tete(x),y)" on s'intéresse alors à la queue de x et à la liste obtenue en supprimant de y le premier élément de x. (x et y jouent le rôle de paramètres formels).

Cette manière de présenter ce schéma reflète bien un énoncé particulier sous une forme algorithmique : l'application de celui-ci permet de vérifier si 2 listes sont la permutation l'une de l'autre.

Les schémas complexes sont construits intuitivement.

Le choix des schémas à appliquer est laissé entièrement à l'utilisateur.

d) La construction d'un programme comprend deux phases :

- une première dite "traduction du problème", qui consiste à fournir à CATY l'énoncé initial. Cette phase n'est ni automatisée ni assistée elle est laissée entièrement à l'utilisateur.
- une seconde dite "découpage du problème", elle consiste à construire un programme à partir de l'énoncé initial. Ce processus de construction repose sur une démarche descendante par les résultats et les données.

La phase "découpage du problème" est mise en oeuvre par trois mécanismes de constructions particuliers qui sont des stratégies de construction :

- la stratégie "découpage transversalement aux programmes", elle consiste à faire une décomposition fonctionnelle de l'énoncé initial.

- la stratégie "découpage transversalement aux données", elle consiste à faire une décomposition des variables données ou de sortie.

- la stratégie "reprise de sous problèmes", elle consiste à caractériser un énoncé intermédiaire à construire.

Donnons un exemple de construction.
Soit le problème : trier une séquence par insertion.

a) traduction du problème par l'utilisateur

t ← sort(s) s,t : séquence

b) découpage du problème :

1. l'utilisateur choisit un découpage transversalement aux données. Il sélectionne la variable s,
2. application d'un schéma de décomposition, l'utilisateur a le choix entre le schéma CONS et le schéma APPEND (voir précédemment). Supposons qu'il choisisse le schéma CONS.

Ce schéma est appliqué à la variable s. On obtient :

```
PROCEDURE  SORT (t : OUT sequence ; s : IN séquence)
           IS
           $1 : entier ;
           $2 : sequence ;
begin
           if videp(s) then ... (1) ...
           else $1 := deb(s)
                $2 := dern(s)
                   ... (2) ...
           end if ;
END ;
```

La partie (1) de ce programme est construite de la façon suivante :
- CATY détermine l'ensemble des variables données d'entrée, les variables à calculer et les constantes associées au cas videp(s) : V = {vide, t}.
- t est la seule variable à calculer figurant dans V il introduit alors un nouveau sous-problème $3 et génère alors pour (1) :

t := $3(vide).

La partie (2) est construite comme suit :
- V = {$1, $ 2, t},
- puisque $2 est du même type que s et est inférieure à s relativement à un ordre bien fondé sur le type séquence alors CATY propose un appel récursif au programme SORT et crée une nouvelle variable $rec.

$rec := SORT($2)

L'ensemble V devient : V = {$1, $rec, t}. Les variables $1 et $rec ont déjà été calculées. Il reste alors à calculer t :

t := $4 ($1, $rec).

Toutes les variables ayant été calculées on revient alors aux choix d'une stratégie.
L'utilisateur choisit la stratégie "reprise des sous problèmes".
Le problème $3 dans t := $3(vide) désigne le constructeur tvide ($3 est d'arité 0). Alors l'utilisateur demande le renommage de $3 en tvide ce qui donne : t := tvide.
Le problème $4 dans t := $4($1, $rec) est l'opération insert : insérer un entier dans une séquence triée. Le renommage produit :

t := insert($1, $rec)

Si l'on suppose que insert et tvide ne sont pas connus de CATY alors $3 et $4 sont de nouveaux problèmes à construire.

L'utilisation intensive de graphismes facilite la construction d'un programme en offrant un support conceptuel. On peut regretter cependant l'absence d'aide à la construction de schémas complexes.
En outre les schémas utilisés ne permettent pas des transformations puissantes. Cela nécessite alors de disposer d'un grand nombre de schémas.
CATY est un système général, il peut traiter différentes classes de problèmes de taille réduite.
Enfin CATY assure la correction du programme après sa construction par un mécanisme de certification ayant en entrée une spécification formelle du problème. Ce mécanisme est en cours de développement.

VI.2.4. Le système DEDALUS

a) L'énoncé initial est une spécification non algorithmique exprimant dans un pseudo-Lisp la relation entre résultat et donnée.

b) L'énoncé final est un programme récursif exprimé par des constructeurs du langage Lisp. Ce programme final est alors prêt à être traduit en Lisp.

c) DEDALUS distingue les règles de transformations suivantes :
- les règles générales de programmation (introduction d'une conditionnelle, d'un appel récursif ...)
- les règles liées aux structures de données,
- les règles exprimant des propriétés sur un domaine,
- les règles logiques,
- des règles pour l'opérateur wp (Dijkstra [DIJ-76]).

Toutes les règles sont des programmes QLISP. Par exemple la règle exprimant une propriété sur le domaine des entiers :

```
U/V =>  true si U est un entier et V = 0 s'écrit en QLISP
(QLAMBDA   (DIVIDE ← U ← V)
           (INSIST (PROVE ('(INTEGER $U))))
           (INSIST (PROVE ('(EQUAL $V 0))))
           TRUE)
```

Le lecteur ne connaissant pas QLISP pourra cependant remarquer que cette représentation se rapproche fortement de son énoncé informel.
Le choix d'une règle est commandé par des stratégies. Ces dernières sélectionnent un sous ensemble de règles et les ordonne. A chaque fois que DEDALUS doit calculer la valeur de vérité d'une expression la sélection d'une règle se fait de la manière suivante :

1. sélection des règles par identification entre la partie gauche et l'expression (transformations dirigées par la forme),

2. application d'une stratégie d'ordonnancement : les règles sélectionnées en 1. sont ordonnées en fonction de la facilité à prouver une terminaison, du temps, de la complexité de la partie droite d'une règle, de l'introduction d'un constructeur Lisp.

3. Vérifications de certaines conditions stratégiques associées aux règles (par exemple pour un appel récursif f(t) on exige que t soit un objet Lisp).

d) La transformation est automatique et déductive. Elle utilise un démonstrateur de théorème. Le principe général de la transformation est :
l'application d'une règle à un énoncé initial implique la nécessité de prouver certaines expressions. On doit alors pour cela appliquer des règles jusqu'à ce que l'expression "vrai" soit produite. Si "faux" est produit le système de façon générale revient sur des règles précédemment appliquées (valeur arrière) pour en appliquer d'autres. Et dans le cas où ces retours échouent alors le système s'arrête.

La preuve des expressions permet d'effectuer et de valider la transformation de l'énoncé initial. Par exemple lors de l'application d'une règle :
1. si une condition ne peut pas être prouvée alors une analyse par cas est introduite,
2. lorsqu'une expression est une instance de l'énoncé initial, dans le cas où la terminaison est garantie, un appel récursif est introduit,
3. si une expression est une instance d'une expression plus générale on choisit d'introduire une procédure pour calculer l'expression plus générale.

Donnons un exemple simple de transformation.
Soit l'énoncé formel :

lessall(x l) ⇐ compute x < all(l)
where x est un entier et l une liste d'entiers.

Le programme lessall vérifie si un entier x donné est plus petit que tous les éléments d'une liste d'entiers l donnée.
L'expression compute ... est le résultat du problème, et where ... spécifie les données.

Le résultat de lessall est de la forme P(all(l)). Supposons que le système se trouve en présence des règles suivantes :

1) P(all(l)) ⇒ true si l est la liste vide

2) P(all(l)) ⇒ P(heard(l)) and P(all(tail(l))) si l est une liste non vide.

La première tâche du système est donnée par l'expression initiale et le choix d'une règle.

Goal 1 : compute x < all(l)

Supposons que le système choisisse la règle 1) précédente. Cette règle nous permet de réécrire notre expression par "true" à condition de prouver l'expression "l est une liste vide". Cela nous conduit à la seconde tâche suivante :

Goal 2 : prove l est une liste vide.

l est une donnée et rien n'indique si l est ou n'est pas vide. DEDALUS ne pouvant pas calculer la valeur de cette expression, applique la règle d'introduction d'une conditionnelle.

case "l est vide" au Goal 1

alors la règle 1) est applicable et on obtient :

```
lessall(x l) ⇐  si vide(l)
                alors true
                sinon  ...
```

case "l est non vide" au Goal 1

la règle 1) n'est pas applicable alors on essaie la 2) :

P(all(l)) ⇒ P(head(l)) and P(all(tail(l))) si l est non vide

L'application de cette règle nous conduit à la tâche :

Goal 3 : prove l est non vide

Cette preuve est immédiate puisque l'on suppose l non vide.

Le résultat de l'application de cette règle produit :

Goal 4 : compute x < head(l) and x < all(tail(l))

L'expression x < head(l) est primitive. Il reste donc à transformer x < all(tail(l)).

Goal 5 : compute x < all(tail(l))

DEDALUS détecte que l'expression de Goal 5 est une instance de celle de Goal 1. Alors il applique la règle d'introduction de la récursivité. La terminaison est prouvée relativement à un ordre bien fondé sur les listes : tail(l) ≺ l (on a identifié x à x et tail(l) à l - entre Goal 5 et Goal 1).

L'introdution d'une récursivité nécessite de prouver :

Goal 6 : prove tail(l) est une liste

On suppose disposer de la règle : tail(l) est une liste ⇒ true si l est non vide,

et Goal 7 : prove lessall(x tail(l)) termine

L'expression lessall(x tail(l)) est primitive. La preuve de sa terminaison est directe, on a bien : tail(l) ≺ l et tail(l) se réduit à une liste vide.

Le programme final s'écrit alors :

```
lessal(x l) ⇐  si vide(l)
               alors true
               sinon x < head(l) and lessall(x tail(l)).
```

DEDALUS dispose d'un grand nombre de règles lui permettant de construire des programmes "puissants". Et le catalogue de ces règles n'est pas structuré. Il ne fait pas de raffinements successifs sur les données.

DEDALUS est un système général (il permet la modification de programmes, la transformation de types abstraits). Sa structure est complexe, et il est peu performant (utilisation intensive du "backtracking", consultation de tout le catalogue à chaque tâche (Goal). Cependant il faut noter qu'il est le premier système de transformation opérationnel (expérimental) ayant en entrée une spécification non algorithmique.

VI.2.5. Le projet CIP

Le système de ce projet est basé sur le point de vue algébrique des types abstraits. Plus précisément, les auteurs de ce projet définissent un langage noyau CIP-L comme un type abstrait auquel s'ajoutent des règles de transformations représentant de nouveaux axiomes.

CIP-L est la base formelle du processus de transformation. Il comprend un niveau applicatif et un niveau procédural.

a) L'énoncé initial est un énoncé "pre-algorithmique", c'est-à-dire cette spécification utilise la logique des prédicats et toutes les opérations figurant dans l'énoncé appartiennent à des types abstraits.

b) L'énoncé final est un algorithme exprimé en PASCAL. (PASCAL peut être vu comme le niveau impératif de CIP-L).

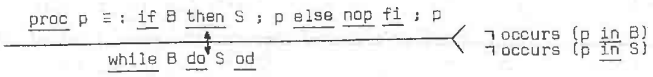
c) Les règles de transformations sont de 2 sortes :

- les règles de bases :
  - . les règles fondamentales (pliage et dépliage)
  - . les axiomes de types abstraits,
  - . les règles caractérisant des extensions du langage noyau (telles que les règles d'implémentation, de construction d'itération ...)
  - . Les règels d'inférence du calcul des prédicats.
- les règles dérivées utilisées pour construire des versions impératives du programme ou modifier des programmes.

Une règle est décrite comme un couple de schémas de programmes :

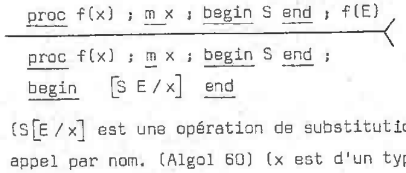
Exemples de règles

Un exemple de règle de base (elle est dite aussi définitionnelle) est :



Cette règle donne une autre définition équivalente d'une procédure P.

Un exemple de règle dérivée (on dit aussi règle sémantique) est :



(S[E / x] est une opération de substitution). Cette règle exprime un appel par nom. (Algol 60) (x est d'un type "m").

Les règles de transformations peuvent s'appliquer non seulement aux programmes, mais aussi aux schémas de programmes figurant dans la partie entrée d'une règle. Une telle application produit une nouvelle règle.

Les règles sont utilisées soit comme des axiomes, soit comme des théorèmes.

Il n'existe pas de stratégies de choix de règles. L'utilisateur à chaque étape de la construction choisit une règle du catalogue ou crée une nouvelle règle soit intuitivement soit en décidant de l'application d'une règle sur un schéma.

d) Le mécanisme de transformation permet de manipuler des schémas de programmes. Schémas et énoncés transformés sont représentés sous forme d'arbres abstraits et le mécanisme consiste à faire des remplacements d'arbres (ROSEN [ROS-73]).

Toute décision de transformation est prise par le programmeur et donne lieu à la définition d'une nouvelle règle ou à un choix d'une règle. Il n'existe pas de stratégies globales.

Finalement le système de transformation est fondé sur un point de vue algébrique de CIP-L étendu par des règles définitionnelles. Plus précisément, les auteurs du projet CIP définissent un langage noyau par une sémantique mathématique, et décrivent alors le noyau étendu (CIP-L) comme une hiérarchie de types de données abstraits algébriques où les règles de transformations spécifient la sémantique de CIP-L.

Ce système transforme à la fois des énoncés et des types abstraits.

L'utilisateur transforme formellement l'énoncé initial, en utilisant une méthode descendante.

A chaque étape il crée ou choisit la règle à appliquer, la valide et demande au système son application.

L'inconvénient majeur dans ce système est la compléxité de la condition d'applicabilité d'une règle.

Une caractéristique intéressante de ce projet est l'indépendance de la définition du système et de son implémentation de tout langage particulier. Le prototype actuel est très limité et un nouveau système de transformation est en cours de développement.

VI.3. Conclusion

La programmation transformationnelle existe grâce aux systèmes de transformations. Cette proposition est valable même si l'objectif des différentes équipes de travail en programmation transformationnelle est la formalisation rigoureuse de l'activité de développement de programmes.

Tous ces systèmes sont expérimentaux, inefficaces et ils ne traitent que des exemples jouets. Malgré des particularités spécifiques, on peut noter certains points communs :

- l'emploi d'une méthode descendante,
- l'utilisation des types abstraits,
- l'emploi des techniques d'identification, de filtrage,
- la construction d'un arbre de transformation précisant les différents choix,
- l'utilisation de la récursivité, de l'itération,
- la définition d'un catalogue de règles.

On peut aussi remarquer deux groupes de systèmes. Un premier privilégie les règles par rapport au mécanisme de transformation. Au contraire le second insiste sur les mécanismes de transformation.
Dans le premier cas, le mécanisme consiste à manipuler les règles, à mettre en oeuvre un principe d'induction, une technique de filtrage. C'est là où l'on situe PECOS, SIE, CIP, DEDALUS.
Dans le second cas, le mécanisme va plus loin que le précédent : il dispose, de façon générale, d'algorithmes de mise en oeuvre de règles, c'est-à-dire il ne se contente pas de les appliquer mais il en déduit des informations nécessaires à la transformation proprement dite. Ainsi dans ZAP et CATY un mécanisme de transformations particulier est associé aux différents types de règles.
Mais l'aspect le plus important de tout système de transformation est bien le problème de la découverte de règles, de stratégies globales.

Et à ce sujet au § III.1. nous avons voulu montrer notre manière intuitive de mettre en évidence des stratégies en voulant faire de SIE aussi un système d'apprentissage (nous nous appuyons sur une certaine classification de stratégies, les notions d'induction sur une suite, et d'invariants pour exprimer d'abord informellement des stratégies et ensuite pour les formaliser).
Les systèmes décrits précédemment ne sont pas ouvert à un tel apprentissage que nous estimons nécessaire afin de contribuer à une formalisation rigoureuse de la programmation transformationnelle.
Nous croyons que la programmation transformationnelle sera l'élément fondamental de la "programmation scientifique" et ce grâce aux efforts continus de développements de systèmes de transformations. En conséquence ces systèmes deviendront des produits industriels et donc ils pourront être intégrés dans les systèmes de développement de logiciels à venir.

CONCLUSION

CONCLUSION

I. RESULTATS

Nous venons de préciser une méthode d'explicitation d'énoncés et un système supportant cette méthode.
Cette démarche d'explicitation a fait apparaître le problème de la découverte de stratégies d'explicitations comme étant fondamental.
Nous croyons que notre système ouvert à l'apprentissage est un pas vers une solution à ce problème.
Le chapitre I présente cette démarche en précisant qu'elle repose sur les principes de la méthode de programmation déductive et surtout sur la notion de transformations par applications successives de stratégies. Et il relève l'importance des notions d'extensibilité et d'assistance à l'explicitation en tant que caractéristiques essentielles de cette approche d'explicitation étudiée dans les chapitres II et III.
Pour mener alors des expériences sur les mécanismes à la base de l'explicitation d'énoncés nous nous appuyons sur la mise en oeuvre et le développement des deux caractéristiques précédentes sur lesquelles le systèmes est bâti.
Les particularités de ce système, décrit au chapitre IV, sont :

- l'énoncé de départ est une spécification non algorithmique, exprimant en Spes le résultat du problème,
- l'énoncé explicite est un énoncé récurrent, exprimé en Médée, correct vis-à-vis de l'énoncé initial,
- le passage du premier énoncé vers le second se fait par l'application répétée de stratégies d'explicitations supposées valides.

De plus l'utilisation de ce système suppose que l'on se place dans une classe de problèmes donnée, comme par exemple celle des "TABLEAUX" présentée au chapitre V.
La principale contribution de notre système est donc de permettre une étude de l'explicitation d'énoncés. Une telle étude est rendue possible par SIE grâce :

1. à sa paramétrisation,
2. à sa capacité de construire "à la main",
3. à ses aides à l'explicitation.

La paramétrisation est un mécanisme d'introduction de nouvelles stratégies et de création de nouvelles classes de stratégies.

Durant le processus d'explicitation le programmeur peut faire appel au mécanisme précédent. Un tel appel lui permet de transformer "à la main" un énoncé en précisant à SIE la stratégie à appliquer.

SIE assiste le programmeur dans sa démarche en lui proposant un sous-ensemble de stratégies (problème du choix de la stratégie), en entretenant un arbre des choix, et en le libérant de certains calculs (application et sélection de stratégies, substitutions particulières). En outre sa démarche est guidée par un dialogue structuré par une technique de menus.

Notre travail veut donc apporter une contribution aux recherches menées sur la construction systématique, voire automatique, de programmes à partir d'une spécification de problème ne contenant pas nécessairement un mode de calcul.

II. NOTES SUR LA REALISATION

Notre système, SIE, est implenté en Franz Lisp (Liszt) [FOD-83].
Ce système dispose actuellement de stratégies propres à la classe de problèmes sur les "TABLEAUX". Leur représentation a été discutée au § IV. Certaines (les SEL) sont des expressions listes et d'autres (les SEG) sont des programmes Liszt.
SIE compte environ 1900 lignes Liszt (non compris les modules caractérisant le mode Edition).

Les modules fondamentaux du système sont IDENTIFIEUR (calcul du filtre d'identification σ) et TRANSFORMATEUR (calcul du résultat de l'application d'une stratégie). Leur implémentation est telle que le module IDENTIFIEUR sert à la fois à SELECT-S (sélection de stratégies) en rendant une valeur booléenne (échec ou réussite) et à TRANSFORMATEUR en rendant σ. Cela a été précisé au § IV.3.4.

De façon générale SIE n'est pas très efficace. Certes des améliorations du code actuel sont possibles mais celles-ci ne suffiront pas à diminuer sensiblement son inefficacité qui est due essentiellement à la complexité des stratégies à appliquer.

Notons en outre qu'il est préférable de donner naissance à un système inefficace que d'écrire une maquette. Le premier est destiné à évoluer alors qu'une fin tragique est réservée au second. Et lors de la conception d'un système il est souhaitable d'abord de mieux maîtriser les programmes fondamentaux (par des commentaires, des annotations particulières, des tests individuels) et ensuite de s'intéresser à l'efficacité du système et à le tester dans son ensemble.

Enfin, revenons à SIE en disant que celui-ci a été conçu pour évoluer en laboratoire et non dans un milieu industriel. Cependant il pourrait servir d'outil pédagogique.

III. PROLONGEMENTS : Vers un Système Automatique d'Explicitation

Nous avons signalé quelques extensions possibles de SIE (cf. plus particulièrement § IV.4.). Parmi toutes ces extensions nous retiendrons principalement les développements sur :
(a) - le mécanisme de paramétrisation,
(b) - un mécanisme d'apprentissage,
(c) - la technique de remplacements.

D'un point de vue pratique on veut développer

(a) un système d'interrogation des connaissances dans le but d'aider le programmeur à définir des stratégies,

(b) un système de construction de stratégies dans le but de mieux les maîtriser, (cf. § III.1.3.), et

(c) rendre notre technique de remplacements moins restrictive (§ II.4.4.) dans le but de permettre des transformations plus puissantes.

Les études (a) et (b) sont fortement liées. La première peut contribuer à analyser (b). Et si l'on écrit alors un système de construction de stratégies, le mécanisme de paramétrisation pourra être vu comme étant intégré dans ce dernier système.
Ceci nous conduit à essayer de donner un point de vue théorique à la construction de stratégies (b) et à la méthode de transformations (c).

L'objectif visé par une étude théorique de la technique de remplacements est la définition des "limites" de notre transformation d'énoncés. Pour cela on pense à l'adaptation du formalisme de [ROS-73] aux calculs de transformations d'ordre plus élevé.
Le système de remplacements de sous arbres défini dans [ROS-73] est en fait un système de réécriture de premier ordre. Notre but est d'utiliser le formalisme employé par Rosen pour définir des remplacements moins restrictifs, plus forts en précisant certaines propriétés autorisant de tels remplacements.

Notre étude théorique sur la découverte de stratégies est orientée de manière à résoudre d'une certaine façon le problème de leur validité par SIE, et également de tendre vers une définition formelle de leur découverte. Cette orientation est donnée par l'adaptation des travaux sur la théorie et les méthodes d'inférences inductives [ANG-83] à l'explication d'énoncés. Notre but est de déterminer des procèdures d'inférence inductive appliquées à la construction de programmes itératifs. Ces procédures devraient conduire à la définition d'une méthode d'apprentissage de découverte de stratégies et à la conception d'un système de construction de stratégies.

Et on veut alors qu'au fur et à mesure que toutes ces études théoriques et pratiques avancent rendre SIE de plus en plus automatique.
Certes beaucoup de travaux insistent davantage sur l'évolution de programmes [DERS-83 , DON-83, WAT-78]. Cependant nous sommes convaincus que l'on peut passer progressivement du logiciel SIE à un logiciel SAE. Pour être plus précis nous croyons au passage de la construction assistée d'un programme à partir d'une spécification de problème, ne contenant aucune information sur sa résolution, à sa construction automatique. A condition que l'on se restreigne à une classe de problèmes.

Notre proposition consiste alors à tendre vers l'explicitation automatique d'énoncés d'une classe de problèmes donnée, caractérisée par des stratégies propres, en des énoncés itératifs corrects.

Finalement, soulignons que l'explicitation s'appuie sur des choses précises (§ III.1.) qu'il faut savoir reconnaître. Pour cela on travaille par apprentissage vu comme l'outil de base des études mentionnées précédemment. Et en parallèle on écrit des mécanismes de constructions de stratégies. Ainsi on devrait aboutir à la définition d'un système automatique de construction d'énoncés itératifs.

ANNEXES

A N N E X E - A -

UN EXEMPLE D'APPLICATION DE REGLES DE CONSTRUCTIONS

a) Spécification du problème

L'indice du plus grand élément d'un tableau

Lexique	Profils	énoncé formel e_r
r l'indice résultat d le tableau donné	e : ENT d : TAB	résultat r r tq $\forall$ k [1..bs(d)] d[r] $\geqslant$ d[k] donnée d

b) Explicitation du problème

but : expliciter e_r

choix de la règle de construction à appliquer : ITERC, SIMPC, CONDC, PREMC

choix de : ITERC $\quad e_r$ :: r == der u_i pour i de 1 jqa a_i, e_u, e_a

instantiation de ITERC :

choisissons :

e_{u_i} : u_i tq $\forall$ k [1..bs($d1_{i-1}$)] $\Rightarrow$ d[u_i] $\leqslant$ d[k]
et k = i $\Rightarrow$ d[u_i] $\leqslant$ d[i]

e_{a_i} : a_i == (i = bs(d))

prouvons :

a_i = vrai et l = μ(i) et e_{u_i}[i $\leftarrow$ l] $\Rightarrow$ e_r [r $\leftarrow$ u_l]

immédiat.

* On calcule u_i par récurrence.

<u>but</u> expliciter e_{u_i}

<u>choix de la règle de construction à appliquer</u> : ITERC, SIMPC, CONDC, PREMC

<u>choix de</u> : CONDC

e_{u_i} :: u_i == <u>si</u> b_i <u>alors</u> r'_i <u>sinon</u> r''_i

(b_i <u>et</u> $e_{r'_i}$) <u>ou</u> (<u>non</u> b_i <u>et</u> $e_{r''_i}$)

<u>instantiation</u> de CONDC

<u>choisissons</u> :

$e_{r'_i}$: r'_i <u>tq</u> $r'_i = u_{i-1}$

$e_{r''_i}$: r''_i <u>tq</u> $r''_i = i$

e_{b_i} : b_i <u>tq</u> $b_i = d[u_{i-1}] \leqslant d[i]$

<u>prouvons</u>

b_i <u>et</u> $r'_i = u_{i-1} \Rightarrow e_{u_i}[u_i \leftarrow r'_i]$

<u>non</u> b_i <u>et</u> $r''_i = i \Rightarrow e_{u_i}[u_i \leftarrow r''_i]$

la preuve ici ne présente aucune difficulté

<u>but</u> expliciter e_{u_o} (e_{u_i} est un invariant)

<u>choix de la règle de construction</u> : SIMPC, ITERC, PREMC, CONDC

<u>choix de</u> : SIMPC e_r :: r == $f(\bar{x})$ et e_{x_1}, ..., e_{x_n}

<u>instantiation de</u> SIMPC

<u>choisissons</u>

u_o == 1

<u>prouvons</u>

$u_o = 1 \Rightarrow e_{u_i}[i \leftarrow 0, u_o \leftarrow 1]$

cette preuve est immédiate

<u>Commentaire</u>

Cet exemple montre bien que l'utilisateur à chaque application d'une règle est en présence de plusieurs explicitations possibles ou que le système a une infinité de façon d'instantier une règle.
Et en outre lors d'une instantiation choisie il est nécessaire de faire une certaine preuve justifiant ainsi l'application d'une règle.

ANNEXE -B-

GRAMMAIRE Spes UTILISEE

Nous utilisons la notation BNF de Backus.

<Spes> ::= <Enonce>

<Enonce> ::= <Entete> <Corps>

<Entete> ::= nom du résultat NON-DE-L'ENONCE

<Corps> ::= <DEFINITION> <CORPS>

<DEFINITION> ::= <Deftype> | <Defimplicite> | <Defexplicite>

Syntaxe d'une définition implicite

<Defimplicite> ::= <Listes-résultat> tq <Prédicat>

<Listes-résultat> ::= <lettres minuscules> ! <Listes-résultat>

<Prédicat> ::= <Prédicat-Simple> | qq <Deftype> !! [<Prédicat>]

<Prédicat-Simple> ::= <Prédicat> => <Implicant> |
<Prédicat> <=> <Implicant> |
<Implicant>

<Implicant> ::= <Implicant> ou <Terme> | <Terme>

<Terme> ::= <Facteur> | <Terme> et <Facteur>

<Facteur> ::= non [<Prédicat>] | <exp-p>

<Exp-p> ::= <Exp-Simple> <Op-Comp> <Exp-simple> | <Exp-Simple>

<Exp-Simple> ::= + <Op> | - <Op> | <Exp-Simple> <Op-prior-c> <Op>

<Op> ::= <Fact-p> | <Op> <Op-prior-b> <Fact-p>

<Fact-p> ::= <Prim-p> | <Prim-p> <Op-prior-a> <Prim-p>

<Prim-p> ::= [<Prédicat>] | <Entité>

<Entité> ::= <Constante> | <Variable> | <Appel> | <Variable indicée>

<Appel> ::= <identificateur> [<Expr-lst>] | <identificateur> [ ]

<Expr-lst> ::= <Prédicat>

Syntaxe d'une définition explicite

<Defexplicite> ::= <identificateur-résultat> == <Formule>
<Formule> ::= <Conditionnelle> | <Iteration> | <Implicant>
<Conditionnelle> ::= <si-Lst> sinon <Exp-p> | <Si-Lst>
<Si-Lst> ::= <Si-partie> | <Si-partie> sinon <si-Lst>
<Si-partie> ::= si <Implicant> alors <Exp-p>
<Itération> ::= der <Var> pour <variagle> de <Expr> jqa <Exp-p>
<Var> ::= <Variable> | <Variable indicée>
<Expr> ::= <Exp-p> (en changeant <Prim-p> par <Pirm-p> ::= <Entité>

Syntaxe d'une définition de type

<deftype> ::= <Liste-type> | <Liste-type> <deftype>
<Liste-type> ::= <Variable> : <sorte>

A N N E X E - C -

UNE ILLUSTRATION DU MODE CONNAISSANCE

Cette annexe illustre un exemple d'introduction d'une règle dans le catalogue

1)

```
I = introduction d'une règle  (stratégies locales)
S = supprimer une règle
V = visualisation d'une règle
L = liste de tous les noms de règles
Q = quitte
* I
MODE CONNAISSANCE  définition de stratégies
```

(* réponse de l'utilisateur)

2)

```
! DONNER NOM DE LA REGLE avec sa CLASSE :
            (nom  classe)
! CLASSES CHOIX ENTRE L(logic), B(valeur),
      T(terminale), P(prop), N(n-ter)
* (stratd N)
MODE CONNAISSANCE  définition de stratégies
```

3)

```
? UNE PRE-CONDITION
* non
! INTRODUCTION DE LA FORME
! INTRODUIRE UNE DEF IMPLICITE
* t tq R(t,d) et R1(t) &
! INTRODUIRE LA DEF TYPE
* t : TAB  d : TAB &
MODE CONNAISSANCE  définition de la forme
```

4)

```
! INTRODUCTION DU RESULTAT
! CHOIX ENTRE DEF IMPLICITE ou EXPLICITE
* EXPLICITE
* t == der u_i pour i de 1 jqa a_i
? UNE DEF TYPE
* non
MODE CONNAISSANCE   ... partie résultat ...
```

Cette dernière étape se répète autant de fois qu'il existe de définitions à introduire.

5)

```
? UN POST-CONDITION
* non
? REGLE A ENREGISTRER
  vrai
  t tq R(t,d) et R1(t)
* oui
MODE CONNAISSANCE  définition de stratéties
```

Après ce dernier affichage le système retour au menu 1).
(l'annexe E montre la manière dont une règle est visualisée)

A N N E X E - D -

VISUALISATION DE L'ARBRE DES CHOIX et des STRATEGIES

1) Visualisation de l'arbre des choix

La visualisation de l'arbre se fait soit à la demande de l'utilisateur doit de façon automatique. Nous illustrons l'arbre des choix de la figure IV.2. (chapitre IV)

a)

```
ARBRE DES CHOIX
                    I
                    I E, initial
I stratd1           I strata1
I t, stratd         I t, strata
(nom règle)     =   visualisation
(P index)       =   remonter
(F index        =   descendre
(ident index)   =   zoom
 q              =   quitte
* (F strata1)
* (t, strata1)
MODE CONNAISSANCE  ... arbre ...
```

vacing

index = stratd1, strata1
l'arbre montre que l'utilisateur a d'abord appliqué stratd à t et ensuite renoncé à ce choix pour appliquer strata à t

(voir écran b))
(voir écran c))

b)

```
ARBRE DES CHOIX
            I strata1
            I t,strata
  I sac1
  I u_i,sac1
   ....
MODE CONNAISSANCE   .....
```

c)

```
! LISTE des identificateurs introduits

  u_i, a_i, t

  ident = visualisation de la définition

  r = visualisation de la définition initiale du résultat

  q = quitte

* r

  t tq ssuite(t,d) et ∀ k [1..bs(d)] t[k] > 0
       et bs(t) = nbrintq(d,d[1] > 0)

* t

  t == der u_i pour i de 1 jqa a_i

MODE CONNAISSANCE    ... arbre : zoom ...
```

2) Visualisation d'une stratégie locale

L'écran peut ne pas suffire à afficher toute une SEL alors le système propose un affichage par partie.

a)

```
REGLE strata

    P = Pré-condition
    F = Forme
    R = Résultat
    C = Post-condition
    V = Affichage écran

* V

MODE CONNAISSANCE   visualisation progressive de strata
```


b)

```
* Pré-condition
    vrai
* Forme
  r tq R(t,s) et ∀ l P(r[l])
  r : TAB   d : TAB
* Résultat
  t = der u_i pour i de 1 jqa a_i
  u_i tq R'u_i d1_i) et P(u_i[i])
  a_i == i = bs(d)
* Post-condition
    vrai

MODE CONNAISSANCE   ... strata ...
```

A N N E X E - E -

UN EXEMPLE D'UNE SESSION DE TRAVAIL

Nous avions déjà montré au § V.3. en quoi consiste une session de travail.

Cette annexe illustre une session complète de travail en considérant l'exemple de la figure IV.2.

1)

```
        MENU MODES

esc1 = MODE EDITION
esc2 = MODE CONNAISSANCE
esc3 = MODE RESOLUTION
esc4 = QUITTE
* esc

MODE ?  édition, connaissance, résolution
```

2)

```
? NOM DE L'ENONCE à EXPLICITER
* SUITE
? NOM DU FICHIER OU SE TROUVE SSUITE
* essai1

MODE RESOLUTION   initialisation de l'explicitation
```

3)

```
! VISUALISATION DE L'ENONCE SSUITE
  $$ t tq ssuite(t,d) et ∀ k [1..bs(d)]
          t[k] > 0 et bs(t) = nbrintq(d,d[1] > 0

MODE RESOLUTION 1 recherche de l'ident. à expliciter
```

4)

```
! UN SEUL IDENT A EXPLICITER : t
! VISUALISATION SQUELLE OU TEXTE OU QUITTE
* QUITTE

MODE RESOLUTION 1 recherche de l'ident. à expliciter
```

5)

```
! CHOIX de la CLASSE

    C = Analyse par cas, sac
    I = Initialisation
    R = Instantiation
    N = Stratégies non terminales
    L = Logiques
    B = Valeurs
    P = Propriétés
    M = Nouvelles définitions de l'ident.
* N

MODE RESOLUTION  ... opération à exécuter ...
```

6)

```
! REGLES RETENUES : strata  ....
? VISUALISATION  ARBRE CHOIX/REGLES
* non
? ACCEPTES-TU CE SOUS-ENSEMBLE
* oui
? QUELLE STRATEGIE CHOISIS-TU
* stratat

MODE RESOLUTION 1 choix de la stratégie à appliquer
```

7)

```
! CHOIX de : strata
! RENOMMAGE DES VARIABLES
! INSTANTIATION du MEMBRE DROIT de la REGLE
  r == der u_i pour i de 1 jqa a_i
  u_i tq R(u_i, tr(s,1,i))
  a_i == (i = bs(s))
MODE RESOLUTION  Application de la stratégie
```


8)

```
! IDENTIFICATION des VARIABLES de PREMIER ORDRE
      r ← t
      s ← d
      l ← k

MODE RESOLUTION   instantiation du premier ordre
```

9)

```
! IDENTIFICATION du SECOND ORDRE
      R(r,s) ← ssuite(t,d)
      P(r[l]) ← t[k] > 0

MODE RESOLUTION   instantiation du second ordre
```

10)

```
! RESULTAT
  t == der u_i pour i de 1 jqa a_i
  u_i tq ssuite(u_i, d1_i) et u_i [i] > 0
  a_i == i = bs(d)

MODE RESOLUTION   application de strata sur t
```

11)

```
! A PROPOS DU R2SULTAT DE L'INSTANTIATION

    A = TRANSFORMATION ACCEPTEE
    B = TRANSFORMATION REJETEE
    V = VISUALISATION DU RESULTAT
    D = VISUALISATION DE L'IDENT. EN COURS
* A

MODE RESOLUTION    ... fin d'une transformation ...
```

12)

```
! NOUVELLE ETAPE
! VISUALISATION DE L'EONCE OBTENU
  $$ t SSUITE
     t == .... (ecran 9)
  et bs(t) = nbrintq(d,d[l])

MODE RESOLUTION 2   recherche de l'ident. à expliciter
```

L'utilisateur à présent explicite u_i : considérons alors directement l'écran suivant :

13)
```
! CHOIX de la CLASSE
   ..... (voir écran 5)
* sac

MODE RESOLUTION     opération à expliciter
```

14)
```
! PROPOSER DES DEFINITIONS
  Si □... (1)
     alors .... (2)

MODE RESOLUTION    analyse par cas  u_i == .... ?
```

("□" = curseur)

L'utilisateur propose successivement :

(1) = $d[i] > 0$, (2) = $concat(u_{i-1}, di_i)$

15)
```
? une partie sinon, d'autre cas, fin
* sinon

MODE RESOLUTION 2   analyse par cas  u_i == ...
```

16)
```
! PROPOSER DES DEFINITIONS
  u_i == si  d[i] > 0
          alors concat(u_{i-1}, di_i)
                sinon □... (3)

MODE RESOLUTION 2   analyse par cas  u_i == ...
```

((3) = u_{i-1})

17)
```
! FIN ANALYSE PAR CAS
  u_i == si  d[i] > 0
          alors concat(u_{i-1}, di_i)
          sinon u_{i-1}

MODE RESOLUTION 2   résultat ....
```


18) A présent l'utilisateur explicite u_o. Il choisit <sei> (initialisation) ce qui donne :

19)
```
? CHOIX d'un PREDICAT
  1 ... ssuite(u_o, d1_o)
  2 ... u_o[0] > 0
* 1

MODE RESOLUTION 2   initialisation
```

20)
```
! PROPOSITION
! u_o == vide
? ACCEPT
* oui

MODE RESOLUTION 3   initialisation
```

Ce dernier écran termine l'explicitation alors le système affiche

21)
```
! RESOLUTION DE L'EXPLICITATION
  t == der u_i pour i de 1 jqa a_i
  u_i == si d[i] > 0 alors
                      concat(u_{i-1}, di_i)
                    sinon
                      u_{i-1}
  u_o == tvide
 u : SUITE-DE-TAB   t : TAB  d : TAB

 MODE RESOLUTION ...  FIN de l'explicitation
```

22)
```
     r = retour au menu MODES
     a = voir arbre des choix

 MODE RESOLUTION ...  FIN, que puis-je faire ?
```

A N N E X E - F -

ALGORITHMES DE FILTRAGE ET DE TRANSFORMATION

a) L'algorithme "testform" ci-dessous est le noyau du module IDENTIFIEUR.

Le programme "testforme" est écrit d'une façon simple. Il utiliser les fonctions suivantes :

- "&donevar" qui rend les variables de premiers ordres et les fonctions définies figurant comme paramètres de la variable du second ordre désignée par "R".

- "doneap/var" est identique à la précédente mais travaille sur la définition implicite effective.

- "sig-feuil" indique si l'on peut identifier toutes les variables de premiers ordres précédemment calculées aux variables effectives.

Cet algorithme est utilisé par le module "TRANSFORMATEUR", et dans ce cas là "Testforme" retourne la valeur de "sigmavar", c'est-à-dire le filtre d'identification.

```
(def testform
  (lambda (en-depart forme-str)
    (prog nil
          (cond ((null forme-str) (return t))
                ((null en-depart) (return nil))
                ((equal (car (explode (caadar forme-str))) 'R)
                 (setq lesop nil)
                 (setq varen
                       (donevar
                        (cdr (extr.tree (caar en-depart) defimplob))))
                 (setq lesop/s nil)
                 (setq varstr
                       (&donevar (cdr (extr/tree (caar forme-str) forme))))
                 (cond ((sig-feuil varstr varen)
                        (cond
                         ((not (zerop i/t))
                          (setq sigmaop
                                (append sigmaop
                                        (list
                                         (list (caadar forme-str)
                                               (list (caar en-depart)))))))
                        (return
                         (testforme (eliminel (caar en-depart) en-depart)
                                    (eliminel (caar forme-str) forme-str))))
                       (t (return nil))))
                ((member (lettre (explode (caar forme-str)))
                         '(vari vindi))
                 (cond ((zerop i/t)
                        (return
                         (testforme (eliminel (caar en-depart) en-depart)
                                    (eliminel (caar forme-str) forme-str))))
                       (t (cond ((setq sig
                                       (assoc (list (caadar forme-str))
                                              sigmavar))
                                 (setq sigmavar
                                       (subst (list
                                               (list (caadar en-depart)
                                                     (caar en-depart)))
                                              (cadr sig)
                                              sigmavar)))
                                (t
                                 (setq sigmavar
                                       (append sigmavar
                                               (list
                                                (list (list
                                                       (caadar
                                                        forme-str))
                                                      (list
                                                       (list (caadar
                                                              en-depart)
                                                             (caar
                                                              en-depart)))))))))
                        (return
                         (testforme (eliminel (caar en-depart) en-depart)
                                    (eliminel (caar forme-str)
                                              forme-str)))))))
```

/Algorithme de filtrage

b) L'algorithme "instancvar" ci-dessous est la fonction principale du module "TRANSFORMATEUR".

Le programme "instancvar" est itératif puisque nous limitons la profondeur de la transformation à un niveau (et aussi pour des raisons de dépassements de capacité d'empilement).

Notons que dans ce programme :

- sigma1 désigne l'ensemble des substitutions de premier ordre et sigma2 celles du second ordre, et tree (tree-strat) est la partie résultat de la règle.

- la partie (1) effectue les remplacements de premier ordre,

- la partie (2) effectue les remplacements du second ordre,

- la partie (3) remplace les intermédiaires par leur valeur effective,

- la partie (4) permet à l'utilisateur d'introduire des intermédiaires : remplacement des variables du second ordre figurant dans la partie résultat de la règle mais pas dans sa partie forme.

```
def instancevar
 (lambda (sigma1 sigma2 tree)
  (prog (sig)
   ins  (cond ((null tree) (return t))
              ((setq sig (assoc (caadar tree) sigma1))
               (cond ((member (lettre (explode (caar tree)))
                              (dtyp vari))
                      (rplaca (cadar tree) (copy (caadr sig)))
                      (setq tree (cdr tree))
                      (go ins))
                     ((equal (lettre (explode (caar tree)))  vindi)
                      (replace tree-strat
                               (&subst-ar (extr/tree (cadadr sig)
                                                     defimplob)
                                          (extr/tree (caar tree)
                                                     tree-strat)
                                          tree-strat))
                      (setq tree (eliminel (caar tree) tree))
                      (go ins))
                     (t (setq tree (cdr tree)) (go ins))))
              ((setq sig (assoc (caadar tree) sigma2))
               (setq sigl
                     (&subst (copy (extr/tree (caadr sig) defimplob))
                             (cdr (extr/tree (caar tree) tree))
                             (cdr
                              (extr/tree (get-pere (caadar tree) forme)
                                         forme))))
               (do ((x sigma1 (cdr x)))
                   ((null x))
                   (replace sigl (subst (caadar x) (caar x) sigl)))
               (replace tree-strat
                        (&subst-ar sigl
                                   (extr/tree (caar tree) tree-strat)
                                   tree-strat))
               (setq tree (eliminel (caar tree) tree))
               (go ins))
              ((equal (car (explode (caadar tree)))  R)
               (patom "! INTRODUIRE UN INTERMEDIARE ")
               (setq ar  nil)
               (replace tree-strat
                        (&subst-ar (cddddr
                                    (&strat-def 'impl
                                                '(fimpl)))
                                   (extr/tree (caar tree) tree-strat)
                                   tree-strat))
               (&autredef))
              (t (setq tree (cdr tree)) (go ins))))))
```

(1) (2) (3) (4)

Algorithme de transformation

BIBLIOGRAPHIE

[ADM-81] A. ADAM Programmation automatique, Synthèse de programmes. Colloque I.A. de Toulouse, 6-10 juillet 1981.

[AMY-78] B. AMY, M. CAPLAIN Invariances algorithmiques et récurrences. 3ème colloque international sur la programmation, DUNOD, 1978.

[ANG-83] D. ANGULIN, C.H. SMITH Inductive Inference : Theory and Methods. Computing Surveys, vol 15, n° 3, sept. 1983.

[ASH-75] E.A. ASHCROFT, W.W. WADGE Lucid : a system formel for writing and proving programs. SIAM J. Comput. 5, 3, septembre 1975.

[ASH-77] E.A. ASHCROFT, W.W. WADGE Lucid a non procedural language with iteration. CACM, July 1977 vol. 20, n° 7.

[ARS-77] J. ARSAC La construction de programmes structurés. Dunod, Paris 1977.

[ARS-82] J. ARSAC, J. KODRATOFF Some techniques for recursion removal from recursive fonctions TOPLAS 4, 2, 1982.

[BAK-80] R.J. BACK Semantic correction of invariant based programs International Workshop on Program Construction. Château de Bonas, septembre 1980, INRIA.

[BAL-81] R. BALZER Transformational Implementation : An Example. IEEE, vol SE-7, n° 1, Janvier 1981.

[BAR-79] D.R. BARSTOW a) Knowledge-Based Program Construction. The computer science Library, Ed. E CHEATHAM, 1979. b) An experiment in Knowledge-based automatic programming A.I. vol 12, n° 2, août 1979.

[BAR-84] D.R. BARSTOW The roles of Knowledge and Deduction in Algorithm design. MacMillan 1984. Editors A.W. BIERMANN, G. GUIHO, Y. KODRATOFF.

[BART-81] A. BARTELS, W. OLTHOFF, P. RAULEFS A.P.E. An Expert System for automatic programming from abstract specification of data types and algorithmes.
SEKI-PROJEKT, INSTITUT FUR Informatik
Universitat Bonn - Post fach 2220-D.5300 BONN 1,
WEST - Germany

[BAS-77] K. BASU, J. MISRA Proving Lopp Programs
IEEE Vol SE-1, n° 1, mars 1977.

[BAS-80] K. BASU a) A note on Synthesis of inductive assertions
IEEE Vol SE-6, n° 1, Janvier 1980.
b) On developpment of iterations from function specifications.
IEEE vol SE-6, n° 2, March 1980.

[BAU-76] F.L. BAUER Programming as an evolutionary process
LNCS 46, 1976, Springer Verlag.

[BAU-79] F.L. BAUER Program Development by stepwise transformations : The projet CIP.
LNCS-69-1979.

[BAU-79a] F.L. BAUER, M. BROY, N. PARSTCH, P. PEPPER, H. WÖSSNER Systematics of Transformation Rules.
Springer Verlag L.N.C.S. 69, 1979.

[BAU-79b] F.L. BAUER, P. PEPPER, H. WÖSSNER Note of the projet CIP : Outline transformation System.
L.N.C.S. 69, 1979 Springer Verlag.

[BAU-81] F.L. BAUER, M. BROY, W. DOSH, H. PARTSH, P. PEPPER, M. WIRSING Programming in a Wide Spectrum Language a collection of Examples.
Science of Computing programming 1, 1981.
North-Holland (073-114).

[BAU-82] F.L. BAUER From Specifications to machine code : Program Construction through formal reasonning.
6th international Conference on S.E. TOKYO, 1982.

[BEL-84] F. BELLEGARDE Thèse d'état en cours de préparations. 1984.

[BEN-84a] J. BENTLEY Algorithm Design techniques (programming pearls)
CACM, vol 27, number 9, 1984.

[BEN-84b] J. BENTLEY Algorithm Design techniques
CACM, vol 27, n° 2, 1984.

[BER-82] J.A. BERGSTRA, J.W. KLOP A formalized proof System for total Correctness of while programs
LNCS Springer Verlag, 137, 1982.

[BID-80] W. BIDEL Syntax-directed, Semantics-supported program Synthesis.
A.I. 1980, vol 14, n° 3.

[BID-84] W. BIDEL, K.M. HÖRNING LOPS A system Based on a Strategical Aproach to Program Synthesis.
MacMillan 1984, Editors : A.W. BIERMANN, G. GUIHO, Y. KODRATOFF.

[BIE-76] A.W. BIERMANN, R. KRISHNASWAMY Construction Programs from example Computations
IEEE, SE-2, 1976.

[BLI-77] A. BLIKLE An analysis approach to the verifications of iterative programs
Inf. processing 1977, pp 285-290.

[BRO-80] M. BROY, B. KRIEG-BRÜCKNER Derivation of invariant assertions during program development by transformation.
TOPLAS vol 2, n° 3, July 1980.

[BRO-81] M. BROY, P. PEPPER Program Development as formal activity
IEEE 1981, vol SE-7, n° 1, Janvier 1981.

[BRO-83] M. BROY Program Construction by transformations : a family tree of sorting programs.
NATO Advanced study - Edited by A.W. BIERMANN, G. GUIHO, 1983.

[BUC-77] D. BUCHNANAN Production rules as a representation for a Knowledge-based Construction Programs.
A.I. 8 (1) Feb. 1977.

[BUR-69] R.M. BURSTALL Proving Properties of program by structural induction.
The Computer Journal 12 (1), 1969.

[BUR-77] R.M. BURSTALL, J. DARLINGTON A transformation System for developping recursive programs.
JACM 24 (1), 1977.

[CHAN-73] C. CHANG, R.C. LEE Symbolic Logic and Mechanical Theorem Proving,
Academic Press, 1973.

[CHAT-77] CHATELIN Self-redefinition as a program manipulation strategy.
ACM SIGPLAN NOTICES, SIGART Newsletter 174-179 (1977).

[CHE-72] T.E. CHEATHAM, B. WEGBREIT A Laboratory for the study of automatic programming
AFIPS 1972, vol 1.

[CLA-81] K.L. CLARK The synthesis and verification of Logic programs
Report 81/36, 1981. Imperial College of London.

[DAH-72] O.J. DAHL, E.W. DIJKSTRA, C.A.R. HOARE Structured Programming.
Academic Press 1972.

[DAR-77] J. DARLINGTON Program Transformation and Synthesis : Present Capabilities.
Sept. 1977, publication n° 77/43, College Imperial London.

[DAR-81] J. DARLINGTON The structured Description of Algorithm Derivations. Algorithmic Languages de Bakker/Van Vliet (eds)
IFIP, North Holland, 1981.

[DAR-83] J. DARLINGTON The synthesis of implementations for abstract data types.
Editors : G. GUIHO, A.W. BIERMANN, NATO - 1983.

[DER-79] J.C. DERNIAME, J.P. FINANCE Spécification, utilisation et réalisation : les types abstraits.
Ecole d'été de l'AFCET, Monastir, Rapport CRIN 79-E-57.

[DERS-81] N. DERSHOWITZ, Z. MANNA Inference rules for program annotation
IEEE, vol SE-7, n° 2, March 1981.

[DERS-83] N. DERSHOWITZ The evolution of programs.
Birkhäuser, 1983 edited by J. Bentley, E. Coffman, R.I. Granam, D. Kuck, N. Pippenger.

[DERS-84] N. DERSHOWITZ Computing with Rewrite Systems
Proceedings of an NSF Workshop on the rewrite rule laboratory. Report n° 84 GEN 008, April 1984.

[DIJ-76] E.W. DIJKSTRA A discipline of programming.
Prentice-Hall 1976.

[DON-80] V. DONZEAU-GOUGE, G. HUET, G. KAHN, B. LANG
Programming Environments based on structured editors : The MENTOR Experience.
Rapport INRIA 26, 1980.

[DON-83] V. DONZEAU, G. KAHN, G. HUET, B. LANG, B. MELESE
MENTOR-METAL. Une vue dans cours et séminaires "les éditeurs dirigés par la syntaxe" Tome 1, 2, Avril 1983, AUSSOIS (FRANCE)

[DUB-84] E. DUBOIS Cadre et méthode de spécifications de systèmes d'informations fondés sur les types de données.
Thèse de docteur-ingénieur, INPL, Nancy, 1984.

[ELL-81] H.A. ELLOZY The determination of Loop invariants for programs with arrays.
IEEE, vol SE-2, March, 1981.

[FEA-79] M.S. FEATHER Development of a simple compiler by transformation.
August 1979, Draft version Edinburgh University.

[FEA-82] M.S. FEATHER A system for assisting program transformation. ACM TOPLAS, vol 4, n° 1, Janvier 1982.

[FIN-77] J.P. FINANCE Formalisation de la sémantique d'un langage de type déclaratif. Rapport 77-R-017, CRIN Nancy 1.

[FIN-77a] J.P. FINANCE, A. QUERE Essai de construction rigoureuse d'un algorithme de tri. Rapport 77-E-05, CRIN Nancy 1.

[FIN-78] J.P. FINANCE, P. LESCANNE Trois approches sémantiques d'un langage permettant l'écriture de programmes analysés déductivement. AFCET, Théorie et Technique de l'informatique, Tome 1, 1978.

[FIN-79] J.P. FINANCE Etude de la construction de programmes : Méthodes et langages de spécification et résolution de problèmes. Thèse d'état, Université de Nancy 1, 1979.

[FIN-82] J.P. FINANCE Cours de licence : la programmation déductive, 1982.

[FIN-84] J.P. FINANCE, M. GRANDBASTIEN, N. LEVY, A. QUERE, J. SOUQUIERES Spes : Un système pour spécifier et transformer. C.G.L.2. Nice, juin 1984.

[FLO-67] R.W. FLOYD Assigning Meaning to programs, Proceedings of the symposium in Applied Mathematic. A.M.S. 1967.

[FOD-83] J.K. FODERARO, K.L. SKLOWER, K. LAYER The Franz LIPS Manuel, 1983.

[GER-83] M. GEBIACH A second order Matching Procedure for the pratical use in program transformation system. Memo Seki-83-13, Université de Kaiserslautern.

[GOG-78] J. GOGUEN, J.W. THATCHER, E.C. WAGNER An intial Algebra approach to the specification Correctness, and Implementation of abstract data types. Programming Methodology, vol IV, (ed. Raymond Yeh) 1978, Prentice Hall.

[GRE-81] C. GRESSE A propos de la synthèse de programmes à partir de spécifications. AFCET R. de F. et I.A., Nancy, septembre 1981.

[GRE-84] C. GRESSE CATY : un système de construction assistée de programmes Thèse d'état 1984. Université d'Orsay.

[GUI-79] G. GUIHO, C. GRESSE, M. BIDOIT A system wich synthezizes Array-manipulating programs from Spécifications. Février 1979, LRI, rapport n° 28.

[GUI-80] G. GUIHO, C. GRESSE, M. BIDOIT Conception et certification de programmes à partir d'une décomposition par les données. RAIRO Informatique/Computer Science, vol 14, n° 4, 1980.

[GUI-83] G. GUIHO Automatic programming using abstract data types IFIP 83.

[GUT-78] G. GUTTAG, F. HOROWITZ, D. MUSSER The design of data types specification. Ed. R.T. Yeh Current trends in programming methodology, vol 4, 1978.

[GUY-84] J. GUYARD; J.P. JACQUOT "MAIDAY : An environment for guided programming with a definitional langage" 7th conference sur le Génie Logiciel, Orlando (Floride) 1984.

[HOA-69] C.A.R. HOARE An axiomatic basic for computer programming. CACM vol 12, n° 10, octobre 1969.

[HUET-77] G. HUET, B. LANG Proving and Applying Program Transformations expressed with second-order patterns. Rapport INRIA, n° 266, novembre 1977.

[HUL-83] J.M. HULLOT CEYX : a multiformalism programming environment. INRIA, Cours et séminaire "les éditeurs dirigés par la syntaxe" tome 2, avril 1983.

[IGA-73] S. IGARASHI, R. LONDON, D.C. LUCKHAM
Automatic Program Verification I : A Logical Basis and its Implementation.
Report STAN-CS-73-365, May 1973.
Computer Science Department STANDFORD University.

[JAF-82] J. JAFFAR, J.L. LASSEZ Reasonning about array segments.
ECAI 1982.

[JEF-80] D. JEFFERSON, N. SUZUKI Verification Decidability of Presburger Array programs.
JACM, vol 27, n° 1, Janvier 1980.

[JOH-73] B. JOHNSON On the power of arrays in universal language
7th annual Conference Princeton
On information Science and Systems (March 1973)
pp. 292-296.

[JON-80] CLIFF B. JONES Software Development : A rigorous approach.
Prentice Hall, 1980.

[JOU-84] J.P. JOUANNAUD, Y. KODRATOFF In Automatic program construction techniques
MacMillan 1984, editors : A.W. BIERMANN, G. GUIHO, Y. KODRATOFF

[KIN-80] J.C. KING Program Correctness : on inductive assertions Methods.
IEEE vol SE-6, n° 5, septembre 1980.

[KLE-71] S.C. KLEENE Logique Mathematique.
Collection U. Armand Colin, 1971.

[KOD-83] Y. KODRATOFF, M. PICARD Completion de systèmes de réécritures et synthèses de programmes à partir de leur spécification. BIGRE 83. Cap d'Agde.

[KOT-78] L. KOTT About System Transformation : A theorical Study
3ème colloque international sur la programmation, Dunod 1978.

[KOW-79] R. KOWALSKI ALGORITHM = Logic + Control
CACM, July 1979, vol 22, n° 7.

[LAM-83] A. VAN LAMSWEERDE Automatisation de la production de logiciels (2 parties)
T.S.I. vol 1, n° 6, 1982, T.S.I. vol 2, n° 1, 1983.

[LAU-82] J.L. LAURIERE Représentation et utilisation des connaissances.
T.S.I. vol 1, n° 2, 1982 (première et deuxième partie)

[LEI-83] P. LEITH Hierarchically structured production rules.
The Computer Journal vol 26, n° 2, 1983.

[LES-78] P. LESCANNE Sémantique d'un langage adapté à la construction déductive de programmes.
Nancy, 1978, CRIN 78-R-015.

[LES-82] P. LESCANNE Behavioral Categoricity of abstract data type Specifications.
1982, Rapport CRIN Nancy 82-R-011.

[LES-83] P. LESCANNE Computer experiments with the REVE term rewriting system generator.
10th annual ACM Symposium on the Principles of Programming Languages, Jan. 1983, Austin Texas.

[LEV-84] N. LEVY Outils d'aide à la construction et transformation de types abstraits algébriques.
Thèse de 3ème cycle, Nancy 1984.

[LIS-75] B. LISKOV, S.N. ZILLES Specifications techniques for data abstractions.
IEEE, vol SE 1-1, mars 1975.

[LIV-78] C. LIVERCI Théorie des programmes. Dunod 1978.

[LOV-77] D.B. LOVEMAN Program Improvement by source to source transformation.
JACM, vol 24, n° 1, Jan. 1977.

[MAN-71] Z. MANNA, J. WALDINGER Towards automatic program Synthesis.
CACM 14,3, 1971.

[MAN-74] Z. MANNA Theory of Computation.
The Graw-Hill, 1974.

[MAN-78] Z. MANNA, J. WALDINGER The logic of Computer Programming.
IEEE SE-4, n° 3, May 1978.

[MAN-79] Z. MANNA, J. WALDINGER Synthesis : Dream ⇒ Programs.
IEEE, vol SE-5, n° 4, 1979.

[MAN-80] Z. MANNA, J. WALDINGER A deductive approach to program synthesis.
ACM TOPLAS, vol 2, n° 1, Jan. 1980.

[MEY-80] B. MEYER La spécification.
EDF-GDF, rapport de recherche HI/3520- Juillet 1980.

[MIS-77] J. MISRA Prospects and limitations of automatic assertion generation for loop programs.
Vol 6, n° 4, Dec. 1977.

[MOR-78] J.H. MORRIS, B. WEGBREIT Program verification by subgoal induction Programming Methodology, vol III, (Software testing, validation and verification)
Ed. R.T. Yeh, 1978, Prentice Hall.

[PAG-83] R. PAIGE Transformational programming Applications to Algorithms and Systems.
10th annual ACM symposium, Principles of programming Languages. Austin, Texas, Jan. 1983.

[PAI-78] C. PAIR La programmation : de l'énoncé au programme.
Rapport CRIN, Nancy 78-P-061.

[PAI-80] C. PAIR Cours 2ème année. Ecole des Mines de Nancy : Algorithmique non numérique. 1980.

[PAR-83] H. PARSTCH, R. STEINBRÜGGEN Program Transformation systems.
Computing Surveys, vol 15, n° 3, Sept. 83.

[PEP-79] P. PEPPER A study on transformations Semantics
LNCS, N° 69, 1979, Springer Verlag.

[PRY-77] N.S. PRIWES Automatic generation for computer programs.
National Computer Conference 1977, pp 679-689.

[QUE-83] A. QUERE Spes : Notice du langage.
Rapport CRIN, Nancy, 83-R-014.

[REM-84] J.H. REMMERS A technique for developping loop invariants.
Information processing letters 18 (1984).

[REY-75] J.C. REYNOLDS Reasoning about array.
CACM, vol 22, n° 5, May 1975.

[ROS-73] B.K. ROSEN Tree - Manipulating Systems and Church-Rosser theorem.
JACM, vol 20, N° 1, Jan. 1973.

[SCH-79] P.C. SCHOLL Vers une programmation systématique : étude de quelques méthodes, techniques et outils.
Thèse d'état, USMG, Grenoble, 1979.

[SCT-83] W.L. SCHERLIS, D.S. SCOTT First steps towards inferential programming.
IFIP, 1983, pp 199-212.

[SCW-78] J.C. SCHWARTZ Verifying the safe use of destructive operations in applicative programs.
3ème colloque international sur la programmation, Dunod, Paris, 1978.

[SHA-80] M. SHARIR Set theoric constructs as tool in program Construction.
International Workshop on construction program, Château de Bonas, sept. 1980, IRIA.

[SOU-82] J. SOUQUIERES Construction et transformations d'algorithmes itératifs.
Thèse de docteur-ingénieur, 1982, Nancy.

[SRI-84] M. SRIVAS Rewrite rules Based Program Transformation.
Proceedings of an NSF Workshop on the rewrite Rule Laboratory. Report N° 84 GEN 008, April 1984.

[TAM-80] M. TAMIR ADI : Automatic Derivation of Invariants.
IEEE, vol SE-6, n° 1, Jan. 1980.

[TUR-84] W.M. TURSKI On programming by itérations.
IEEE, vol SE-10, n° 2, 1984.

[WAN-80] M. WAND Continued-Based program Transformation strategies.
JACM, 27, 1, 1980.

[WAT-78] R.C. WATERS Automatic Analys of the logical structure of programs.
MIT/AI/TR-492, 1978.

[WAT-83] R.C. WATERS Expressional Loops.
10th annual ACM principles of programming Languages
Jan. 1983, Austin, Texas.

[WEG-74] J.B. WEGBREIT The synthesis of Loop Predicates
CACM, Febr. 1974, vol 17, N° 2.

[WEG-76] J.B. WEGBREIT Goal - Directed Program Transformation
IEEE, vol SE-2, N° 2, 1976.

[WEG-77] J.B. WEGBREIT Complexity of Synthesizinz inductive assertions.
JACM, vol 24, n° 3, 1977.

[WIR-71] N. WIRTH Program development by stepwise refinement.
CACM, 14,4, 1971.

[WIR-73] N. WIRTH Systematic programming.
Prentice-Hall, 1973.

[WIR-74] N. WIRTH On the composition of well-structured programs.
Computing surveys, 6, N° 4, 1974.

[WIS-81] P.H. WISTON, B.K. PAULHORN LISP.
Addison Wesley, 1981.

[ZEL-79] M.V. ZELKOWITZ Perspectives on Software Engineering.
Computing surveys, vol. 10, n° 2, 1979.

institut
national
polytechnique
de lorraine

Le Président

N/Réf. : Scol.

AUTORISATION DE SOUTENANCE DE THESE DE DOCTORAT-D'INGENIEUR

-:-:-:-:-:-:-:-:-:-:-:-:-:-:-

VU LE RAPPORT ETABLI PAR :

Monsieur le Professeur FINANCE

le Président de l'Institut National Polytechnique de Lorraine autorise :

Monsieur SOUFI Loutfi

à soutenir, devant l'I.N.P.L., une thèse intitulée :

"Un système d'aide à la construction de programme itératifs"

en vue de l'obtention du titre de DOCTEUR-INGENIEUR

Spécialité "INFORMATIQUE"

UNIVERSITE NANCY BIBLIOTHEQUE SCIENCES

Fait à NANCY, le 5 Décembre 1984

Le Président de l'I.N.P.L.

M. LUCIUS

Résumé

Les années 80 sont marquées par les efforts visant à formaliser l'activité de développement de programmes et à réaliser des systèmes de transformations dans le but d'automatiser leur construction. Le système décrit ici est un exemple de cette démarche. Il est fondé sur des stratégies de constructions définies à partir de règles de preuves et sur une méthode de programmation descendante guidée par les résultats. De plus il est évolutif.

Schématiquement notre système assure la transformation d'une spécification formelle non algorithmique en un algorithme par applications successives de stratégies qu'il suppose valides. Une conséquence est que l'algorithme obtenu est correct vis-à-vis de la spécification initiale.

Au cours de ce "calcul de programmes" apparaît principalement le problème du choix de la stratégie à appliquer. Il est alors utile de définir un cadre pour mener des expériences sur ce processus de construction. Pour cela le système est paramétré au sens où il permet l'adjonction de nouvelles stratégies durant la construction. Nous croyons qu'un tel "apprentissage manuel" est un pas vers une solution au problème de la découverte de stratégies d'explicitations.

Mots clés :

transformation de programmes, correction, système de transformation, règles de transformations, méthode de programmation, spécification.