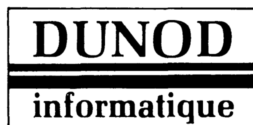
théorie des programmes

schémas, preuves, sémantique



L'informatique

- **Ainsi naquit l'informatique**, par R. Moreau
- **Principes des ordinateurs**, par P. de Miribel
- **Emploi des ordinateurs**, par J.-C. Faure
- **Aide-mémoire d'informatique**, par Ch. Berthet
- **Let's talk D.P.** ; **lexique d'informatique**, par J.-P. Drieux et A. Jarlaud
- **D.P. words ; dictionnaire d'informatique anglais-français, français-anglais**, par G. Fehlmann

La microinformatique

- **Le choix d'un microordinateur**, par H.-P. Blomeyer-Bartenstein
- **Introduction aux microprocesseurs et aux microordinateurs**, par C. Pariot
- **Microprocesseurs : du 6800 au 6809, modes d'interfaçage**, par G. Révellin
- **Interfaçage des microprocesseurs**, par M. Robin et T. Maurin

Les applications de l'informatique

- **Analyse organique, tomes 1 et 2**, par C. Cochet et A. Galliot
- **Analyse fonctionnelle**, par H. Briand et C. Cochet
- **Systèmes d'exploitation des ordinateurs**, par Crocus
- **Bases de données, méthodes pratiques sur maxi et mini-ordinateurs**, par D. Martin
- **Les fichiers**, par C. Jouffroy et C. Létang
- **Bases de données et systèmes relationnels**, par C. Delobel et M. Adiba
- **Systèmes informatiques répartis**, par Cornafion
- **Téléinformatique**, par C. Macchi et J.-F. Guilbert

L'art de programmer

- **Initiation à l'analyse et à la programmation**, par J.-P. Laurent
- **Les bases de la programmation**, par J. Arsac
- **Proverbes de programmation**, par H.-F. Ledgard
- **Théorie des programmes**, par C. Livercy
- **Algorithmique**, par P. Berlioux et Ph. Bizard
- **Synchronisation de programmes parallèles**, par F. André, D. Herman et J.-P. Verjus

Les langages de programmation

- **La programmation en assembleur**, par J. Rivière
- **Exercices d'assembleur et de macro-assembleur**, par J. Rivière
- **Basic : programmation des microordinateurs**, par A. Checroun
- **Introduction à A.P.L.**, par S. Pommier
- **Cobol. Initiation et pratique**, par M. Barès et H. Ducasse
- **Fortran IV**, par M. Dreyfus
- **La pratique du fortran**, par M. Dreyfus et C. Gangloff
- **Le langage de programmation PL/1**, par Ch. Berthet
- **Le langage ADA : manuel d'évaluation**, par D. Le Verrand

théorie des programmes

schémas, preuves, sémantique

par
C. LIVERCY

Nom collectif de :

Jean-Pierre FINANCE
Monique GRANDBASTIEN
Pierre LESCANNE
Pierre MARCHAND
Roger MOHR
Alain QUÉRÉ
Jean-Luc RÉMY

Chercheurs au Centre de Recherche
en Informatique de Nancy
et enseignants dans les Universités
de Nancy I et Nancy II

Préface du Professeur **C. PAIR**
Président de l'Institut National
Polytechnique de Lorraine

publié avec le concours du CNRS

DUNOD
informatique

© BORDAS, Paris, 1978
ISBN 2-04-010516-6

" Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, ou de ses ayants-droit, ou ayants-cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 n'autorise, aux termes des alinéas 2 et 3 de l'article 41, que les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part, et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration "

Préface

Depuis une vingtaine d'années, on a écrit beaucoup de programmes, souvent faux d'ailleurs. Tout est-il donc dit sur la programmation ? Les praticiens de l'informatique espèrent bien que non, eux qui, malgré quelques progrès récents, ne possèdent guère de guide pour construire leurs programmes, savoir s'ils sont corrects, les transformer ; et qui commencent à se demander pourquoi l'ordinateur ne les aiderait pas davantage dans ces tâches.

Construire, prouver, transformer les programmes, « à la main », automatiquement ou de manière assistée, voilà les principaux buts d'une réflexion sur la programmation. Il faut y ajouter la conception et l'implantation de langages commodes.

Cette réflexion doit, à mon sens, être entreprise de deux manières. D'une part, elle peut être *méthodologique*, c'est-à-dire viser à guider la conduite du programmeur. D'autre part, elle peut être *théorique*, c'est-à-dire vouloir comprendre ces objets mystérieux que sont les programmes.

L'informatique a introduit en effet des artefacts, les programmes, qu'on est amené à étudier comme des objets naturels pour les comprendre et les maîtriser. Pour cela, on va les modéliser, en donner des formalisations de caractère mathématique. C'est ce qui a été entrepris depuis une bonne dizaine d'années, par les théoriciens de l'informatique.

Il est maintenant possible et important que cette théorie de la programmation sorte du cadre restreint des initiés. C'est le but de cet ouvrage qui est le résultat d'un travail de synthèse effectué par un groupe de sept chercheurs et enseignants de Nancy. Son utilité est apparue clairement à la session 1974 de l'Ecole d'Eté d'Informatique de l'A.F.C.E.T., où les auteurs ont présenté un cours sur quelques aspects de cette théorie. Depuis, le contenu de ce cours a été largement approfondi, de nouveaux résultats y ont été intégrés et l'aspect synthétique encore accentué.

Le point de vue théorique ne saurait s'opposer au point de vue méthodologique et ils doivent s'éclairer l'un l'autre. C'est en particulier ce qu'essaie de réaliser le groupe de recherche nancéien. Aussi, même si ce livre se penche d'abord sur la théorie, comme en témoigne son titre, il ne néglige pas pour autant le côté méthodologique. D'ailleurs, tout au long de l'ouvrage est présenté le lien entre les recherches théoriques et leurs applications pratiques : preuves de programmes, construction de programmes, résolution de problèmes récursifs, ...

Je pense que l'audience de cet ouvrage sera très large et qu'il permettra aux informaticiens professionnels de dépasser les limites de leurs travaux quotidiens en leur présentant à la fois les résultats et l'esprit des recherches actuelles sur les fondements scientifiques de l'informatique.

C. PAIR.

Avertissements aux lecteurs et plan de lecture

Des connaissances élémentaires en mathématiques sont suffisantes pour aborder ce livre. Seules quelques notions courantes de théorie des ensembles (union, intersection, relation d'ordre, relation d'équivalence, récurrence sur $\mathbb{N}$, ...) ne sont pas redéfinies dans le texte. On utilise aussi un peu d'arithmétique simple dans les exemples (pgcd, ppcm, division euclidienne, par exemple). Cet ouvrage s'adresse à des lecteurs possédant un minimum de connaissances en informatique. Les étudiants de deuxième et troisième cycles d'université, les étudiants de grandes écoles ayant bénéficié d'une initiation à l'informatique et souhaitant approfondir les aspects théoriques de cette matière, mais aussi les informaticiens professionnels, devraient tirer profit de cette lecture.

Ce livre a été écrit en vue d'un parcours linéaire mais chaque chapitre est développé de façon indépendante des autres ; ainsi un lecteur averti ou pressé peut commencer la lecture où bon lui semble. Nous indiquons, ci-dessous un graphe donnant les liaisons entre les paragraphes. Le lecteur dispose aussi à la fin du livre d'un index des notations et des termes définis dans ce livre qui renvoie au paragraphe de première utilisation. Certaines parties de l'ouvrage sont en petits caractères, ces développements sont en général plus théoriques et peuvent être passés en première lecture sans que cela gêne la compréhension de la suite. De nombreux exercices sont insérés dans le corps du texte, la plupart d'entre eux font l'objet d'une correction ou d'indications de solution. Certains, précédés d'une étoile, sont plus difficiles que les autres. A la fin de chaque chapitre, on trouve une bibliographie succincte et commentée donnant les ouvrages et les articles fondamentaux concernant ce chapitre. La bibliographie générale est en fin d'ouvrage.

Les auteurs remercient leurs collègues qui, par leurs commentaires, leurs critiques et leurs conseils, ont permis d'améliorer la qualité de cet ouvrage, citons en particulier J. ARSAC, M. C. GAUDEL, C. PAIR, M. SINTZOFF.

Ils remercient aussi les secrétaires des divers établissements universitaires de Nancy qui ont contribué à la réalisation matérielle de ce travail.

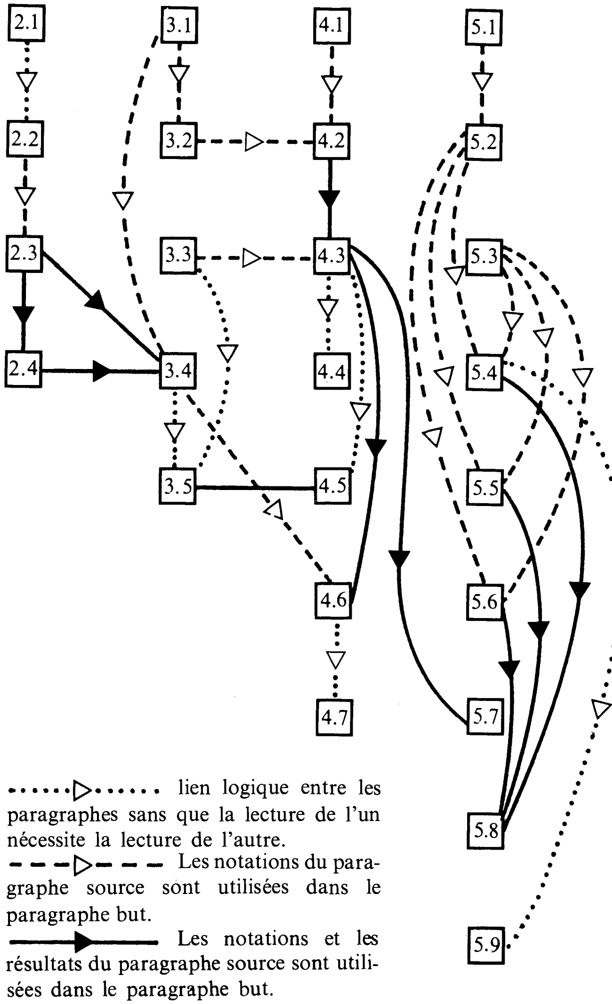


Figure 1 Graphe logique entre les différents paragraphes.

Sommaire

CHAPITRE 1 Problèmes, algorithmes et programmes 1

CHAPITRE 2 Problèmes récursifs et théorie du point fixe 7

1 *Généralités sur les problèmes* 7

1.0 Introduction 7
1.1 Quelques exemples 7
1.2 Concepts fondamentaux 8

2 *Enoncés récursifs* 11

2.1 De l'intérêt des énoncés récursifs 12
2.2 Notions de fonctionnelle et de point fixe 14
2.3 Calcul du plus petit point fixe d'une fonctionnelle par approximations successives 15
2.4 Calcul par substitution 16
2.5 Conclusion 17

3 *Etude mathématique : théorème du point fixe* 18

3.1 Ensembles inductifs et fonctions continues 18
3.2 Théorème du point fixe : énoncé et démonstration 21
3.3 Deux applications du théorème du point fixe 21
3.4 Généralisations du théorème au cas de fonctions non continues 22

4 *Preuve de propriétés du plus petit point fixe d'une fonctionnelle* 23

4.1 Méthode de récurrence sur la fonctionnelle 24
4.2 Exemples d'applications de la règle d'induction de Scott 26
4.3 Règle d'induction structurelle 30
4.4 Conclusion 33

5 *Conclusion et commentaires bibliographiques* 33

6 *Solution des exercices* 34

CHAPITRE 3 Schémas de programmes 47

1 *Schémas fonctionnels* 47

1.1 Présentation 47
1.2 Définition formelle de la première approche 48
1.3 Commentaires 49
1.4 Interprétations 49

2 *Schémas de programmes avec branchements* 50
2.1 Définition d'un schéma de programme 50
2.2 Interprétation, initialisation 52
2.3 Description de l'exécution d'un programme 53
2.4 Interprétations libres 56
2.5 Les schémas à une variable et les schémas de Ianov 58
2.6 Isologie entre schémas de programmes 60
2.7 Problèmes d'indécidabilité 67

3 *Schémas itératifs* 72
3.1 Introduction 72
3.2 Structures de contrôle, interprétation et calculs 73
3.3 Définition des $\mathscr{D}$-Schémas 74
3.4 Comparaisons entre structures 78
3.5 Etude des $\mathscr{D}$-Schémas 81
3.6 Autres structures 85
3.7 Résultats généraux et conclusion 92

4 *Schémas récursifs* 93
4.0 Introduction 93
4.1 Définitions 94
4.2 Interprétation d'un schéma récursif 95
4.3 Fonctionnelle associée à un terme 97
4.4 Calculs et règles de calcul 100
4.5 Relations entre fonctions calculées et plus petit point fixe 102
4.6 Relation entre les règles d'appel par valeur et d'appel par nom 109
4.7 Complexité des calculs. Une règle optimale 111
4.8 Conclusion 112

5 *Calcul relationnel récursif* 113
5.1 Présentation 113
5.2 Opérations sur les relations et constructions de programmes 114
5.3 Approche axiomatique 118
5.4 Approche opérationnelle 119
5.5 Approche dénotationnelle : le μ-calcul 122

6 *Commentaires bibliographiques* 126

7 *Solution des exercices* 127

CHAPITRE 4 Vérification et conception de programmes 145

1 *Introduction* 145

2 *Preuves de programmes avec branchements* 146
2.1 Introduction 146
2.2 Correction partielle et méthode des assertions de Floyd 148
2.3 Méthodologie pour vérifier la correction partielle 152
2.4 Problème de la terminaison 167
2.5 Méthodologie pour démontrer la terminaison 170

3 *Preuves de programmes itératifs* 176
3.1 Introduction 176

3.2 Correction partielle et méthode de Hoare 177
3.3 Exemple 187
3.4 Correction totale 189

4 *Construire des programmes corrects* 191

4.1 Introduction 191
4.2 Construction d'un programme par raffinements successifs 192
4.3 Construction d'un programme par transformations d'énoncés 200
4.4 Perspectives 204

5 *Application du calcul relationnel à la théorie des preuves de programmes* 205

5.1 Introduction rappel et but poursuivi 205
5.2 Correction partielle d'un cycle tant que... faire... fait 206
5.3 Opérations o et → sur les prédicats 207
5.4 Complétude de la méthode de Floyd pour une itération 209
5.5 Correction partielle d'un schéma avec branchements 211

6 *Preuves de programmes récursifs* 215

6.1 Introduction 215
6.2 Règle de preuve des appels de procédure 216
6.3 Etude d'un exemple 217
6.4 Correction partielle 218
6.5 Terminaison 220

7 *Preuves de correction et tests de validité des programmes* 221

7.1 Inconvénients des preuves a posteriori 221
7.2 Vers la conception de programmes fiables 222
7.3 Techniques d'évaluation symbolique 222

8 *Commentaires bibliographiques* 224

9 *Solution des exercices* 224

CHAPITRE 5 Sémantique d'un langage de programmation 241

1 *Buts d'une formalisation de la sémantique d'un langage* 241

1.1 Introduction 241
1.2 Le concept de sémantique d'un langage de programmation 242
1.3 Intérêts d'une formalisation de la sémantique 243

2 *Principaux concepts utilisés dans les méthodes de formalisation* 246

2.1 Structure d'information 246
2.2 Fonctions et calculs 247
2.3 Fonction sémantique 247
2.4 Langage pivot 248

3 *Un petit langage de programmation* 250

4 *Sémantique interprétative opérationnelle* 251

4.1 Notion de machine abstraite 251
4.2 Etat de mémoire et modifications élémentaires 251
4.3 Interprète abstrait de NAIN 256
4.4 Sémantique interprétative de NAIN 258

5 *Sémantique calculatoire* ... 262
5.1 Introduction ... 262
5.2 Notion de calcul ... 263
5.3 Système de calculs associé à une phrase de NAIN ... 265
5.4 Sémantique calculatoire de NAIN ... 267

6 *Sémantique fonctionnelle ou dénotationnelle* ... 269
6.1 Sémantique des entiers ... 270
6.2 Sémantique des expressions ... 270
6.3 Une première sémantique des instructions ... 272
6.4 Une autre approche de la sémantique des instructions : les continuations. 275
6.5 Sémantique des programmes ... 276
6.6 Etude d'un exemple ... 279

7 *Sémantique axiomatique* ... 280
7.1 Axiomes et règles d'inférence relatives à NAIN ... 280
7.2 Etude d'un exemple ... 281

8 *Complémentarité* ... 282
8.1 Introduction ... 282
8.2 Complémentarité des trois premières sémantiques de NAIN ... 282
8.3 Comparaison des sémantiques fonctionnelle et axiomatique ... 284
8.4 En guise de conclusion sur les définitions sémantiques ... 284

9 *Méthode des attributs* ... 285
9.1 Introduction ... 285
9.2 Attributs synthétisés ... 287
9.3 Attributs hérités : un exemple ... 291
9.4 Définition formelle de la méthode des attributs ... 293
9.5 Algorithmes testant les circularités dans les définitions des attributs ... 295
9.6 Rôle de sélection des attributs ... 302

10 *Commentaires bibliographiques* ... 307
11 *Solution d'exercices* ... 309

Bibliographie ... 315

Notations ... 323

Index des sujets ... 325

CHAPITRE 1

Problèmes, algorithmes et programmes

La programmation est souvent comparée à un art. Il y a des programmes élégants et bien conçus, et d'autres qui sont illisibles et mal structurés. Dans un cas comme dans l'autre ces programmes peuvent être faux ; ils peuvent l'être d'emblée ou ils peuvent le devenir à la suite de modifications (optimisation, amélioration !). Ce livre n'a pas la prétention d'apporter la solution qui permet d'écrire des programmes corrects, mais il résulte d'une réflexion sur les différents concepts de l'activité informatique afin de donner un cadre mathématique et formalisé à de nombreuses questions qui concernent l'« art de la programmation ».

Voici une liste non exhaustive de problèmes liés à l'activité informatique :

— Qu'est-ce qu'un programme, de quoi est-il composé, que définit-il ?

— Quels sont les objets manipulés par un programme ? Comment peut-on les définir ?

— Quand peut-on dire qu'un programme est correct ?

— Que faut-il faire pour construire des programmes qui soient corrects ?

— Que font exactement les instructions d'un langage évolué ?

Ces questions se posent sous différentes formes tout au long de la chaîne des travaux lors de la conception d'un programme. Cette chaîne peut être décomposée en plusieurs étapes fondamentales suivant le schéma simplifié suivant :

Problème → énoncé → énoncé algorithmique → programme → calcul

Le plan de ce livre suit approximativement cette décomposition et à chaque étape on essaie de préciser en quels termes se posent les questions évoquées plus haut et comment y répondre.

Tout au début de cette chaîne se pose un problème à résoudre. En général on a seulement une vague idée de ce que l'on cherche, des objets que l'on va manipuler, de ceux qui seront des données, de ceux qui seront des résultats intermédiaires ou finals. Pire encore, il est probable que l'on ne conçoit pas à ce niveau de méthode de résolution de ce problème. La première étape du travail de l'informaticien est alors de mettre un peu d'ordre dans ces idées. Pour cela deux niveaux sont à considérer :

— D'une part, il faut définir logiquement les objets que l'on va manipuler. A ce niveau il ne s'agit pas de leur implémentation mais de définir leur type

abstrait, c'est-à-dire les ensembles abstraits qu'ils vont parcourir, leur mode de construction, les moyens d'y accéder ainsi que leurs propriétés logiques. Dans le chapitre 2 on propose des outils pour atteindre ce but en présentant brièvement le concept de structure de donnée.

— D'autre part, on doit décrire à l'aide de formules logiques les liens existant entre les objets que l'on va manipuler. A priori dans cette étape on ne devrait pas se préoccuper du caractère algorithmique de ces formules et toute la puissance d'un langage logique comme par exemple le calcul des prédicats du 1er ordre doit pouvoir être utilisée.

Ces deux démarches, qui en général doivent être menées de front, conduisent à un énoncé formel du problème initial. Le chapitre 2 est consacré à l'étude de différents types d'énoncés que l'on peut obtenir. En particulier, on y étudie en détail les énoncés récursifs et le théorème du plus petit point fixe qui est fondamental pour la suite. Ce théorème est tout à fait analogue aux théorèmes de point fixe que l'on rencontre en analyse mathématique, et permet pour les problèmes dont l'énoncé est un système d'équations fonctionnelles récursives de trouver la « plus petite » solution par approximations successives.

La deuxième étape dans la résolution d'un problème informatique est de passer d'un énoncé général de ce problème à un énoncé algorithmique. Dans un petit nombre de cas particuliers, ce passage est trivial et il s'agit par exemple de réordonner un ensemble d'égalités. Mais dans la plupart des cas ce passage est non évident et constitue un des aspects les plus difficiles de l'informatique. On peut aborder cette question de deux façons différentes, suivant que l'on espère complètement automatiser le passage d'un énoncé à un énoncé algorithmique voire à un programme ou que l'on désire plus modestement donner des outils à l'informaticien lui permettant d'élaborer le programme en contrôlant à tout moment la validité de ce qui est déjà construit. La première attitude conduit à la théorie de la synthèse des programmes qui, malgré les nombreux travaux déjà effectués, n'en est encore qu'à ses balbutiements et reste un des sujets de recherche en intelligence artificielle. La deuxième est celle des auteurs qui ont proposé des méthodologies de la programmation. Sans aucunement nier la valeur de ces travaux fondés sur la programmation structurée ou modulaire et sur des méthodes d'analyse descendante ou par raffinements successifs, nous nous occupons assez peu ici de ce genre de questions. Cependant au paragraphe 4 du chapitre 4, nous donnons deux exemples d'approches différentes de ce problème et tout au cours de l'ouvrage les heuristiques ayant conduit à tel ou tel algorithme sont mises en évidence.

Une fois obtenu un énoncé algorithmique d'un problème informatique, transformer cet énoncé en un programme est en principe un simple problème de traduction. Cependant suivant le type de langage que l'on utilise les programmes que l'on obtient sont fort différents. C'est en particulier les structures de contrôle fournies à l'utilisateur par le langage de programmation qui conditionnent la forme du programme. Au chapitre 3, nous étudions d'un point de vue formel les différents types de programmes en les classifiant en fonction des structures de contrôle que l'on peut utiliser. On obtient ainsi

les notions de schéma de programme avec branchement (correspondant plus ou moins à une écriture des programmes en un langage du type FORTRAN), de schéma itératif de divers types (correspondant à la comparaison de différentes formes d'itération) et de schéma récursif (correspondant à l'introduction des fonctions procédures récursives). Pour pouvoir comparer ces différents types de schémas de programmes on a exhibé dans chaque cas les fonctions calculées par ces programmes en définissant les calculs qui font passer d'une donnée à un résultat ; un calcul est intuitivement la suite des états d'une machine depuis son état initial jusqu'à son arrêt qui donne le résultat.

La dernière étape de l'activité informatique est de faire exécuter ce programme sur un ordinateur, et chaque fois que l'on fournira des données à ce programme, par l'intermédiaire d'un calcul, il nous fournira un résultat. Cette étape parfaitement automatique ne semble donc pas justifier une étude théorique. Mais on constate que, même si l'on a scrupuleusement suivi les différentes étapes de la conception du programme, il se peut que des erreurs se soient malencontreusement glissées quelque part. L'attitude classique et pas toujours efficace pour détecter ces erreurs est d'utiliser des jeux de tests en essayant d'examiner le maximum de cas particuliers. Ces jeux de tests n'étant évidemment pas exhaustifs, cette attitude conduit à douter de la validité universelle du programme conçu et à craindre qu'une utilisation dans des cas limites ne conduise à des résultats erronés ou à un bouclage infini. Remarquons, bien que nous n'abordions pas ces questions ici, que l'utilisation des tests est pratiquement sans intérêt dès que l'exécution de diverses procédures se fait en parallèle ; en effet en cas d'erreur on est dans l'impossibilité de reconstituer la configuration qui l'a provoquée. Pour remédier à cela, il est nécessaire de prouver la validité d'un programme, de même qu'un mathématicien n'admettra pas un théorème après en avoir vu la vérification sur des cas particuliers mais demandera une démonstration de ce théorème.

Au chapitre 4 nous avons repris chacun des types de schémas de programmes et exhibé à chaque fois des méthodes appropriées permettant de prouver logiquement des propriétés de correction à propos de ces programmes. Ces propriétés doivent montrer l'adéquation entre le programme et l'énoncé du problème. En général on procède en deux temps, d'une part on montre que le programme est partiellement correct par rapport à l'énoncé, c'est-à-dire que s'il fournit des résultats ce seront ceux que l'on attendait, d'autre part on démontre l'arrêt du programme pour toutes les données que l'on est susceptible de lui fournir. Ce chapitre a été écrit dans un but essentiellement didactique et contient de nombreux exemples, cependant nous avons cru bon d'y insérer un paragraphe plus théorique sur le calcul relationnel dans lequel on justifie théoriquement certaines des méthodes de ce chapitre. Pour clarifier l'exposition et sérier les problèmes nous avons dans cet ouvrage dissocié la description des divers schémas de programme et les méthodes de preuve associées. Dans la réalité, les activités d'écriture de programme et

de preuve de ces programmes doivent être menées en parallèle, et tout informaticien devrait garder présent à l'esprit le précepte de DIJKSTRA dont une traduction libre est « aucune boucle ne doit être écrite sans en donner une preuve et sans en prouver la terminaison ». Nous avons donné quelques indications sur la pratique de cette double activité au chapitre 4 dans le paragraphe 4 concernant l'écriture de programmes corrects.

Les chapitres 2, 3 et 4 de cet ouvrage ont donc l'ambition de recouvrir l'ensemble des problèmes de l'informatique et effectivement leur lecture devrait fournir aux utilisateurs de cette science des outils leur permettant de mieux comprendre leur activité et, par là même, de mieux la dominer. Cependant tous les exemples que l'on a développés dans ces chapitres sont relativement simples et leur complexité n'a rien de comparable avec celle d'un projet plus conséquent comme la réalisation d'un compilateur, d'un moniteur ou d'un système de gestion de banque d'information.

De plus, dans ces chapitres, nous avons généralement écrit les algorithmes dans des langages plus ou moins idéaux sans nous préoccuper de l'exécution sur machine, qui nécessite la traduction en un programme d'un langage de programmation réel. Il faut donc définir de façon précise la signification des programmes de ces langages et pour cela se donner des outils formels permettant d'associer au texte de chaque programme une signification.

C'est le but du chapitre 5 où l'on traite les principales méthodes de description de la sémantique des langages de programmation. Si les théories que l'on obtient sont plus complexes et d'un abord plus rébarbatif, ceci est dû à la nature même des langages de programmation évolués. En effet, ils mettent à la disposition des utilisateurs de multiples possibilités et celles-ci s'imbriquent pour conduire à des programmes dont la signification n'est pas évidente a priori. Ce type de théorie doit éviter en particulier que l'on soit obligé de tester par un passage machine la signification d'un programme sans que l'on sache d'ailleurs si le résultat obtenu est dû à des propriétés intrinsèques du langage communes à tous les compilateurs ou à un choix particulier d'implémentation. D'autre part, ces théories mettent en évidence parmi les structures de contrôle de programme des types qui s'expriment très bien formellement et d'autres dont l'expression formelle est beaucoup plus compliquée. Il est alors remarquable de constater que ce sont justement les structures dont l'expression est aisée qui de l'avis même des utilisateurs de l'informatique sont essentielles pour bien écrire des programmes et que les autres sont à proscrire dans la mesure du possible. Le cas de l'utilisation excessive des instructions de branchement (goto) est à cet égard tout à fait significatif de la convergence de vue des théoriciens et des praticiens de l'informatique.

Les principales méthodes de description de la sémantique, présentée au chapitre 5, ont pour but commun d'associer à un texte de programme, une fonction mathématique, qui à une donnée associe le résultat du programme pour cette donnée. Ces méthodes diffèrent par le mode de définition de cette fonction, et on peut distinguer deux types : Les méthodes interprétatives, où l'on associe au langage de programmation un automate qui simule l'exé-

cution des programmes de ce langage, et les méthodes dénotationnelles (ou mathématiques), où l'on définit directement les fonctions ci-dessus à partir du texte du programme.

Cet ouvrage se termine par l'exposition de la méthode des attributs qui est moins une méthode au sens précédent qu'un outil d'aide à la description de la sémantique pouvant être utilisé dans différentes méthodes.

En conclusion ce livre veut aborder certains aspects fondamentaux de l'activité informatique en essayant d'en dégager une synthèse. On est conduit à formaliser différents concepts pour mieux les appréhender. Indiquons tout de suite que d'autres questions, tout aussi fondamentales, telles que la complexité des programmes n'ont pas été abordées ici, et que certaines comme les structures de données, la synthèse de programmes, la méthodologie de la programmation, n'ont été qu'effleurées.

CHAPITRE 2

Problèmes récursifs et théorie du point fixe

1 GÉNÉRALITÉS SUR LES PROBLÈMES

Problème : question à résoudre, portant soit sur un résultat inconnu à trouver à partir de certaines données, soit sur la détermination de la méthode à suivre pour obtenir un résultat supposé connu.

(Paul ROBERT, dictionnaire.)

1.0 Introduction

Avant d'étudier les notions d'algorithmes et de programmes, il est utile de se souvenir qu'un algorithme est écrit pour résoudre un problème ou, ce qui revient au même, pour calculer un certain nombre d'objets. Nous allons, très brièvement, préciser ici ce qu'on peut entendre par problème et en profiter pour introduire quelques outils utiles pour la suite. Nous présentons, au paragraphe 1.1, quelques exemples portant sur des domaines diversifiés : arithmétique, théorie des graphes, bases de données. Au paragraphe 1.2, nous dégageons les concepts essentiels en montrant la difficulté de présenter une théorie unitaire des problèmes.

1.1 Quelques exemples

Présentons tout d'abord un exemple tiré de l'arithmétique.

Exemple 1 : Plus grand commun diviseur de deux entiers.
Etant donné deux entiers a et b, calculer le pgcd p de a et b.

Dans ce premier énoncé, on distingue les **données** a, b qui sont des entiers, **l'inconnue** p qui est un entier, et la définition de p (pgcd de a et b). Cette définition peut être détaillée dans un **énoncé** du type :
p divise a et p divise b et, pour tout entier q, si q divise a et b,
alors q divise p. □

Le problème suivant, extrait de la théorie des graphes, admet un énoncé plus complexe.

Exemple 2 : Fonction ordinale d'un graphe.

Rappelons que la fonction ordinale d'un graphe associe à chaque point x du graphe la longueur, c'est-à-dire le nombre d'arcs, d'un plus long chemin d'une origine à x. L'énoncé du problème peut être ainsi formulé :
Etant donné un graphe $G = (E, \Gamma)$, calculer la fonction ordinale f de G, définie pour tout x de E, par :

$$f(x) = \begin{cases} \text{SI, pour tout } y \text{ de } E, \text{ non } y\Gamma x \text{ ALORS } 0 \\ \text{SINON } 1 + \max \{ f(y) \mid y \in E \text{ et } y\Gamma x \} . \end{cases}$$

Outre les opérations arithmétiques, nous avons besoin, pour étudier ce problème de graphe, d'opérations et de fonctions sur les ensembles : maximum des éléments d'un ensemble ordonné, image d'un ensemble par une application, etc... □

Exemple 3 : Interrogation d'une banque de données.

Le problème est de construire, à partir des renseignements obtenus au cours d'un recensement, l'ensemble des individus mariés habitant la même agglomération que l'un de leurs parents. Le problème suppose l'existence d'un environnement formé des éléments suivants : un ensemble d'individus, un ensemble de villes et trois relations binaires « est marié avec », « a pour parent » et « habite ». Les individus x cherchés peuvent être définis par la propriété I suivante :

$$\begin{aligned} I(x) : \exists y\, \exists z\, \exists v [& x \text{ est marié avec } y \\ & \underline{\text{et}}\ (x \text{ a pour parent } z\ \underline{\text{ou}}\ y \text{ a pour parent } z) \\ & \underline{\text{et}}\ \ x \text{ habite } v\ \underline{\text{et}}\ z \text{ habite } v] . \end{aligned}$$

Résoudre le problème revient à chercher tous les individus vérifiant I. □

1.2 Concepts fondamentaux

A partir de ces trois exemples, nous pouvons dégager les remarques suivantes :

a) Un **problème** fait intervenir une **donnée structurée,** c'est-à-dire un objet mathématique constitué d'objets élémentaires et de **fonctions d'accès.** Ainsi dans l'exemple 1, les opérations arithmétiques permettent de passer d'un couple d'entiers à un autre entier. En théorie des graphes, les fonctions successeur et prédécesseur associent à chaque point du graphe les points adjacents.

On se donne également un certain nombre de **relations** ou de **prédicats.** Ainsi, en arithmétique, on se donne la relation « divise » alors que le prédicat « est un nombre premier » peut être défini à partir de relations plus élémentaires. Dans l'exemple 3, on se donne les prédicats « a pour parent » et « habite »

alors que le prédicat « habite la même ville qu'un de ses parents » est calculé à partir des précédents. Notons sur cet exemple, que nous avons en fait introduit deux types d'objets : PERSONNE, VILLE et qu'il n'est pas question d'écrire « Paul habite Pierre » ou « Nice a pour parent Paul ». Résumons cette remarque dans une définition.

Définition 1 : Une donnée structurée est formée d'une famille d'ensembles, de fonctions et de prédicats. □

b) **Enoncer un problème** revient à définir formellement un ou plusieurs objets. Dans l'exemple 3 il s'agit de l'ensemble de tous les individus vérifiant la propriété I. Plus généralement, un énoncé définit une (ou plusieurs) fonction d'accès : dans l'exemple 1, la fonction pgcd.

La fonction d'accès cherchée est définie en utilisant les fonctions et les prédicats de la donnée du problème. Un énoncé sera donc d'autant plus long que l'ensemble de fonctions et de prédicats de base sera réduit. Ainsi si l'on ne dispose pas de la relation « divise » dans l'exemple 1, il faudra remplacer partout « x divise y » par la formule « $\exists z \; (y = x * z)$ ». De même dans l'exemple 2, si la donnée ignore la fonction max, on peut définir cette fonction en posant, par exemple :

$$\max(\{ x \}) = x$$
$$\max(E \cup \{ x \}) = \underline{\text{si}}\ x > \max(E)\ \underline{\text{alors}}\ x\ \underline{\text{sinon}}\ \max(E)\ \underline{\text{fsi}}\,.$$

Exercice 1

a) Enoncer le problème du tri d'un tableau d'entiers.

b) Ecrire un énoncé du problème d'interclassement de deux tableaux triés. On prendra soin de définir complètement le problème [cf. exemple 6]. □

Enoncer un problème suppose que l'on dispose d'un **langage d'énoncés.** Les exemples 1 et 3 sont écrits dans le langage du calcul des prédicats avec un ensemble de symboles représentant les fonctions et prédicats de base. L'exemple 2 est écrit dans le langage plus riche de la théorie des ensembles qui utilise, outre les symboles de base précédents, des symboles tels que $\in$, $\cup$, $\cap$, $\subset$, $\{$, $\}$, etc.

On peut au contraire imaginer des langages plus pauvres en excluant, par exemple, les quantificateurs. C'est ainsi que le problème du calcul des racines d'une équation du second degré peut être posé sous la forme :

Etant donné trois réels a, b, c ($a \neq 0$) trouver x tel que

$$ax^2 + bx + c = 0$$

ou sous la forme beaucoup plus explicite.

Etant donnés trois réels a, b, c ($a \neq 0$) trouver deux réels x_1, x_2 tels que

$$x_1 = (-b + \sqrt{b^2 - 4\,ac})/(2\,a) \qquad x_2 = (-b - \sqrt{b^2 - 4\,ac})/(2\,a)\,.$$

Concluons cette seconde remarque par une nouvelle définition.

Définition 2 : Un langage d'énoncés de problèmes sur une donnée structurée D est un langage formel sur un alphabet contenant, en particulier, des symboles représentant les fonctions et prédicats de base de la donnée. Les éléments du langage sont généralement appelés **formules**. Un énoncé du langage est un ensemble de formules. □

Les symboles représentant des fonctions sont encore appelés **symboles fonctionnels.** Leur ensemble est noté en général $\mathscr{F}$. De même, les symboles représentant des prédicats sont appelés **relationnels** et leur ensemble noté $\mathscr{P}$. Il est important de bien distinguer les symboles formels des objets, fonctions ou prédicats qu'ils représentent. Généralement en effet, un problème est posé pour toute une famille de données similaires, en ce sens que les fonctions et prédicats de base vérifient des propriétés semblables. (Il en va ainsi des problèmes de graphe.) L'énoncé, par contre, est donné une fois pour toutes. Quelquefois même, des problèmes, en apparence différents, peuvent être considérés comme des cas particuliers d'un même problème plus général. Ainsi le problème de trouver le plus grand entier k tel que $k.y \leqslant x$ ($y > 0$, $x \geqslant 0$) et le problème de trouver le plus grand entier k tel que y^k divise x ($x > 0$, $y > 1$) sont deux cas particuliers du problème général suivant :

Etant donné un ensemble $(D, \prec)$ ordonné muni d'une opération $*$ et deux éléments x, y de D, trouver le plus grand entier k tel que

$$\underbrace{y * y * \cdots * y}_{\text{k fois}} \prec x .$$

Ici, le symbole fonctionnel $*$ représente dans un cas l'addition, dans l'autre la multiplication. De même le symbole relationnel $\prec$ représente alternativement l'ordre usuel sur $\mathbb{N}$ et la relation « divise ».

Cette notion d'énoncé formel interprétable de plusieurs manières différentes est assez importante. On retrouve un point de vue similaire, au chapitre 3, dans l'étude des schémas de programmes et des schémas récursifs.

c) **Résoudre un problème,** c'est donner une définition équivalente de l'accès défini par l'énoncé formel, à partir de laquelle on puisse effectuer un calcul de celui-ci. Une telle définition est appelée **définition algorithmique.** Ainsi pour le calcul des racines d'une équation du second degré, le deuxième énoncé peut être considéré comme algorithmique. Il va de soi que le concept de définition algorithmique dépend lui-même du concept d'algorithme. Dans un langage admettant la récursivité, on pourra considérer comme algorithmique la définition suivante de la fonction pgcd :

$$\text{pgcd}(x, y) = \underline{\text{si}}\ y = 0\ \underline{\text{alors}}\ x\ \underline{\text{sinon}}\ \text{pgcd}(y, \text{reste}(x, y))\ \underline{\text{fsi}} .$$

En résumé nous venons de présenter très brièvement trois aspects de la théorie des problèmes : données, énoncé et résolution algorithmique. Nous insisterons assez peu, dans les chapitres suivants, sur les deux premiers aspects pour nous pencher essentiellement sur le dernier. En effet, jusqu'à une date assez récente, les spécialistes de la programmation n'ont envisagé que le

côté algorithmique des choses, principalement pour les deux raisons suivantes : d'abord, il y avait beaucoup à faire en ce domaine comme la suite de l'ouvrage le montre amplement, ensuite, les problèmes numériques abordés dans un premier temps ne nécessitaient pas, du moins apparemment, de réflexion approfondie sur les données et les spécifications de l'énoncé. Au paragraphe suivant, nous allons décrire un langage permettant de spécifier à la fois les énoncés, les données et les algorithmes de calcul : le langage des définitions récursives.

2 ÉNONCÉS RÉCURSIFS

L'énoncé proposé dans l'exemple 1 pour définir le pgcd de deux entiers ne donne pas directement d'indications pour le calcul. Par contre, il permet de prouver quelques propriétés telles que

(1) $\mathrm{pgcd}(x, 0) = x$

(2) $\mathrm{pgcd}(x, y) = \mathrm{pgcd}(x - y, y)$

(3) $y \neq 0 \Rightarrow \mathrm{pgcd}(x, y) = \mathrm{pgcd}(y, \mathrm{reste}(x, y))$

où reste(x, y) désigne le reste de la division euclidienne de x par y.

Vérifions, par récurrence sur y, que les relations (1) et (3) permettent de calculer pgcd(x, y) pour tous x, y. Ceci est clair pour $y = 0$. Supposons que nous sachions calculer $\mathrm{pgcd}(x', y')$ pour tout $y' < y$. En particulier reste(x, y) est strictement inférieur à y. Nous savons donc calculer pgcd(y, reste(x, y)) et, par là, pgcd(x, y). C.Q.F.D.

Nous obtenons ainsi l'énoncé suivant du problème posé

$$\mathrm{pgcd}(x, y) \underset{\mathrm{def}}{=} \underline{\mathrm{si}}\ y = 0\ \underline{\mathrm{alors}}\ x\ \underline{\mathrm{sinon}}\ \mathrm{pgcd}(y, \mathrm{reste}(x, y))\ \underline{\mathrm{fsi}}\,.$$

On dira que cet énoncé est **récursif** dans la mesure où l'inconnue (pgcd) apparaît à la fois à gauche et à droite du signe $\underset{\mathrm{def}}{=}$ (qui se lit « est défini par »).

Les définitions que nous étudions dans ce paragraphe sont toutes de la forme

$$\varphi(x_1, \ldots, x_n) \underset{\mathrm{def}}{=} \varepsilon$$

où φ est une fonction inconnue, $x_1, \ldots, x_n$ des variables et ε une expression contenant des symboles fonctionnels et relationnels de base, les variables $x_1, \ldots, x_n$ et (en général) l'inconnue. Reportant une définition rigoureuse du langage d'énoncés récursifs au paragraphe 4, nous présentons d'abord quelques exemples pour concrétiser notre propos.

2.1 De l'intérêt des énoncés récursifs

> Diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre.
>
> (R. DESCARTES, Discours de la Méthode.)

L'idée fondamentale justifiant l'importance des énoncés récursifs peut s'exprimer ainsi :

« Pour résoudre un problème sur une donnée complexe, décomposer cette donnée en données plus simples et ramener la solution du problème compliqué à celles de ces problèmes plus simples ». Le calcul du pgcd par l'algorithme d'Euclide en est un premier exemple. Considérons maintenant plusieurs problèmes, d'abord sur les entiers, puis sur des structures de données plus riches.

Exemple 4 : Nombre de partitions d'un entier m en au plus n sommants.

On est amené à répartir une prime de m francs entre n employés $e_1, ..., e_n$; mais les habitudes de la maison veulent que compte tenu de l'ancienneté la prime de e_1 soit supérieure ou égale à celle de e_2, elle-même supérieure ou égale à celle de e_3, etc. La prime de certains employés peut éventuellement être nulle. Combien y a-t-il de manières de répartir cette prime en francs entiers ?

Notons q(m, n) ce nombre ; on l'appelle aussi nombre de partitions de m en au plus n sommants. Si m ou n sont nuls le calcul est simple. En effet $q(0, n) = 1$ et $q(m, 0) = 0$ si $m > 0$. Si maintenant m et n sont strictement positifs, essayons de calculer q(m, n) en fonction de $q(m', n')$ avec $m' < m$ ou $n' < n$. Si $n \geqslant m$, $q(m, n) = q(m, m)$. Sinon, q(m, n) est la somme de $q(m, n - 1)$ et du nombre de partitions de m en exactement n sommants non nuls. Ce dernier nombre est égal à $q(m - n, n)$. On obtient finalement l'équation récursive

$$\begin{aligned} q(m, n) \underset{\text{def}}{=} & \ \underline{\text{si}}\ m = 0\ \underline{\text{alors}}\ 1 \\ & \underline{\text{sinon}}\ \underline{\text{si}}\ n = 0\ \underline{\text{alors}}\ 0 \\ & \underline{\text{sinon}}\ \underline{\text{si}}\ m < n\ \underline{\text{alors}}\ q(m, m) \\ & \underline{\text{sinon}}\ q(m, n - 1) + q(m - n, n)\ \underline{\text{fsi}}\ \underline{\text{fsi}}\ \underline{\text{fsi}}\ . \end{aligned}$$ □

Exercice 2

Calculer q(5, 3).

Exemple 5 : Une autre définition de factorielle.

La définition classique de factorielle induit un calcul « à rebours » : $n ! = n * (n - 1) * \cdots * 1$. On pourra constater que l'équation suivante permet de calculer la factorielle d'une autre manière.

$$g(x, y) \underset{\text{def}}{=} \underline{\text{si}}\ x = y\ \underline{\text{alors}}\ 1\ \underline{\text{sinon}}\ g(x, y + 1) * (y + 1)\ \underline{\text{fsi}}\ .$$ □

Exercice 3

a) Montrer que $g(x, 0) = fact(x)$.

b) $g(x, y)$ est-il défini lorsque x est plus petit que y ? □

Les définitions récursives dépassent cependant largement le cadre de l'arithmétique. Sans vouloir faire une étude exhaustive, présentons toutefois trois exemples impliquant des structures de données plus riches.

Exemple 6 : Tri par interclassement.

Pour trier un tableau t non réduit à un élément, on peut le diviser en 2 tableaux t_1, t_2 sensiblement de même taille, trier t_1 et t_2 et interclasser les tableaux résultants. Le tri peut donc être défini par l'équation récursive suivante dans laquelle t[i : j] représente la restriction de t à l'intervalle [i : j] et ÷ la division euclidienne.

$$\text{tri}(t[i : j]) \underset{\text{def}}{=} \underline{\text{si}}\ i \geqslant j\ \underline{\text{alors}}\ t[i : j]$$
$$\underline{\text{sinon}}\ \text{interclass}(\text{tri}(t[i : (i+j) \div 2]), \text{tri}(t[(i+j) \div 2+1 : j]))\ \underline{\text{fsi}}\ .$$

Cette équation définit sans ambiguïté tri(t[i : j]). C'est vrai pour $i \geqslant j$. C'est aussi vrai, par récurrence, pour $i < j$ car les tableaux $t[i : (i + j) \div 2]$ et $t[(i + j) \div 2 + 1 : j]$ sont plus petits que t[i : j]. Nous laissons au lecteur le soin de définir la fonction interclass. □

Exemple 7 : Parcours d'un arbre binaire.

Sans donner une définition formelle, convenons qu'un arbre binaire sur un ensemble E est soit réduit à un point (on dit alors qu'il est atomique), soit formé d'un nœud (sa racine) et de deux arbres respectivement appelés sous-arbre gauche et droit.

Pour parcourir un tel arbre binaire T ((figure 1), on peut adopter la stratégie suivante :

— parcourir d'abord le sous-arbre gauche g(T) de T,
— passer par la racine de T,
— parcourir le sous-arbre droit d(T) de T,

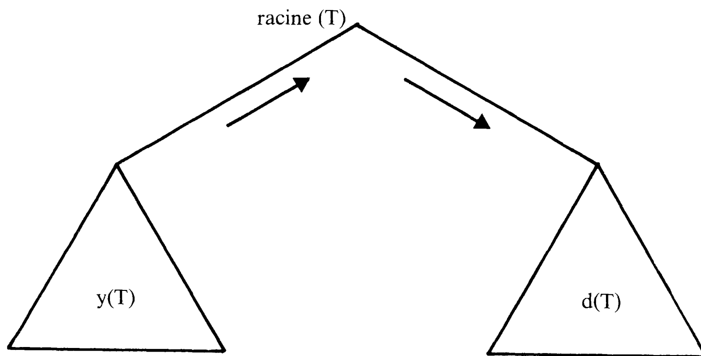


Figure 1 Parcours symétrique d'un arbre T.

La liste des nœuds de T, tels qu'ils apparaissent au long de ce parcours, peut être définie par :

$$\text{liste}(T) \underset{\text{def}}{=} \underline{\text{si}}\ T \text{ atomique } \underline{\text{alors}}\ T$$
$$\underline{\text{sinon}}\ \text{liste}(g(T)).\text{racine}(T).\text{liste}(d(T))\ \underline{\text{fsi}}$$

où . représente la concaténation des listes.

* *Exercice* 4 : Problème des tours de Hanoï.

Au milieu du temple de Hanoï se trouvent 3 pieux de diamant. Autour du premier de ces pieux, 60 disques d'or étaient empilés il y a longtemps, par ordre de taille décroissante. Les bonzes ont reçu pour mission de déplacer un par un ces soixante disques sur le troisième pieu, en utilisant les 3 pieux et sans que jamais un disque se retrouve au-dessus d'un disque plus petit. Pour formaliser ce problème notons Hanoï(n, i, j) la liste des déplacements nécessaires pour amener les n disques du pieu numéro i jusqu'au pieu numéro j (un déplacement est un triplet (n, i, j) où i désigne le pieu de départ, j le pieu d'arrivée et n le numéro du disque). En s'inspirant de l'exemple précédent définir récursivement la procédure Hanoï. □

2.2 Notions de fonctionnelle et de point fixe

Commençons par introduire une notation fréquemment utilisée dans la suite. Si u(n) est une expression contenant éventuellement la lettre n ($n * f(n-1)$ par exemple), $\lambda n.u(n)$ désigne l'application f qui, à tout n, associe la valeur obtenue en évaluant u(n). Autrement dit $f = \lambda n.u(n)$ a le même sens que la notation classique $f : n \to u(n)$.

Cette notation permet d'exprimer d'une autre manière les définitions récursives. Donnons-en trois exemples :

a) L'application factorielle est l'unique solution, dans l'ensemble des applications de $\mathbb{N}$ dans lui-même, de l'équation :

$$f \underset{\text{def}}{=} \lambda n.\underline{\text{si}}\ n = 0\ \underline{\text{alors}}\ 1\ \underline{\text{sinon}}\ n * f(n-1)\ \underline{\text{fsi}}\,.$$

b) L'équation suivante admet au moins deux solutions dans l'ensemble des applications de $\mathbb{N}^2$ dans $\mathbb{N}$ (exercice 5) :

$$f \underset{\text{def}}{=} \lambda x, y.\underline{\text{si}}\ x = y\ \underline{\text{alors}}\ y + 1\ \underline{\text{sinon}}\ f(x, f(x-1, y+1))\ \underline{\text{fsi}}\,.$$

Exercice 5

Vérifier que

$$g = \lambda x, y.x + 1 \quad \text{et} \quad h = \lambda x, y.\underline{\text{si}}\ x \geqslant y\ \underline{\text{alors}}\ x + 1\ \underline{\text{sinon}}\ y + 1\ \underline{\text{fsi}}$$

sont des solutions de l'équation ci-dessus. □

c) L'équation

$$f \underset{\text{def}}{=} \lambda x.\underline{\text{si}}\ f(x) = 0\ \underline{\text{alors}}\ 1\ \underline{\text{sinon}}\ 0\ \underline{\text{fsi}}$$

n'admet quant à elle aucune solution dans l'espace des applications partout définies de $\mathbb{N}$ dans $\mathbb{N}$. Cependant, si l'on étend le domaine des solutions possibles, on trouve comme solution la fonction nulle part définie. C'est pourquoi nous allons étudier à partir de maintenant les équations récursives sur l'espace $\mathscr{F}(E, F)$ des fonctions partiellement définies sur E à valeurs dans F.

Précisons quelques termes. Une **fonctionnelle** (ou **opérateur**) est une application τ d'un espace de fonctions $\mathscr{F}(E, F)$ dans lui-même. Une fonction f de E dans F est un **point fixe** de τ si $\tau(f) = f$.

On peut associer à chaque équation du type $f \underset{def}{=} \lambda x . u(x)$ une fonctionnelle $\tau = \lambda f . \lambda x . u(x)$ admettant comme points fixes les solutions de l'équation. Par exemple les fonctions g et h de l'exercice 5 sont deux points fixes de la fonctionnelle

$$\tau = \lambda f . \lambda x, y . \underline{si}\ x = y\ \underline{alors}\ y + 1\ \underline{sinon}\ f(x, f(x - 1, y + 1))\ \underline{fsi}\ .$$

D'autre part on peut définir sur $\mathscr{F}(E, F)$ une relation d'ordre et une notion de convergence. Toute équation récursive admet alors une plus petite solution calculable par une méthode d'approximations successives. Ce sont ces idées que nous développons maintenant.

Définition 3 : Soit f et g deux fonctions de E dans F. On dit que f est moins définie que g (et on écrit $f \sqsubseteq g$) si, pour tout x de E tel que f(x) soit définie, g(x) est aussi définie et $g(x) = f(x)$.

La relation $\sqsubseteq$ est une relation d'ordre dans $\mathscr{F}(E, F)$. La fonction nulle part définie, notée Ω est le plus petit élément de cet ensemble. □

Définition 4 : Une suite (f_n) converge vers une fonction f si f_n est moins définie que f pour tout n et si, pour tout x tel que f(x) soit défini il existe n_0 vérifiant

$$n \geqslant n_0 \Rightarrow f_n(x) = f(x)\ . \qquad \square$$

Exercice 6

a) Vérifier que la fonction $f = \lambda x, y . \underline{si}(x \geqslant y\ \underline{et}\ \text{pair}(x - y))\ \underline{alors}\ x + 1\ \underline{fsi}$ est une solution de l'équation définie en b) ci-dessus.

b) Vérifier que f est moins définie que les solutions mentionnées à l'exercice 5.

c) Vérifier que f est la plus petite solution de cette équation. □

2.3 Calcul du plus petit point fixe d'une fonctionnelle par approximations successives

Considérons à nouveau la fonctionnelle

est le plus petit point fixe de τ. Pour cela construisons une suite $(f_n)_{n \geqslant 0}$ de fonctions de plus en plus définies convergeant vers f. Soit $f_0 = \Omega$ et pour tout $n \geqslant 0$, $f_{n+1} = \tau(f_n)$. La démonstration comporte deux parties :

a) (f_n) est une suite croissante convergeant vers f.

Montrons par récurrence sur n que, pour tout (x, y), $f_n(x, y)$ est défini si et seulement si $x - y$ est pair et $x - y$ appartient à l'intervalle $[0 : 2n - 2]$ et que, dans ce cas, $f_n(x, y) = x + 1$. Pour $n = 0$, $f_n = \Omega$ et l'assertion est vraie. Supposons l'assertion vérifiée pour un entier $n \geqslant 0$ quelconque et considérons deux entiers x, y tels que $x-y$ soit pair et $0 \leqslant x-y \leqslant 2(n+1)-2$:

— Si $x = y$ alors $f_{n+1}(x, y) = y + 1 = x + 1$.

— Si $x \neq y$, $f_n(x - 1, y + 1) = x - 1 + 1$ par hypothèse de récurrence et donc $f_{n+1}(x, y) = f_n(x, f_n(x - 1, y + 1)) = f_n(x, x) = x + 1$ ce qui termine la démonstration de l'assertion.

Montrons maintenant que (f_n) converge vers f. En effet (f_n) est moins défini que f pour tout n. D'autre part pour tout (x, y) tel que $x - y$ soit pair et $x - y \geqslant 0$, il existe n_0 tel que $x - y < 2n_0$. Et alors pour tout $n \geqslant n_0$, $f_n(x, y) = x + 1 = f(x, y)$ ce qui prouve la convergence.

b) f est le plus petit point fixe de τ.

Remarquons tout d'abord que la fonctionnelle τ vérifie la propriété suivante (de continuité) :

Si $(f_n)_{n \geqslant 0}$ converge vers f alors $(\tau(f_n))_{n \geqslant 0}$ converge vers $\tau(f)$. Autrement dit $\tau\left(\lim_n f_n\right) = \lim_n \tau(f_n)$.

Alors, comme $f_{n+1} = \tau(f_n)$ pour tout $n \geqslant 0$ on en déduit que

$$f = \lim_n f_n = \lim_n \tau(f_n) = \tau\left(\lim_n f_n\right) = \tau(f)$$

c'est-à-dire que f est un point fixe de τ.

D'autre part, montrons que si f' est une fonction vérifiant $f' = \tau(f')$ alors f est moins définie que f' ce qui achèvera la démonstration.

Tout d'abord, si $f' = \tau(f')$ alors $f_n \sqsubseteq f'$ pour tout $n \geqslant 0$. En effet $f_0 \sqsubseteq f'$ et, pour $n \geqslant 0$, $f_n \sqsubseteq f'$ implique $f_{n+1} = \tau(f_n) \sqsubseteq \tau(f') = f'$. En passant à la limite on déduit de $f_n \sqsubseteq f'$ que $f = \lim_n f_n \sqsubseteq f'$. C.Q.F.D.

Nous verrons au paragraphe 3 que cette technique d'approximation se généralise entièrement à toute fonctionnelle « continue ».

2.4 Calcul par substitution

Avant d'entreprendre une étude plus théorique, nous montrons, sur un exemple, que la plus petite solution d'une définition récursive est calculable algorithmiquement. Le chapitre suivant, consacré à l'étude des algorithmes, permettra d'étendre ce type de résultat.

Notre propos se réduit à montrer qu'une procédure récursive du type

<u>procedure</u> *fact* = (<u>entier</u> *n*) <u>entier</u> : <u>si</u> *n* = 0 <u>alors</u> 1 <u>sinon</u> *n* * *fact*(*n* − 1) <u>*fsi*</u>

calcule effectivement la plus petite solution de l'équation

$$\mathrm{f} \underset{\mathrm{def}}{=} \lambda \mathrm{f}.\lambda \mathrm{n}.\underline{\mathrm{si}}\ \mathrm{n} = 0\ \underline{\mathrm{alors}}\ 1\ \underline{\mathrm{sinon}}\ \mathrm{n} * \mathrm{f}(\mathrm{n} - 1)\ \underline{\mathrm{fsi}}\,.$$

Pour évaluer une expression e contenant des appels à la procédure *fact*, on applique autant de fois que nécessaire les deux règles suivantes :

a) Simplifier en effectuant chaque opération dont tous les arguments sont connus ou en évaluant les expressions conditionnelles dont l'argument booléen est connu.

b) Développer l'expression en remplaçant chaque terme $fact(e')$ par le terme

$$\underline{si}\ e' = 0\ \underline{alors}\ 1\ \underline{sinon}\ e' * fact(e' - 1)\ \underline{fsi}\,.$$

Désignons par Simpl(e) et Dev(e) les expressions obtenues en appliquant respectivement les règles a) et b). L'évaluation de e, si elle est possible, peut être réalisée en calculant une suite e_0, e'_1, e_1, ..., e'_n, e_n vérifiant $e_0 = e$, $e_n = \mathrm{Simpl}(e'_n)$ et $e'_n = \mathrm{Dev}(e_{n-1})$. L'évaluation s'arrête lorsque e_n est une valeur simple. Par exemple, pour $e = fact(2)$ on obtient le calcul suivant :

$$\begin{aligned}
e_0 &= fact(2)\\
e'_1 &= \underline{si}\ 2 = 0\ \underline{alors}\ 1\ \underline{sinon}\ 2 * fact(2 - 1)\ \underline{fsi}\\
e_1 &= 2 * fact(1)\,.\\
e'_2 &= 2 * \underline{si}\ 1 = 0\ \underline{alors}\ 1\ \underline{sinon}\ 1 * fact(1 - 1)\ \underline{fsi}\\
e_2 &= 2 * (1 * fact(0))\\
e'_3 &= 2 * (1 * \underline{si}\ 0 = 0\ \underline{alors}\ 1\ \underline{sinon}\ 0 * fact(0 - 1)\ \underline{fsi})\\
e_3 &= 2\,.
\end{aligned}$$

Plus généralement, pour tout n et pour tout x, posons $f'_n(x) = y$ si et seulement si l'évaluation de $fact(x)$ s'est terminée en y avant la n-ième étape (par exemple, pour $x = 2$, f'_0, f'_1, f'_2 sont indéfinis en x tandis que $f'_n(2) = 2$ pour $n \geqslant 3$). On peut montrer que la suite (f'_n) converge vers *fact* ce qui fournit un procédé de calcul effectif.

2.5 Conclusion

Résumons l'étude de l'exemple entreprise dans ce paragraphe. Nous avons vu qu'une définition récursive pouvait admettre dans l'espace des fonctions de E dans F plusieurs solutions. La plus petite de ces solutions, pour la relation d'ordre « moins définie que », peut être calculée par approximations successives. Ce premier résultat sera généralisé après une étude mathématique de la théorie du plus petit point fixe (§ 3). Cette solution peut aussi être calculée algorithmiquement, ce qui fait tout l'intérêt des définitions récursives. Ce second résultat sera prouvé rigoureusement au paragraphe 4 du chapitre 3. Pour mettre en évidence le fait que nous nous intéressons uniquement à la

plus petite solution, nous notons à partir de maintenant les équations sous la forme :

$$\varphi(x_1, \ldots, x_n) \Leftarrow \varepsilon .$$

3 ÉTUDE MATHÉMATIQUE : THÉORÈME DU POINT FIXE

> Le jour où, placée devant le problème de la résolution d'une équation $x = g(x)$ une machine sans aide apparente ou cachée de son cornac sera parvenue à l'idée de former la suite x_0, $x_1 = g(x_0)$, $x_2 = g(x_1)$, etc. Je ne serai pas loin de penser, moi aussi, à la possibilité de faire « penser » les ordinateurs.
>
> (J. DIEUDONNÉ, préface du livre « Computer in mathematical research ».)

Nous avons vu au paragraphe précédent que toute définition récursive admettait une plus petite solution, définie comme le plus petit point fixe d'une fonctionnelle. L'objectif du présent paragraphe est de prouver l'existence d'un plus petit point fixe pour une classe importante de fonctions. Le cadre dans lequel nous nous plaçons est assez vaste pour inclure le théorème des approximations successives de l'analyse mathématique aussi bien que le théorème d'existence d'une solution pour les systèmes algébriques en théorie des langages.

Le résultat général et fondamental que nous voulons démontrer est le suivant (les termes continue et inductif sont définis au § 3.1).

Théorème du point fixe

Soit $(X, \leqslant)$ un ensemble ordonné inductif et f une application continue de X dans X. Alors f admet un plus petit point fixe $\mu(f)$. De plus, $\mu(f) = \sup_{n \geqslant 0} \{ f^n(\perp) \}$ où $\perp$ désigne l'élément minimal de X. □

Avant de donner la démonstration de ce théorème, énonçons quelques définitions et propriétés immédiates.

3.1 Ensembles inductifs et fonctions continues

Soit $(X, \leqslant)$ un ensemble ordonné, non nécessairement totalement ordonné, et Y une partie de X.

Définition 5 :

- Un **majorant** (resp. un **minorant**) de Y est un élément a de X tel que, pour tout x de Y, on ait $x \leqslant a$ (resp. $x \geqslant a$).
- Le **maximum** (resp. le **minimum**) de Y (s'il existe) est un majorant (resp. un minorant) de Y qui appartient à Y. On montre facilement que, s'il existe, il est unique. On parle encore de plus grand (resp. plus petit) élément de Y.

• La **borne supérieure** (resp. la **borne inférieure**) de Y est le minimum (resp. le maximum), s'il existe, des majorants (resp. des minorants) de Y. On la note sup Y (resp. inf Y).

• Une **chaîne** est une suite croissante $x_0 \leqslant x_1 \leqslant \cdots \leqslant x_n \leqslant x_{n+1} \leqslant \cdots$ S'il existe n_0 tel que $n \geqslant n_0$ implique $x_n = x_{n_0}$, on dit que la chaîne est **stationnaire.**

• Un ensemble ordonné est **inductif** (au sens des chaînes) s'il admet un plus petit élément (noté $\perp$) et si toute chaîne admet une borne supérieure.

• Une application f d'un ensemble inductif X vers un autre ensemble inductif Z est continue (au sens des chaînes) si, pour toute chaîne $(x_n)_{n \geqslant 0}$ la suite $(f(x_n))_{n \geqslant 0}$ admet une borne supérieure vérifiant :

$$\sup_n f(x_n) = f\left(\sup_n (x_n)\right)$$ □

Exemple 8

• Etant donné deux ensembles E, F l'espace $\mathscr{F}(E, F)$ des fonctions partielles de E dans F ordonné par $\sqsubseteq$ est inductif (exercice 7).

• Si $(X, \leqslant)$ est un ensemble ordonné, l'ensemble X^n peut être muni d'une relation d'ordre encore notée $\leqslant$ et définie par :

$$(x_1, \ldots, x_n) \leqslant (y_1, \ldots, y_n) \Leftrightarrow x_1 \leqslant y_1 \ \underline{\text{et}} \ldots \underline{\text{et}}\ x_n \leqslant y_n\,.$$

Il est alors immédiat que si X est inductif il en est de même de X^n. □

Exercice 7

Montrer que l'ensemble $\mathscr{F}(E, F)$ muni de la relation $\sqsubseteq$ (moins défini que) vérifie les 2 propriétés suivantes :

• toute partie totalement ordonnée admet une borne supérieure,
• toute partie non vide admet une borne inférieure.

En déduire que $\mathscr{F}(E, F)$ est inductif. □

Exercice 8

Montrer que :

• Toute fonction continue est croissante.
• Si f et g sont continues alors $f \circ g$ est continue. □

Exercice 9

Si f est une fonction de $(E, \leqslant)$ dans $(F, \sqsubseteq)$ et si X est une partie de E, on note $f(X) = \{ y \in F \mid \exists x \in X, y = f(x) \}$. Montrer que, si f est croissante et si sup X et sup f(X) existent, alors $\sup f(X) \sqsubseteq f(\sup X)$. □

Les exercices suivants peuvent être omis en première lecture.

Exercice 10

a) Construire des ensembles ordonnés contenant des parties n'ayant pas de maximum, ou pas de majorant ou pas de borne supérieure.

b) Etudier ces concepts sur l'ensemble des parties finies de $\mathbb{N}$ ordonnées par inclusion. □

Exercice 11

Montrer que $\sup \varnothing$ existe si et seulement s'il y a un plus petit élément $\perp$ et que $\sup \varnothing = \perp$. □

Exercice 12

Montrer que les deux propositions suivantes sont équivalentes :
(i) toute chaîne et toute paire admettent une borne supérieure,
(ii) toute partie au plus dénombrable admet une borne supérieure. □

Exercice 13

1) Soit $(E, \leqslant)$ et $(F, \sqsubseteq)$ deux ensembles inductifs et soit f une fonction croissante de E vers F, montrer que :

a) pour toute chaîne C de E, f(C) est une chaîne (donc $\sup f(C)$ existe),

b) pour toute chaîne stationnaire C de E $\sup f(C) = f(\sup C)$,

c) En déduire que si toutes les chaînes de E sont stationnaires, une application de E vers F est continue si et seulement si elle est croissante.

2) En déduire que, pour les ensembles finis, les notions de continuité et de croissance sont identiques.

3) Trouver un exemple d'une fonction croissante et non continue (on pourra choisir pour E et F l'intervalle [0, 1] muni de l'ordre habituel sur les réels).

4) a) Montrer que si une chaîne (x_n) est non stationnaire alors $(\forall k \in \mathbb{N})$ $\left(\sup_n x_n \neq x_k\right)$.

b) En déduire que, si $(E, \leqslant)$ est inductif et contient une chaîne non stationnaire et si card $F \geqslant 2$, il existe une fonction croissante non continue de E vers F. □

L'exercice suivant montre le lien entre la notion de continuité au sens des chaînes et celle de continuité au sens topologique habituel.

* *Exercice* 14

Si $(E, \leqslant)$ est un ensemble inductif, on dit qu'une partie U de E est ouverte si :

a) $x \in U$ <u>et</u> $x \leqslant y$ implique $y \in U$

b) si $x_0 \leqslant \cdots \leqslant x_n \leqslant \cdots$ est une chaîne et

$$\sup_n x_n \in U \text{ alors } \{ x_n \}_{n \in \mathbb{N}} \cap U \neq \varnothing .$$

1) Montrer que ces ouverts définissent une topologie sur E c'est-à-dire une famille ζ d'ensemble appelés « ouverts » telle que

a) $\varnothing$ et E sont des ouverts,

b) une réunion d'ouverts est un ouvert,

c) une intersection finie d'ouverts est un ouvert.

2) Si (E, τ) et (F, τ') sont deux espaces topologiques, une application f de E vers F est continue si, pour tout ouvert U de F, $f^{-1}(U) = \{ x \in E \mid f(x) \in U \}$ est un ouvert.

Montrer que, si $(E, \leqslant)$ et $(F, \sqsubseteq)$ sont munis chacun de la topologie définie ci-dessus, une fonction est continue au sens topologique si et seulement si elle est continue au sens des chaînes. □

3.2 Théorème du point fixe : énoncé et démonstration

Théorème 1

Soit $(X, \leqslant)$ un ensemble inductif, f une fonction continue de X vers X. Alors f admet un plus petit point fixe $\mu(f)$ unique qui vérifie

$$\mu(f) = \sup_{n \geqslant 0} (f^n(\perp)) . \quad \square$$

Démonstration : La suite $(f_n(\perp))_{n\geqslant 0}$ est une chaîne. En effet, $\perp \leqslant f(\perp)$ et, puisque f est croissante (exercice 8), $f^{n-1}(\perp) \leqslant f^n(\perp)$ implique $f^n(\perp) \leqslant f^{n+1}(\perp)$. Puisque l'ensemble est inductif, cette chaîne admet une borne supérieure $\mu(f)$ et, f étant continue,

$$f(\mu(f)) = f(\sup_{n\geqslant 0} f_n(\perp)) = \sup_{n\geqslant 0} f^{n+1}(\perp) = \sup_{n\geqslant 1} f^n(\perp) = \mu(f) .$$

De plus, si x est un élément de X tel que $f(x) \leqslant x$, on a $f^n(\perp) \leqslant x$ pour tout $n \geqslant 0$. En effet, $\perp \leqslant x$ et, si $f^n(\perp) \leqslant x$, alors $f^{n+1}(\perp) \leqslant f(x) \leqslant x$ puisque f est croissante. Donc, $\mu(f) \leqslant x$. C.Q.F.D.

Remarque : Nous avons montré, en fait, que $\mu(f) = \inf \{ y \mid f(y) \leqslant y \}$.

3.3 Deux applications du théorème du point fixe

Avant de nous intéresser aux conséquences de ce théorème dans l'étude des définitions récursives, nous en donnons, sous forme d'exercices, quelques applications et généralisations.

Exercice 15 : Langages algébriques.

Soit A un alphabet, $\mathscr{L}(A)$ l'ensemble des parties de A^* (ou langages sur A) muni de la relation d'inclusion. Vérifier que $\mathscr{L}(A)$ est inductif.

On appelle expression régulière sur des variables $X_1 \ldots X_n$ toute expression formée, à partir de $X_1, \ldots, X_n$ et des parties finies de A^* en utilisant les opérations de réunion, de produit et de produit itéré. Par exemple

$$X_1 X_2 \cup (ab)^* X_1^* .$$

Si $U(X_1, \ldots, X_n)$ est une expression régulière, montrer que l'application

$$\lambda X_1, X_2, \ldots, X_n . U(X_1, \ldots, X_n)$$

de $(\mathscr{L}(A))^n$ dans $\mathscr{L}(A)$ est continue. En déduire que le système d'équations

$$X_i = U_i(X_1, \ldots, X_n) \qquad 1 \leqslant i \leqslant n$$

où les U_i sont des expressions régulières, admet une plus petite solution dans $(\mathscr{L}(A))^n$. $\square$

L'exercice suivant montre que le théorème du point fixe de l'analyse mathématique est un cas particulier du théorème 1.

* *Exercice* 16 : Théorème du point fixe de l'analyse classique.

1) a) Soit I_0 un intervalle fermé de $\mathbb{R}$. Montrer que l'ensemble $\mathrm{Int}(I_0)$ des intervalles fermés de $\mathbb{R}$ contenus dans I_0, muni de la relation : $I \leqslant I' \Leftrightarrow I \supset I'$ est inductif

(un intervalle est fermé s'il est de l'une des formes $]-\infty, +\infty[$, $]-\infty, b]$, $[a, +\infty[$, $[a, b]$ avec $a, b \in \mathbb{R}$).

b) Toute application f de I_0 dans I_0 induit une application $\tilde{f}$ de $Int(I_0)$ dans lui-même définie par

$$\tilde{f}(I) = \sup \{ J \in Int(I_0) \mid J \supseteq f(I) \} .$$

Compte tenu de la définition de $\leqslant$, $\tilde{f}(I)$ est le plus petit intervalle fermé contenant $f(I)$. Montrer que f est une application continue (au sens de l'analyse) dans I_0 si et seulement si $\tilde{f}$ est une application continue (au sens des chaînes) dans $Int(I_0)$.

2) Soient I un intervalle fermé de $\mathbb{R}$, non nécessairement borné, f une application de I dans I et k un nombre strictement inférieur à 1 tels que $| f(x) - f(y) | \leqslant k | x - y |$ pour tous x, y de I. Soit enfin $x_0 \in I$.

a) Montrer qu'il existe un intervalle $I_0 \subset I$, borné, contenant x_0 tel que $f(I_0)$ soit inclus dans I_0.

b) Soit $\tilde{f}$ l'application de $Int(I_0)$ dans lui-même induite par f. Vérifier que $\tilde{f}$ admet un plus petit point fixe, réduit à un point x de I_0 et que x est en réalité l'unique point fixe de f. □

3.4 Généralisation du théorème au cas de fonctions non continues

Indiquons, toujours sous forme d'exercices, quelques généralisations et propriétés classiques du théorème du point fixe :

Exercice 17

a) Montrer que, si $(X, \leqslant)$ est tel que toute partie admet une borne inférieure, alors c'est un treillis complet, c'est-à-dire que toute partie (y compris la partie vide) admet une borne inférieure et une borne supérieure.

b) Montrer que, dans ces conditions, toute fonction croissante admet un plus petit point fixe (on donnera une définition de ce dernier en terme de borne inférieure d'une partie de X).

c) Donner des exemples de treillis complets et de fonctions croissantes non continues. □

Exercice 18 : Une « construction » abstraite du plus petit point fixe lorsque f est seulement croissante.

Lorsqu'on essaie d'appliquer la méthode des approximations successives, on obtient une chaîne (x_n), admettant une borne supérieure x_ω ; comme f est croissante, $f(x_n)$ est plus petit que $f(x_\omega)$ pour tout n, donc $x_\omega \leqslant f(x_\omega)$. Mais f n'étant pas continue, on peut ne pas avoir l'égalité. L'idée est alors de définir $x_{\omega+n}$ de la même manière en partant de x_ω puis $x_{2\omega}$, $x_{3\omega}$, etc. Plus généralement on définit x_α pour tout ordinal α, par récurrence. Précisément, $x_0 = \bot$, $x_{\alpha+1} = f(x_\alpha)$ pour tout α et $x_\alpha = \sup_{\beta<\alpha} (x_\beta)$ si α est un ordinal limite (tel que ω, $2\,\omega$, ...). Soit Y l'ensemble des éléments de X qui sont de la forme x_α. Montrer qu'il existe α tel que $x_\alpha = x_{\alpha+1}$. Vérifier que x_α est le plus petit point fixe de f. □

Exercice 19

Soit $(X, \leqslant)$ un espace inductif et soient f et g deux fonctions continues de $(X, \leqslant)$ vers $(X, \leqslant)$ telles que $f \circ g = g \circ f$.

a) Montrer que l'ensemble des points fixes communs à f et g est non vide et admet un élément minimum. Donner l'expression de ce dernier.

b) Est-ce nécessairement le plus petit point fixe de f, de g ?

c) Est-ce un point fixe de $f \circ g$? Est-ce le plus petit ? □

Exercice 20

Soit $(X, \leqslant)$ un espace inductif et soient f, g deux fonctions continues de $(X, \leqslant)$ vers $(X, \leqslant)$ telles que $f(\perp) = g(\perp)$ et $f \circ g = g \circ f$. Montrer que f et g ont même plus petit point fixe. □

4 PREUVE DE PROPRIÉTÉS DU PLUS PETIT POINT FIXE D'UNE FONCTIONNELLE

Lorsque l'on se trouve en présence d'une fonctionnelle, on peut vouloir, soit calculer explicitement son plus petit point fixe, soit en rechercher des propriétés. En particulier, on peut essayer de montrer que deux fonctionnelles ont même plus petit point fixe ou que le plus petit point fixe est plus (ou moins) défini qu'une fonction g donnée.

Un certain nombre de règles ont été proposées pour prouver ces propriétés. Elles peuvent être réparties en 2 classes suivant qu'elles sont fondées sur la structure de la fonctionnelle (règles de Park et de Scott) ou sur la structure du domaine (règle d'induction structurelle). Ces règles peuvent être intégrées dans un système automatique de preuve. C'est le cas, en particulier, de la règle de Scott, base du système LCF développé par MILNER [MILR 76]. Un tel système suppose que les propriétés du domaine de définition, si elles sont utiles à la preuve, sont représentées par un ensemble d'axiomes et de règles d'inférence. Nous verrons, sur des exemples, comment réaliser une telle formalisation.

Donnons d'abord quelques exemples de propriétés. Dans ce paragraphe, toutes les fonctionnelles sont continues. Elles sont notées τ, τ_1 et leurs points fixes $\mu(\tau)$ et $\mu(\tau_1)$.

Exemple 9

a) Vérifier que le plus petit point fixe de la fonctionnelle

$$\tau = \lambda f.\lambda x.\underline{\text{si}}\ x > 100\ \underline{\text{alors}}\ x - 10\ \underline{\text{sinon}}\ f(f(x+11))\ \underline{\text{fsi}}$$

est moins défini que la fonction f_{91} définie par

$$[f_{91}(x) = \lambda x.\underline{\text{si}}\ x > 100\ \underline{\text{alors}}\ x - 10\ \underline{\text{sinon}}\ 91].$$

b) Soit τ la fonctionnelle $\lambda f.\lambda x.\underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ x\ \underline{\text{sinon}}\ f(f(h(x)))\ \underline{\text{fsi}}$ dans laquelle p est un prédicat et h une fonction. Vérifier que

$$\mu(\tau) \circ \mu(\tau) = \mu(\tau)$$

c) Considérons la définition récursive suivante de la concaténation de 2 listes

$$\alpha * \beta \Leftarrow \underline{\text{si}}\ \alpha = \Lambda\ \underline{\text{alors}}\ \beta\ \underline{\text{sinon}}\ \text{tête}(\alpha).(\text{queue}(\alpha) * \beta)\ \underline{\text{fsi}}\,.$$

Vérifier que $(\alpha * \beta) * \gamma = \alpha * (\beta * \gamma)$. □

4.1 Méthode de récurrence sur la fonctionnelle

a) Règle de Park

> Soit τ une fonctionnelle croissante. S'il existe une fonction g telle que $\tau(g) \sqsubseteq g$, alors τ admet un plus petit point fixe et $\mu(\tau) \sqsubseteq g$.

Justifions cette règle :

i) Si Λ est une famille non vide de fonctions, on peut définir Inf Λ par la relation

$$\forall x\ \forall y \quad \text{Inf}\ \Lambda(x) = y \Leftrightarrow \text{pour tout } f \in \Lambda, \quad f(x) = y\,.$$

ii) S'il existe g tel que $\tau(g) \sqsubseteq g$, l'ensemble $\Lambda = \{ h : D \to D \mid \tau(h) \sqsubseteq h \}$ est non vide et admet une borne inférieure f. Vérifions que $f = \mu(\tau)$.

— Pour tout $h \in \Lambda$, $f \sqsubseteq h$ et, puisque τ est croissante,

$$\tau(f) \sqsubseteq \tau(h) \sqsubseteq h\,.$$

Donc $\tau(f)$ est un minorant de Λ et :

$$\tau(f) \sqsubseteq f\,.$$

— Puisque τ est croissante, il vient de l'inégalité précédente

$$\tau^2(f) \sqsubseteq \tau(f)\,.$$

Donc $\tau(f) \in \Lambda$ et $f \sqsubseteq \tau(f)$. Ainsi, f est un point fixe de τ. Puisque tout point fixe de τ appartient à Λ, f est le plus petit.

iii) Si g vérifie $\tau(g) \sqsubseteq g$, g appartient à Λ et $f \sqsubseteq g$. C.Q.F.D.

Exemple 10 : Application de la règle de Park.

Considérons la fonctionnelle

$$\tau = \lambda f.\lambda x.\underline{\text{si}}\ x > 100\ \underline{\text{alors}}\ x - 10\ \underline{\text{sinon}}\ f(f(x + 11))\ \underline{\text{fsi}}$$

introduite dans l'exemple précédent et vérifions qu'elle admet un plus petit point fixe moins défini que f_{91}.

D'après la règle précédente, il suffit de montrer que $\tau(f_{91}) \sqsubseteq f_{91}$.

$$\tau(f_{91})(x) = \underline{\text{si}}\ x > 100\ \underline{\text{alors}}\ x - 10\ \underline{\text{sinon}}\ f_{91}(f_{91}(x + 11))\ \underline{\text{fsi}}$$

$$\begin{aligned}\tau(f_{91})(x) = \underline{\text{si}}\ x > 100\ &\underline{\text{alors}}\ x - 10\\ &\underline{\text{sinon}}\ \underline{\text{si}}\ x + 11 > 100\ \underline{\text{alors}}\ f_{91}(x + 1)\\ &\underline{\text{sinon}}\ f_{91}(91)\ \underline{\text{fsi}}\ \underline{\text{fsi}}\end{aligned}$$

Alors si $x = 100, \tau(f_{91})(x) = f_{91}(x+1) = f_{91}(101) = 91$

si $89 < x < 100, \tau(f_{91})(x) = f_{91}(x+1) = 91$

si $x \leqslant 89, \tau(f_{91})(x) = f_{91}(91) = 91$. □

b) Règle de troncation

> Si σ_1, σ_2, τ sont trois fonctionnelles continues telles que
> i) $\sigma_1(\Omega) \sqsubseteq \sigma_2(\Omega)$
> ii) pour tout $i \geqslant 0, \sigma_1(\tau^i(\Omega)) \sqsubseteq \sigma_2(\tau^i(\Omega)) \Rightarrow \sigma_1(\tau^{i+1}(\Omega)) \sqsubseteq \sigma_2(\tau^{i+1}(\Omega))$
> alors $\sigma_1(\mu(\tau)) \sqsubseteq \sigma_2(\mu(\tau))$.

Justification : $\mu(\tau) = \sup_i \tau^i(\Omega)$. Par récurrence sur l'indice i, l'assertion $\sigma^1(\tau^i(\Omega)) \sqsubseteq \sigma^2(\tau^i(\Omega))$ pour tout $i \geqslant 0$. Mais, par continuité,

$$\sigma_1(\mu(\tau)) = \sup_i \sigma_1(\tau^i(\Omega)) \sqsubseteq \sup_i \sigma_2(\tau^i(\Omega)) = \sigma_2(\mu(\tau)).$$

Cette règle permet essentiellement de justifier la règle suivante dite règle d'induction de Scott.

c) Règle de Scott (forme simple)

> Soient σ_1, σ_2, τ trois fonctionnelles continues telles que
> i) $\sigma_1(\Omega) \sqsubseteq \sigma_2(\Omega)$
> ii) pour tout $f, \sigma_1(f) \sqsubseteq \sigma_2(f) \Rightarrow \sigma_1(\tau(f)) \sqsubseteq \sigma_2(\tau(f))$
> alors $\sigma_1(\mu(\tau)) \sqsubseteq \sigma_2(\mu(\tau))$.

Cette règle s'avère très puissante dans la pratique et présente l'avantage de ne pas faire référence aux entiers. Cela permet de réaliser des systèmes automatiques de preuves plus simples. Donnons immédiatement quelques généralisations de cette règle.

La première consiste à considérer l'espace $(\mathscr{F}(E, F))^n$ des n-uplets $(f_1, \ldots, f_n)$ de fonctions. Nous avons signalé à l'exemple 8 que $(\mathscr{F}(E, F))^n$ est un espace inductif. Si $\tau_1, \ldots, \tau_n$ sont des fonctionnelles continues à n variables, l'application $\tau = (\tau_1, \ldots \tau_n)$ est une fonctionnelle de $(\mathscr{F}(E, F))^n$ sur lui-même. Son plus petit point fixe n'est autre que la plus petite solution du système d'équations :

$$\begin{cases} f_1(x) = \tau_1(f_1, \ldots, f_n)(x) \\ \quad \vdots \qquad\qquad \vdots \\ f_n(x) = \tau_n(f_1, \ldots, f_n)(x) \end{cases}$$

Nous noterons f le n-uplet $(f_1, \ldots, f_n)$.

La deuxième généralisation concerne la notion de propriété admissible.

Définition 6 : Une propriété P sur $(\mathscr{F}(E, F))^n$ est dite admissible s'il existe deux familles $(\sigma_i)_{i \in I}$ et $(\sigma'_i)_{i \in I}$ de fonctionnelles continues telles que P puisse se mettre sous la forme

$$P(f) \;=\; \sigma_i(f) \sqsubseteq \sigma'_i(f) \quad \text{pour tout} \quad i \in I\,. \qquad \square$$

Exemple 11 : Quelques propriétés admissibles.

i) « $f \circ f = f$ » est une propriété admissible. En effet cette égalité équivaut à :

$$f \circ f \sqsubseteq f \qquad \underline{\text{et}} \qquad f \sqsubseteq f \circ f\,.$$

ii) « $f_1 = f_2$ » est une propriété admissible. En effet il suffit de prendre dans la définition 6

$$I = \{1, 2\}\,, \qquad \sigma_1(f_1, f_2) = \sigma'_2(f_1, f_2) = f_1\,,$$
$$\sigma_2(f_1, f_2) = \sigma'_1(f_1, f_2) = f_2\,.$$

iii) Par contre « f n'est pas partout définie » n'est pas admissible.

En effet cette dernière propriété peut encore s'écrire « $\exists x f(x)$ indéfini » qui ne peut s'exprimer comme une conjonction d'inclusions. Vérifions d'ailleurs que la règle de Scott ne s'applique pas à cette propriété en considérant la fonctionnelle

$$\tau = \lambda f.\lambda x.\underline{\text{si}}\ x = 0\ \underline{\text{alors}}\ 1\ \underline{\text{sinon}}\ x * f(x-1)\ \underline{\text{fsi}}$$

$\mu(\tau)$, qui est la procédure *fact* comme nous l'avons vu au paragraphe 2.4 est partout définie. Par contre, pour tout $n \geqslant 0$, la fonction $\tau^n(\Omega)$ définie par :

$$\tau^n(\Omega)\,(x) = \underline{\text{si}}\ x < n\ \underline{\text{alors}}\ x\,!\ \underline{\text{sinon}}\ \text{indéfini}\ \underline{\text{fsi}}$$

est une fonction partielle. $\square$

d) Règle de Scott généralisée

> Soient P une propriété admissible et τ une fonctionnelle continue sur $(\mathscr{F}(E, F))^n$. Si
>
> $.P(\Omega)$
>
> et $.P(f) \Rightarrow P(\tau(f))$ pour tout f appartenant à $(\mathscr{F}(E, F))^n$
>
> alors
>
> $P(\mu(\tau))$

4.2 Exemples d'applications de la règle d'induction de Scott

a) Propriétés indépendantes du domaine d'interprétation

Soit la fonctionnelle

$$\tau = \lambda f.\lambda x\ \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ x\ \underline{\text{sinon}}\ f(f(h(x)))\ \underline{\text{fsi}}\,.$$

Montrons que $\mu(\tau)$ est idempotente c'est-à-dire que $\mu(\tau) \circ \mu(\tau) = \mu(\tau)$.

Considérons pour cela la propriété P(f)

$$\mu(\tau) \circ f = f$$

et raisonnons par induction de Scott. Montrons d'abord $P(\Omega)$ c'est-à-dire $\mu(\tau) \circ \Omega = \Omega$. Pour tout x, Ω est indéfini en x, donc aussi $\mu(\tau) \circ \Omega$.

Montrons que $P(f) \Rightarrow P(\tau(f))$.

$$\begin{aligned}\mu(\tau)\,(\tau(f)\,(x)) &= \mu(\tau)\,(\underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ x\ \underline{\text{sinon}}\ f(f(h(x)))\ \underline{\text{fsi}}) \\ &= \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ \mu(\tau)\,(x)\ \underline{\text{sinon}}\ \mu(\tau)\,(f(f(h(x))))\ \underline{\text{fsi}} \\ &= \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ \mu(\tau)\,(x)\ \underline{\text{sinon}}\ f(f(h(x)))\ \underline{\text{fsi}} \quad \text{par hypothèse} \\ &= \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ x\ \underline{\text{sinon}}\ f(f(h(x)))\ \underline{\text{fsi}} = \tau(f)\,(x)\end{aligned}$$

puisque $\mu(\tau)\,(x) = \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ x\ \underline{\text{sinon}}\ \mu(\tau)\,(\mu(\tau)\,(h(x)))\ \underline{\text{fsi}}$. C.Q.F.D.

* *Exercice* 21

Soit

$$\tau_1 = \lambda f.\lambda x, y.\underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ y\ \underline{\text{sinon}}\ h(f(k(x), y))\ \underline{\text{fsi}}$$

et

$$\tau_2 = \lambda f.\lambda x, y.\underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ y\ \underline{\text{sinon}}\ f(k(x), h(y))\ \underline{\text{fsi}}\,.$$

Vérifier que $\mu(\tau_1) = \mu(\tau_2)$. □

b) Equivalence de deux langages définis par des systèmes à point fixe.

La règle de Scott s'applique également aux fonctions continues sur des espaces plus généraux. Il suffit d'y remplacer Ω par l'élément minimum $\perp$. En particulier, en théorie des langages (voir l'exercice 15), on considère des applications continues sur l'ensemble des langages sur un alphabet A, c'est-à-dire sur l'ensemble $\mathscr{P}(A^*)$. Cet ensemble est inductif, admet $\varnothing$ comme plus petit élément et est muni de deux opérations continues, l'union et le produit, défini par la relation

$$AB = \{\, \alpha\beta \mid \alpha \in A, \beta \in B \,\}\,.$$

Si $A = \{\alpha\}$, on écrit simplement αB (resp. $\alpha \cup B$) au lieu de $\{\alpha\}\,B$ (resp. $\{\alpha\} \cup B$).

Considérons ici l'alphabet $\{\alpha, \beta, \gamma, \delta, \varepsilon\}$ et les applications F_1, F_2, F_3 définies par

$$\begin{aligned}F_1(X) &= (\alpha X\beta \cup \gamma)\,(\delta X \cup \varepsilon) \\ F_2(Y, Z) &= (\alpha YZ\beta \cup \gamma) \\ F_3(Y, Z) &= \delta YZ \cup \varepsilon\end{aligned}$$

et désignons par L_X le plus petit point fixe de F_1 et par (L_Y, L_Z) celui de (F_2, F_3). Vérifions que

$$\boxed{L_X = L_Y L_Z}\,.$$

Soit $P(X, Y, Z) =$ « $X = YZ$ ». P est une propriété admissible. Clairement $P(\varnothing, \varnothing, \varnothing)$ est vrai. Supposons maintenant $P(X, Y, Z)$ vérifiée et montrons que $P(X', Y', Z')$ est vrai où

$$X' = F_1(X)\,, \quad Y' = F_2(Y, Z)\,, \quad Z' = F_3(Y, Z)\,,$$

$$\begin{aligned} X' &= a(\alpha X\beta \cup \gamma)\,(\delta X \cup \varepsilon) \\ &= a(\alpha YZ\beta \cup \gamma)\,(\delta YZ \cup \varepsilon) \quad \text{par récurrence} \\ &= Y'Z'\,. \end{aligned}$$

Cette propriété d'équivalence a des applications intéressantes en programmation dont nous décrivons un exemple ci-dessous.

Exemple 12 : Equivalence entre deux stratégies de parcours d'une forêt.

Une forêt est une suite d'arbres. Un arbre est un graphe connexe sans circuit, muni d'une racine r. Chaque point est donc relié à la racine de l'arbre par un chemin unique. On dit que y est fils de x si x est le prédécesseur de y sur le chemin reliant r à y. Inversement, on dit que x est le père de y. L'arbre est dit orienté si l'on ordonne les fils de tout point (ou nœud) de l'arbre. Si un nœud admet un ou plusieurs enfant, sa descendance est encore une forêt. Trois fonctions fondamentales permettent de passer d'un nœud à l'autre de la forêt :

F(x) est le premier fils de x s'il existe
C(x) est le premier frère de x s'il existe (frère cadet)
P(x) est le père de x si x n'est pas une racine.

Considérons d'autre part les deux prédicats

f(x) vrai si et seulement si x a un fils
c(x) vrai si et seulement si x a un frère cadet

et notons $\bar{f}$, $\bar{c}$ les négations de f, c.

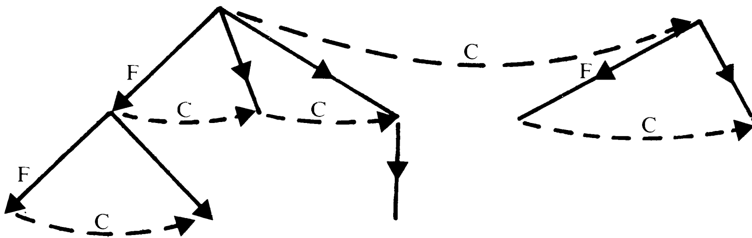


Figure 2 Schéma d'une forêt.

On dit qu'une forêt est parcourue si tous ses nœuds ont été visités en utilisant les fonctions F, C, P et les prédicats f, c. Le parcours d'une forêt peut être représentée par la suite des opérations réalisées ou calcul, c'est-à-dire par un mot sur l'alphabet $\{ F, P, C, f, \bar{f}, c, \bar{c} \}$. Décrivons deux stratégies possibles de parcours d'une forêt.

Stratégie 1 : Pour parcourir une forêt, visiter la racine r de son premier arbre, parcourir la forêt, si elle existe, constituée par la descendance de r, parcourir la forêt, si elle existe, constituée par les frères cadets de r et leur descendance.

Stratégie 2 : Pour parcourir une forêt, parcourir successivement chaque arbre de cette forêt. Pour parcourir un arbre, visiter sa racine, puis parcourir successivement chaque arbre de sa descendance.

Désignons par $\mathscr{C}$, $\mathscr{C}'$ les ensembles de calculs associés aux traversées de toutes les forêts selon les stratégies 1 et 2. Nous voulons montrer que $\mathscr{C}$ et $\mathscr{C}'$ coïncident et, pour cela, nous assurer qu'ils vérifient un système d'équations du type précédent.

Il est facile de voir que $\mathscr{C}$ est solution de l'équation

$$X = (\mathrm{fFXP} \cup \bar{\mathrm{f}})(\mathrm{cCX} \cup \bar{\mathrm{c}}).$$

qui se met sous la forme $X = F_1(X)$ en posant

$$\alpha = \mathrm{fF}\,,\quad \beta = \mathrm{P}\,,\quad \gamma = \bar{\mathrm{f}}\,,\quad \delta = \mathrm{cC}\,,\quad \varepsilon = \bar{\mathrm{c}}$$

(avec les notations du début de paragraphe)

Pour la 2e stratégie, introduisons l'ensemble $\mathscr{C}_2$ des calculs associés aux parcours des arbres et l'ensemble $\mathscr{C}_3$ des calculs associés aux parcours d'une suite éventuellement vide d'arbres. $\mathscr{C}'$ vérifie

$$\mathscr{C}' = \mathscr{C}_2 \mathscr{C}_3\,.$$

D'autre part $\mathscr{C}_2$ et $\mathscr{C}_3$ sont liés par les relations

$$\mathscr{C}_3 = \mathrm{c}\mathrm{C}\mathscr{C}_2 \mathscr{C}_3 \cup \bar{\mathrm{c}}$$

et $$\mathscr{C}_2 = \mathrm{fF}\mathscr{C}'\mathrm{P} \cup \bar{\mathrm{f}} = \mathrm{fF}\mathscr{C}_2 \mathscr{C}_3 \mathrm{P} \cup \bar{\mathrm{f}}\,.$$

En définitive, $(\mathscr{C}_2, \mathscr{C}_3)$ est solution du système

$$\begin{aligned} Y &= \mathrm{fFYZP} \cup \bar{\mathrm{f}} \\ Z &= \mathrm{cCYZ} \cup \bar{\mathrm{c}} \end{aligned}$$

qui se met encore sous la forme

$$\begin{aligned} Y &= F_2(Y, Z) \\ Z &= F_3(Y, Z)\,. \end{aligned}$$

Il résulte donc du résultat précédent que

$$\mathscr{C} = L_X = L_Y L_Z = \mathscr{C}_2 \mathscr{C}_3 = \mathscr{C}'\,.$$

Le formalisme des ensembles de calculs sera développé au paragraphe 5 du chapitre 3. □

c) Propriétés dépendant du domaine d'interprétation

Soit A^* l'ensemble des mots sur l'alphabet A ; on se donne trois fonctions de base :

— $a.\alpha$ qui est l'adjonction de la lettre a en tête du mot α
— $\underline{\text{tête}}(\alpha)$ qui donne la première lettre de α.
— $\underline{\text{queue}}(\alpha)$ qui donne le mot β tel que $\alpha = \underline{\text{tête}}(\alpha).\beta$.

Ces fonctions vérifient les propriétés suivantes :

Si $a \in V$ et $\alpha \in V^*$ alors $a.\alpha \neq \Lambda$, $\underline{\text{tête}}(a.\alpha) = a$, $\underline{\text{queue}}(a.\alpha) = \alpha$ et si $\alpha \neq \Lambda$ alors $\underline{\text{tête}}(\alpha).\underline{\text{queue}}(\alpha) = \alpha$.
Considérons alors la fonctionnelle

$$\tau = \lambda f.\lambda\alpha, \beta.\underline{\text{si}}\ \alpha = \Lambda\ \underline{\text{alors}}\ \beta\ \underline{\text{sinon}}\ \underline{\text{tête}}(\alpha).f(\underline{\text{queue}}(\alpha), \beta)\ \underline{\text{fsi}}$$

son point fixe $\mu(\tau)(\alpha, \beta)$ est la concaténation de α et β que nous noterons $\alpha * \beta$. Montrons que cette opération est associative, c'est-à-dire que :

$$[(\alpha * \beta) * \gamma = \alpha * (\beta * \gamma)] .$$

Remarque 1 : Avant de poursuivre, le lecteur est invité à chercher la propriété sur laquelle portera l'induction. □

Remarque 2 : Vérifions que $(a.\alpha) * \beta = a.(\alpha * \beta)$.
En effet $(a.\alpha) * \beta = \underline{\text{si}}\ a.\alpha = \Lambda\ \underline{\text{alors}}\ \beta\ \underline{\text{sinon}}\ \underline{\text{tête}}(a.\alpha).(\underline{\text{reste}}(a.\alpha) * \beta)\ \underline{\text{fsi}}$
$= a.(\alpha * \beta)$. □
La propriété $P(f)$ est $f(\alpha, \beta) * \gamma = f(\alpha, \beta * \gamma)$.
En effet, $P(\Omega)$ est $\Omega(\alpha, \beta) * \gamma = \Omega(\alpha, \beta * \gamma)$.
Et les deux membres de l'équation sont également indéfinis.
Montrons que $P(f) \Rightarrow P(\tau(f))$.
Par définition de τ :

$$\begin{aligned}\tau(f)(\alpha, \beta * \gamma) &= \underline{\text{si}}\ \alpha = \Lambda\ \underline{\text{alors}}\ \beta * \gamma\ \underline{\text{sinon}}\ \underline{\text{tête}}(\alpha).f(\underline{\text{queue}}(\alpha), \beta * \gamma)\ \underline{\text{fsi}}\\ &= \underline{\text{si}}\ \alpha = \Lambda\ \underline{\text{alors}}\ \beta * \gamma\ \underline{\text{sinon}}\ \underline{\text{tête}}(\alpha).\ f(\underline{\text{queue}}(\alpha), \beta) * \gamma]\ \underline{\text{fsi}}\\ &\quad \text{par hypothèse de récurrence}\\ &= \underline{\text{si}}\ \alpha = \Lambda\ \underline{\text{alors}}\ \beta * \gamma\ \underline{\text{sinon}}[\underline{\text{tête}}(\alpha).f(\underline{\text{queue}}(\alpha), \beta)] * \gamma\ \underline{\text{fsi}}\\ &\quad \text{d'après la remarque 2}\\ &= [\tau(f)(\alpha, \beta)] * \gamma . \quad \text{C.Q.F.D.}\end{aligned}$$

On constate sur cet exemple que le choix de la propriété P est fondamental et qu'il ne peut en général être automatisé.

4.3 Règle d'induction structurelle

a) Notion d'ensemble bien fondé

Exemple 13 : Retour sur l'associativité de la concaténation.

Démontrons maintenant la propriété $\alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma$ précédente par récurrence sur α.
Si $\alpha = \Lambda$, $\alpha * (\beta * \gamma) = \beta * \gamma = (\alpha * \beta) * \gamma$

Si $\alpha = a.\alpha'$ alors

$$\begin{aligned}
\alpha * (\beta * \gamma) &= a.\alpha' * (\beta * \gamma) \\
&= a.(\alpha' * (\beta * \gamma)) && \text{par définition} \\
&= a.((\alpha' * \beta) * \gamma) && \text{par hypothèse de récurrence} \\
&= (a.(\alpha' * \beta)) * \gamma && \text{par définition} \\
&= ((a.\alpha') * \beta) * \gamma && \text{par définition} \\
&= (\alpha * \beta) * \gamma .
\end{aligned}$$

Cette méthode de preuve s'avère ici plus simple que la précédente ; il ne s'agit cependant pas d'un phénomène général. □

Dans l'exemple précédent, la « récurrence sur α » est basée sur l'ordre : $\alpha \leqslant \beta$ si α est facteur droit de β (en effet $\alpha \leqslant a.\alpha$ pour tout a, α).

Cet ordre possède la propriété d'être **bien fondé** au sens de la définition suivante :

Définition 7

Une relation d'ordre sur un ensemble E est bien fondée si toute chaîne décroissante de E est stationnaire. □

b) Enoncé de la règle d'induction structurelle

Soient $(D, \leqslant)$ un ensemble muni d'une relation d'ordre bien fondée, P une propriété sur D. Si pour tout $a \in D$ (P(b) pour tout $b < a$) $\Rightarrow$ P(a) Alors pour tout $a \in D$, P(a) est vrai.

($<$ désigne la relation d'ordre stricte induite par $\leqslant$).

c) Application

Vérifions que le plus petit point fixe de la fonctionnelle

$$\tau = \lambda f.\lambda x.\underline{\text{si}}\ x > 100\ \underline{\text{alors}}\ x - 10\ \underline{\text{sinon}}\ f(f(x + 11))\ \underline{\text{fsi}}$$

est bien la fonction f_{91} définie dans l'exemple 9.

L'ordre choisi est $y \prec x \Leftrightarrow x < y \leqslant 101$. On constate en effet expérimentalement que le calcul du plus petit point fixe en 91 par exemple implique ce calcul en 92, 93.

Supposons alors que pour tout y tel que $y \prec x$ on ait $\mu(\tau)(y) = f_{91}(y)$ et montrons que $\mu(\tau)(x) = f_{91}(x)$:

— Si $x > 100$ c'est immédiat

— Si $100 \geqslant x \geqslant 90$, $\mu(\tau)(x) = \mu(\tau)(\mu(\tau)(x + 11)) = \mu(\tau)(x + 11 - 10)$
$= \mu(\tau)(x + 1)$

et puisque $x + 1 \prec x$ on en déduit

$$\mu(\tau)(x) = \mu(\tau)(x + 1) = f_{91}(x + 1) = f_{91}(x)$$

— Si $x < 90$, $\mu(\tau)(x) = \mu(\tau)(\mu(\tau)(x + 11))$

$$= \mu(\tau)(f_{91}(x) + 11)) \quad \text{puisque} \quad x + 11 \prec x$$

mais $f_{91}(x + 11) = 91$ et $\mu(\tau)(91) = 91$ puisque $91 \prec x$

donc $\mu(\tau)(x) = \mu(\tau)(91) = 91 = f_{91}(x)$. C.Q.F.D.

Comparons cette preuve avec une preuve utilisant la règle de Scott. On a déjà montré, en utilisant la règle de Park que $\mu(\tau) \sqsubseteq f_{91}$. Pour montrer l'inclusion inverse, notons que f_{91} est le plus petit point fixe de la fonctionnelle :

$$\sigma = \lambda f.\lambda x.\underline{\text{si}}\ x > 100\ \underline{\text{alors}}\ x - 10\ \underline{\text{sinon}}\ f(x + 1)\ \underline{\text{fsi}}.$$

En posant $g = \mu(\tau)$ il nous faut vérifier que g est plus défini que $\mu(\sigma)$. Soit encore, en utilisant la règle de Park que

$$\sigma(g) \sqsubseteq g \quad \text{ou que} \quad \sigma(g) \sqsubseteq \tau(g).$$

Or $\sigma(g) = \lambda x.\underline{\text{si}}\ x > 100\ \underline{\text{alors}}\ x - 10\ \underline{\text{sinon}}\ g(x + 1)\ \underline{\text{fsi}}$
et $\tau(g) = \lambda x.\underline{\text{si}}\ x > 100\ \underline{\text{alors}}\ x - 10\ \underline{\text{sinon}}\ g(g(x + 11))\ \underline{\text{fsi}}$
Ainsi $\sigma(g) \sqsubseteq \tau(g)$ est vérifié si l'on peut prouver que

$$g \sqsubseteq \lambda x.g(g(x + 10)).$$

Pour cela appliquons la règle de Scott à la double propriété suivante

$$P(f)\lambda = f \sqsubseteq \lambda x.g(g(x + 10))\ \underline{\text{et}}\ f \sqsubseteq g$$

$P(\Omega)$ est immédiat.

Supposons $P(f)$ et prouvons $\tau(f) \sqsubseteq \lambda x.g(g(x + 10))$ et $\tau(f) \sqsubseteq g$.

Par hypothèse de récurrence : $f \sqsubseteq g$

donc $\tau(f) \sqsubseteq \tau(g)$ par monotonie de τ

ainsi (1) $\tau(f) \sqsubseteq g$ car $g = \mu(\tau)$

ce qui prouve la deuxième partie de $P(\tau(f))$.

Pour démontrer que $\tau(f)$ est moins définie que $\lambda x.g(g(x + 10))$ commençons par expliciter $g(g(x + 10))$.

$$g(g(x + 10)) = g(\tau(g)(x + 10)) \quad \text{car} \quad g = \mu(\tau)$$

(2) $g(g(x+10)) = \underline{\text{si}}\ x+10 > 100\ \underline{\text{alors}}\ g(x)\ \underline{\text{sinon}}\ g(g(g(x+10+11)))\ \underline{\text{fsi}}$.

Effectuons alors un raisonnement par cas :

— $x + 10 > 100$, on tire alors immédiatement de (1) et (2) que

$$\tau(f) \sqsubseteq \lambda x.g(g(x + 10))$$

— $x + 10 \leqslant 100$, des hypothèses de récurrence :

$$f \sqsubseteq g$$

et $\lambda x.f(x + 11) \sqsubseteq \lambda x.g(g(x + 21))$

on tire $\lambda x.f(f(x + 11)) \sqsubseteq \lambda x.g(g(g(x + 21)))$

et donc avec (2) : $\tau(f) \sqsubseteq \lambda x.g(g(x + 10))$.

Ainsi, d'après la règle d'induction de Scott, $\mu(\tau) = g$ est moins définie que $\lambda x.g(g(x + 10))$ et donc g est plus défini que $\mu(\sigma)$. C.Q.F.D.

4.4 Conclusion

Deux règles se révèlent particulièrement puissantes :

a) La règle de Scott, malgré son apparente simplicité, permet à notre connaissance de prouver toutes les propriétés effectivement démontrables. En particulier, notons que la théorie du point fixe permet d'axiomatiser les structures de données, récursives ou non, et que la règle de Scott est ainsi un outil de preuves sur les données. Celui-ci est effectivement réalisé dans le système LCF de MILNER.

La principale difficulté dans l'application de la règle, commune à toutes les règles de récurrence, résulte du fait que pour prouver une propriété P_0, il faut, en général démontrer, par récurrence, une propriété P beaucoup plus forte que P_0. L'exercice suivant illustre bien cette difficulté.

Exercice 22

Soient τ_1, τ_2 deux fonctionnelles vérifiant les propriétés

$$\tau_1(\Omega) = \tau_2(\Omega) \quad \text{et} \quad \tau_1^i\,\tau_2 = \tau_2\,\tau_1 \quad \text{pour un entier} \quad i \geqslant 1.$$

Vérifier que $\mu(\tau_1) = \mu(\tau_2)$ en appliquant :

i) la règle d'induction par troncation,
ii) la règle de Scott. □

b) La règle d'induction structurelle dépasse le cadre de la théorie du point fixe. Il s'agit de vérifier une propriété P sur un domaine D et P peut être absolument quelconque. On utilisera d'ailleurs au chapitre 4 cette règle pour effectuer des preuves de terminaison de programmes. L'usage de cette règle dans des systèmes automatiques de preuves se révèle assez difficile.

Notons enfin que l'on peut prouver certaines propriétés en utilisant la règle de Scott sans qu'on puisse le faire en utilisant uniquement la règle de Park.

5 CONCLUSION ET COMMENTAIRES BIBLIOGRAPHIQUES

Ce chapitre s'est attaché à deux objectifs.

a) Dégager les notions de problème et d'énoncé. Poser un problème revient à définir de nouveaux accès à partir d'accès primitifs. Il existe une grande variété de langages permettant d'énoncer un problème. On trouvera dans les

ouvrages de SHOENFIELD [SHO, 67] et de MANNA [MAN, 74] de bonnes présentations du calcul des prédicats. Nous avons pris le parti de présenter de façon plus approfondie les énoncés récursifs qui ont fait l'objet de nombreux travaux. A ce sujet on pourra se reporter à un article synthétique de MANNA, NESS et VUILLEMIN [MAN, 73].

b) Le deuxième objectif, très lié au premier, consiste en la présentation de la théorie du point fixe. Celle-ci, connue depuis longtemps en mathématiques (méthode des approximations successives en analyse), après avoir été utilisée en logique [KLE, 71] et en théorie des langages [CHO, 69] a pris un essor en sémantique des programmes à la suite des travaux de SCOTT [SCO, 69]. Outre le livre de MANNA déjà cité [MAN, 74], on pourra consulter un article de SCOTT [SCO, 77]. Cette théorie est utilisée à de multiples reprises dans la suite de cet ouvrage : au chapitre 3 lors de l'étude des schémas récursifs (§ 4) et du calcul relationnel (§ 5), au chapitre 4 pour la justification des méthodes de preuves de programmes (§ 5, § 6), et enfin comme outil de définition de la sémantique d'un langage de programmation au chapitre 5.

A l'intérieur même de ce chapitre nous avons utilisé cette théorie du point fixe comme support de techniques de preuves. On pourra lire à ce propos l'article de PARK [PAR, 70] concernant la règle d'induction portant son nom et celui de BURSTALL [BUR, 69].

Terminons en présentant brièvement les relations entre ce chapitre et les suivants : le chapitre 3 étudie diverses classes d'algorithmes en fonction de structures de contrôles permises. De même que ce chapitre 2 s'est particulièrement intéressé aux énoncés récursifs, le paragraphe 4 du chapitre 3 étudie spécialement les schémas de programmes récursifs. Il se trouve que le passage de l'énoncé au programme est dans ce cas plus facile mais que la technique de calcul est plus compliquée. Dans le cas général, il est nécessaire de montrer qu'un algorithme calcule effectivement les accès spécifiés par un problème ; en d'autres termes, qu'un algorithme est correct par rapport à un énoncé. C'est l'objet du chapitre 4 de développer des méthodes de preuves de correction.

6 SOLUTION DES EXERCICES

Exercice 1

a) L'énoncé du problème peut être :

Etant donné un tableau t[1 : n] trouver un tableau t'[1 : n] équivalent à t[1 : n] et qui soit trié.

Si elles ne font pas partie du contexte il faut expliciter les deux propriétés « équivalent » et « trié » :

— t[1 : n] équivalent à t'[1 : n] est une abréviation de :

$$(\exists\sigma)\,(\sigma : [1:n] \rightarrow [1:n] \ \underline{\text{et}}\ \sigma \text{ bijective } \underline{\text{et}}\ (\forall i \in [1:n])\,(t(i) = t'(\sigma(i))))\,;$$

— t[1 : n] trié est une abréviation de :

$$(\forall i \in [1 : n - 1])(t[i] \leqslant t[i + 1]).$$

b) Enoncé : Etant donné deux tableaux triés t[1 : n] et t′[1 : p] trouver un tableau t″[1 : q] qui soit trié et équivalent à la concaténation de t[1 : n] et t′[1 : p].

La seule nouvelle propriété à définir est « concaténation ».

— t″[1 : k] est la concaténation de t[1 : n] et t′[1 : p] est une abréviation de :

$$k = n + p \ \underline{\text{et}}\ (\forall i \in [1 : n])(t''[i] = t[i])\ \underline{\text{et}}\ (\forall i \in [n + 1 : k])(t''(i) = t'(i - n))).$$

Exercice 2

Remarquons tout d'abord que $q(m, 1) = 1$ pour tout $m \geqslant 0$: C'est vrai pour $m = 0$.

Si $m > 0$, $q(m, 1) = q(m, 0) + q(m - 1, 1)$
$= 0 + 1$ par définition de q et hypothèse de récurrence.

$$\begin{aligned}
\text{Alors } q(5,3) &= q(5,2) + q(2,3) && \text{car } 5 \geqslant 3\\
&= (q(5,1) + q(3,2)) + q(2,3) && \text{car } 5 \geqslant 2\\
&= 1 + (q(3,1) + q(1,2)) + q(2,3) && \text{car } 3 \geqslant 2\\
&= 1 + 1 + q(1,1) + q(2,2) && \text{puisque } 1 < 2 \text{ et } 2 < 3\\
&= 2 + 1 + q(2,1) + q(0,2) && \text{puisque } 2 \geqslant 2\\
&= 3 + 1 + 1 = 5.
\end{aligned}$$

Plus précisément, les cinq partitions de 5 en au plus 3 sommants sont :

$$3 + 1 + 1 = 2 + 2 + 1 = 3 + 2 = 4 + 1 = 5.$$

Exercice 3

a) Montrons par récurrence sur $x - y$, avec $x \geqslant y$, que

$$g(x, y) * \text{fact}(y) = \text{fact}(x).$$

Si $x = y$, $g(y, y) * \text{fact}(y) = \text{fact}(y)$ par définition de g.

Si $x > y$ supposons que $g(x', y') * \text{fact}(y') = \text{fact}(x')$ pour tout (x', y') tel que $x' - y' < x - y$. Alors :

$$\begin{aligned}
g(x, y) * \text{fact}(y) &= g(x, y + 1) * (y + 1) * \text{fact}(y) && \text{par définition de g}\\
&= g(x, y + 1) * \text{fact}(y + 1) && \text{par définition de fact}\\
&= \text{fact}(x) && \text{puisque } x - (y + 1) < x - y
\end{aligned}$$

on en déduit que $g(x, 0) = \text{fact}(x)$ pour tout entier x.

b) Remarquons que si $x < y$ la définition récursive ne permet pas de calculer $g(x, y)$. En effet :

$$g(x, y) = (y + 1) * g(x, y + 1) = (y + 1) * (y + 2) * g(x, y + 2) = \cdots.$$

On pourrait, pour $y > x + 1$, « retourner » la définition et écrire :

$$g(x, y) = \frac{1}{y} g(x, y - 1) = \frac{1}{y * (y - 1)} g(x, y - 2) = \cdots =$$

$$= \frac{1}{y * (y - 1) * \cdots * (x + 2)} g(x, x + 1)$$

mais alors $g(x, x + 1)$ et laissé complètement indéterminé.

La seule solution, dans l'ensemble des applications de $\mathbb{N}^2$ dans $\mathbb{N}$ est

$$g(x, y) = \underline{\text{si}}\ x < y\ \underline{\text{alors}}\ 0\ \underline{\text{sinon}}\ \text{fact}(x)/\text{fact}(y)\ \underline{\text{fsi}}$$

mais une telle solution n'est pas calculable par substitutions.

Exercice 4

Si $n = 0$, il n'y a rien à déplacer : hanoï $(0, i, j) = \Lambda$ (liste vide).
Si $n > 0$, déplacer n disques de i vers j consiste simplement à :
— déplacer $n - 1$ disques de i vers le troisième pieu (noté troisième(i, j)) « libérant » ainsi le disque n,
— déplacer le disque n de i vers j,
— déplacer les $n - 1$ disques de troisième(i, j) vers j.
Ainsi donc :

$$\begin{aligned}\text{hanoï}(n, i, j) = {} & \underline{\text{si}}\ n = 0\ \underline{\text{alors}}\ \Lambda \\ & \underline{\text{sinon}}\ \text{hanoï}(n - 1, i, \text{troisième}(i, j)).(n, i, j). \\ & \text{hanoï}(n - 1, \text{troisième}(i, j), j)\ \underline{\text{fsi}}\end{aligned}$$

notons que $\text{troisième}(i, j) = 6 - i - j$ si les pieux sont numérotés 1, 2 et 3.

Exercice 5

— g est solution :

$$\text{si } x = y,\ g(x, y) = x + 1 = y + 1$$
$$\text{si } x \neq y,\ g(x, g(x - 1, y + 1)) = g(x, x - 1 + 1) = x + 1 = g(x, y)$$

— h est solution :

$$\begin{aligned}&\text{si } x = y,\ h(x, y) = x + 1 = y + 1 \\ &\text{si } x \leqslant y + 1,\ h(x, h(x - 1, y + 1)) = h(x, y + 1 - 1) \quad \text{car} \quad x - 1 < y + 1 \\ &\qquad = h(x, y) \\ &\text{si } x > y + 1,\ h(x, h(x - 1, y + 1)) = h(x, x - 1 + 1) \quad \text{puisque } x - 1 \geqslant y + 1 \\ &\qquad = x + 1 \quad \text{puisque } x \geqslant x \\ &\qquad = h(x, y) \quad \text{puisque } x \geqslant y.\ \text{C.Q.F.D.}\end{aligned}$$

Exercice 6

a) Si $x = y$, $f(x, y) = x + 1 = y + 1$.
Si $x > y$ <u>et</u> $x - y$ pair, $x - 1 \geqslant y + 1$ <u>et</u> $x - y - 2$ pair donc

$$f(x, f(x - 1, y + 1)) = f(x, x - 1 + 1) = x + 1 = f(x, y)$$

Si $x<y$ ou $x-y$ impair, $x-1<y+1$ ou $x-1-(y+1)$ impair donc

$$f(x-1, y+1) \quad \text{est indéfini ainsi que} \quad f(x, f(x-1, y+1)).$$

b) Si $x \geqslant y$ et $x-y$ pair, $f(x, y)=x+1=g(x, y)=h(x, y)$ dans les autres cas f n'est pas définie, donc

$$f \sqsubseteq g \quad \text{et} \quad f \sqsubseteq h.$$

c) cf. paragraphe 2.3.

Exercice 7

• Soit X une partie totalement ordonnée de $\mathscr{F}(E, F)$ et posons, pour tout x de E, $f(x) = y$ si et seulement s'il existe g dans X tel que $g(x) = y$. On vérifie aisément que f est bien définie et que f est la borne supérieure de X.

• Soit X une partie non vide de $\mathscr{F}(E, F)$. Posons pour tout x de E, $f(x) = y$ si et seulement si $g(x) = y$ pour tout g de X. On vérifie que g est borne inférieure de X.

Exercice 8

— Soit f continue. Si $x \leqslant y$ considérons la chaîne $x_0 = x$ et $x_i = y$ pour tout $i > 0$. Alors $\sup\limits_i x_i = y$ et par hypothèse sur f :

$$f(x_0) \leqslant \sup_i f(x_i) = f(y) \quad \text{donc f est croissante}.$$

— Si f et g sont continues,

$$f\left(g\left(\sup_i x_i\right)\right) = f\left(\sup_i g(x_i)\right) = \sup_i f(g(x_i))$$

ce qui prouve la continuité de $f \circ g$.

Exercice 9

Si $x \in X$ alors $x \leqslant \sup X$ et si f est croissante alors $f(x) \sqsubseteq f(\sup X)$. Par conséquent $f(\sup X)$ est un majorant de $f(X)$, donc

$$\sup f(X) \sqsubseteq f(\sup X).$$

Exercice 10

a) Dans l'ensemble ordonné (figure 3) la partie { a, b, c } n'a pas de maxi-

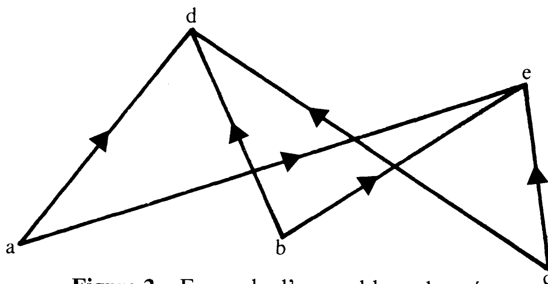


Figure 3 Exemple d'ensemble ordonné.

mum, a deux majorants d et e, mais n'a pas de borne supérieure la partie $\{ d, e \}$ n'a pas de majorant.

b) Dans l'ensemble des parties finies de $\mathbb{N}$, ordonné par inclusion, la partie constituée de tous les singletons n'a pas de majorant, donc pas de borne supérieure et pas de maximum.

Exercice 11

a est majorant de A si $x \in A \Rightarrow x \leqslant a$; si l'on remplace A par $\varnothing$, la proposition $x \in \varnothing$ étant fausse pour tout x l'implication est toujours vraie, par conséquent tout élément de E est majorant de $\varnothing$; donc la borne supérieure de $\varnothing$ ou minimum des majorants de $\varnothing$ est le plus petit élément de E.

Exercice 12

Seul i) $\Rightarrow$ ii) présente de l'intérêt.

Si toute paire admet une borne supérieure toute partie finie admet une borne supérieure.

Soit $A = \{ a_0, a_1, \ldots, a_n, \ldots \}$ une partie dénombrable soit $x_0 = a_0$

$$x_1 = \sup \{ a_0, a_1 \}$$
$$x_n = \sup \{ a_0, \ldots, a_n \} .$$

Les x_n forment une chaîne, cette chaîne admet une borne supérieure qui est aussi la borne supérieure de A, en effet x est majorant de A et si a est un autre majorant de A, pour chaque n on a $x_n \leqslant a$ donc $x \leqslant a$.

Exercice 13

3) $f : [0, 1] \rightarrow [0, 1]$ définie par :

$$f(x) = \underline{\text{si}}\ x < \frac{1}{2}\ \underline{\text{alors}}\ 0\ \underline{\text{sinon}}\ 1\ \underline{\text{fsi}}$$

f est croissante mais non continue puisque

$$\sup_{n \in \mathbb{N}} f\left(\frac{1}{2} - \frac{1}{n}\right) = 0 < f\left(\sup_{n \in \mathbb{N}} \left(\frac{1}{2} - \frac{1}{n}\right)\right) = 1 .$$

En revanche $g : [0,1] \rightarrow [0, 1]$ définie par :

$$g(x) = \underline{\text{si}}\ x \leqslant \frac{1}{2}\ \underline{\text{alors}}\ 0\ \underline{\text{sinon}}\ 1\ \underline{\text{fsi}}$$

est continue au sens des ensembles inductifs bien qu'elle ne le soit pas au sens de la continuité habituelle.

On se persuadera facilement que les fonctions croissantes $f : [0, 1] \rightarrow [0, 1]$ qui sont continues pour la structure d'ordre habituelle de $\mathbb{R}$ sont exactement les fonctions croissantes continues à gauche au sens de l'analyse.

4) a) Montrons la contraposée de la proposition :

$$(\exists k \in \mathbb{N}) \left(x_k = \sup_n x_n\right) \Rightarrow (\exists k \in \mathbb{N})\ (\forall i \geqslant k) \left(x_k \leqslant x_i \leqslant \sup_n x_n = x_k\right)$$
$$\Rightarrow (\exists k \in \mathbb{N})\ (\forall i \geqslant k)\ (x_i = x_k) .$$

b) Soit (x_n) une chaîne non stationnaire de E et $a = \sup_n x_n$. Considérons dans F deux éléments m et M tels que $m < M$ et définissons f par :

$$f(x) = \underline{\text{si}}\ x \geqslant a\ \underline{\text{alors}}\ M\ \underline{\text{sinon}}\ m\ \underline{\text{fsi}}\ .$$

Il est immédiat que f est croissante. D'autre part $f(x_n) = m$ et donc

$$\left(\sup_n f(x_n)\right) = m < f(\sup x_n) = M\ ,$$

c'est-à-dire que f n'est pas continue.

Exercice 14

1) *a*) $\varnothing$ est ouvert en effet les propositions

$$\text{« } x \in \varnothing\ \underline{\text{et}}\ x \leqslant y \text{ »} \quad \text{et} \quad \text{« } \sup_n x_n \in \varnothing \text{ »}$$

sont fausses donc les implications correspondantes sont vraies ; E est un ouvert de façon triviale.

b) Si $(\mathcal{O}_i)_{i \in I}$ est une famille d'ouverts on a :

$$\left(x \in \bigcup_{i \in I} \mathcal{O}_i\ \underline{\text{et}}\ x \leqslant y\right) \Rightarrow ((\exists i \in I)\ x \in \mathcal{O}_i\ \underline{\text{et}}\ x \leqslant y) \Rightarrow$$

$$((\exists i \in I)\ y \in \mathcal{O}_i) \Rightarrow y \in \bigcup_{i \in I} \mathcal{O}_i$$

(x_n) est une chaîne et $\sup_n x_n \in \bigcup_{i \in I} \mathcal{O}_i$ implique qu'il existe $i \in I$ tel que $\sup_n x_n \in \mathcal{O}_i$ et donc $\{x_n \mid n \in \mathbb{N}\} \cap \mathcal{O}_i \neq \varnothing$.

Finalement on a $\{x_n \mid n \in \mathbb{N}\} \cap \bigcup_{i \in I} \mathcal{O}_i \neq \varnothing$.

c) si $\mathcal{O}_i$ est une famille finie d'ouverts

$$x \in \bigcap_{i \in I} \mathcal{O}_i \quad \text{et} \quad x \leqslant y \Rightarrow (\forall i \in I)\ x \in \mathcal{O}_i\ \underline{\text{et}}\ x \leqslant y$$

$$\Rightarrow (\forall i \in I)\ y \in \mathcal{O}_i \Rightarrow y \in \bigcap_{i \in I} \mathcal{O}_i\ .$$

Si $\{x_n \mid n \in \mathbb{N}\}$ est une chaîne et si $\sup_n x_n \in \mathcal{O}_i$ alors pour chaque i il existe n_i tel que $k \geqslant n_i$ implique $x_k \in \mathcal{O}_i$, si l'on prend $n = \max(n_i)$ $k \geqslant n$ implique $x_k \in \bigcap_{i \in I} \mathcal{O}_i$.

2) Remarquons que : $x \leqslant y$ si et seulement si y appartient à chaque ouvert contenant x.

L'une des implications est évidente ; pour l'autre si $x \nleqslant y$ alors

$$U = \{z \in E \mid z \nleqslant y\}$$

est un ouvert tel que $x \in U$ et $y \notin U$.

— Si f est topologiquement continue, f est croissante.

En effet, soit $x \leqslant y$ et U un ouvert quelconque contenant f(x) alors $f^{-1}(U)$ contient x, est ouvert, donc contient y, ainsi $f(y) \in U$ et $f(x) \leqslant f(y)$.

Soit f une fonction topologiquement continue, $\{ x_n \mid n \in \mathbb{N} \}$ une chaîne et U un ouvert quelconque contenant $f\left(\sup_n x_n\right)$, $f^{-1}(U)$ est un ouvert qui contient $\sup_n x_n$ donc l'un des x_n de la chaîne $f(x_n)$ appartient à U et $\sup_n f(x_n)$ aussi, ainsi $\sup_n f(x_n)$ appartient à tout ouvert contenant

$$f\left(\sup_n x_n\right) \quad \text{donc} \quad f\left(\sup_n x_n\right) \leqslant \sup_n f(x_n) .$$

Comme f est croissante on a (exercice 9)

$$\sup_n f(x_n) \sqsubseteq f\left(\sup_n x_n\right)$$

d'où le résultat :

— Si f est continue au sens des chaînes et si U est un ouvert de F : $x \in f^{-1}(U)$ et $x \leqslant y$ implique $y \in f^{-1}(U)$ car f est croissante. Si $\{ x_n \mid n \in \mathbb{N} \}$ est une chaîne telle que $\sup_n (x_n \in f^{-1}(U))$ c'est-à-dire

$$f\left(\sup_n x_n\right) = \sup_n f(x_n) \in U ,$$

alors il existe x_m tel que $f(x_m) \in U$ c'est-à-dire $x_m \in f^{-1}(U)$.

Exercice 15

Soit $X = (X_i)_{i \in I}$ une famille non vide de parties de A*. X admet pour borne supérieure $Y = \bigcup_{i \in I} X_i$. En effet pour tout $i \in I$, $X_i \subset Y$ et si X_i est inclus dans Z pour tout $i \in I$ il en est de même de Y. La partie vide $\varnothing$ étant un élément minimum de $\mathscr{L}(A)$ on en déduit que cet ensemble est inductif.

Soit $U(X_1, \ldots, X_n)$ une expression régulière sur A. Vérifions par récurrence sur la longueur de U que $\lambda X_1, \ldots, X_n . U(X_1, \ldots, X_n)$ est continue.

i) Si $|U| = 1$, $U = a \in A$ ou $U = X_i$ ($|U|$ désigne la longueur de U)

ii) Si $|U| > 1$ on peut distinguer trois cas

- $U = U_1 \cup U_2$
- $U = U_1 . U_2$
- $U = (U_1)^*$.

Il suffit donc de vérifier que les applications

$\lambda X_1, X_2 . X_1 \cup X_2$, $\lambda X_1, X_2 . (X_1 . X_2)$ et $\lambda X . X^*$ sont continues .

Démontrons cette propriété pour la première application ; compte tenu de la caractérisation des bornes supérieures dans $\mathscr{L}(A)$ mise en évidence ci-dessus il suffit de prouver :

$$\bigcup_{n,m} (X_n \cup Y_m) = \left(\bigcup_n X_n\right) \cup \left(\bigcup_m Y_m\right) \quad \text{ce qui est immédiat .}$$

Exercice 16

1) b) — Soit f une application continue au sens de l'analyse et soit $(I_n)_{n \geqslant 0}$ une chaîne de $\mathrm{Int}(I_0)$. Par définition $I_n \supset I_{n+1}$ pour tout $n \geqslant 0$, donc $\bigcap_{n \geqslant 0} I_n$ contient nécessairement un point a et l'on peut ainsi se ramener au cas où les intervalles sont de la forme $[a, b_n]$ ou $[a, +\infty[$ avec a fixé.

Si $I_n = [a, +\infty[$ pour tout $n \geqslant 0$,

$$\bigcap_{n \geqslant 0} I_n = [a, +\infty[.$$

Si par contre il existe n_0 tel que $I_n = [a, b_n]$ pour $n \geqslant n_0$, dans ce cas $\tilde{f}(I_n) = f(I_n)$. De plus la suite $(b_n)_{n \geqslant 0}$ est décroissante, minorée donc converge vers un nombre b. Alors :

— $f\left(\bigcap_{n \geqslant 0} I_n\right) \subset \bigcap_{n \geqslant 0} f(I_n)$ de manière évidente .

— Soit $f([a, b]) = [c, d]$; pour tout $\varepsilon > 0$ il existe $\eta > 0$ tel que

$$f([a, b + \eta]) \subset [c - \varepsilon, d + \varepsilon] .$$

En utilisant la convergence de (b_n) il existe d'autre part $n_1 \in \mathbb{N}$ tel que

$n \geqslant n_1 \Rightarrow [a, b_n] \subset [a, b + \eta]$

et donc $n \geqslant n_1 \Rightarrow f([a, b_n]) \subset [c - \varepsilon, d + \varepsilon]$

ainsi $\bigcap_{n \geqslant 0} f(I_n) \subset [c - \varepsilon, d + \varepsilon]$ pour tout $\varepsilon > 0$

donc $\bigcap_{n \geqslant 0} f(I_n) \subset [c, d] = f\left(\bigcap_{n \geqslant 0} I_n\right)$.

— Soit maintenant $\tilde{f}$ une application continue au sens des chaînes et x un point de I_0. Considérons la chaîne $I_n = \left[x - \frac{1}{n}, x + \frac{1}{n}\right]$;

$$f(x) = \tilde{f}(x) = \tilde{f}\left(\bigcap_{n \geqslant 0} I_n\right) = \bigcap_{n \geqslant 0} \tilde{f}(I_n) = \bigcap_{n \geqslant 0} f(I_n) .$$

Ainsi, pour tout $\varepsilon > 0$, il existe n_0 tel que

$$n \geqslant n_0 \Rightarrow f\left(\left[x - \frac{1}{n}, x + \frac{1}{n}\right]\right) \subset [f(x) - \varepsilon, f(x) + \varepsilon]$$

ce qui prouve la continuité de f.

2) a) Soit $x_0 \in I$, $A = |f(x_0) - x_0|/(1 - k)$

et $I_0 = I \cap [x_0 - A, x_0 + A]$

pour $x \in I_0$, $|f(x) - f(x_0)| \leqslant k\,|x - x_0|$

or $|f(x) - x_0| \leqslant |f(x) - f(x_0)| + |f(x_0) - x_0|$

donc $|f(x) - x_0| \leqslant |f(x_0) - x_0| + k\,|x - x_0|$

soit encore $|f(x) - x_0| \leqslant A[(1 - k) + k] = A$

ainsi $f(x) \in I_0$.

b) Puisque $\mathrm{Int}(I_0)$ est inductif et que $\tilde{f}$ est continue, $\tilde{f}$ admet un plus petit point fixe. Il existe donc [a, b] tel que

$$f([a, b]) = \tilde{f}([a, b]) = [a, b] .$$

Supposons $a \neq b$ et soit $c, d \in [a, b]$ tels que $f(c) = a, f(d) = b$ alors $|a - b| = |f(c) - f(d)| < k|c - d| \leqslant k|a - b|$ ce qui est absurde si $k < 1$.

$\{a\}$ est l'unique point fixe de $\tilde{f}$: $\{a\}$ est maximal dans $\mathrm{Int}(I_0)$; donc a est aussi l'unique point fixe de f.

Exercice 17

a) Il faut montrer que toute partie Y de X admet une borne supérieure. Appelons Y′ l'ensemble des majorants de Y. Par hypothèse Y′ a une borne inférieure qui est, par définition, la borne supérieure de Y.

b) Soit $f : X \to X$ croissante,

$$Y = \{x \mid f(x) \leqslant x\} \text{ et } y = \inf(Y) .$$

Alors, pour tout x de Y on a $y \leqslant x$ et donc $f(y) \leqslant f(x) \leqslant x$ puisque f est croissante. Ainsi f(y) est un minorant de Y, donc $f(y) \leqslant y$. Mais on tire de cette inégalité : $f(f(y)) \leqslant f(y)$ qui équivaut à : $f(y) \in Y$. Comme par hypothèse y est la borne inférieure de Y : $y \leqslant f(y)$. Finalement on obtient $y = f(y)$ qui est un point fixe de f.

c) D'après l'exercice 13 il est nécessaire de choisir des exemples de treillis complets infinis. Prenons le plus simple d'entre eux $\mathbb{N} \cup \{\infty\}$ avec la relation : $(\forall n \in \mathbb{N})\,(n < \infty)$. Alors la fonction $f : \mathbb{N} \cup \{\infty\} \to \mathbb{N} \cup \{\infty\}$ définie par $f(x) = \underline{\text{si}}\ x = \infty\ \underline{\text{alors}}\ \infty\ \underline{\text{sinon}}\ 0\ \underline{\text{fsi}}$ est trivialement croissante et non continue.

Exercice 18

La solution de cet exercice nécessite quelques connaissances de théorie des ensembles que l'on peut trouver dans [KRI, 67].

Montrons par récurrence sur α que $x_\alpha \leqslant x_{\alpha+1}$.

— Pour $\alpha = 0$ c'est immédiat.

— Si $\alpha = \alpha' + 1$, $x_\alpha = f(x_{\alpha'}) \leqslant f(x_{\alpha'+1}) = x_{\alpha+1}$ puisque f est croissante.

— Si α est un ordinal limite,

$$x_\alpha = \sup_{\gamma<\alpha} x_\gamma \leqslant \sup_{\gamma<\alpha} x_{\gamma+1} = \sup_{\gamma<\alpha} f(x_\gamma)$$

$$\leqslant f(x_\alpha) = x_{\alpha+1} .$$

Plus généralement on prouve que $\alpha \leqslant \beta \Rightarrow x_\alpha \leqslant x_\beta$.

Considérons la relation $R(y, \alpha) \Leftrightarrow y = x_\alpha$. Si pour tous ordinaux α, β on a $\alpha < \beta \Rightarrow x_\alpha < x_\beta$ cette relation est fonctionnelle. Mais d'après le

schéma de remplacement (ou de substitution) on en déduit alors que la collection des ordinaux est un ensemble ce qui est faux. Ainsi il existe deux ordinaux α et β tels que $x_\alpha = x_\beta$ et $\alpha < \beta$. Comme $\alpha < \alpha + 1 \leqslant \beta$ on en déduit $x_\alpha = x_{\alpha+1} = x_\beta$, c'est-à-dire $f(x_\alpha) = x_\alpha$. x_α est donc un point fixe de f. Montrons que c'est le plus petit en prouvant que

$$(\forall x)\,(f(x) = x \Rightarrow (\forall \alpha)\,(x_\alpha \leqslant x)))$$

par récurrence sur α :

— Si α est non limite, $\alpha = \alpha' + 1$ et

$$x_{\alpha'} \leqslant x \Rightarrow f(x_{\alpha'}) = x_\alpha \leqslant f(x) = x\,.$$

— Si α est limite, $\alpha = \sup\limits_{\gamma<\alpha} \gamma$ et par hypothèse de récurrence on a $x_\gamma \leqslant x$ pour tout $\gamma < \alpha$. Donc $x_\alpha = \sup\limits_{\gamma<\alpha} x_\gamma \leqslant x$. C.Q.F.D.

Exercice 19

a) Soit $Y = \{\, x \mid g(x) = x \,\}$. Y est non vide et la restriction à Y de la relation $\leqslant$ donne à Y une structure d'espace inductif (le lecteur est invité à détailler cette démonstration).

Montrons que $x \in Y \Rightarrow f(x) \in Y$.

En effet

$$x \in Y \Leftrightarrow g(x) = x \Rightarrow f \circ g(x) = f(x) \Rightarrow g \circ f(x) = f(x) \Leftrightarrow f(x) \in Y\,.$$

Donc f/Y (f resteint à Y) admet un plus petit point fixe noté $\mu(f, g)$ qui de façon évidente est le plus petit point fixe commun à f et g.

Expression de ce plus petit point fixe :

Le plus petit élément de Y est $\sup\limits_{n\in\mathbb{N}} g^n(\perp)$ d'après le théorème 1.

Et donc, toujours d'après ce théorème :

$$\mu(f, g) = \sup_{n\in\mathbb{N}} f^n\left(\sup_{n\in\mathbb{N}} g^n(\perp)\right)\,.$$

Comme f et g sont continues $\mu(f, g)$ peut encore s'écrire :

$$\mu(f, g) = \sup_{n\in\mathbb{N}}\ \sup_{m\in\mathbb{N}} f^n \circ g^m(\perp)\,.$$

b) Pas nécessairement ; considérons l'ensemble X schématisé par la figure 4

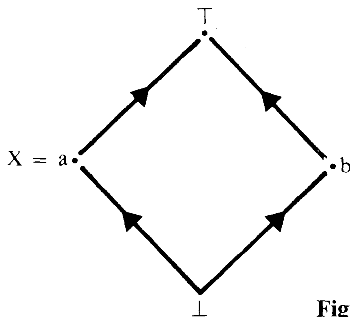


Figure 4.

et les fonctions f; g définies par :

$$f(\bot) = a\,;\quad f(a) = a\,;\quad f(b) = T\,;\quad f(T) = T$$
$$g(\bot) = b\,;\quad g(b) = b\,;\quad g(a) = T\,;\quad g(T) = T\,.$$

De façon évidente f et g sont croissantes et on a

$$f \circ g(\bot) = g \circ f(\bot) = T$$

d'où $f \circ g = g \circ f$ et cette fonction est la fonction constante T. Dans ce cas

$$\mu(f) = a\,,\quad \mu(g) = b \quad \text{et} \quad \mu(f, g) = T\,.$$

c) Oui aux deux questions. En effet $f^n \circ g^m(\bot) \leqslant f^k \circ g^k(\bot)$ avec $k = \sup(n, m)$ et comme f et g commutent $f^k \circ g^k = (f \circ g)^k$.

Ainsi

$$\mu(f \circ g) = \sup_{k \in \mathbb{N}} (f \circ g)^k(\bot) = \sup_{k \in \mathbb{N}} f^k \circ g^k(\bot) = \sup_{n \in \mathbb{N}} \sup_{m \in \mathbb{N}} f^n \circ g^m(\bot) = \mu(f, g)\,.$$

Exercice 20

Pour tout $n \geqslant 1, f^n(\bot) = g^n(\bot)$:

— Pour $n = 1$ c'est vrai.

— Supposons $f^{n-1}(\bot) = g^{n-1}(\bot)$ alors

$$\begin{aligned} f^n(\bot) &= f(f^{n-1}(\bot)) \\ &= f(g^{n-1}(\bot)) \\ &= g^{n-1}(f(\bot)) \\ &= g^{n-1}(g(\bot)) \\ &= g^n(\bot) \end{aligned}$$

ainsi $\mu(f) = \sup_{n \geqslant 0} f^n(\bot) = \sup_{n \geqslant 0} g^n(\bot) = \mu(g)$.

Exercice 21

On introduit la propriété plus forte :

$$\Phi(f, g) : \forall x, y \quad f(x, y) = g(x, y) \ \underline{\text{et}}\ h(g(x, y)) = g(x, h(y))\,.$$

— $\Phi(\Omega, \Omega)$ est immédiat.

— Prouvons maintenant que $\Phi(f, g)$ implique $\Phi(\tau_1(f), \tau_2(g))$.

Supposons donc

(1) $\forall x, y \quad f(x, y) = g(x, y)$

(2) $\forall x, y \quad h(g(x, y)) = g(x, h(y))$

alors :

$$\begin{aligned} \tau_1(f)\,(x, y) &= \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ y\ \underline{\text{sinon}}\ h(f(k(x), y))\ \underline{\text{fsi}} \\ &= \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ y\ \underline{\text{sinon}}\ h(g(k(x), y))\ \underline{\text{fsi}} \quad \text{d'après (1)} \\ &= \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ y\ \underline{\text{sinon}}\ g(k(x), h(y)\ \underline{\text{fsi}} \quad \text{d'après (2)} \\ &= \tau_2(g)\,(x, y) \end{aligned}$$

et d'autre part :

$$\begin{aligned} h(\tau_2(g(x, y))) &= h(\underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ y\ \underline{\text{sinon}}\ g(k(x), h(y))\ \underline{\text{fsi}}) \\ &= \underline{\text{si}}\ p(x)\ \underline{\text{alors}}\ h(y)\ \underline{\text{sinon}}\ g(k(x), h^2(y))\ \underline{\text{fsi}} \text{ d'après (2)} \\ &= \tau_2(g)(x, h(y)) \end{aligned}$$

ce qui termine la preuve de Φ. En utilisant la règle de Scott on en tire $\Phi(\mu(\tau_1), \mu(\tau_2))$ dont on déduit $\mu(\tau_1) = \mu(\tau_2)$. C.Q.F.D.

Remarquons que la propriété $h(g(x, y)) = g(x, h(y))$ est indépendante de τ_1.

Exercice 22

i) Par troncation :

Montrons par récurrence que $\tau_2^n(\Omega) = \tau_1^{1+i+i^2+\ldots+i^{n-1}}(\Omega)$.

— Pour $n = 1$ on obtient $\tau_2(\Omega) = \tau_1(\Omega)$ ce qui est l'hypothèse.

— Si l'égalité est vérifiée pour $n \geqslant 1$ quelconque :

$$\tau_2^{n+1}(\Omega) = \tau_2\, \tau_1^{1+i+\ldots+i^{n-1}}(\Omega) \quad \text{par hypothèse de récurrence}$$

$$\begin{aligned} \tau_2^{n+1}(\Omega) &= \tau_1^{i+i^2+\ldots+i^n}\, \tau_2(\Omega) \quad \text{car} \quad \tau_2\, \tau_1^k = \tau_1^{ik}\, \tau_2 \text{ pour tout } k \geqslant 1 \\ &= \tau_1^{1+i+i^2+\ldots+i^n}(\Omega) \quad \text{car} \quad \tau_1(\Omega) = \tau_2(\Omega)\,. \end{aligned}$$

On déduit de cette propriété :

$$\bigcup_{n \geqslant 0} \tau_2^n(\Omega) = \bigcup_{n \geqslant 0} \tau_1^n(\Omega) \quad \text{et donc} \quad \mu(\tau_1) = \mu(\tau_2)\,.$$

ii) Par induction de Scott :

— $\mu(\tau_2) \sqsubseteq \mu(\tau_1)$. Pour prouver cette inégalité il suffit de montrer par induction de Park que $\tau_2(\mu(\tau_1)) \sqsubseteq \mu(\tau_1)$.

Pour cela considérons l'assertion $\tau_2(f) \sqsubseteq \mu(\tau_1)$:

- $\tau_2(\Omega) = \tau_1(\Omega) \sqsubseteq \mu(\tau_1)$.
- Si $\tau_2(f) \sqsubseteq \mu(\tau_1)$ alors $\tau_2(\tau_1(f)) = \tau_1^i(\tau_2(f))$
  donc $\tau_2(\tau_1(f)) \sqsubseteq \tau_1^i(\mu(\tau_1))$
  ainsi $\tau_2(\tau_1(f)) \sqsubseteq \mu(\tau_1)$.

On en déduit bien que $\tau_2(\mu(\tau_1)) \sqsubseteq \mu(\tau_1)$. C.Q.F.D.

— $\mu(\tau_1) \sqsubseteq \mu(\tau_2)$. Pour prouver cette deuxième inégalité introduisons l'assertion $\Phi(f)$:

(1) $f \sqsubseteq \tau_1^{i-1}(f)$

(2) $\underline{\text{et}}\ \tau_1(f) \sqsubseteq \tau_2(f)$

(3) $\underline{\text{et}}\ f \sqsubseteq \mu(\tau_2)$

il est bien clair que $\Phi(\mu(\tau_1))$ implique l'inégalité cherchée. Pour prouver cette assertion utilisons l'induction de Scott :

- $\Phi(\Omega)$ est évident.
- Supposons $\Phi(f)$ alors :

$$f \sqsubseteq \tau_1^{i-1}(f) \Rightarrow \tau_1(f) \sqsubseteq \tau_1^{i-1}(\tau_1(f)) \quad \text{car } \tau_1 \text{ est croissante}$$

$$
\begin{array}{llll}
\text{d'autre part} & \tau_1^2(f) & \sqsubseteq \tau_1^{i+1}(f) & \text{d'après (1) et la croissance de } \tau_1 \\
 & & \sqsubseteq \tau_1^i\, \tau_2(f) & \text{d'après (2)} \\
 & & \sqsubseteq \tau_2\, \tau_1(f) & \text{par hypothèse} \\
\text{enfin} & \tau_1(f) & \sqsubseteq \tau_2(f) & \text{d'après (2)} \\
 & & \sqsubseteq \tau_2(\mu(\tau_2)) & \text{d'après (3)} \\
 & & \sqsubseteq \mu(\tau_2). & \text{C.Q.F.D.}
\end{array}
$$

CHAPITRE 3

Schémas de programme

Le chapitre précédent a traité des méthodes de définition des problèmes. La résolution des problèmes, ainsi définie, conduit à des programmes qui vont exécuter un calcul et parfois donner un résultat. Ce sont ces notions qui vont être développées maintenant.

Dans le chapitre précédent, pour poser les problèmes, nous nous sommes intéressés à la signification des fonctions ; or lorsqu'on étudie un programme ou une classe de programmes, un certain nombre de propriétés ne sont pas liées à cette signification, mais seulement à l'ordre des instructions ; les objectifs à atteindre sont des théorèmes généraux indépendants de la structure particulière du domaine sur lequel on travaille (réels, listes, chaînes de caractères, ...). Nous allons donc nous intéresser à l'ordre dans lequel s'effectuent les opérations, autrement dit, à la partie contrôle du programme. Nous obtiendrons des résultats sur la terminaison, l'isologie (propriété de deux programmes qui calculent la même chose) et les transformations de programmes. Beaucoup d'auteurs considèrent cette approche comme syntaxique, celle des chapitres 2 et 4 étant considérée comme sémantique.

Signalons que beaucoup de résultats sont des résultats d'indécidabilité (§ 2.7) du genre : « Il n'y a pas d'algorithme pour prouver qu'une telle classe de programmes se termine ». Un autre ensemble de propriétés concerne la puissance ou l'efficacité d'une famille de schémas par rapport à une autre avec des résultats disant : « Telle famille de schémas est strictement moins puissante que telle autre ».

1 SCHÉMAS FONCTIONNELS

1.1 Présentation

La technique d'écriture, la plus classique, d'une fonction composée consiste, comme nous l'avons appris dans l'enseignement secondaire, à utiliser un système de parenthèses, de virgules et de symboles de fonctions et de variables.

Une deuxième technique est d'utiliser une représentation arborescente, comme sur la figure 1.

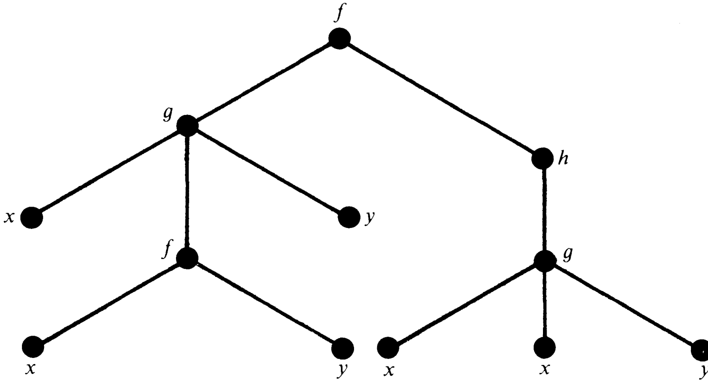


Figure 1 Redrésentation arborescente du schéma fonctionnel

$f(g(x, f(x, y), y)\, h(g(x, x, y)))$.

Une troisième technique est de supprimer les parenthèses. Connaissant l'arité **ar** de chaque fonction, c'est-à-dire le nombre de paramètres qu'elle admet, on peut reconstituer l'écriture avec parenthèses et virgules automatiquement, si ce n'est sans difficulté. Par exemple la fonction de la figure 1 s'écrit *fgxfxyyhgxxy* où l'on sait que l'arité des fonctions est :

$$\mathbf{ar}(f) = 2\,, \quad \mathbf{ar}(g) = 3\,, \quad \mathbf{ar}(h) = 1\,.$$

Exercice 1

Ecrire un algorithme qui rétablit les parenthèses et les virgules pour des schémas fonctionnels d'arité connue et sans parenthèse. □

1.2 Définition formelle de la première approche

Soient $\mathscr{F} = \bigcup_{n \in \mathbb{N}} \mathscr{F}_n$ un ensemble de symboles de fonctions et **ar** une application donnant l'arité de chaque symbole ; $\mathscr{F}_n$ est l'ensemble des symboles d'arité n. Soit $\mathscr{X}$ un ensemble de symboles de variables.

L'ensemble des schémas fonctionnels sur $\mathscr{F}$ et $\mathscr{X}$ noté $\mathrm{sch}(\mathscr{F}, \mathscr{X})$ est la plus petite solution de l'équation à point fixe, d'inconnue X :

$$X = \mathscr{X} \cup \mathscr{F}_0 \cup \bigcup_{n \geqslant 1} \mathscr{F}_n\underbrace{(X, .., X)}_{n \text{ fois}}\,.$$

Remarquons que si $\mathbf{ar}(f) = 0$, alors $f \in \mathrm{sch}(\mathscr{F}, \mathscr{X})$; ces symboles d'arité nulle représentent les constantes.

Exercice 2

1) Dans chacun des cas suivants, écrire quelques schémas de sch($\mathcal{F}$, $\mathcal{X}$).

a) $\mathcal{F} = \{ f \}$, $\mathbf{ar}(f) = 1$, $\mathcal{X} = \{ x \}$.

b) $\mathcal{F} = \{ f, g \}$, $\mathbf{ar}(f) = \mathbf{ar}(g) = 1$, $\mathcal{X} = \{ x \}$.

c) $\mathcal{F} = \{ f \}$, $\mathbf{ar}(f) = 2$, $\mathcal{X} = \{ x_i \mid i \in \mathbb{N} \}$.

d) $\mathcal{F} = \{ f, g \}$, $\mathbf{ar}(f) = 2$, $\mathbf{ar}(g) = 1$, $\mathcal{X} = \{ x, y \}$.

e) $\mathcal{F} = \{ f, g \}$, $\mathbf{ar}(f) = 2$, $\mathbf{ar}(g) = 1$, $\mathcal{X} = \varnothing$.

f) $\mathcal{F} = \{ f, g \}$, $\mathbf{ar}(f) = 2$, $\mathbf{ar}(g) = 0$, $\mathcal{X} = \varnothing$.

2) Dessiner les arborescences associées.

1.3 Commentaires

Cette technique d'écriture permet de faire la différence entre la structure interne d'une fonction (aspect syntaxique) et les valeurs particulières de cette fonction (aspect sémantique), mais il s'agit d'un processus figé, indépendant des dites valeurs des variables. Il manque dans la description un moyen de contrôle opérant des choix de calcul, aussi est-elle impropre à rendre compte des calculs en général et ne peut décrire que les problèmes les plus simples où aucune variante n'est possible à l'exécution.

Par exemple, possédant la multiplication m d'arité 2, on décrit la fonction $x \to x^2$ par le schéma : $m(x, x)$. Mais si l'on ne veut utiliser que l'addition, un schéma fonctionnel unique ne peut pas décrire l'application $x \to x^2$.

1.4 Interprétations

Soit D un ensemble non vide. Supposons que $\mathcal{X} = \{ x_1, \ldots, x_n \}$. Pour définir une interprétation, il faut d'abord se donner une interprétation des symboles fonctionnels : c'est une application I_0 qui, à f d'arité k, fait correspondre une application $I_0[f]$ de D^k vers D. L'interprétation I induite par I_0 est une application de sch($\mathcal{F}$, $\mathcal{X}$) vers l'ensemble $D^n \to D$ des applications de D^n vers D telles que :

$$I[x_i] \text{ est la i-ième projection}$$

et
$$I[f(\alpha_1, \ldots, \alpha_k)] = I_0[f]\,(I[\alpha_1], \ldots, I[\alpha_k])\,.$$

autrement dit :

$$I[x_i]\,(a_1, \ldots, a_n) = a_i$$

et
$$I[f(\alpha_1, \ldots, \alpha_k)]\,(a_1, \ldots, a_n) = I_0[f]\,(I[\alpha_1]\,(a_1, \ldots, a_n), \ldots, I[\alpha_k]\,(a_1, \ldots, a_n))\,.$$

Exercice 3

Calculer l'interprétation du schéma fonctionnel de la figure 1 en prenant

$$\begin{aligned} D &= \mathbb{N}\,; \\ I_0[f] &= \lambda x, y . x + y\,; \\ I_0[g] &= \lambda x, y, z . x + 2 - y * z\,, \\ I_0[h] &= \lambda x . 5 * x\,. \end{aligned}$$ □

2 SCHÉMAS DE PROGRAMME AVEC BRANCHEMENTS

2.1 Définition d'un schéma de programme

Un schéma de programme est un objet purement syntaxique. Si on attribue une signification aux symboles, un schéma de programmes devient un programme. Les symboles élémentaires utilisés pour décrire les schémas de programmes avec branchement sont les suivants :

1) Des **noms de variables** formant un ensemble $\mathscr{X} = \{ x_1, \ldots, x_n \}$.
2) Des **noms de fonctions** formant un ensemble $\mathscr{F} = \{ f, g, \ldots \}$.
3) Des **noms de prédicats** formant un ensemble $\mathscr{P} = \{ p, q, \ldots \}$.
4) Un symbole $:=$ représentant l'affectation.
5) *STOP* notant l'arrêt.
6) Les entiers naturels $\{ 1, 2, \ldots \}$ pour numéroter les instructions d'un schéma.
7) L'ensemble des symboles { (,), , }.

Nous ne considérons que trois types d'instructions :

— l'instruction d'arrêt : *STOP* ;
— les instructions d'affectation $x_i := x_j$, ou bien $x_i := f(y_1, \ldots, y_m)$ où $f \in \mathscr{F}$, $\mathbf{ar}(f) = m$ et les y_i sont des symboles de variables appartenant à $\mathscr{X}$ pour $i = 1, \ldots, m$;
— les instructions de test : $p(y_1, \ldots, y_m)\ h\,, k$ où $h, k \in \mathbb{N}$, $p \in \mathscr{P}$, $\mathbf{ar}(p) = m$ et les y_i sont des symboles de variables appartenant à $\mathscr{X}$ pour $i = 1, \ldots, m$.

Un **schéma de programme avec branchement** π est alors défini comme une suite finie de la forme $(1, \varepsilon_1), (2, \varepsilon_2), \ldots, (r, \varepsilon_r)$ où chaque ε_i est une instruction d'un des trois types précédents. De plus, on suppose d'une part que toutes les instructions de test de π sont de la forme $p(y_1, \ldots, y_m)\ h\,, k$ avec $1 \leqslant h \leqslant r$ et $1 \leqslant k \leqslant r$ et d'autre part que toute affectation est suivie d'une instruction.

Exemple 1

1 $x_1 := f$
2 $p(x_2)\ 6\,, 3$
3 $x_1 := g(x_2, x_1)$
4 $x_2 := h(x_2)$
5 $p(x_2)\ 6\,, 3$
6 *STOP* □

Exercice 4

Dans l'exemple précédent, définir les plus petits ensembles $\mathscr{X}$, $\mathscr{F}$ et $\mathscr{P}$ permettant de reconstruire π. □

Il n'est pas nécessaire d'introduire un branchement inconditionnel car il peut être simulé par des instructions du type :

$$\text{i } p \text{ k , k}$$

qui est un branchement de l'instruction i à l'instruction k. Dans la suite de ce chapitre on utilise indifféremment les locutions « schémas de programme », « schémas avec branchements » ou « schémas de programmes avec branchements ».

De façon évidente les schémas de programme avec branchements peuvent être représentés par des organigrammes (voir par exemple la figure 2).

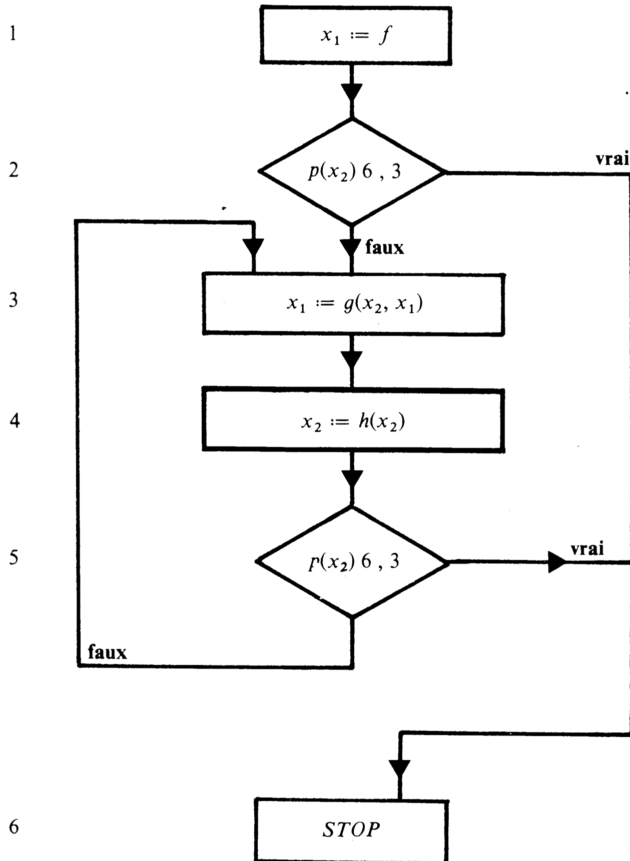


Figure 2 Un organigramme associé au schéma de l'exemple 1.

Exercice 5

Construire l'organigramme associé au schéma de programme suivant :

1 $p(x_2)$ 2 , 3
2 *STOP*
3 $x_3 := f(x_2)$
4 $x_2 := g(x_1, x_2)$
5 $x_1 := f(x_3)$
6 $p(x_2)$ 1 , 1 . □

2.2 Interprétation, initialisation

Nous avons décrit la syntaxe de notre langage et maintenant, pour transformer un schéma de programme en un programme, il faut donner un sens aux symboles de fonctions et de prédicats. Nous pourrons alors décrire le déroulement de ce programme.

Une **interprétation** I_0 des symboles d'un schéma de programmes est la donnée :

— d'un ensemble non vide, D, appelé domaine de l'interprétation ;
— pour chaque $f \in \mathscr{F}$, d'une application $I_0[f]$ de D^k vers D avec $k = \mathbf{ar}(f)$;
— pour chaque $p \in \mathscr{F}$, d'une application $I_0[p]$ de D^k vers { **vrai, faux** } avec $k = \mathbf{ar}(p)$.

I_0 se prolonge en une application I qui, à un programme, fait correspondre une fonction (partiellement définie) de D^n vers D^n appelée interprétation du programme.

On appelle **initialisation** un point $d = (d_1, \ldots, d_n)$ de D^n.

Exemple 2

Pour le schéma de programme de l'exemple 1, nous pouvons prendre comme interprétation

a) $D = \mathbb{N}$; $I_0[f] = 1$; $I_0[g](x, y) = x * y$;

$I_0[h](x) = \text{SI } x > 0 \text{ ALORS } x-1 \text{ SINON } 0$; $I_0[p](x) = (x = 0)$.

On voit facilement que le programme obtenu calcule la fonction factorielle.

b) Soit $D = V^*$ l'ensemble des mots sur V ;

$I_0[f] = \Lambda$; $I_0[g](x, y) = \mathbf{cons}(\mathbf{t\hat{e}te}(y), x)$;

$I_0[h](x) = \text{SI } x \neq \Lambda \text{ ALORS } \mathbf{queue}(x) \text{ SINON } \Lambda$;

$I_0[p](x) = (x = \Lambda)$.

cons(a, x) construit une liste par adjonction de $a \in V$ en tête de la liste $x \in V^*$; **tête** est la fonction qui donne l'élément en tête d'une liste non vide (c'est-à-dire différente de Λ) ; **queue** donne la liste que l'on obtient quand on a enlevé la tête d'une liste non vide.

Dans ce cas, le programme obtenu calcule la fonction **miroir ;** par exemple pour l'initialisation $(\Lambda, abcd) \in D^2$, le programme rend le résultat (dcba, Λ) ; en d'autres termes **miroir** (abcd) = dcba. □

Exercice 6

Quelle est la fonction calculée quand on prend comme interprétation du schéma de programme de l'exemple 1 :

$$D = \mathbb{N}\ ;\quad I_0[p]\,(x) = (x = 0)\ ;\quad I_0[f] = 0\ ;$$
$$I_0[g]\,(x, y) = \text{SI } y \neq 0 \text{ ALORS } x + y + y - 1 \text{ SINON } 0\ ;$$
$$I_0[h]\,(x) = \text{SI } x \neq 0 \text{ ALORS } x - 1 \text{ SINON } 0\ ?$$

□

Exercice 7

Trouver d'autres interprétations pour le schéma de programme de l'exemple 1. En existe-t-il une pour laquelle, intuitivement, le programme obtenu ne termine jamais ?

En existe-t-il une pour laquelle la terminaison dépend de l'initialisation ?

□

Exercice 8

Pour le schéma de programme de l'exercice 5, trouver trois interprétations telles que :

— la première calcule la fonction pgcd ;
— la deuxième ne termine jamais ;
— la troisième ne termine que pour certaines initialisations.

□

2.3 Description de l'exécution d'un programme

Etant donné un programme π (formé d'un schéma et d'une interprétation) (*) et une initialisation $d \in D^n$, décrivons le calcul effectué par π à partir de la première instruction. A un instant donné, l'état du programme est caractérisé par la valeur du compteur d'instruction, c'est-à-dire le numéro de l'instruction qui va être exécutée et l'état mémoire, c'est-à-dire les valeurs actuelles des variables. Définissons donc une **description instantanée** d'un programme π comme un couple (σ, x) où σ est le numéro de la prochaine instruction et x est le n-uple de D^n formé des valeurs des variables à cet instant.

Nous écrivons $(\sigma, x) \xrightarrow{1} (\sigma', x')$ si l'on passe d'une description instantanée (σ, x) à (σ', x') en exécutant l'instruction ε_σ de numéro σ, c'est-à-dire si et seulement si une des conditions suivantes est vérifiée :

(a) ε_σ est une affectation du type : $\varepsilon_\sigma = x_i := f(x_{j(1)}, \ldots, x_{j(m)})$

$$x'_k = x_k \quad \text{pour} \quad k \neq i$$
$$x'_i = I_0[f]\,(x_{j(1)}, \ldots, x_{j(m)})$$
$$\sigma' = \sigma + 1\ .$$

(*) En toute rigueur il faudrait noter (π, I) le programme formé du schéma π et de l'interprétation I.

(b) ε_σ est une affectation du type : $\varepsilon_\sigma = x_i := x_j$

$$x'_k = x_k \quad \text{pour} \quad k \neq i$$
$$x'_i = x_j$$
$$\sigma' = \sigma + 1 .$$

(c) ε_σ est une instruction de test : $\varepsilon_\sigma = p(x_{j(1)}, \ldots, x_{j(m)})$ h, k

$$x' = x$$
$$\sigma' = \text{SI } I_0[p]\,(x_{j(1)}, \ldots, x_{j(m)}) \text{ ALORS h SINON k} .$$

Un **calcul de longueur** m pour le couple (I, d) est une suite $(\sigma_1, x^1), \ldots, (\sigma_m, x^m)$ telle que :

* $\sigma_1 = 1$ et $x^1 = d$ (le calcul commence au début du programme).
* Pour $i = 1, \ldots, m - 1$, $(\sigma_i, x^i) \xrightarrow{I} (\sigma_{i+1}, x^{i+1})$.

On dit que le calcul est réussi si l'instruction de numéro σ_m est *STOP*. Remarquons qu'un calcul réussi est unique d'après la définition de la relation $\xrightarrow{I}$, en effet, nous avons choisi des interprétations déterministes (voir l'exercice 11).

Si $(\sigma_1, x^1), \ldots, (\sigma_m, x^m)$ est un calcul réussi du programme π pour l'interprétation I et l'initialisation d, on appelle **résultat** du programme l'élément x^m et l'on note $I[\pi]\,(d)$ ce résultat. Si I est une interprétation sur le domaine D, $I[\pi]$ est une fonction partielle de D^n dans D^n.

On peut ne s'intéresser qu'à l'une des composantes (la i-ième par exemple) de $I[\pi]\,(d)$. On note alors $I[\pi]_i\,(d)$ cette composante.

La suite $\sigma_1, \ldots, \sigma_m$ d'un calcul $(\sigma_1, x^1), \ldots, (\sigma_m, x^m)$ s'appelle une **séquence d'exécution** elle nous indique le cheminement du programme lors d'un calcul.

Exemple 3

Si, pour le schéma de programme de l'exemple 1 muni de l'interprétation proposée à l'exemple 2 a), nous prenons pour initialisation : d = (7, 2), on obtient alors le calcul réussi suivant (voir la figure 3)

$$(1, (7, 2)), \quad (2, (1, 2)), \quad (3, (1, 2)), \quad (4, (2, 2)), \quad (5, (2, 1)),$$
$$(3, (2, 1)), \quad (4, (2, 1)), \quad (5, (2, 0)), \quad (6, (2, 0)) .$$

Nous avons donc $I[\pi]\,(d) = (2, 0)$, c'est le résultat du programme π. Plus généralement, $I[\pi]\,(x, y) = (y\,!, 0)$. En fait, ce qui nous intéresse ici, c'est que $I[\pi]_1\,(x, y) = y\,!$ ($I[\pi]_1\,(x, y)$ désigne la première composante de $I[\pi]\,(x, y)$).

Exercice 9

Donner le calcul du programme si l'on prend l'interprétation de l'exemple 2 b) et comme initialisation, d = (Λ, abcd). □

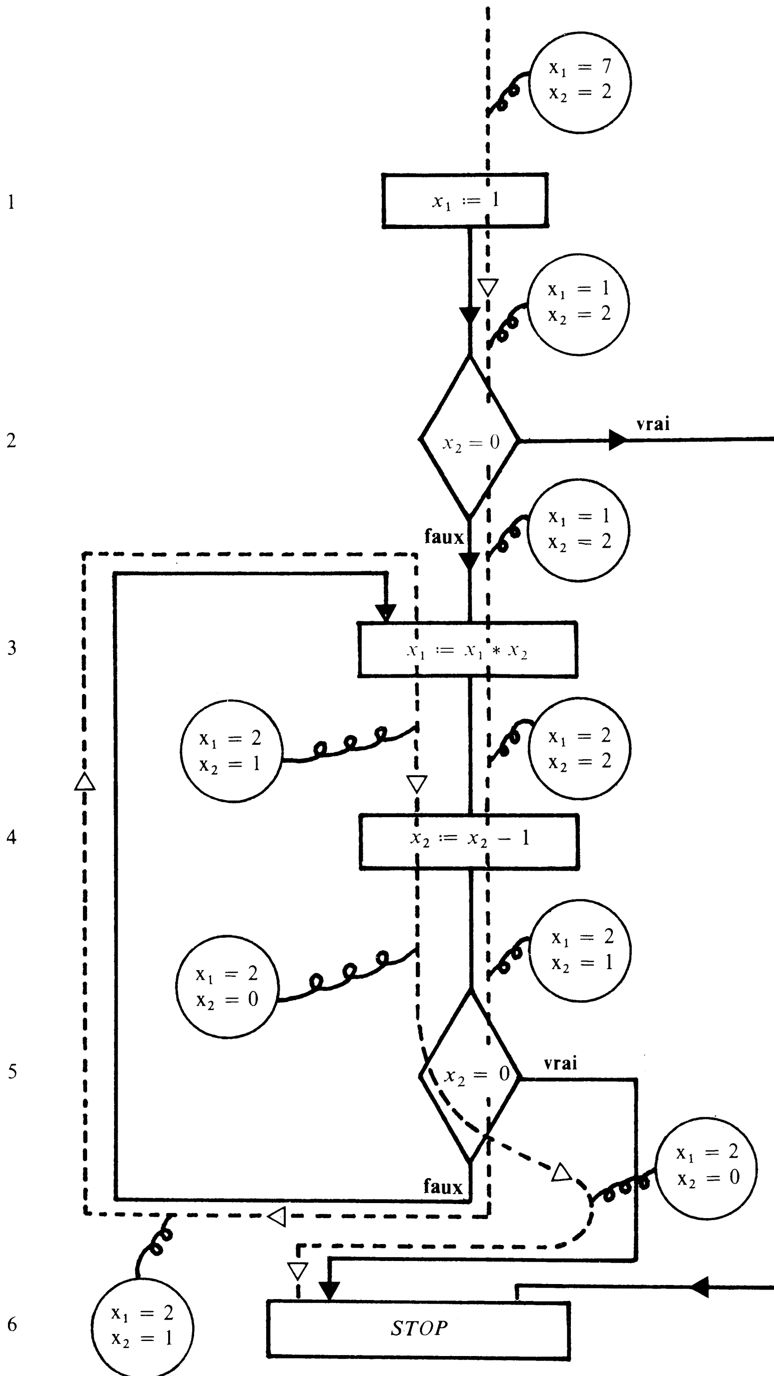


Figure 3 Une séquence de calcul représentée sur un organigramme.

Exercice 10

Examiner les notions de calcul réussi et de séquence d'exécution sur le programme π décrit par le schéma :

1 $p(x)$ 1 , 1
2 STOP

et une interprétation I quelconque.
Quelle est la fonction $I[\pi]$? □

Exercice 11

1) Imaginer une définition de la relation $\xrightarrow{I}$, des calculs et des résultats dans le cas des interprétations indéterministes, c'est-à-dire le cas où $I_0[f]$ est une relation de D^n vers D, autrement dit un sous-ensemble de $D^n \times D$.

2) Trouver le résultat du programme déduit du schéma de l'exemple 1, par l'interprétation

$$D = V^* \; ;$$
$$I_0[f] = \Lambda \; ; \quad I_0[g] = \{ (x, y, z) \mid z = \mathbf{cons}(\mathbf{t\hat{e}te}(y), x) \quad \text{ou} \quad z = x \} \; ;$$
$$I_0[h] = \{ (x, y) \mid y = \mathbf{queue}(x) \} \; ;$$
$$I_0[p] = (x = \Lambda) \, .$$

2.4 Interprétations libres

Nous cherchons à obtenir des résultats indépendants d'interprétations particulières. Pour cela, nous allons introduire une classe d'interprétations qui présente un caractère universel et qui permet de rendre compte de toutes les autres : les **interprétations libres** ou **interprétations de Herbrand.** Il y a abus de langage car on devrait dire interprétation et initialisation libres.

Le domaine considéré est celui des schémas fonctionnels $\text{sch}(\mathcal{F}, \mathcal{X})$ (§ 1). L'initialisation est toujours l'initialisation canonique qui est $(x_1, \ldots, x_n)$, l'interprétation de f est définie par

$$I_0[f] \, (\alpha_1, \ldots, \alpha_p) = f(\alpha_1, \ldots, \alpha_p)$$

et aucune condition n'est imposée aux interprétations des prédicats.

L'intérêt des interprétations libres réside en particulier dans le résultat suivant :

Théorème 1

Soit π un schéma de programme. Il existe une interprétation I telle que $\sigma_1, \ldots, \sigma_m$ soit une séquence d'exécution si et seulement si c'est une séquence d'exécution pour une interprétation libre. □

Démonstration : La condition est évidemment suffisante. L'idée de la démonstration de la réciproque est la suivante : si l'on s'est donné une interprétation I_0 des symboles fonctionnels de $\mathcal{F}$ et une valeur $d = (d_1, \ldots, d_n)$ des

variables, chaque schéma fonctionnel u peut être « évalué » ; on lui donne la valeur $\varphi(u)$ qui vérifie :

$$\varphi(u) = \mathrm{I}[u]\,(\mathrm{d})\,.$$

Plus généralement si $u = (u_1, \ldots, u_k)$, on note $\varphi(u)$ le k-uplet $(\varphi(u_1), \ldots, \varphi(u_k))$. A partir d'un prédicat R sur D^k, on déduit immédiatement un prédicat R′ sur $\mathrm{sch}(\mathscr{F}, \mathscr{X})^\mathrm{k}$ vérifiant :

$$\mathrm{R}'(u) = \mathrm{R}(\varphi(u)) = \mathrm{R} \circ \varphi(u)\,.$$

Soient maintenant un schéma de programme π, une interprétation I_0 de $\mathscr{F} \cup \mathscr{P}$ et une initialisation d. Le prolongement I de I_0 induit une interprétation libre I^d et, par suite, l'exécution d'un programme sur $\mathrm{sch}(\mathscr{F}, \mathscr{X})$.

Pour chaque symbole de prédicat p d'arité k, on définit le prédicat $\mathrm{I}_0^\mathrm{d}[p]$ sur $\mathrm{sch}(\mathscr{F}, \mathscr{X})^\mathrm{k}$ en posant

$$\mathrm{I}_0^\mathrm{d}[p] = \mathrm{I}_0[p] \circ \varphi$$

où φ est la fonction définie ci-dessus, c'est-à-dire :

$$\mathrm{I}_0^\mathrm{d}[p]\,(u) = \mathrm{I}_0[p]\,(\mathrm{I}[u]\,(\mathrm{d}))\,.$$

Si $(\sigma_1, \mathrm{y}^1), \ldots, (\sigma_\mathrm{m}, \mathrm{y}^\mathrm{m})$ est une séquence d'exécution de π pour I et l'initialisation d, alors π admet pour I^d et l'initialisation x une séquence d'exécution de la forme $(\sigma_1, \mathrm{u}^1), \ldots, (\sigma_\mathrm{m}, \mathrm{u}^\mathrm{m})$ avec $\mathrm{y}^\mathrm{k} = \varphi(\mathrm{u}^\mathrm{k})$ pour $1 \leqslant \mathrm{k} \leqslant \mathrm{m}$. En d'autres termes, le programme défini par π, I^d et l'initialisation x calcule, à chaque étape k, les schémas fonctionnels $\mathrm{u}_\mathrm{i}^\mathrm{k}$, qui conservent l'« histoire » du calcul des valeurs $\mathrm{y}_\mathrm{i}^\mathrm{k}$ à cet instant. Montrons cela par récurrence sur k :

— Pour $\mathrm{k} = 1$ et $\sigma_1 = 1$, on a $\mathrm{y}^1 = \mathrm{d}$ et $\mathrm{u}^1 = x$.

— Pour $\mathrm{k} > 1$, supposons que $\mathrm{y}^{\mathrm{k}-1} = \varphi(u^{k-1})$. Si $\sigma_{\mathrm{k}-1}$ est le numéro d'instruction d'une affectation $x_j := f(x_{i_1}, \ldots, x_{i_r})$, alors

$$\mathrm{y}_\mathrm{j}^\mathrm{k} = \mathrm{I}_0[f]\,(\mathrm{y}_{\mathrm{i}_1}^{\mathrm{k}-1}, \ldots, \mathrm{y}_{\mathrm{i}_r}^{\mathrm{k}-1}) \quad \text{et} \quad u_j^k = f(u_{i_1}^{k-1}, \ldots, u_{i_r}^{k-1})$$

et donc par hypothèse de récurrence $\mathrm{y}_\mathrm{j}^\mathrm{k} = \varphi(u_j^k)$ et $\mathrm{y}^\mathrm{k} = \varphi(u^k)$. S'il s'agit d'une affectation $x_j := x_i$, c'est encore plus simple car $\mathrm{y}_\mathrm{j}^\mathrm{k} = \mathrm{y}_\mathrm{i}^{\mathrm{k}-1}$ et $\mathrm{u}_\mathrm{j}^\mathrm{k} = \mathrm{u}_\mathrm{i}^{\mathrm{k}-1}$ et donc par hypothèse de récurrence $\mathrm{y}_\mathrm{j}^\mathrm{k} = \varphi(u_j^k) = \mathrm{y}_\mathrm{i}^{\mathrm{k}-1} = \varphi(u_i^{k-1})$. Enfin si $\sigma_{\mathrm{k}-1}$ est le numéro d'un test $p(x_{i_1}, \ldots, x_{i_r})\,\sigma, \sigma'$, alors $\mathrm{y}^\mathrm{k} = \mathrm{y}^{\mathrm{k}-1}$, $u^k = u^{k-1}$ et donc $\mathrm{y}^\mathrm{k} = \varphi(u^k)$. D'autre part

$$\mathrm{I}_0^\mathrm{d}[\mathrm{p}]\,(u^{k-1}) = \mathrm{I}^\mathrm{d}[\mathrm{p}] \circ \varphi(\mathrm{u}^{\mathrm{k}-1}) = \mathrm{I}_0[\mathrm{p}]\,(\mathrm{y}^{\mathrm{k}-1})\,.$$

et donc le numéro de l'instruction suivant σ_k est le même dans les deux cas. c.q.f.d.

Exercice 12

Quelle est l'interprétation libre correspondant à l'interprétation et à l'initialisation de l'exemple 3 ? Quel est le calcul associé ? □

Le théorème précédent est à la base d'une technique de vérification de programmes appelée « évaluation symbolique ». Pour savoir ce que fait un

programme ou plus exactement s'il fait bien ce qu'on attend de lui, on le fait calculer sur les interprétations libres (ou des interprétations déduites des interprétations libres en tenant compte de règles de simplification, par exemple $x + x - x = x$).

2.5 Les schémas à une variable et les schémas de Ianov

a) Schémas de Ianov

Supposons que l'ensemble $\mathscr{X}$ soit réduit à une seule variable x. On peut alors considérer que toutes les fonctions sont monadiques en posant par exemple :

$$h(x) = f(x, x, x) .$$

Voici un exemple de schéma à une variable :

1 $x := h(x)$
2 $x := k(x)$
3 $q(x)$ 1 , 4
4 *STOP*

Ces schémas de programme simulent assez bien le comportement d'une machine, à condition de considérer x comme un vecteur d'état qui contient à chaque instant tous les éléments variables de la machine.

Exercice 13

Décrire un schéma de programme à une seule variable et une interprétation qui « calcule » la fonction factorielle en s'inspirant des exemples 1 et 2. □

Il n'y a plus d'ambiguïté, maintenant, sur les affectations; ainsi on peut écrire simplement h à la place de $x := h(x)$.
Le schéma de l'exemple devient donc

1 h
2 k
3 q 1 , 4
4 *STOP*

De tels schémas où l'on trouve trois types d'instructions : actions, tests et arrêts s'appellent des schémas de Ianov.

b) Trace

Les suites des instructions des calculs réussis du schéma de Ianov que nous avons vu, forment le langage

$$hk(qhk)^* \, \overline{q} \, STOP$$

où $\overline{q}$ signifie, qu'après le test, on a exécuté l'instruction 4. De plus, comme toute suite d'instructions d'un calcul réussi se termine par *STOP*, sa présence est inutile. Nous ne le noterons pas. Le langage $hk(qhk)^* \, \overline{q}$ s'appelle la **trace** du schéma de Ianov.

Soit π un schéma de Ianov à n instructions, utilisant les vocabulaires

$$\mathscr{F} = \{ f, g, \dots \} \quad \text{et} \quad \mathscr{P} = \{ p, q, \dots \} .$$

La trace du schéma π, notée $\mathbf{Tr}(\pi)$, est le langage sur $\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}} = \mathscr{V}$ (où $\overline{\mathscr{P}} = \{ \overline{p}, \overline{q}, \dots \}$) défini ainsi :

Considérons le langage $L(\pi)$ inclus dans $([1 : n].(\mathscr{V} \cup \{ STOP \}))^*$, formé de tous les mots $\alpha = 1 \dots m$ STOP qui vérifient :

- si iaj est un sous-mot de α alors :

— ou bien l'instruction de rang i s'écrit i*f* et alors $a = f$ et $j = i + 1$

— ou bien elle s'écrit i*p*k, l et alors $a = p$ et $j = k$ ou bien $a = \overline{p}$ et $j = l$

- et, de plus, m *STOP* est une instruction de π.

On suppose, en outre, que dans une suite de prédicats consécutifs on ne rencontre pas un prédicat et son contraire.

La trace est le langage obtenu en supprimant dans tous les mots de $L(\pi)$ les occurrences de numéros d'instructions k appartenant à [1 : n], ainsi que le signe *STOP* qui termine ce mot.

Exercice 14

Quelle est la trace du schéma suivant ?

1 *h*
2 *k*
3 *q* 5 , 4
4 *STOP*
5 *h*
6 *k*
7 *q* 1 , 8
8 *STOP*

Comparer avec le schéma de l'exemple précédent. □

Exercice 15

Montrer que les traces sont des langages réguliers. □

c) Comparaison

Un programme, c'est-à-dire un schéma de programme interprété, peut toujours être considéré comme l'interprétation d'un certain schéma de Ianov en utilisant le concept de variable d'état ; par conséquent, les schémas de Ianov et les schémas de programme avec branchements formalisent les mêmes notions, mais en fait à des niveaux différents ; les schémas de Ianov constituent une abstraction supérieure à celle des schémas avec branchements et cette abstraction perd de l'information sur le programme qu'elle formalise : il y a plus d'interprétations possibles dans le cas des schémas de Ianov. Ceci explique les différences de résultats que l'on obtient.

Chaque interprétation d'un schéma avec branchement π peut être considérée comme l'interprétation d'un schéma de Ianov σ toujours le même. On peut admettre que ce schéma de programme π particularise parmi les interprétations

de σ une certaine famille. Par exemple considérons que π est un schéma de programme avec branchement non interprété utilisant deux variables x, y et une fonction monadique f et supposons que nous ayons deux instructions

$$\begin{array}{ll} \mathrm{i} & x := f(x) \\ \mathrm{j} & y := f(y) . \end{array}$$

Pour chaque interprétation, l'effet de l'instruction i sur la variable x est identique à l'effet de l'instruction j sur la variable y (c'est ce fait important qui sera utilisé dans l'exemple 5). Ces instructions n'agissent que sur une variable à la fois : x pour i et y pour j. Ces faits ne pourront pas être traduits au niveau du schéma de Ianov σ, on écrira par exemple :

$$\begin{array}{ll} \mathrm{i} & f_1 \\ \mathrm{j} & f_2 \end{array}$$

où f_1 et f_2 sont deux actions différentes ; ce qui permet beaucoup d'interprétations de σ, certaines étant très éloignées de celles de π.

Aussi peut-on dire que la classe des interprétations de σ « contient » celle des interprétations de π. Au paragraphe 3.3 cette approche est développée avec une présentation différente.

2.6 Isologie entre schémas de programme

Considérons les trois schémas de programmes de la figure 4 (donnés sous forme d'organigrammes).

Les schémas π_1 et π_2 diffèrent seulement par l'introduction du test $q(y)$ Soit (I, d) un couple formé d'une interprétation et d'une initialisation. Si $\mathrm{I}[q]\,(d_2) =$ **vrai**, alors π_1 et π_2 se déroulent de la même manière pour le couple (I, d) ; en revanche, si $\mathrm{I}[q]\,(d_2) =$ **faux**, le programme π_2 ne termine pas et il se peut que π_1 donne un résultat. Par contre, le schéma de programme π_3 calcule dans tous les cas la même fonction que π_1 (voir exercice 16).

Une des idées qui va nous guider est celle-ci : nous voudrions remplacer un programme par un programme qui fasse la même chose (plus simple par exemple) et peut-être nous ramener à une forme canonique. Nous sommes donc amenés à introduire une relation entre schémas de programme que nous appelons isologie (*) ; cette relation peut être considérée soit en se plaçant sur une interprétation donnée, soit indépendamment de toute interprétation. La seconde solution présente l'avantage de ne pas faire intervenir la structure particulière de D (et des fonctions définies sur D) ; malheureusement, dans ce cas, beaucoup de résultats trop généraux sont négatifs par manque de renseignements sur le programme.

(*) Nous utilisons le mot isologie de préférence à équivalence pour éviter les confusions car les relations d'isologies définies ne sont pas toujours des relations d'équivalence.

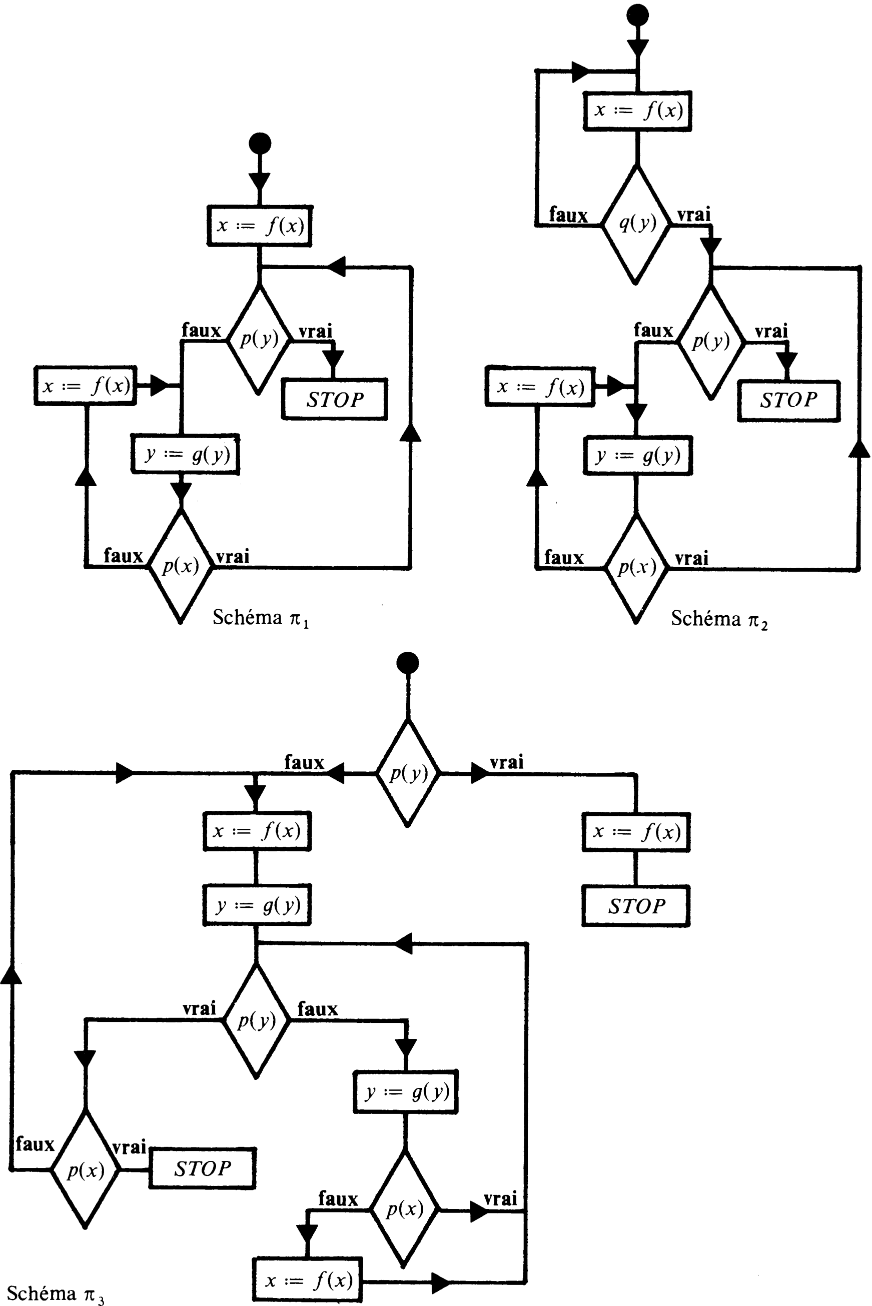


Figure 4 Trois schémas de programme.

a) *Définitions*

Nous considérons les cinq relations suivantes. Soient π et π' deux schémas de programme construits à l'aide des mêmes ensembles $\mathscr{X}$, $\mathscr{F}$, $\mathscr{P}$; on dit que :

1) π et π' sont **fortement isologues** si, pour toute interprétation I on a :

$$I[\pi] = I[\pi'] , \quad \text{ce qui est noté} \quad \pi \equiv \pi' .$$

Autrement dit, ou bien les deux programmes terminent et donnent le même résultat, ou bien les deux programmes ne terminent pas.

2) π et π' sont **faiblement isologues** si pour toute interprétation et pour toute initialisation d telle que $I[\pi]$ (d) et $I[\pi']$ (d) sont définis on a :

$$I[\pi]\,(d) = I[\pi']\,(d) , \quad \text{cette relation est notée} \quad \pi \simeq \pi' .$$

3) π et π' sont **finiment isologues** si pour toute interprétation I de domaine fini :

$$I[\pi] = I[\pi'] , \quad \text{ce qui est noté} \quad \pi \equiv_F \pi' .$$

4) π et π' sont **fortement isologues sur I** si

$$I[\pi] = I[\pi'] , \quad \text{ce qui est noté} \quad \pi \equiv \pi' \ (I) .$$

5) π et π' sont **faiblement isologues sur I** si pour toute initialisation telle que $I[\pi]$ (d) et $I[\pi']$ (d) est défini on a :

$$I[\pi]\,(d) = I[\pi']\,(d) , \quad \text{ce qui est noté} \quad \pi \simeq \pi' \ (I) .$$ □

Remarque

L'isologie forte et l'isologie finie sont des relations d'équivalence, tandis que l'isologie faible n'est pas une relation d'équivalence.

Exemple 4

Considérons les trois schémas de programme suivants (donnés sous forme d'organigrammes dans la figure 5)

π_4 : 1 $p(x)$ 1 , 1
2 *STOP*

π_5 : 1 $y := f_0(z)$
2 $x := f_1$
3 $p(y)$ 7, 4
4 $x := f_2(x, y)$
5 $y := f_3(y)$
6 $p(y)$ 7 , 4
7 *STOP*

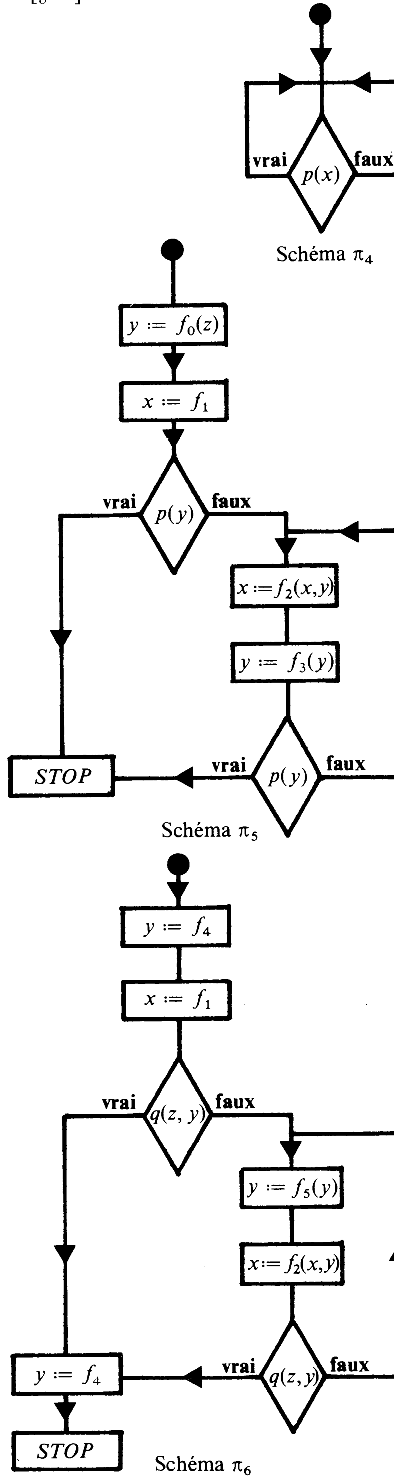


Figure 5 Les trois organigrammes de l'exemple 5.

π_6 : 1 $y := f_4$
2 $x := f_1$
3 $q(z, y)$ 7 , 4
4 $y := f_5(y)$
5 $x := f_2(x, y)$
6 $q(z, y)$ 7 , 4
7 $y := f_4$
8 *STOP* .

Il est clair que ni $\pi_4 \equiv \pi_5$, ni $\pi_4 \equiv \pi_6$. Remarquons cependant que quel que soit I

$$I[\pi_4] \sqsubseteq I[\pi_6], \quad \text{donc} \quad \pi_4 \simeq \pi_6$$
$$I[\pi_4] \sqsubseteq I[\pi_5], \quad \text{donc} \quad \pi_4 \simeq \pi_5,$$

mais que pour certaines interprétations I (en chercher une simple), on peut avoir $I[\pi_5]$ (d) et $I[\pi_6]$ (d) tous deux définis mais distincts.

Prenons maintenant l'interprétation suivante :

$D = \mathbb{N}$
$I_0[f_0](x) = x$; $I_0[f_1] = 1$; $I_0[f_2](x, y) = x * y$; $I_0[f_3](y) = y - 1$;
$I_0[f_4] = 0$; $I_0[f_5](y) = y + 1$;
$I_0[p](y) = (y = 0)$; $I_0[q](x, y) = (x = y)$.

Nous constatons que $I_0[\pi_5] = I_0[\pi_6]$, plus exactement :

$$I_0[\pi_4](x, y, z) = (z, 0, z) = I_0[\pi_5](x, y, z) .$$ □

* *Exercice* 16

Montrer que les schémas π_1 et π_3 de la figure 4 sont fortement isologues. Montrer que le schéma π_7 de la figure 6 n'est pas faiblement isologue à π_1. □

Exercice 17

Soient les schémas :

π : 1 $x := f$
2 *STOP*

ρ : 1 $p(x)$ 2 , 7
2 $q(x)$ 3 , 5
3 $x := g(x)$
4 $p(x)$ 2 , 7
5 $x := h(x)$
6 $q(x)$ 2 , 7
7 *STOP* .

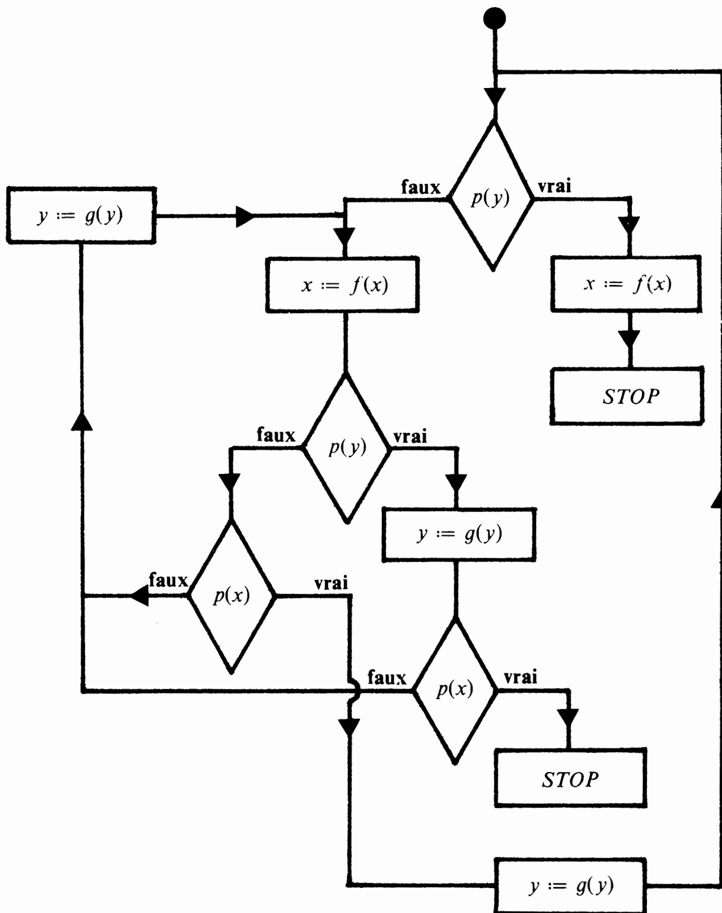


Figure 6 Schéma π_7.

a) Considérons l'interprétation I suivante :

D = ℕ

I[f] = 1 ; I[g] (x) = x ÷ 2 (÷ désigne la division entière) ; I[h] (x) = x − 1 ;
I[p] (x) = (x ≠ 1) ; I[q] (x) = (x est pair) .

Montrer que $\pi \equiv \rho(\mathrm{I})$.

b) Soit maintenant l'interprétation I :

D = ℕ
I[f] = 1 ; I [g] (x) = x ÷ 2 ; I[h] (x) = 3 ∗ x + 1
I[p] (x) = (x ≠ 1) ; I[q] (x) = (x est pair) .

Vérifier que $\pi \simeq \rho$ (I) .
A-t-on $\pi \equiv \rho$ (I) ?

(Cette dernière question difficile n'a pas encore, à notre connaissance, reçu de solution.) □

b) Liaisons entre les différentes isologies

Théorème 2

Si $\pi \equiv \pi'$ alors $\pi \equiv_F \pi'$

et si $\pi \equiv_F \pi'$ alors $\pi \simeq \pi'$,

mais les réciproques sont fausses. □

Démonstration : La première implication est évidente. Pour démontrer la seconde, il suffit de montrer que si $\pi \equiv_F \pi'$, alors pour une interprétation I et une initialisation d $I[\pi](d) = I[\pi'](d)$, dès qu'ils sont tous deux définis. Remarquons que si $I[\pi](d)$ et $I[\pi'](d)$ sont définis, les valeurs manipulées durant le déroulement du programme sont en nombre fini, même si le domaine d'interprétation est infini. Donc nous pouvons nous restreindre à un domaine fini, l'ensemble des valeurs manipulées par π et π' et considérer une interprétation finie I' sur ce domaine telle que

$$I[\pi](d) = I'[\pi](d)$$
$$I[\pi'](d) = I'[\pi'](d)$$

or $\pi \equiv_F \pi'$

donc $I'[\pi](d) = I'[\pi'](d)$.

La réciproque de la seconde implication est évidemment fausse, en particulier tout schéma de programme est faiblement équivalent au schéma de programme :

$$1\ p(x)\ 1\ ,\ 1\ .$$

L'exemple suivant nous montre que la réciproque de la première implication est fausse.

Exemple 5

Considérons le schéma de programme π décrit par l'organigramme de la figure 7.

L'exercice 13 du chapitre 4 montre que ce schéma de programme boucle si et seulement si

$$I[p(f^n(x)](d) = \textbf{faux} \text{ est équivalent à } \exists k > 0 \left(n = \frac{k(k+1)}{2}\right)$$

c'est-à-dire si $I[p(f^n(x))](d)$ prend successivement les valeurs

faux , **vrai** , **faux** , **vrai** , **vrai** , **faux** , **vrai** , **vrai** , **vrai** , **faux** , ...

Cette éventualité n'est possible que si l'interprétation est faite sur un domaine infini, car sur un domaine fini, la fonction $\lambda n . I[p(f^n(x))](d)$ a toujours une période. □

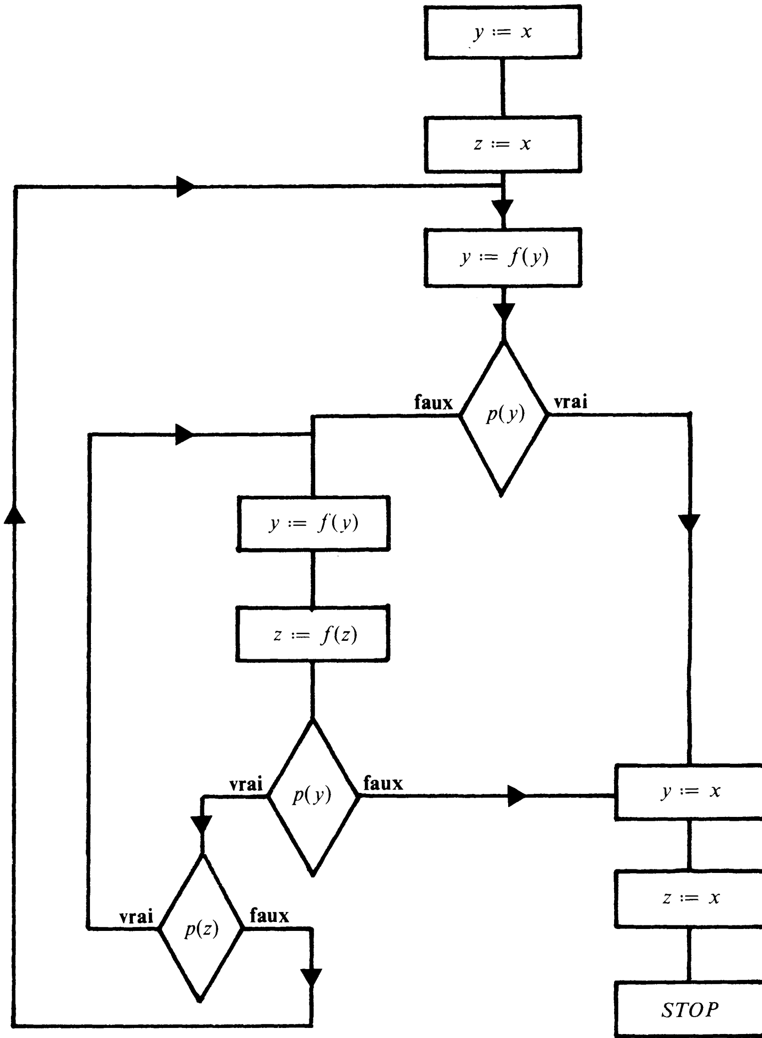


Figure 7 Schéma π.

2.7 Problèmes d'indécidabilité

a) Rappels

On dit qu'une propriété est **décidable** sur un ensemble E s'il existe un algorithme (décrit par un programme) qui, pour chaque élément a ∈ E, répond au bout d'un temps fini par oui si la propriété est vraie pour a, par non si la propriété est fausse pour a.

On dit qu'une propriété est **semi-décidable** sur E s'il existe un algorithme qui, pour chaque élément a ∈ E, répond, au bout d'un temps fini, par oui si la

propriété est vraie pour a et calcule éventuellement indéfiniment si la propriété est fausse pour a.

La propriété P(π) : « le programme π s'arrête » est une propriété semi-décidable, l'algorithme correspondant s'écrit : « activer π ; sortir (« oui ») ».

Une fonction est intuitivement **calculable** si elle peut être calculée par un programme (thèse de Church).

Les problèmes d'indécidabilité se résolvent habituellement de la manière suivante :

— ou on démontre qu'un problème est indécidable (presque toujours par le procédé de la diagonale dont nous verrons un exemple plus loin),

— ou on ramène ce problème à un problème dont l'indécidabilité a été démontrée (chaîne de Church).

b) Théorème de l'indécidabilité de l'arrêt des programmes sur les entiers

On s'intéresse ici aux schémas de programmes sur $\mathscr{F} = \{z, pr, s\}$, $\mathscr{P} = \{t\}$

(où $\mathbf{ar}(z) = 0$, $\mathbf{ar}(pr) = \mathbf{ar}(s) = 1$, $\mathbf{ar}(t) = 1$)

et à l'interprétation particulière I suivante :

$D = \mathbb{N}$ et

$I[z] = 0$;

$I[pr](a) = \text{SI} \quad a \neq 0 \quad \text{ALORS} \quad a - 1 \quad \text{SINON} \quad 0$;

$I[s](a) = a + 1$;

$I[t](a) = (a = 0)$.

Théorème 3

Avec les notations précédentes, la propriété « π se termine pour I » est semi-décidable et non décidable sur l'ensemble des schémas de programmes construits sur $\mathscr{F}$ et $\mathscr{P}$. □

Principe de la démonstration : (On trouvera une démonstration complète fondée sur ce principe, dans [ENG, 73]).

— Codage des programmes et des initialisations : on définit une application injective qui, à chaque programme π, associe un entier $\bar{\pi}$ (son code ou nombre de Gödel) et une autre application injective calculable qui, à chaque initialisation d, associe un entier $\bar{d}$ (son code ou nombre de Gödel).

— Principe de la diagonale : soit δ la fonction de $\mathbb{N}$ dans $\mathbb{N}$ telle que

$\delta(\bar{\pi}) = \text{SI } I[\pi](\bar{\pi}, 0, \ldots, 0)$ n'est pas défini

(le programme π ne termine pas pour $(\bar{\pi}, 0, \ldots, 0)$) ALORS 0

SINON indéfini

δ n'est pas calculable. En effet, si δ était décrite par un programme ρ, $I[\rho](\bar{\rho})$ différerait de $\delta(\bar{\rho})$, ce qui serait contradictoire.

— Indécidabilité de l'arrêt : Soit Δ une fonction telle que

$\Delta(\bar{\pi}, \bar{d})$ = SI I[π] (d, 0, ..., 0) n'est pas défini ALORS 0 SINON 1

Si Δ était décrite par un programme ρ, on pourrait construire un programme qui calcule δ, ce qui est impossible d'après ce qui précède. Le programme déduit de ρ et calculant δ est décrit sommairement par :
— *calculer* $\bar{d}$ *à partir de l'initialisation* ($\bar{\pi}$, 0, ..., 0)
— *activer* ρ *pour l'initialisation* ($\bar{\pi}$, $\bar{d}$)
— *si le résultat est* 0 *alors STOP sinon activer un programme qui ne termine pas.*

Corollaire

Il n'existe pas d'algorithme ayant pour données une interprétation I, un schéma de programme π et une initialisation d qui détermine si I[π] (d) est défini. □

Démonstration : D'un tel algorithme on déduirait facilement en prenant l'interprétation du théorème précédent un algorithme d'arrêt des programmes sur les entiers.

c) Indécidabilité des isologies dans le cas général

Les théorèmes qui vont suivre sont en général des résultats d'indécidabilité. On peut se restreindre au cas où les fonctions et les prédicats sont monadiques, c'est-à-dire d'arité un ; on parle alors de **schémas de programme monadiques** (on se gardera de les confondre avec les schémas à une variable étudiés en 2.5). Notons que dans ce cas, les schémas fonctionnels associés sont linéaires, c'est-à-dire formés de mots sur l'ensemble $\mathscr{F}$ des symboles fonctionnels, suivis d'un symbole de variable : ils appartiennent à $\mathscr{F}^* \mathscr{X}$. Parmi toutes les variables, nous distinguons x_1 que nous appelons variable de sortie.

Nous allons montrer que, dans ce cadre, les problèmes d'isologie se traduisent en des problèmes d'égalité de langages. Ainsi, d'un problème sur les langages connu comme indécidable, on déduira l'indécidabilité du problème correspondant pour les schémas monadiques et a fortiori pour les schémas les plus généraux. D'après le théorème sur les interprétations libres, pour démontrer l'équivalence de deux schémas de programme, il suffit de considérer de telles interprétations, c'est ce que nous ferons par la suite, sauf mention du contraire.

Soient π un schéma de programme monadique avec $\mathscr{X} = \{ x_1, \ldots, x_n \}$ et I une interprétation libre. A toute étape du calcul, les valeurs associées aux variables sont de la forme αx_i où $\alpha \in \mathscr{F}^*$ et $x_i \in \mathscr{X}$. Si le programme s'arrête, le résultat est $(\alpha_1 y_1, \ldots, \alpha_n y_n)$ où $\alpha_i \in \mathscr{F}^*$ et $y_i \in \mathscr{X}$.

On appelle **langage associé à** π le langage $\Lambda(\pi)$ formé des α_1 obtenus en considérant les différentes interprétations libres de π.

Exemple 6

Considérons le schéma de programme π de la figure 8.

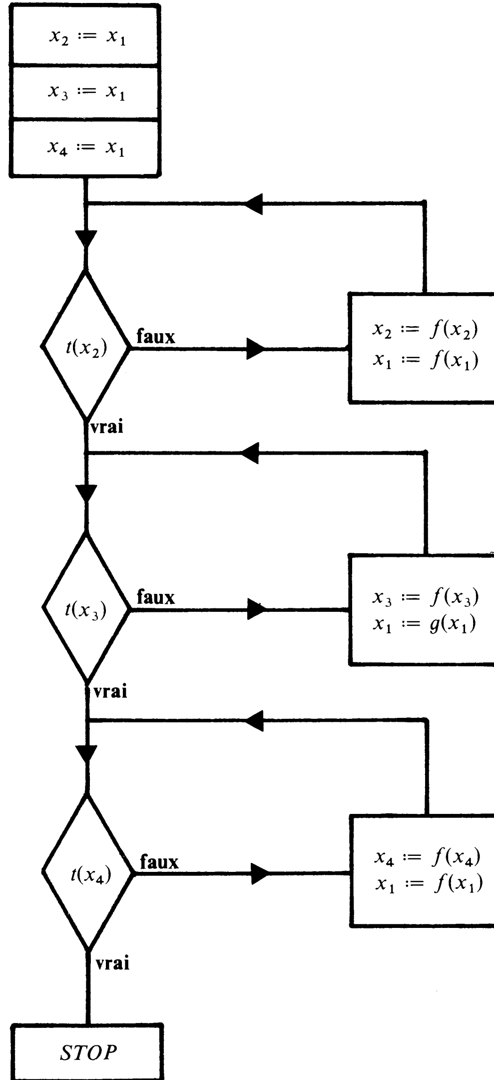


Figure 8 Schéma π.

Soit n le premier entier tel que $\mathrm{I}[t(f^n(x_1))]\,(x) = \mathbf{vrai}$; le résultat de ce programme est

$$(f^n\, g^n\, f^n\, x_1, f^n\, x_1, f^n\, x_1, f^n\, x_1)\,.$$

On en déduit que $\Lambda(\pi) = \{\, f^n\, g^n\, f^n \mid n \geqslant 0 \,\}$.

Remarquons que $\Lambda(\pi)$ n'est pas à contexte libre (c'est-à-dire n'est pas algébrique). □

Théorème 4

La classe des langages associés aux différents schémas de programme monadiques est exactement celle des langages semi-décidables (encore appelés récursivement énumérables ou semi-thuéiens ou de type 0). □

Le résultat suivant précise le lien entre isologie forte de programmes et égalité des langages associés.

Théorème 5

Si $\pi \equiv \rho$ alors $\Lambda(\pi) = \Lambda(\rho)$. □

En effet, si $\alpha \in \Lambda(\pi)$ et $\alpha \notin \Lambda(\rho)$, il existe une interprétation I telle que la première composante du résultat de $I[\pi]$ est αx_i. Mais comme $\alpha \notin \Lambda(\rho)$, la première composante du résultat de $I[\rho]$, s'il existe, n'est pas αx_i, donc nous n'avons pas $\pi \equiv \rho$.

Exercice 18

Trouver deux schémas de programme π et ρ tels que $\Lambda(\pi) = \Lambda(\rho)$ et $\pi \not\equiv \rho$. □

Théorème 6

Les propriétés suivantes ne sont pas semi-décidables :

— $\pi \not\equiv \rho$
— $\pi \equiv_F \rho$
— $\pi \simeq \rho$
— π ne termine jamais
— π s'arrête pour toute interprétation finie □

Théorème 7

Les propriétés suivantes sont partiellement décidables, mais non décidables sur l'ensemble des schémas avec branchement :

— il existe une interprétation et une initialisation telles que π termine,
— π se termine pour toute interprétation et toute initialisation.
— $\pi \not\equiv_F \rho$
— $\pi \not\simeq \rho$. □

Ces résultats, à première vue décevants, ont cependant l'intérêt de nous éviter de rechercher des solutions algorithmiques à des problèmes n'en ayant pas. Par exemple, il est illusoire d'espérer qu'un compilateur Fortran prédise systématiquement l'arrêt de l'exécution d'un programme ; de même, il ne saurait prédire, dans le cas le plus général, si, lors de l'exécution du programme, telle instruction serait exécutée ; en effet, si l'on remplace cette instruction par *STOP*, ceci nous ramène au problème de l'arrêt. De même, il n'est pas possible de construire un programme général de vérification de l'équivalence de programmes.

d) Décidabilité dans le cas des schémas à une variable

Théorème 8

Pour les schémas de programme à une seule variable, le problème de l'isologie forte est décidable. □

Corollaire

Pour les schémas de programme à une seule variable, le problème de savoir s'ils ne terminent jamais est décidable.

Pour démontrer le corollaire, il suffit de considérer l'isologie forte avec un schéma qui ne termine jamais.

La démonstration du théorème utilise la notion de trace d'un schéma de programme définie au paragraphe 2.5. Elle repose sur les points suivants :

1) Deux schémas de programme ayant les mêmes ensembles de symboles de prédicats et de fonctions sont fortement isologues si et seulement si leurs traces sont égales.

2) La trace d'un schéma de programme est un langage régulier.

3) L'égalité de deux langages réguliers est décidable [HOP, 69].

Théorème 9

Pour les schémas à une variable, le problème de l'arrêt pour toute interprétation est décidable. □

En effet, le problème se ramène à tester la finitude d'un langage régulier, ce qui est décidable [HOP, 69].

3 SCHÉMAS ITÉRATIFS

3.1 Introduction

L'étude du paragraphe précédent portait sur des schémas de programmes tous définis de la même façon : on y retrouvait comme instructions de base les affectations et les tests ; le contrôle était réalisé par un traitement séquentiel des instructions ou par des branchements : tout cela correspond à une formalisation de la notion d'organigramme bien connue des programmeurs d'antan.

Les progrès dans la conception des langages de programmation ont abouti, entre autres, à proposer de nouvelles structures pour la partie contrôle des programmes. Les motivations pour introduire ces nouvelles structures sont diverses : faciliter l'analyse des problèmes, faciliter la preuve des programmes, améliorer l'efficacité des programmes,... Les structures proposées en pratique sont très variées et amènent à se poser deux questions :

1) sur le plan de « l'efficacité », peut-on comparer les différentes structures proposées ?

2) sur le plan de la facilité de programmation, quelles sont les structures les plus naturelles et les plus « simples » pour analyser les problèmes ?

La deuxième question sera abordée, de façon indirecte, dans le chapitre 4 lorsque nous étudierons le passage d'un énoncé à un algorithme. Nous allons ici résumer quelques résultats obtenus à propos de 1). Il faut naturellement fixer un cadre plus précis pour aborder cette question : il s'agit d'une part de définir différentes structures de contrôle et, d'autre part, de se donner les moyens de les comparer en définissant des relations entre elles. Dans ce qui suit, nous nous intéresserons essentiellement aux structures itératives. Des études analogues ont été réalisées sur le parallélisme et la collatéralité.

3.2 Structures de contrôle, interprétation et calculs

a) Structures de contrôle

Dans tout ce qui suit, nous négligeons le détail de l'instruction d'affectation ; autrement dit, comme dans les schémas de Ianov (§ 2.5), les affectations sont représentées par un ensemble d'actions de base.

Les constructions envisagées ici sont définies à partir :

— d'un ensemble d'actions $\mathscr{F}$ (chaque élément de $\mathscr{F}$ représente une affectation),
— d'un ensemble de prédicats (ou tests) $\mathscr{P}$.

On se donne d'autre part un ensemble $\overline{\mathscr{P}}$ et on associe à chaque prédicat $p \in \mathscr{P}$ sa négation $\overline{p}$ dans $\overline{\mathscr{P}}$. On pose de plus $\overline{\overline{p}} = p$.

On peut définir plusieurs types de composition pour combiner les éléments de $\mathscr{F}$ et $\mathscr{P}$ afin d'obtenir des assemblages que nous appelons **schémas de programme.** Les symboles auxiliaires (si, alors, ...) sont adjoints à $\mathscr{F}$ et $\mathscr{P}$ pour construire les schémas de programme. En groupant plusieurs types de composition, on définit des structures de contrôle.

Pour chaque structure $\mathscr{S}$ que nous allons étudier, nous donnons une définition intuitive et une définition syntaxique de l'ensemble des schémas de programme correspondants que nous appelons $\mathscr{S}$-programmes.

Exemple 7

Si on veut définir une structure $\mathscr{S}_0$ comprenant seulement la composition séquentielle qu'on note par un symbole « ; », le langage L correspondant à $\mathscr{S}_0$ est défini sur le vocabulaire $\mathscr{V} = \mathscr{F} \cup \{ ; \}$ par

$$\mathrm{L} = \mathscr{F} \cup \mathrm{L} ; \mathscr{F} \qquad \square$$

b) Interprétation

Pour donner un sens à un schéma de programme, on l'interprète ; comme précédemment, une interprétation est formée :

— d'un ensemble non vide D,
— pour chaque action $a \in \mathscr{F}$ d'une application $\mathrm{I}_0[a]$ de D vers D,
— pour chaque prédicat $p \in \mathrm{D}$, d'une application $\mathrm{I}_0[p]$ de D vers { **vrai**, **faux** } telle que $\mathrm{I}_0[p]$ soit la négation de $\mathrm{I}_0[p]$.

Dans ce paragraphe les symboles sont donc tous d'arité un. Pour chaque structure $\mathscr{S}$ nous indiquerons comment prolonger I_0 en une interprétation I des schémas de programme de $\mathscr{S}$.

Exemple 8

Pour la structure $\mathscr{S}_0$ définie dans l'exemple 7, choisissons $\mathscr{F} = \{ a, b \}$ et définissons une interprétation de la façon suivante :

D est l'ensemble des entiers positifs à quatre chiffres décimaux notés $\overline{\mathrm{xyzt}}$; $I_0[a]\,\overline{\mathrm{xyzt}} = \overline{\mathrm{x'y'z't'}}$ où $\overline{\mathrm{x'y'z't'}}$ est déduit de $\overline{\mathrm{xyzt}}$ par permutation et vérifie $x' \geqslant y' \geqslant z' \geqslant t'$; $I_0[b]\,\overline{\mathrm{xyzt}} = |\,\overline{\mathrm{xyzt}} - \overline{\mathrm{tzyx}}\,|$.

I est défini pour tout schéma de L associé à $\mathscr{L}_0$ de la façon suivante :

— pour $l \in \mathscr{F}$ $\quad I[l] = I_0[l]$;
— pour $\pi \in L$ $\quad I[l\,;\pi] = I[\pi] \circ I_0[l]$.

Calculons pour cette interprétation le résultat du schéma de programme $a\,; b\,; a\,; b$ appliqué à la donnée 9090 :

$$\begin{aligned} I[a\,;b\,;a\,;b]\,(\overline{9090}) &= I[b\,;a\,;b] \circ I_0[a]\,(\overline{9090}) \\ &= I[b\,;a\,;b]\,(\overline{9900}) \\ &= I[a\,;b]\,(\overline{9801}) \\ &= I[b]\,(\overline{9810}) \\ &= \overline{9621}\,. \end{aligned}$$ □

* *Exercice* 19

Sous les hypothèses de l'exemple précédent, interpréter pour toute initialisation de D le schéma :

$$a\,;b\,;a\,;b\,;a\,;b\,;a\,;b\,;a\,;b\,;a\,;b\,;a\,;b\,;a\,;b\,.$$ □

c) Calculs

Etant donné un schéma de programme π, une interprétation I et une initialisation $d \in D$, un calcul de π est un mot qui indique la suite des actions et tests à composer pour construire le résultat $I[\pi]\,(d)$; c'est donc un mot sur l'ensemble $\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}}$.

3.3 Définition des $\mathscr{D}$-schémas

a) Le langage des $\mathscr{D}$-schémas

La structure que nous allons présenter a été proposée par DIJKSTRA [DAH, 72] Nous la notons $\mathscr{D}$; elle a été introduite comme un outil permettant une simplification dans la construction de programmes. En général, un schéma admet un point d'entrée et un point de sortie ; on voudrait ici que tout « morceau » de schéma vérifie cette propriété (unicité du point d'entrée et du point de sortie) ; ceci correspond à une analyse « structurée » ; par ailleurs, les preuves de programmes ainsi conçus peuvent être plus systématiques.

Intuitivement, on accepte pour la structure $\mathscr{D}$ la composition séquentielle, la composition conditionnelle, la composition itérative de type « tant que ». Ici $\mathscr{V} = \mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}} \cup \{$;, <u>si</u>, <u>alors</u>, <u>sinon</u>, <u>fsi</u>, <u>tantque</u>, <u>faire</u>, <u>fait</u> $\}$.

Le langage $L_{\mathscr{D}}$ des $\mathscr{D}$-schémas est défini par :

$$L_{\mathscr{D}} = \mathscr{F} \cup L_{\mathscr{D}} \,;\, L_{\mathscr{D}}$$
$$\cup\ \underline{si}\ (\mathscr{P} \cup \overline{\mathscr{P}})\ \underline{alors}\ L_{\mathscr{D}}\ \underline{sinon}\ L_{\mathscr{D}}\ \underline{fsi}$$
$$\cup\ \underline{tant\ que}\ \mathscr{P} \cup \overline{\mathscr{P}}\ \underline{faire}\ L_{\mathscr{D}}\ \underline{fait}\ .$$

b) Organigramme associé à un $\mathscr{D}$-schéma

On peut aussi donner une représentation imagée des compositions proposées (voir figure 9) : elles correspondent bien à des organigrammes à un point d'entrée et un point de sortie. Ainsi, comme pour les schémas avec branchements, on peut associer un **organigramme** à un $\mathscr{D}$-schéma.

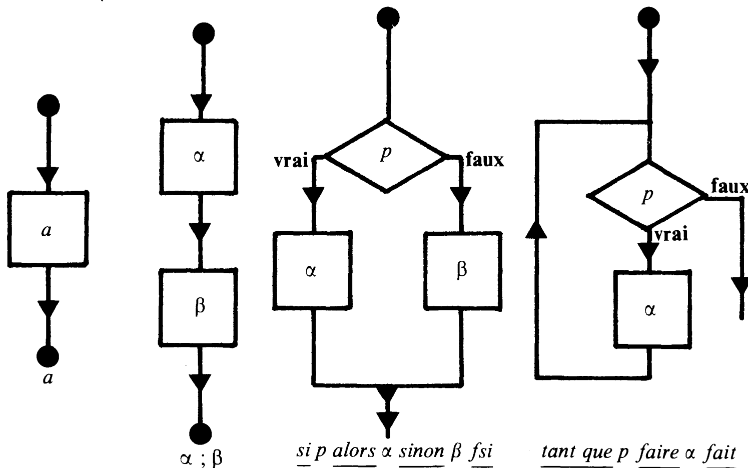


Figure 9 Fragments d'organigrammes associés aux $\mathscr{D}$ schémas.

Exercice 20

Dessiner l'organigramme associé au $\mathscr{D}$-schéma suivant et comparer avec l'exercice 1.

$$\sigma : a_1 \,;\, \underline{si}\ p\ \underline{alors}\ a_2\ \underline{sinon}\ a_3 \,;\, a_4 \,;\, \underline{tantque}\ p\ \underline{faire}\ a_3 \,;\, a_4\ \underline{fait}\ \underline{fsi}$$

(ce schéma est construit sur $\mathscr{F} = \{ a_1, a_2, a_3, a_4 \}$ et $\mathscr{P} = \{ p \}$.) □

Réciproquement, peut-on traduire tout organigramme en un organigramme associé à un $\mathscr{D}$-schéma ? Tout organigramme est un graphe orienté dont chaque point est étiqueté par une instruction, les nœuds sans successeur sont étiquetés par des instructions *STOP* ; soit un circuit γ d'un organigramme. Appelons **sortie** du circuit γ un nœud i tel qu'il existe un chemin de i vers un point *STOP*, aucun autre nœud de ce chemin ne figurant sur le circuit γ.

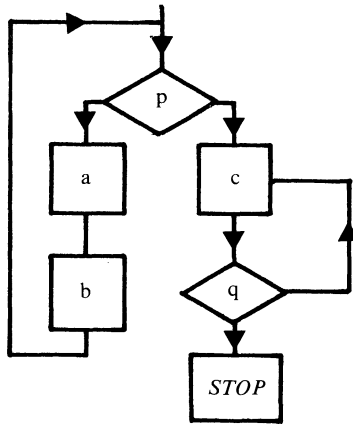


Figure 10.

Sur la figure 10, p est une sortie du circuit p, a, b : c'est le seul nœud de sortie du circuit p, a, b ; de même q est la seule sortie du circuit c, q. Sans faire de démonstration rigoureuse (qui nécessiterait une définition précise de la construction des organigrammes) une récurrence intuitive amène au résultat suivant :

Proposition 1

Tout circuit d'un organigramme associé à un $\mathscr{D}$-schéma admet une et une seule sortie. □

La démonstration se fait par récurrence sur la complexité des organigrammes représentés à la figure 9.

Exemple 9

L'organigramme de la figure 11 peut être interprété comme le calcul de la longueur d'une suite de caractères ; on a prévu dans le calcul un cas « erreur » en cas de lecture d'un caractère incorrect ; la fin de la suite à lire est signalée par le caractère #.

Ce problème, très banal, est représenté par un schéma avec branchements (voir figure 12) comportant un circuit admettant deux points de sortie.

L'organigramme de la figure 12 n'est pas associé à un $\mathscr{D}$-schéma. Ceci ne signifie pas que le problème de comptage posé ne peut être résolu avec un langage comportant uniquement une $\mathscr{D}$-structure. □

Exercice 21

Résoudre le problème décrit dans l'exemple 9 (figure 11) en construisant un programme n'utilisant qu'une $\mathscr{D}$-structure. □

L'exemple précédent nous conduit à préciser la façon de comparer des

structures puisque l'on obtient des résultats différents sur un organigramme et sur un programme (interprété). Dans le paragraphe suivant, nous allons définir différents types de comparaisons.

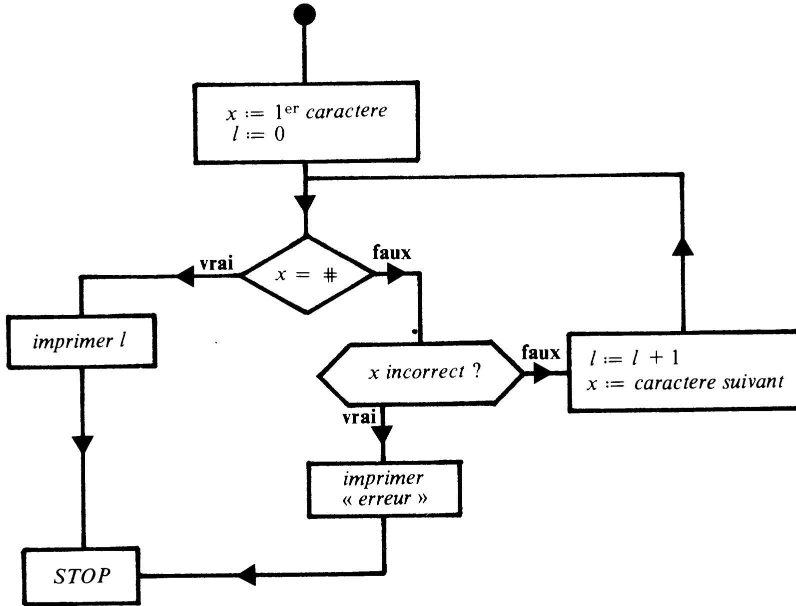


Figure 11.

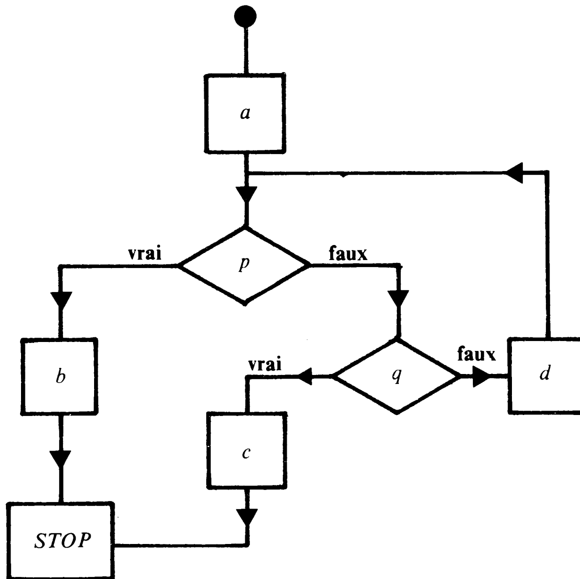


Figure 12 Les nœuds p, q sont deux sorties du circuit p, q, d.

c) Calculs d'un $\mathscr{D}$-schéma

Pour un $\mathscr{D}$-schéma π, une interprétation I_0 et une initialisation $d \in D$ notons $cal(\pi, d)$ le calcul associé.

$cal(\pi, d)$ peut être défini par récurrence sur la complexité de π ; c'est un mot sur $\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}}$. Plus généralement, appelons calcul un mot γ sur $\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}}$ et définissons (pour l'interprétation I) le résultat $res(\gamma, d)$ d'un calcul γ pour l'initialisation d ; $res(\gamma, d)$ est obtenu en composant les fonctions déduites par interprétation des actions de γ ; par récurrence :

$$res(\Lambda, d) = d$$

$$\forall p \in \mathscr{P} \cup \overline{\mathscr{P}} \qquad res(\gamma p, d) = res(\gamma \overline{p}, d) = res(\gamma, d)$$

$$\forall a \in \mathscr{F} \qquad res(\gamma a, d) = I_0[a]\,(res(\gamma, d))$$

En particulier, le calcul du programme π, pour la donnée d admet le résultat $res(cal(\pi, d), d)$.

Définissons enfin $cal(\pi, d)$ par récurrence ; pour $a \in \mathscr{F}, p \in \mathscr{P}, \alpha, \beta \in L_{\mathscr{D}}$:

$cal(a, d) = a$;

$cal(\alpha\ ;\ \beta, d) = cal(\alpha, d).cal(\beta, res(cal(\alpha, d), d))$;

$cal(\underline{si}\ p\ \underline{alors}\ \alpha\ \underline{sinon}\ \beta\ \underline{fsi}, d) =$ SI $I_0[p]$ (d) ALORS $p.cal(\alpha, d)$ SINON $\overline{p}.cal(\beta, d)$;

$cal(\underline{tant\ que}\ p\ \underline{faire}\ \alpha\ \underline{fait}, d) =$ SI $I_0[p]$ (d) ALORS $p.cal(\alpha\ ;\ \underline{tant\ que}\ p\ \underline{faire}\ \alpha\ \underline{fait}, d)$ SINON $\overline{p}$.

Remarquons que ce système d'équations est un système récursif.

De manière analogue l'interprétation I d'un $\mathscr{D}$ schéma peut être définie par récurrence ; pour $p \in \mathscr{P}$, α, $\beta \in L_{\mathscr{D}}$:

$I[\alpha\ ;\ \beta] = I[\beta] \circ I[\alpha]$;

$I[\underline{si}\ p\ \underline{alors}\ \alpha\ \underline{sinon}\ \beta\ \underline{fsi}]$ (d) $=$ SI $I_0[p]$ (d) ALORS $I[\alpha]$ (d) SINON $I[\beta]$ (d) ;

$I[\underline{tantque}\ p\ \underline{faire}\ \alpha\ \underline{fait}]$ (d) $=$ SI $I_0[p]$ (d) ALORS $I[\alpha\ ;\ \underline{tantque}\ p\ \underline{faire}\ \alpha\ \underline{fait}]$ (d) SINON d.

Exercice 22

Montrer par récurrence sur la complexité d'un programme π que si $cal(\pi, d)$ est défini, on a :

$$I[\pi]\,(d) = res(cal(\pi, d), d)$$ □

3.4 Comparaisons entre structures

Dans le paragraphe précédent, nous avons étudié particulièrement la structure des $\mathscr{D}$-schémas. Nous allons présenter ici quelques définitions générales qui précisent différentes façons de comparer des structures. Définis-

sons tout d'abord des relations entre programmes. Soit π_1 et π_2 deux schémas de programmes, R1, R2, R3, R4 sont définies de la façon suivante :

(R1) Pour toute interprétation I de π_1 et π_2 :

$$I[\pi_1] = I[\pi_2] .$$

(R2) Les ensembles d'actions et de prédicats ayant une occurrence dans π_1 ou dans π_2 sont les mêmes.

(R3) Pour toute interprétation I et toute donnée d :

$$\mathrm{cal}(\pi_1, d) = \mathrm{cal}(\pi_2, d) .$$

(R4) Pour toute action ou test, le nombre d'occurrences dans π_1 est supérieur ou égal au nombre d'occurrences dans π_2.

La relation R1 correspond à l'isologie forte sur les schémas avec branchements (§ 2.6.a). Si deux schémas de programme sont liés par la relation R2, le passage de l'un à l'autre se fait sans introduction de nouvelles actions (c'est-à-dire qu'il n'est pas possible d'utiliser de nouvelles variables par exemple). Intuitivement, π_1 R4 π_2 impose qu'aucune « recopie » n'a été faite lors de la transformation de π_1 en π_2.

Exemple 10

Comparons les différents organigrammes de la figure 13.
Intuitivement, on trouve :

π_1 R1 π_2 R1 π_3 (même fonction)
NON π_1 R2 π_3 (e figure dans π_3 et ne figure pas dans π_1) ;
π_1 R3 π_2 R3 π_3 (les calculs sont les mêmes) □

Exercice 23

Vérifier que R1, R2, R3 sont des relations d'équivalence. □

Soit deux structures $\mathscr{S}_1$, $\mathscr{S}_2$ et une relation R :

— $\mathscr{S}_1$ est dite **réductible** en $\mathscr{S}_2$ pour la relation R si pour tout schéma π_1 de la structure $\mathscr{S}_1$, il existe un schéma π_2 de $\mathscr{S}_2$ tel que :

$$\pi_1 \mathrm{R} \pi_2 .$$

Cette relation entre structures est notée $\mathscr{S}_1 \underset{R}{\leqslant} \mathscr{S}_2$.

— $\mathscr{S}_1$ est dite **équivalente** à $\mathscr{S}_2$ pour R si

$$\mathscr{S}_1 \underset{R}{\leqslant} \mathscr{S}_2 \textbf{ et } \mathscr{S}_2 \underset{R}{\leqslant} \mathscr{S}_1$$

cette équivalence est notée $\mathscr{S}_1 \underset{R}{\equiv} \mathscr{S}_2$.

— $\mathscr{S}_1$ est **strictement réductible** en $\mathscr{S}_2$ pour R si

$$\mathscr{S}_1 \underset{R}{\leqslant} \mathscr{S}_2 \textbf{ et non } \mathscr{S}_2 \underset{R}{\leqslant} \mathscr{S}_1$$

cette relation est notée $\mathscr{S}_1 \underset{R}{<} \mathscr{S}_2$.

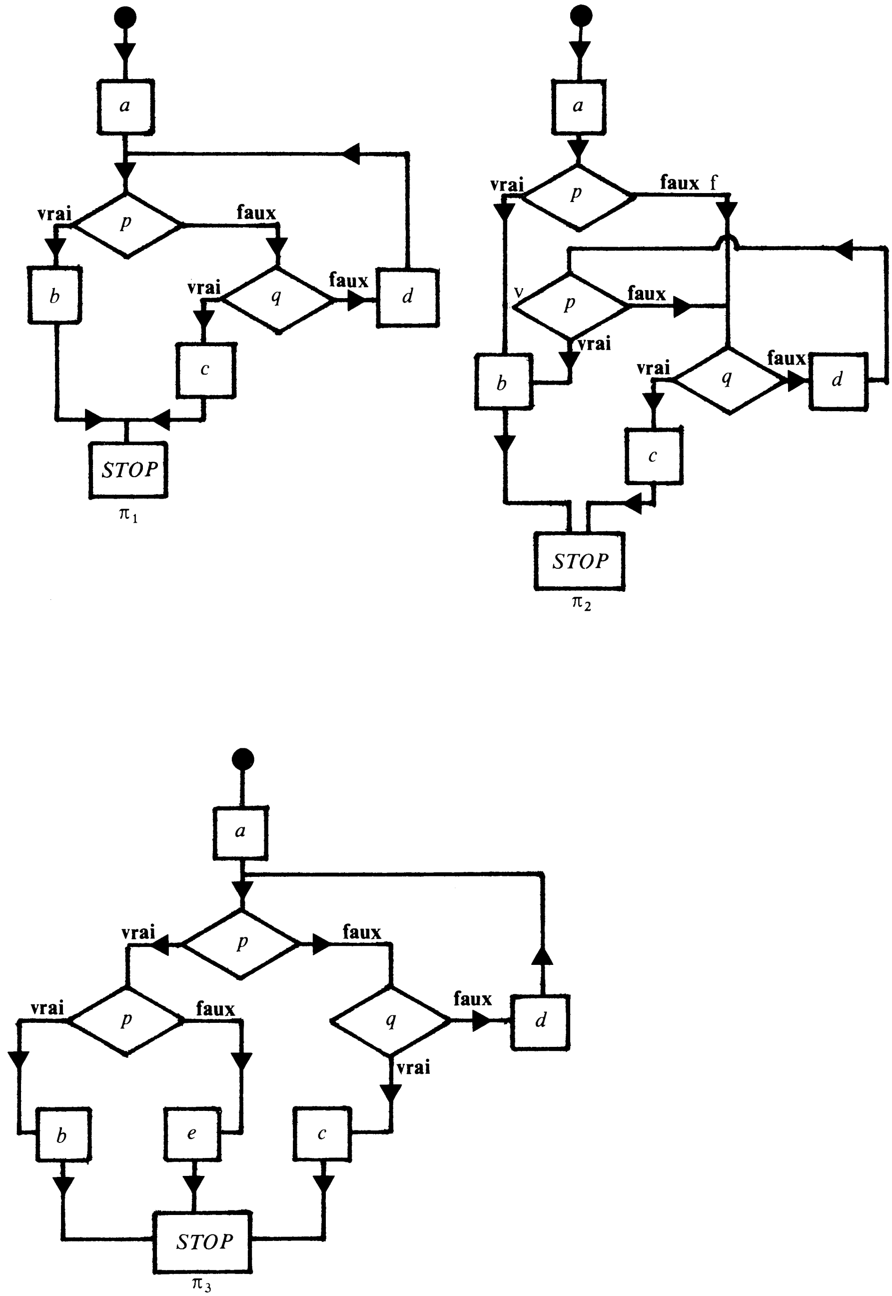


Figure 13.

A partir des relations R1, R2, R3, R4 définies entre schémas, nous pouvons définir des relations entre structures. Les plus couramment étudiées sont les suivantes :

— *Conversion fonctionnelle* elle correspond à la relation R1 et est notée FN : $\mathcal{S}_1 \underset{\text{FN}}{\leqslant} \mathcal{S}_2$ signifie que, pour toute interprétation et tout schéma π_1 de $\mathcal{S}_1$ il existe un schéma π_2 de $\mathcal{S}_2$ tel que $I[\pi_1] = I[\pi_2]$; intuitivement, π_1 et π_2 définissent la même fonction.

— *Conversion sémantique* en prenant la relation SE = R1 **et** R2 : $\mathcal{S}_1 \underset{\text{SE}}{\leqslant} \mathcal{S}_2$ signifie que, pour toute interprétation et tout schéma π_1 de $\mathcal{S}_1$, il existe un schéma π_2 de $\mathcal{S}_2$ construit sans introduire de nouvelles actions et tel que $I[\pi_1] = I[\pi_2]$.

— *Conversion par calcul* en prenant la relation CA = R1 **et** R2 **et** R3 (notez que R1 **et** R3 = R3) :

$\mathcal{S}_1 \underset{\text{CA}}{\leqslant} \mathcal{S}_2$ signifie que, pour toute interprétation et tout schéma π_1 de $\mathcal{S}_1$, il existe un schéma π_2 de $\mathcal{S}_2$ construit sans introduire de nouvelles actions ou de nouveaux tests et effectuant pour toute donnée le même calcul que π_1.

— *Conversion stricte* en prenant la relation ST = R1 **et** R2 **et** R3 **et** R4.

3.5 Etude des $\mathcal{D}$-schémas

Au paragraphe précédent, nous avons constaté qu'à tout $\mathcal{D}$-schéma on peut associer un organigramme mais que la réciproque est fausse. En appelant $\mathcal{G}$ la structure générale des organigrammes (ou des schémas avec branchement déjà définis au § 2), nous avons donc le résultat :

$$[\mathcal{D} \underset{\text{ST}}{\leqslant} \mathcal{G}] .$$

En comparant $\mathcal{D}$ et $\mathcal{G}$, on peut même prouver le résultat plus fort suivant [KOS, 74] :

Proposition 2

$$[\mathcal{D} \underset{\text{SE}}{<} \mathcal{G}] . \qquad \square$$

Démonstration

A tout $\mathcal{D}$-schéma, on peut associer un organigramme d'où $\mathcal{D} \underset{\text{SE}}{\leqslant} \mathcal{G}$. Pour montrer **non** $\mathcal{G} \underset{\text{SE}}{\leqslant} \mathcal{D}$, il suffit de trouver un organigramme π et une interprétation de cet organigramme telle qu'aucun $\mathcal{D}$-schéma (comportant les mêmes actions et prédicats que π) ne calcule la même fonction pour cette interprétation. Choisissons tout d'abord π représenté par l'organigramme de la figure 14.

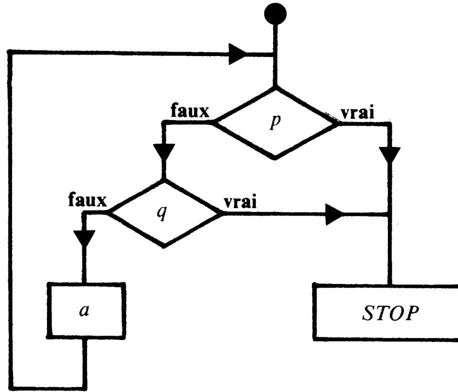


Figure 14 π.

Choisissons sur le domaine D = ℕ l'interprétation suivante ($x \in \mathbb{N}$) :

$$I[p](x) = (\exists n)(x = 10^n) ;$$
$$I[q](x) = (\exists n)(x = 10^n + 10^{n-1}) ;$$
$$I[a](x) = x + 1 .$$

On trouve immédiatement :

$$10^n < d \leqslant 10^n + 10^{n-1} \Rightarrow I[\pi](d) = 10^n + 10^{n-1} ;$$
$$10^n + 10^{n-1} < d \leqslant 10^{n+1} \Rightarrow I[\pi](d) = 10^{n+1} .$$

Supposons qu'il existe un $\mathscr{D}$-schéma π' sémantiquement équivalent à π (π' R1 **et** R2 π), π' contient k occurrences de a. Choisissons un entier m et une donnée $d = 10^m - (k + 1)$ de façon que $10^{m-1} + 10^{m-2} < d \leqslant 10^m$. (Il suffit de prendre $k + 1 < 10^{m-2}$.) Alors,

$$I[\pi'](d) = I[\pi](d) = 10^m.$$

Nécessairement, le calcul d'initialisation d pour π' contient k + 1 occurrences de l'action *a* ; comme *a* admet k occurrences dans π', on trouve dans π' au moins une instruction du type :

(1) *tant que* $\bar{p}$ *faire*... *a*... *fait*.

Considérons la première occurrence d'une instruction de ce type utilisée dans le calcul de d ; avant cette instruction, dans le calcul de d, il ne peut y avoir d'instruction itérative effectuée au moins une fois car une telle instruction serait de la forme :

tant que $\bar{q}$ *faire*... *a*... *fait*

et son « résultat intermédiaire » serait supérieur à 10^m, ce qui est impossible.

Avant le calcul de (1) on a donc au plus k actions *a*. Le calcul de π' d'initialisation $d' = 10^m + 10^{m-1} - (k + 1)$ a donc même début que le calcul

d'initialisation d ; après l'instruction (1), son résultat intermédiaire est 10^{m+1} qui est supérieur au résultat I[π] (d'), ce qui est impossible. C.Q.F.D.

Intuitivement, la proposition que nous venons de prouver montre que les $\mathscr{D}$-schémas sont « sémantiquement » moins puissants que les organigrammes. Le résultat suivant prouve cependant que les $\mathscr{D}$-schémas sont aussi puissants que les organigrammes sur le plan fonctionnel.

Théorème 10 (BÖHM et JACOPINI) [BÖH, 66] :

$$[\mathscr{D} \underset{\text{FN}}{\equiv} \mathscr{G}] . \qquad \square$$

Idée de la démonstration

Comme tout $\mathscr{D}$-schéma est associé à un organigramme, on a $\mathscr{D} \underset{\text{FN}}{\leqslant} \mathscr{G}$.

Il suffit donc de prouver $\mathscr{G} \underset{\text{FN}}{\leqslant} \mathscr{D}$, c'est-à-dire que pour tout organigramme σ, on peut construire un $\mathscr{D}$-schéma associé. Nous allons raisonner par induction sur la structure de l'organigramme. On peut décomposer un organigramme non vide en isolant sa première action ou son premier test : trois cas sont possibles qui sont représentés à la figure 15 : $\mathscr{A}$, $\mathscr{B}$, $\mathscr{C}$ désignent le reste de l'organigramme, les liaisons en traits gras notées L_1 ou L_2 représentent 0, un ou plusieurs arcs, et sont en général plus compliquées que celles autorisées sur les organigrammes associés aux $\mathscr{D}$-schémas.

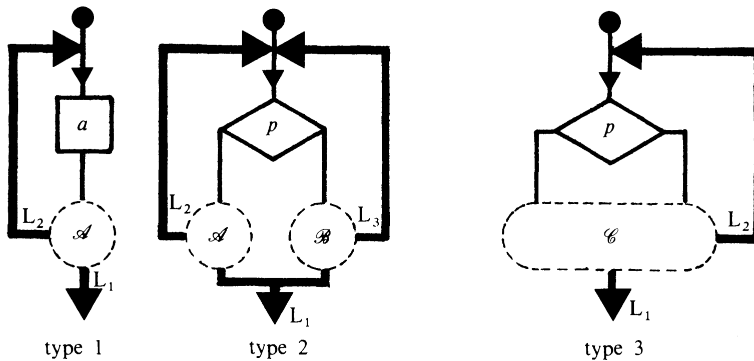


Figure 15 Décomposition d'un organigramme.
(Les liaisons en traits gras représentent zéro, un ou plusieurs arcs.)

Notons que les organigrammes du type 3 peuvent être ramenés à des organigrammes de type 2 en dupliquant éventuellement certaines parties. Nous nous limitons donc, dans la suite, à des organigrammes de type 1 ou 2. Intuitivement, pour construire un $\mathscr{D}$-schéma équivalent il faut « retenir » les chemins utilisés. Pour cela nous allons introduire une nouvelle donnée booléenne b et deux actions v et f qui seront interprétées respectivement comme

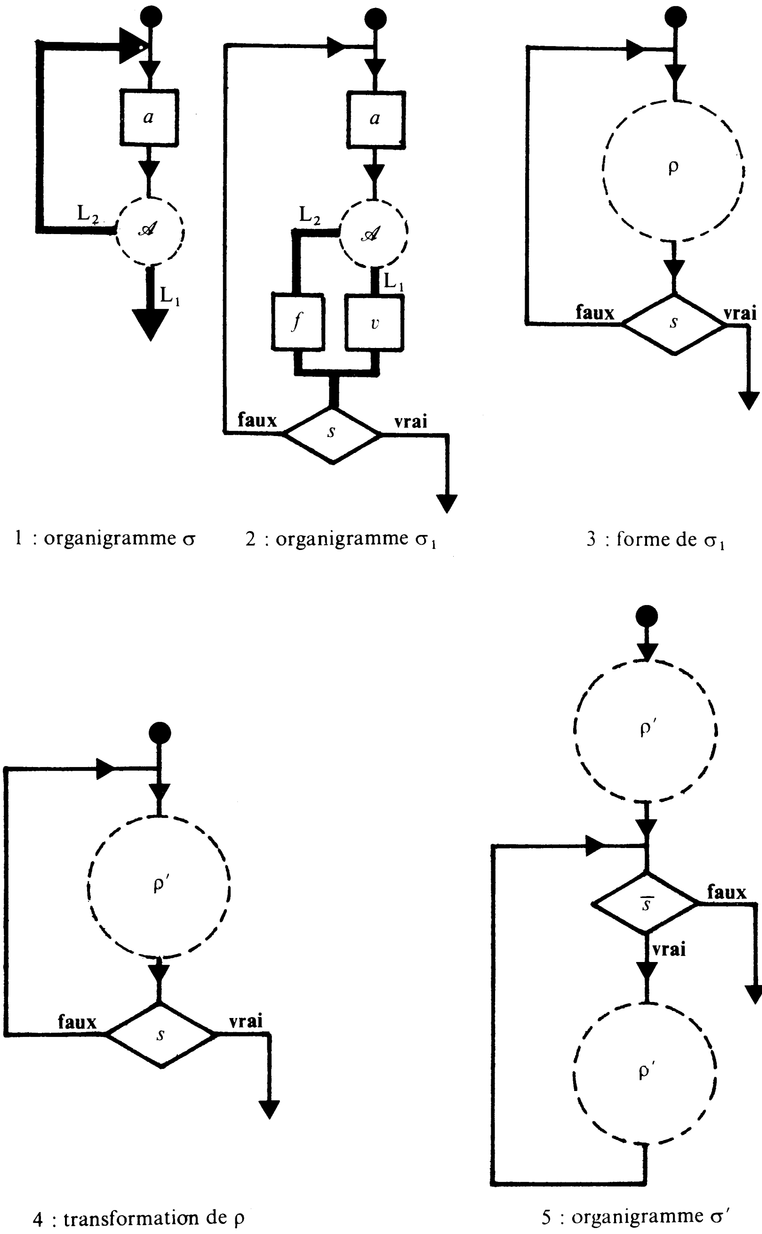


Figure 16 Etapes de la construction de σ' pour un organigramme σ de type 1.

les affectations $b := \mathbf{vrai}$ et $b := \mathbf{faux}$. Enfin un nouveau test s est adjoint à $\mathscr{P}$ et interprété par :

$$s = \mathbf{vrai} \Leftrightarrow b = \mathbf{vrai}\,.$$

Soit un organigramme quelconque σ, de type 1 ou 2 et comportant éventuellement des actions v et f et des tests s en plus des actions de $\mathscr{F}$ et des tests de $\mathscr{P}$, σ admet n occurrences d'éléments de $\mathscr{F} \cup \mathscr{P}$ et m occurrences d'éléments de $\{ v, f, s \}$. Prouvons par récurrence sur $3\,n + m$ qu'on peut construire un organigramme σ' associé à un $\mathscr{D}$-schéma et fonctionnellement équivalent à σ. Si $3\,n + m = 0$ (σ est vide) ou si σ est déjà associé à un $\mathscr{D}$-schéma $\sigma' = \sigma$ convient. Autrement σ se décompose en un organigramme de type 1 ou 2. Selon que σ est de type 1 ou 2 transformons-le en un nouvel organigramme σ_1 ou σ_2 comme cela est indiqué sur les figures 16 et 17 ; la transformation fait apparaître les organigrammes ρ, ρ_1, ρ_2 auxquels on peut appliquer l'hypothèse de récurrence et qu'on peut donc transformer en des organigrammes ρ', ρ'_1, ρ'_2 associés à des $\mathscr{D}$-schémas. Enfin une dernière transformation permet de replacer le test s en début d'itération et d'obtenir ainsi un organigramme σ' associé à un $\mathscr{D}$-schéma. C.Q.F.D.

Exemple 11

Transformons l'organigramme de la figure 12 ; appliquons seulement à la partie $\mathscr{C}$ (figure 18) la transformation proposée dans la démonstration ; l'organigramme obtenu σ' s'écrit encore sous forme d'un $\mathscr{D}$-schéma :

$$a\,;\,\pi\,;\ \underline{\textit{tant que}}\ \bar{s}\ \underline{\textit{faire}}\ \pi\ \underline{\textit{fait}}$$

où π est le $\mathscr{D}$-schéma suivant

$$\underline{\textit{si}}\ p\ \underline{\textit{alors}}\ b\,;\,v$$
$$\underline{\textit{sinon}}\ \underline{\textit{si}}\ q\ \underline{\textit{alors}}\ c\,;\,v\ \underline{\textit{sinon}}\ d\,;\,f\ \underline{\textit{fsi}}\ \underline{\textit{fsi}}\,.$$

□

3.6 Autres structures

Dans les $\mathscr{D}$-schémas, on ne peut quitter une itération que par un point de sortie unique : d'autres structures ne présentent pas cette contrainte, nous allons en étudier sommairement quelques-unes.

a) Les BJ_n-programmes

BÖHM et JACOPINI [BOH, 66] introduisent essentiellement à titre d'exemple contraire de $\mathscr{D}$-schémas la construction de la figure 19 ; plus généralement, les $\mathrm{BJ_n}$-schémas sont construits (figure 20) sur

$$\mathscr{V} = \mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}} \cup \{\ ;,\ \underline{\textit{si}},\ \underline{\textit{sinon}},\ \underline{\textit{alors}},\ \underline{\textit{fsi}},\ \underline{\textit{faire}},\ \underline{\textit{fait}},\ \rightarrow\ \}$$

avec la composition séquentielle $\alpha\,;\,\beta$, la composition conditionnelle $\underline{\textit{si}}\ p\ \underline{\textit{alors}}\ \alpha\ \underline{\textit{sinon}}\ \beta\ \underline{\textit{fsi}}$ et un nouveau type de composition dit **k-itération** qui s'écrit

$$\underline{\textit{faire}}\ p_1 \rightarrow \alpha_1\,;\,p_2 \rightarrow \alpha_2\,;\,\ldots\,;\,p_k \rightarrow \alpha_k\ \underline{\textit{fait}}$$

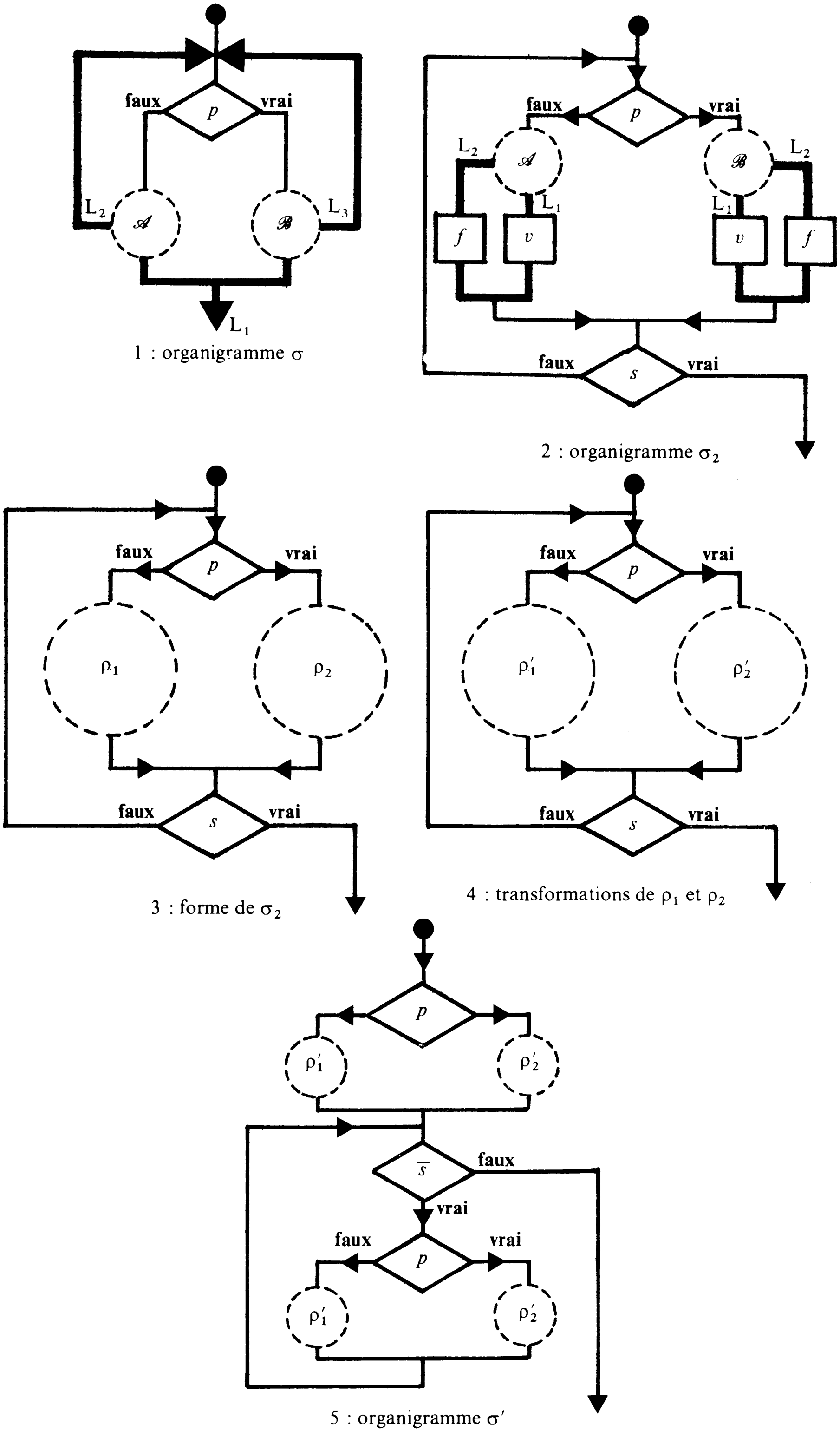
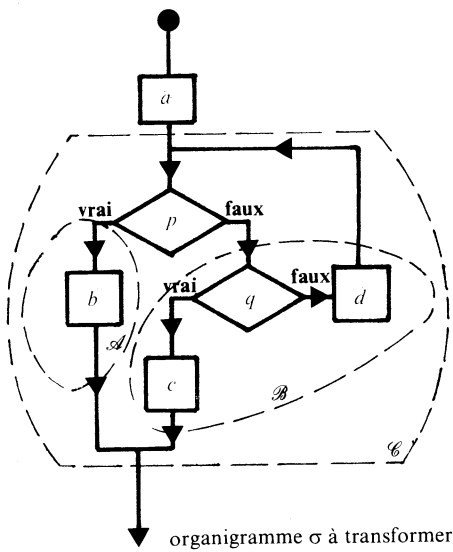
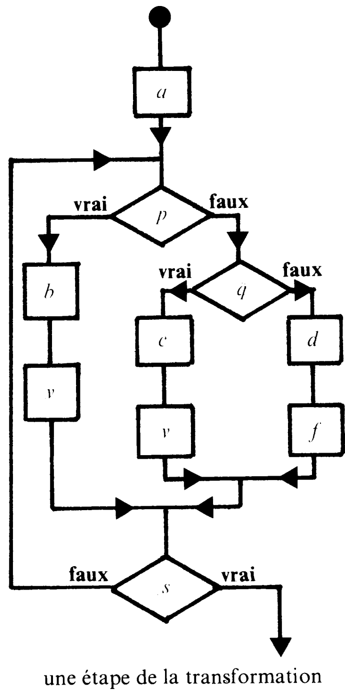


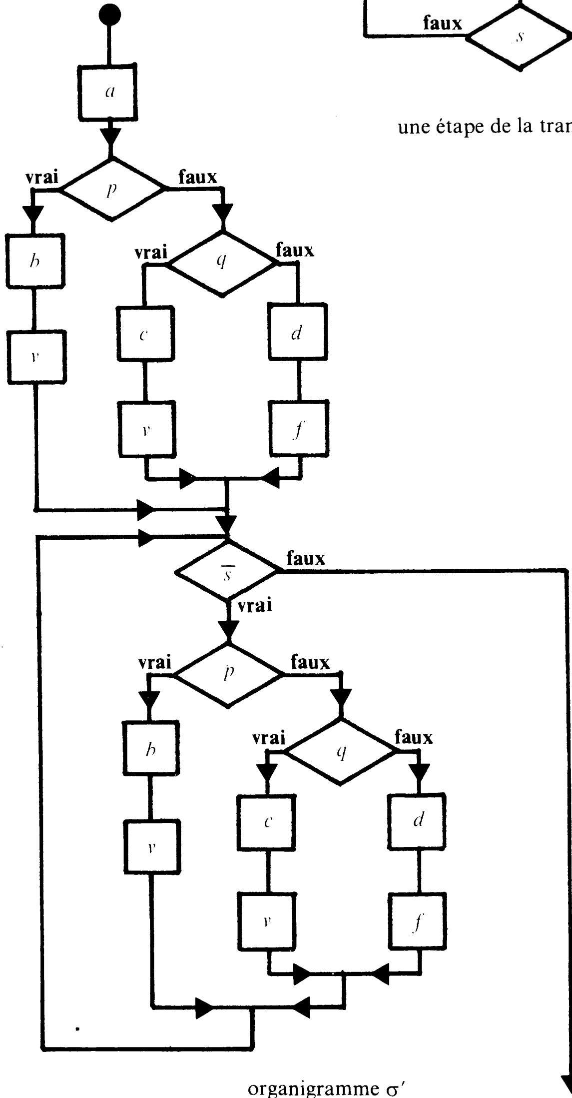
Figure 17 Etapes de la construction de σ' pour un organigramme de type 2.



organigramme σ à transformer



une étape de la transformation



organigramme σ′

Figure 18.

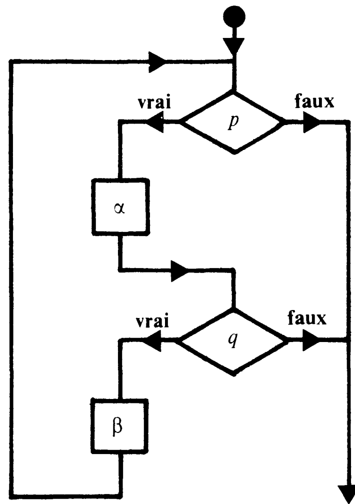


Figure 19 *faire* $p \to \alpha\,;\, q \to \beta$ *fait*.

avec $1 \leqslant k \leqslant n$. Intuitivement, cette construction signifie que, si p_1 est vrai, on effectue α_1 puis, si p_2 est vrai, on effectue α_2, ..., après α_k, on revient en p_1. L'itération est arrêtée dès qu'un p_i est faux.

Syntaxiquement, le langage BJ des BJ_n-schémas est solution de :

$$\text{BJ} = \mathscr{F} \cup \text{BJ}\,;\,\text{BJ} \cup \underline{si}\,(\mathscr{P} \cup \overline{\mathscr{P}})\,\underline{alors}\ \text{BJ}\ \underline{sinon}\ \text{BJ}\ \underline{fsi}$$
$$\cup \bigcup_{1 \leqslant k \leqslant n} \underline{faire}\,[\mathscr{P} \to \text{BJ}\,;]^{k-1}\,\mathscr{P} \to \text{BJ}\,\underline{fait}$$

Exercice 24

Montrer que $\mathscr{D} \underset{CA}{\equiv} BJ_1$.

Montrer que pour $n \geqslant 2$, il existe des BJ_n-organigrammes qui ne sont pas des $\mathscr{D}$-organigrammes. □

Exercice 25

Définir une interprétation de BJ_2-schémas.
Définir les calculs et résultats d'un BJ_2-schéma. □

b) Schémas répétitifs avec exits et entrées

Dans les structures précédentes, les possibilités de sauts sont très limitées : il s'agit soit de sauts à une fin de l'itération en cours, soit de retour en début d'itération mais après la fin de l'étape précédente. Les structures que nous définissons maintenant évitent partiellement ces contraintes :

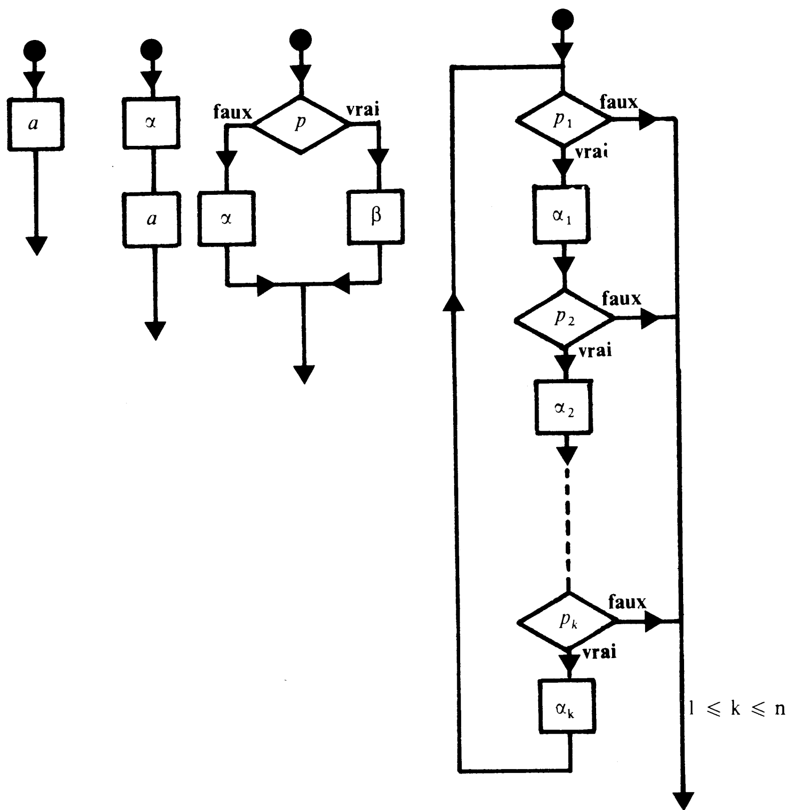


Figure 20 BJ_n-schémas.

RE_n-*schémas*

Lorsque plusieurs itérations sont imbriquées, on voudrait pouvoir sortir directement des i itérations englobantes : on introduit un nouveau type d'instruction *exit*(i) ($1 \leqslant i \leqslant n$). Supposons que *exit*(i) soit placée dans une itération qui admet k itérations englobantes ; pour $i \leqslant k$ *exit*(i) correspond à une rupture de séquence et passage à la fin de la i-ième itération englobante ; pour $k < i$ *exit*(i) correspond à une rupture de séquence à la fin de l'itération englobante la plus externe (figure 21).

En plus des compositions sérielle et conditionnelle on trouve, dans les RE_n-schémas, des répétitions : *faire* α *fait* où α comporte éventuellement des instructions *exit*.

Syntaxiquement, le langage RE des RE_n-schémas est défini sur $\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}} \cup \{$ *si*, *alors*, *sinon*, *fsi*, *faire*, *fait*, ; $\}$ où $\mathscr{F}$ contient des actions et les instructions *exit*(1), ..., *exit*(*n*).

$$RE = \mathscr{F} \cup RE \; ; RE \cup \underline{si}(\mathscr{P} \cup \overline{\mathscr{P}}) \; \underline{alors} \; RE \; \underline{sinon} \; RE \; \underline{fsi} \cup \underline{faire} \; RE \; \underline{fait} \; .$$

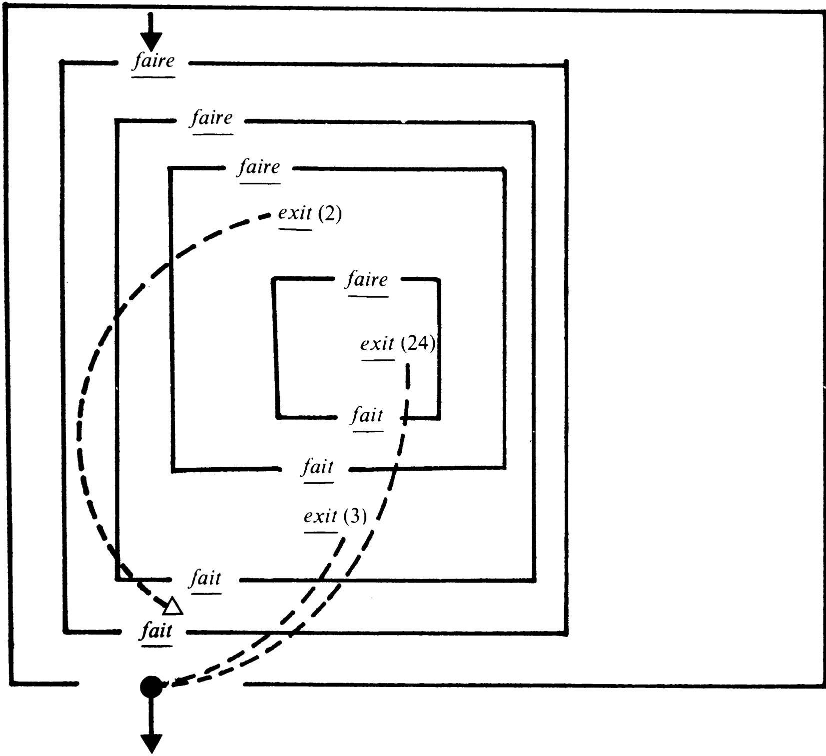


Figure 21 Instructions *exit*.

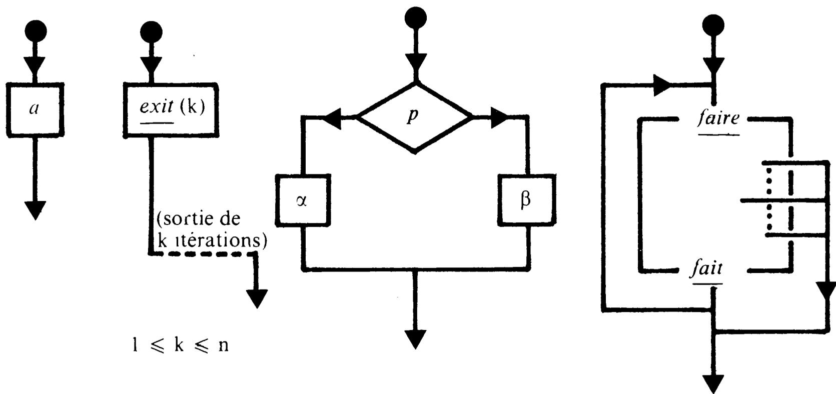


Figure 22 RE_n-schémas.

Exercice 26

Montrer que $BJ_n \underset{CA}{\leqslant} RE_1$. □

REC_n-*schémas*

L'instruction exit permet des sauts en fin d'itération ; on peut lui ajouter une instruction entrée qui réalise un saut en début d'itération. De même que pour exit, on précise entrée(*i*) pour indiquer un retour en début de la i-ième itération englobante (ou à défaut, au début de l'itération englobante la plus externe). Les procédés de composition sont les mêmes pour les RE_n et les REC_n-schémas (figure 23).

Syntaxiquement, le langage REC des REC_n-schémas est défini sur $\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}} \cup$ { si, alors, sinon, fsi, faire, fait, ; } où $\mathscr{F}$ contient des actions et l'ensemble { exit(1), ..., exit(n), entrée(1), ..., entrée(n) }.

REC = $\mathscr{F}$ ∪ REC ; REC ∪ si($\mathscr{P} \cup \overline{\mathscr{P}}$) alors REC sinon REC fsi ∪ faire REC fait .

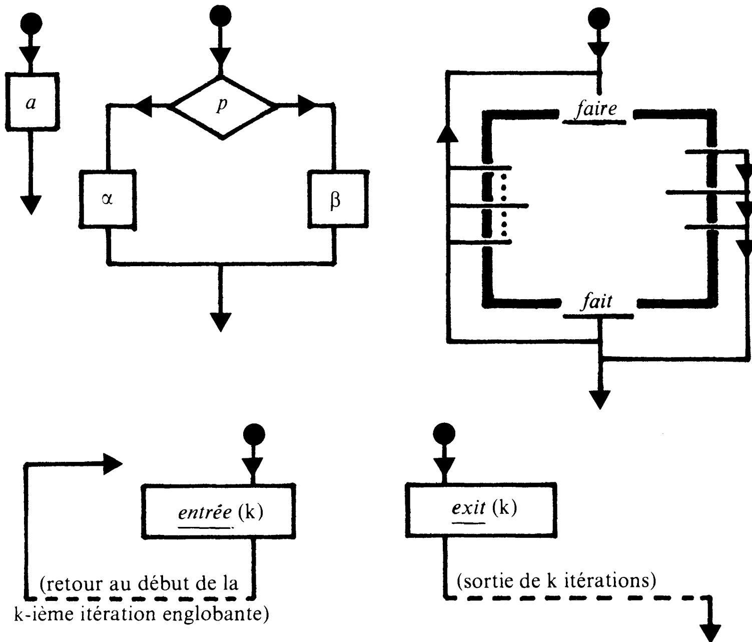


Figure 23 REC_n-schémas.

Variantes

Dans les RE_n, et REC_n-schémas, on ne retrouve pas la construction tant que *p* faire *α* fait des $\mathscr{D}$-schémas. On appelle DRE_n (resp. $DREC_n$) schémas l'ensemble des schémas construits en ajoutant ce mode de composition. Les intructions exit(i) et entrée(i) sont définies de même façon que précédemment par rapport aux itérations faire... fait et ne portent pas sur les constructions tant que *p* faire *α* fait.

3.7 Résultats généraux et conclusion

La grande variété des structures que nous venons de définir invite à rechercher les comparaisons possibles entre ces structures. Les principaux résultats obtenus dans ce domaine sont résumés dans une classification prouvée par RAO-KOSARAJU [RAO, 74] et schématisée par la figure 24 où $<$, $\leqslant$, $\equiv$ sont mis pour $\underset{SE}{<}$, $\underset{SE}{\leqslant}$, $\underset{SE}{\equiv}$.

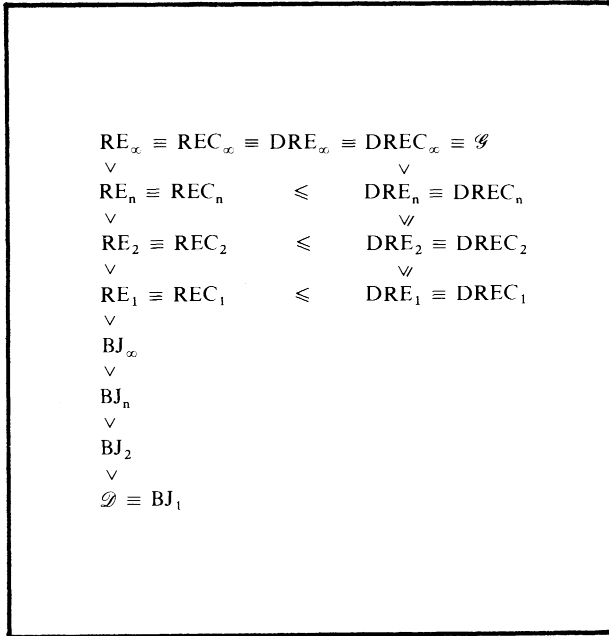


Figure 24 Comparaison sémantique des structures $\mathcal{S}_\infty = \bigcup_{n \geqslant 0} \mathcal{S}_n$.

Soit $\mathcal{S}$ une structure quelconque parmi celles que nous venons de définir : d'après le résultat de RAO-KOSARAJU nous avons :

$$\mathcal{D} \underset{SE}{\leqslant} \mathcal{S} \underset{SE}{\leqslant} \mathcal{G}$$

D'où, a fortiori :

$$\mathcal{D} \underset{FN}{\leqslant} \mathcal{S} \underset{FN}{\leqslant} \mathcal{G}$$

Comme $\mathcal{G}$ est fonctionnellement équivalent à $\mathcal{D}$ (théorème de BÖHM et JACOPINI), on en déduit :

$$\mathcal{S} \underset{FN}{\equiv} \mathcal{G} .$$

Ce résultat peut s'énoncer :

Théorème 11

Toutes les structures présentées dans ce paragraphe sont fonctionnellement équivalentes. □

4 SCHÉMAS RÉCURSIFS

4.0 Introduction

Au paragraphe 2 du chapitre 2, nous avons défini un énoncé récursif de problème comme un ensemble de définitions du type :

$$\varphi_i(x_1, \ldots, x_n) \underset{\text{def}}{=} \varepsilon_i$$

dans lesquelles $\varepsilon_1, \ldots, \varepsilon_m$ représentent les expressions contenant éventuellement les variables $x_1, \ldots, x_n$ et les inconnues $\varphi_1, \ldots, \varphi_m$.

Les schémas récursifs de programmes, objets de ce paragraphe, formalisent le concept de déclarations récursives de fonctions et celui de calcul d'une expression contenant des occurrences de telles fonctions. Nous n'essayons pas de décrire formellement les procédures récursives [GRE, 75].

Un schéma récursif est donc la donnée d'une **partie déclaration** notée $\varphi(x) \Leftarrow \varepsilon$ et d'une expression τ contenant éventuellement des occurrences de φ. Comme les autres schémas de programme présentés aux paragraphes 2 et 3, il s'agit d'un objet syntaxique particulièrement commode pour effectuer des manipulations textuelles comme la substitution. Un schéma peut être interprété à condition de se donner un domaine D et une interprétation I_0 des symboles de base. Une déclaration récursive $\varphi(x) \Leftarrow \varepsilon$ définit alors une fonction $I^\varepsilon[\varphi(x)]$ encore notée Φ par abus de langage. Plus généralement toute expression τ représente une fonction $I^\varepsilon[\tau]$.

Au paragraphe 4.3 nous définissons $I^\varepsilon[\varphi(x)]$ et plus généralement $I^\varepsilon[\tau]$ en procédant par remplacement successif de $\varphi(x)$ par ε (cette démarche n'est autre que celle utilisée dans les langages de programmation pour évaluer une fonction définie récursivement). Il se peut que le procédé ne se termine pas toujours et Φ peut rester indéfinie en certains points. Aussi l'application I^ε est-elle définie comme la plus petite application vérifiant plusieurs conditions, comme par exemple :

$$I^\varepsilon[\varphi(x)] = I^\varepsilon[\varepsilon]$$

ou, plus généralement

$$I^\varepsilon[\tau] = I^\varepsilon[\tau(\varepsilon/\varphi(x))] \, .$$

Au paragraphe 4.3, nous adoptons une démarche différente en commençant par associer à chaque terme τ une fonctionnelle $I[\![\tau]\!]$. Sommairement,

$I[\![\tau]\!](f)$ peut être obtenue à partir de τ en remplaçant le symbole φ par la fonction f et en interprétant les différents symboles de base. En particulier, on associe au corps ε de la déclaration récursive la fonctionnelle $I[\![\varepsilon]\!]$. La fonction Φ n'est autre que le plus petit point fixe de $I[\![\varepsilon]\!]$. Plus généralement, nous montrons que

$$I^{\varepsilon}[\tau] = I[\![\tau]\!](\Phi) .$$

Au paragraphe 4.4, nous développons la première approche en présentant le concept de règle de calcul et en particulier les règles d'appel par valeur et par nom. Au paragraphe 4.5, nous comparons cette approche calculatoire et les approches fonctionnelles des paragraphes précédents, obtenant des résultats différents suivant que la fonction inconnue a un ou plusieurs arguments.

Dans les derniers paragraphes, nous comparons les règles d'appel à la fois du point de vue de leur puissance et du point de vue de la longueur des calculs qu'elles définissent.

4.1 Définitions

Donnons d'abord deux exemples de déclarations récursives :

$$D_1 : \varphi(x) \Leftarrow \underline{si}\ p(x)\ \underline{alors}\ a\ \underline{sinon}\ h(x, \varphi(g(x)))\ \underline{fsi}$$
$$D_2 : \psi(x) \Leftarrow \underline{si}\ p(x)\ \underline{alors}\ g(x)\ \underline{sinon}\ \psi(\psi(h(x)))\ \underline{fsi}$$

Dans le premier exemple, on voit qu'une déclaration récursive se compose d'une partie gauche telle que $\varphi(x)$ et d'une partie droite qui est un **terme** construit à partir de x, des symboles fonctionnels g, h, du symbole propositionnel p et de l'inconnue φ.

Définition 1

a) L'alphabet utilisé pour écrire un schéma récursif se décompose en :
— un ensemble $\mathcal{X} = \{ x_1, \ldots, x_n \}$ de variables,
— un ensemble $\mathcal{F} = \{ f, g, \ldots \}$ de symboles fonctionnels,
— un ensemble $\mathcal{P} = \{ p, q, \ldots \}$ de symboles propositionnels,
— un ensemble $\mathcal{F}' = \{ \varphi_1, \ldots, \varphi_k \}$ d'inconnues (fonctionnelles),

ensembles, auxquels il convient d'ajouter les symboles auxiliaires $\Leftarrow$, $\underline{si}$, $\underline{alors}$, $\underline{sinon}$, $\underline{fsi}$. Chaque symbole ψ de $\mathcal{F} \cup \mathcal{F}' \cup \mathcal{P}$ est affecté d'une arité $\mathbf{ar}(\psi)$. En particulier, l'arité de chaque symbole de $\mathcal{F}'$ est égale à n (où n désigne le nombre de variables de l'alphabet).

b) **L'ensemble des termes** est le langage $\mathcal{T}$ défini par le système d'équations :

$$\begin{cases} \mathcal{T} = \mathcal{X} \cup \mathcal{S} \cup \mathcal{C} \\ \mathcal{S} = \bigcup_{\psi \in \mathcal{F} \cup \mathcal{F}'} \psi \mathcal{T}^{\mathbf{ar}(\psi)} \\ \mathcal{C} = \bigcup_{p \in \mathcal{P}} \underline{si}\ p \mathcal{T}^{\mathbf{ar}(p)}\ \underline{alors}\ \mathcal{T}\ \underline{sinon}\ \mathcal{T}\ \underline{fsi} . \end{cases}$$

On y ajoute des parenthèses et des virgules pour faciliter la lecture.

c) **Un schéma récursif** est un couple π formé d'un ensemble de déclarations $\{ \varphi_i(x_1, ..., x_n) \Leftarrow \varepsilon_i \mid 1 \leqslant i \leqslant k \}$ et d'un terme τ. □
Intuitivement τ est le programme principal. On obtient par exemple un schéma récursif en associant à la déclaration D_1 le terme $\varphi(\varphi(x))$.
Dans la suite, on donnera les définitions et les résultats dans le cas des schémas récursifs à une seule déclaration, du type :

$$\varphi(x_1, ..., x_n) \Leftarrow \varepsilon ,$$

la généralisation aux schémas à plusieurs déclarations étant possible [NIV, 78].

4.2 Interprétation d'un schéma récursif

On interprète un schéma récursif π en se donnant un domaine D et une interprétation I_0 des symboles de $\mathscr{F} \cup \mathscr{P}$. Le couple $\langle \pi, I_0 \rangle$ est **un programme récursif.**

Exemple 12

a) On obtient un programme récursif en associant à la déclaration D_1 le terme $\varphi(x)$ et l'interprétation I_0 sur les entiers définie par $I_0[p]\,(x)=(x=0)$, $I_0[a] = 1$, $I_0[g]\,(x) = x - 1$ et $I_0[h]\,(x, y) = x * y$ (produit). Ce programme peut s'écrire, par abus de notation, sous la forme :

$$\varphi(x) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ 1\ \underline{sinon}\ x * \varphi(x - 1)\,\underline{fsi}$$

et définit la fonction factorielle.

b) Considérons l'interprétation I'_0 sur les entiers, égale à I_0 pour p et g et telle que $I'_0[a] = 0$ et $I'_0[h]\,(x, y) = x + y$. On obtient cette fois le programme récursif $\langle \pi_1, I'_0 \rangle$ suivant :

$$\varphi(x) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ 0\ \underline{sinon}\ x + \varphi(x - 1)\,\underline{fsi}$$

qui calcule la somme des x premiers entiers.

c) Interprétation associée à D_2 : pour l'interprétation I''_0 sur les entiers, définie par $I''_0[p]\,(x) = (x > 100)$, $I''_0[g]\,(x) = x - 10$ et $I''_0[h]\,(x) = x + 11$, on obtient le programme récursif $\langle \pi_2, I''_0 \rangle$:

$$\psi(x) \Leftarrow \underline{si}\ x > 100\ \underline{alors}\ x - 10\ \underline{sinon}\ \psi(\psi(x + 11))\,\underline{fsi}\,. \qquad \square$$

Lorsqu'on cherche à calculer 2 ! en utilisant le programme récursif $\langle \pi_1, I_0 \rangle$, donné en a) de l'exemple précédent, on commence par se ramener au calcul du terme $\underline{si}\ x = 0\ \underline{alors}\ 1\ \underline{sinon}\ x * \varphi(x - 1)\,\underline{fsi}$ pour $x = 2$. On continue en calculant le test $x = 0$ et le terme $x * \varphi(x - 1)$, etc.. Nous sommes ainsi conduits à interpréter chaque terme τ par une fonction $I^{\varepsilon}[\tau]$ de D^n dans D. Donnons-en une définition précise.

Définition 2

Soit $\varphi(x) \Leftarrow \varepsilon$ une déclaration récursive et soit I_0 une interprétation des symboles de $\mathscr{F} \cup \mathscr{P}$. I_0 se prolonge en une interprétation I des schémas

récursifs associés à la déclaration. Pour tout terme τ, on définit une fonction de D^n dans D', notée $I[\varphi(x) \Leftarrow \varepsilon, \tau]$, ou encore $I^\varepsilon[\tau]$. I^ε est l'application la moins définie vérifiant les conditions suivantes :

a) si $\tau = x_i$, $I^\varepsilon[x_i]\,(a) = a_i$;

b) si $\tau = g(\tau_1, ..., \tau_m)$, $I^\varepsilon[g(\tau_1, ..., \tau_m)]\,(a) = I_0[g]\,(I^\varepsilon[\tau_1]\,(a), ..., I^\varepsilon[\tau_m]\,(a))$;

c) si $\tau = \varphi(\tau_1, ..., \tau_n)$,

$$I^\varepsilon[\varphi(\tau_1, ..., \tau_n)]\,(a) = I^\varepsilon[\varepsilon]\,(I^\varepsilon[\tau_1]\,(a), ..., I^\varepsilon[\tau_n]\,(a))\ ;$$

d) si $\tau = \underline{si}\ \beta\ \underline{alors}\ \tau_1\ \underline{sinon}\ \tau_2\ \underline{fsi}$, avec $\beta = p\tau_1, ..., \tau_m$;

$I^\varepsilon[\underline{si}\ \beta\ \underline{alors}\ \tau\ \underline{sinon}\ \tau'\ \underline{fsi}]\,(a) =$ SI $I^\varepsilon[\beta]\,(a)$ ALORS $I^\varepsilon[\tau]\,(a)$ SINON $I^\varepsilon[\tau']\,(a)$

où l'on définit $I^\varepsilon[p(\tau_1, ..., \tau_m)]$ par :

e) $I^\varepsilon[p(\tau_1, ..., \tau_m)]\,(a) = I_0[p]\,(I^\varepsilon[\tau_1]\,(a), ..., I^\varepsilon[\tau_m]\,(a))$.

On appellera par la suite **fonction définie par la déclaration récursive** $\varphi(x) \Leftarrow \varepsilon$ l'interprétation $I^\varepsilon[\varphi(x)]$. On notera cette fonction Φ. □

Remarques

1) Dans la définition 2, il est entendu que, si G, $F_1, ..., F_m$ sont des fonctions, $G(F_1, ..., F_m)$ est définie en un point a si et seulement si $F_1, ..., F_m$ sont définies en a et si G est définie au point $(F_1(a), ..., F_m(a))$. Ceci est essentiel pour obtenir une définition correcte de I^ε. Dans la terminologie introduite au paragraphe 4.5, G est une fonction **stricte.**

2) L'interprétation I^ε existe effectivement. On peut montrer que toute application de $\mathcal{T}$ dans $D^n \to D$ vérifiant les conditions a) à e) est point fixe d'une certaine fonctionnelle Σ. On prouve que Σ vérifie les hypothèses du théorème sur l'existence d'un plus petit point fixe, ce qui assure l'existence de I^ε.

Exemple 13

Calculons $I^\varepsilon[\varphi(x)]\,(2)$ pour le programme récursif définissant factorielle (exemple 12) :

$I^\varepsilon[\varphi(x)]\,(2) = I^\varepsilon[\underline{si}\ p(x)\ \underline{alors}\ a\ \underline{sinon}\ h(x, \varphi(g(x)))\ \underline{fsi}]\,(2)$
d'après c) et a) ;

$I^\varepsilon[\varphi(x)]\,(2) =$ SI $2 = 0$ ALORS 1 SINON $I^\varepsilon[h(x, \varphi(g(x)))]\,(2)$
d'après e), a), d), b) ;

$I^\varepsilon[\varphi(x)]\,(2) = I^\varepsilon[h(x, \varphi(g(x)))]\,(2)$
par définition de SI... ALORS... SINON... ;

$I^\varepsilon[\varphi(x)]\,(2) = I_0[h]\,(I^\varepsilon[x]\,(2), I^\varepsilon[\varphi(g(x))]\,(2))$
d'après b) ;

$I^\varepsilon[\varphi(x)]\,(2) = 2 * I^\varepsilon[\varphi(g(x))]\,(2)$
d'après a) et la définition de $I_0[h]$;

$I^{\varepsilon}[\varphi(x)]\,(2) = 2 * I^{\varepsilon}[\underline{si}\; p(x)\; \underline{alors}\; a\; \underline{sinon}\; h(x, \varphi(g(x)))\; \underline{fsi}]\,(I^{\varepsilon}[g(x)]\,(2))$
d'après c) ;

$I^{\varepsilon}[\varphi(x)]\,(2) = 2 * I^{\varepsilon}[\underline{si}\; p(x)\; \underline{alors}\; a\; sinon\; h(x, \varphi(g(x)))\; \underline{fsi}]\,(1)$
d'après b) et la définition de $I_0[g]$.

En abrégeant les calculs on obtient ensuite :

$$\begin{aligned} I^{\varepsilon}[\varphi(x)]\,(2) &= 2 * 1 * I^{\varepsilon}[\varphi(g(x))]\,(1) \\ &= 2 * I^{\varepsilon}[\underline{si}\; p(x)\; \underline{alors}\; a\; \underline{sinon}\; h(x, \varphi(g(x)))\; \underline{fsi}]\,(0) \\ &= 2 * (\text{SI } 0 = 0 \text{ ALORS } a \text{ SINON } I^{\varepsilon}[h(x, \varphi(g(x)))])\,(0) \\ &= 2 * 1 = 2\,. \end{aligned}$$ □

Exercice 27

Pour le programme récursif de l'exemple 12, montrer à l'aide de la définition 2 que :

(1) $I^{\varepsilon}[\varphi(x)]\,(0) = 1$

(2) pour $n > 0$, $I^{\varepsilon}[\varphi(x)]\,(n) = I^{\varepsilon}[h(x, \varphi(g(x)))]\,(n)$

(3) pour $n > 0$, $I^{\varepsilon}[h(x, \varphi(g(x)))]\,(n) = n * I^{\varepsilon}[\varphi(x)]\,(n-1)$.

En déduire que $I^{\varepsilon}[\varphi(x)]\,(n) = n\,!$ (pour $n \geqslant 0$). □

4.3 Fonctionnelle associée à un terme

Après avoir défini une interprétation des schémas récursifs par des fonctions, nous pouvons associer à chaque terme τ une fonctionnelle $I[\![\tau]\!]$. On transforme ainsi une déclaration récursive du type $\varphi(x) \Leftarrow \varepsilon$ en une équation fonctionnelle $F = I[\![\varepsilon]\!]\,(F)$. Cette équation admet un plus petit point fixe qui coïncide avec la fonction $I^{\varepsilon}[\varphi(x)]$ définie par la déclaration récursive. Reprenons d'abord l'exemple 12 ; en remplaçant φ par une fonction inconnue F et en interprétant chaque symbole de base, on obtient ici l'équation

$$F(x) = \text{SI } x = 0 \text{ ALORS } \bar{1} \text{ SINON } x * F(x-1)$$

dont le plus petit point fixe est la fonction factorielle.

Généralisons maintenant cette démarche.

Définition 3. *Fonctionnelle* $I[\![\tau]\!]$ *associée à un terme* τ.

$I[\![\tau]\!]$ est définie par récurrence sur la complexité de τ par les équations suivantes :

a′) $I[\![x_i]\!]\,(F) = \lambda a . a_i$ (i-ième projection).

b′) $I[\![g(\tau_1, \ldots, \tau_m)]\!]\,(F) = I_0[g]\,(I[\![\tau_1]\!]\,(F), \ldots, I[\![\tau_m]\!]\,(F))$.

c′) $I[\![\varphi(\tau_1, \ldots, \tau_n)]\!]\,(F) = F(I[\![\tau_1]\!]\,(F), \ldots, I[\![\tau_n]\!]\,(F))$.

d′) $I[\![\underline{si}\; \beta\; \underline{alors}\; \tau_1\; \underline{sinon}\; \tau_2\; \underline{fsi}]\!]\,(F)$

$$= \text{SI } I[\![\beta]\!]\,(F) \text{ ALORS } I[\![\tau_1]\!]\,(F) \text{ SINON } I[\![\tau_2]\!]\,(F)\,.$$

De plus, si p est un prédicat on définit $I[\![p(\tau_1, \ldots, \tau_m)]\!]$ par

e') $I[\![p(\tau_1, \ldots, \tau_m)]\!] (F) = I_0[p] (I[\![\tau_1]\!] (F), \ldots, I[\![\tau_m]\!] (F))$. □

Le résultat suivant affirme que $I[\![\tau]\!]$ est une fonctionnelle continue.

Proposition 3

Soit τ un terme sur l'alphabet $x \cup \mathscr{F} \cup \{ \varphi \} \cup \mathscr{P}$. L'application $I[\![\tau]\!]$ associée à τ est une fonctionnelle continue. □

Démonstration par récurrence sur la complexité de τ. Etudions simplement le cas où τ s'écrit $\tau = \varphi(\tau_1, \ldots, \tau_n)$ (les autres cas sont très similaires). D'après c') $I[\![\tau]\!] (F) = F(I[\![\tau_1]\!] (F), \ldots, I[\![\tau_n]\!] (F))$. Soit (f_k) une suite convergente de fonctions. Notons :

$$f_k^i = I[\![\tau_i]\!] (f_k) , \quad f^i = \sup_k f_k^i , \quad f = \sup_k f_k .$$

Il s'agit de montrer que

$$\sup_k I[\![\tau]\!] (f_k) = I[\![\tau]\!] \left(\sup_k f_k\right) ;$$

soit encore

$$\sup_k f_k(f_k^1, \ldots, f_k^n) = I[\![\tau]\!] (f) = f(I[\![\tau_1]\!] (f), \ldots, I[\![\tau_n]\!] (f)) .$$

Par hypothèse de récurrence, $I[\![\tau_i]\!] (f) = \sup_k I[\![\tau_i]\!] (f_k) = \sup_k f_k^i = f^i$. Soient $x \in D^n$, $y \in D$ tels que $f(f^1(x), \ldots, f^n(x)) = y$. Pour $i = 1, \ldots, n$ il existe k_i tels que $k \geqslant k_i$ implique $f_k^i(x) = f^i(x)$. Soit $z = (f^1(x), \ldots, f^n(x))$. Il existe k_0 tel que $k \geqslant k_0$ implique $f_k(z) = f(z) = y$.

Finalement, pour $k \geqslant \mathrm{Max}\,(k_0, \ldots, k_n)$

$$y = f(z) = f_k(f_k^1(x), \ldots, f_k^n(x)) \quad \text{C.Q.F.D.}$$

Comme on l'a montré au chapitre 2, les fonctionnelles $I[\![\tau]\!]$ ainsi définies admettent un plus petit point fixe noté $\mu(I[\![\tau]\!])$. En particulier $I[\![\varepsilon]\!]$ admet un plus petit point fixe que l'on notera FIX. Le théorème suivant exprime en particulier que ce plus petit point fixe coïncide avec la fonction $\Phi = I^\varepsilon[\varphi(x)]$ définie par la déclaration récursive $\varphi(x) \Leftarrow \varepsilon$.

Théorème 12

Soient π le schéma récursif $(\varphi(x) \Leftarrow \varepsilon, \tau)$, I_0 une interprétation. Posons $\mathrm{FIX} = \mu(I[\![\varepsilon]\!])$ et $\Phi = I^\varepsilon[\varphi(x)]$. Alors

$$I[\pi] = I^\varepsilon[\tau] = I[\![\tau]\!] (\mathrm{FIX}) .$$

En particulier, si $\tau = \varphi(x)$,

$$\Phi = \mathrm{FIX} .$$

□

Prouvons d'abord le lemme suivant :

Lemme. Pour tout couple (ε, τ) de termes

$$I^{\varepsilon}[\tau] = I[\![\tau]\!]\,(I^{\varepsilon}[\varphi(x)])\,.$$

La démonstration se fait par récurrence sur τ. Le seul cas non trivial est celui où $\tau = \varphi(\tau_1, \ldots, \tau_n)$. Alors

$$I^{\varepsilon}[\varphi(\tau_1, \ldots, \tau_n)] = I^{\varepsilon}[\varepsilon]\,(I^{\varepsilon}[\tau_1], \ldots, I^{\varepsilon}[\tau_n])$$
d'après la définition 2, condition c) ;
$$= I^{\varepsilon}[\varphi(x)]\,(I^{\varepsilon}[\tau_1], \ldots, I^{\varepsilon}[\tau_n])$$
puisque $I^{\varepsilon}[\varphi(x)] = I^{\varepsilon}[\varepsilon]\,(I^{\varepsilon}[x_1], \ldots, I^{\varepsilon}[x_n]) = I^{\varepsilon}[\varepsilon]$;
$$= I^{\varepsilon}[\varphi(x)]\,(I[\![\tau_1]\!]\,(I^{\varepsilon}[\varphi(x)], \ldots, I[\![\tau_n]\!]\,(I^{\varepsilon}[\varphi(x)]))$$
par hypothèse de récurrence ;
$$= I[\![\varphi(\tau_1, \ldots, \tau_n)]\!]\,(I^{\varepsilon}[\varphi(x)]) = I[\![\tau]\!]\,(I^{\varepsilon}[\varphi(x)])$$
d'après la définition 3 condition c′) .

Démonstration du théorème

a) $I^{\varepsilon}[\tau] \sqsubseteq I[\![\tau]\!]$ (FIX)

Puisque l'application I^{ε} est la plus petite application vérifiant les conditions de la définition 2, considérons l'application J^{ε} définie par $J^{\varepsilon}[\tau] = I[\![\tau]\!]$ (FIX). Il suffit de montrer que J^{ε} vérifie les mêmes conditions. Examinons uniquement la condition c). Il faut prouver l'égalité :

$$J^{\varepsilon}[\varphi(\tau_1, \ldots, \tau_n)] = J^{\varepsilon}[\varepsilon]\,(J^{\varepsilon}[\tau_1], \ldots, J^{\varepsilon}[\tau_n])\,;$$

égalité qui s'écrit encore :

(1) $I[\![\varphi(\tau_1, \ldots, \tau_n)]\!]\,(\mathrm{FIX}) = I[\![\varepsilon]\!]\,(\mathrm{FIX})\,(I[\![\tau_1]\!]\,(\mathrm{FIX}), \ldots, I[\![\tau_n]\!]\,(\mathrm{FIX}))$;

La formule (1) est bien vérifiée puisque, d'après la condition c), de la définition 3,

$$I[\![\varphi(\tau_1, \ldots, \tau_n)]\!]\,(\mathrm{FIX}) = \mathrm{FIX}(I[\![\tau_1]\!]\,(\mathrm{FIX}), \ldots, I[\![\tau_n]\!]\,(\mathrm{FIX}))$$

et que $\mathrm{FIX} = I[\![\varepsilon]\!]\,(\mathrm{FIX})$.

b) $I[\![\tau]\!]\,(\mathrm{FIX}) \sqsubseteq I^{\varepsilon}[\tau]$

En utilisant le lemme, il suffit de démontrer que

(2) $$\mathrm{FIX} \sqsubseteq I^{\varepsilon}[\varphi(x)]\,.$$

En effet, $I^{\varepsilon}[\tau] = I[\![\tau]\!]\,(I^{\varepsilon}[\varphi(x)])$ d'après le lemme
$\sqsupseteq I[\![\tau]\!]\,(\mathrm{FIX})$ d'après (2) et la croissance de $I[\![\tau]\!]$.

Pour démontrer (2), il suffit de prouver que $I^{\varepsilon}[\varphi(x)]$ est un point fixe de $I[\![\varepsilon]\!]$.

Effectivement, $I^{\varepsilon}[\varphi(x)] = I^{\varepsilon}[\varepsilon]$ d'après la condition c)
$= I[\![\varepsilon]\!]\,(I^{\varepsilon}[\varphi(x)])$ d'après le lemme. □

4.4 Calculs et règles de calcul

Dans les deux paragraphes précédents, nous avons associé, de deux manières différentes, une fonction à un programme récursif. Il nous reste maintenant à étudier comment calculer cette fonction en un point $d \in D^n$.

Ajoutons à l'ensemble des constantes (c'est-à-dire des symboles de $\mathscr{F}$ d'arité 0) les éléments du domaine D. Pour chaque d de D^n, un calcul de $\varphi(d)$ est une suite de termes, ou **dérivation**, de premier élément $\varphi(d)$, telle que l'on passe d'un terme au suivant en appliquant certaines règles appelées **règles de dérivations.** Nous allons nous limiter, par la suite, à l'étude de deux types de règles : la **simplification** et la **substitution.**

Dans le reste du paragraphe, on suppose fixées une déclaration récursive $\varphi(x) \Leftarrow \varepsilon$ et une interprétation I_0. Les termes sans occurrence de variables seront notés par des minuscules latines.

Exemple 14

Considérons à nouveau la déclaration récursive interprétée étudiée au paragraphe 2 du chapitre 2

$$\varphi(x) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ 1\ \underline{sinon}\ x * \varphi(x-1)\ \underline{fsi}\ .$$

$t_0 = \varphi(2)$ est un terme simplifié qui se réécrit par substitution en $t_1 = \underline{si}\ 2 = 0\ \underline{alors}\ 1\ \underline{sinon}\ 2 * \varphi(2-1)\ \underline{fsi}$; t_1 peut être simplifié. En effet, $2 = 0$ se simplifie en $\underline{\textbf{faux}}$, $2 * \varphi(2-1)$ en $2 * \varphi(1)$ et

$$\underline{si}\ \underline{\textbf{faux}}\ \underline{alors}\ 1\ \underline{sinon}\ 2 * \varphi(1)\ \underline{fsi}$$

se simplifie en $t_2 = 2 * \varphi(1)$. t_2 lui-même se réécrit, par substitution puis simplification, en $2 * \varphi(0)$ puis en 2. Le calcul de $\varphi(2)$ est la suite de termes simplifiés $\varphi(2)$, $2 * \varphi(1)$, $2 * \varphi(0)$, 2.

Précisons ceci par les définitions suivantes.

a) Simplification

Définition 4

On dit que t se simplifie en t' (ou que $t' = \text{Simpl}(t)$) si toute sous-expression de t sans occurrence de φ a été évaluée et si toute sous-expression de t du type $\underline{si}\ b\ \underline{alors}\ t_1\ \underline{sinon}\ t_2\ \underline{fsi}$ où $b \in \{\ \underline{\textbf{vrai}}\ \ \underline{\textbf{faux}}\ \}$ a été réduite à t_1 ou à t_2. □

Une définition plus rigoureuse de la simplification est possible, mais sans intérêt ici [VUI, 74].

Par exemple :

$\text{Simpl}(\underline{si}\ 100 > 100\ \underline{alors}\ \varphi(100)\ \underline{sinon}\ \varphi(\varphi(100+11))\ \underline{fsi}) = \varphi(\varphi(111))$,

$\text{Simpl}(\underline{si}\ \varphi(5) > 0\ \underline{alors}\ 1\ \underline{sinon}\ 0\ \underline{fsi}) = \underline{si}\ \varphi(5) > 0\ \underline{alors}\ 1\ \underline{sinon}\ 0\ \underline{fsi}$.

b) Substitution

Définition 5

On dira que t **se réécrit en** t' **par substitution** si un certain nombre d'occurrences de φ dans t sont remplacées par ε. □

Les calculs associés à un programme récursif dépendent de façon déterminante du choix des occurrences de φ à substituer. Etudions d'abord un exemple.

Exemple 15

Considérons la déclaration récursive interprétée

$$\varphi(x) \Leftarrow \underline{si}\ x > 100\ \underline{alors}\ x - 10\ \underline{sinon}\ \varphi(\varphi(x + 11))\,\underline{fsi}$$

et le terme $t = \varphi(\varphi(100))$. On obtient trois termes distincts suivant la ou les occurrences de φ sélectionnées (et soulignées par la suite).

1) $t = \varphi(\underline{\varphi}(100))$ **(règle de l'appel par valeur** (V) : choix de l'occurrence de φ la plus à gauche parmi les plus internes)

$$t_V = \mathrm{Simpl}[\varphi(\underline{si}\ 100 > 100\ \underline{alors}\ 100 - 10\ \underline{sinon}\ \varphi(\varphi(100 + 11)))\,\underline{fsi})]$$
$$= \varphi(\varphi(\varphi(111)))\,.$$

2) $t = \underline{\varphi}(\varphi(100))$ **(règle de l'appel par nom** (N) : choix de l'occurrence de la plus à gauche parmi les plus externes)

$$t_N = \underline{si}\ \varphi(100) > 100\ \underline{alors}\ \varphi(100) - 10\ \underline{sinon}\ \varphi(\varphi(\varphi(100) + 11)))\,\underline{fsi}$$

(ici le terme obtenu ne se simplifie pas) .

3) $t = \underline{\varphi}(\underline{\varphi}(100))$ **(règle de substitution pleine** (P) : substitution de toutes les occurrences de φ)

$$t_P = \mathrm{Simpl}[\underline{si}\ t_1 > 100\ \underline{alors}\ t_1 - 10\ \underline{sinon}\ \varphi(\varphi(t_1 + 11))\,\underline{fsi}]$$
$$(\text{où } t_1 = \varepsilon[100] = \underline{si}\ 100 > 100\ \underline{alors}\ 100 - 10\ \underline{sinon}\ \varphi(\varphi(100 + 11))\,\underline{fsi})$$

On a encore, en simplifiant t_1 en $\varphi(\varphi(111))$,

$$t_P = \underline{si}\ \varphi(\varphi(111)) > 100\ \underline{alors}\ \varphi(\varphi(111)) - 10$$
$$\underline{sinon}\ \varphi(\varphi(\varphi(\varphi(111)) + 11))\,\underline{fsi}\,.$$

Cet exemple illustre le fonctionnement de la substitution. Précisons maintenant la notion de règle de calcul. □

Définition 6

Une règle de calcul R est fixée par une convention permettant de sélectionner dans tout terme simplifié une ou plusieurs occurrences sur lesquelles s'effectue la substitution de φ. Si t est un terme simplifié, notons le $t[\underline{\varphi}, \varphi]$ pour distinguer les occurrences de φ sélectionnées (notées $\underline{\varphi}$) ; le terme t' déduit de t par application de la règle R est :

$$t' = \mathrm{Simpl}(t[\varepsilon/\underline{\varphi}, \varphi])\ (*)$$

et on note $t \overset{R}{\Rightarrow} t'$.

(*) $t[\varepsilon/\underline{\varphi}, \varphi]$ désigne le terme obtenu en substituant ε à $\underline{\varphi}$.

Autrement dit, t' est déduit de t en effectuant les substitutions des occurrences de φ désignées par R, puis en simplifiant. □

Précisons à titre d'exemple les trois règles déjà présentées dans l'exemple 4.

Règle de calcul V (appel par valeur) : On sélectionne dans tout terme simplifié l'occurrence de φ la plus à gauche parmi les plus internes.

Par exemple sur le terme $\underline{si}\ \varphi(\varphi(111)) > 100\ \underline{alors}\ \varphi(\varphi(111)) - 10\ \underline{sinon}$
$\varphi(\varphi(\varphi(\varphi(111))+11))\ \underline{fsi}$,
on sélectionne la 2ᵉ occurrence de φ :

$$\underline{si}\ \varphi(\underline{\varphi}(111)) > 100\ldots$$

Règle de calcul N (appel par nom) : On sélectionne dans tout terme simplifié l'occurrence de φ la plus à gauche (donc l'une des plus externes).

Par exemple, sur le terme de l'exemple précédent, on trouve ici :

$$\underline{si}\ \underline{\varphi}\ (\varphi(111)) > 100\ \ldots$$

Règle de calcul P (règle de substitution pleine) : On sélectionne ici toutes les occurrences de φ.

Définissons maintenant les calculs d'un schéma récursif.

Définition 7

Soient $\pi = (\varphi(x) \Leftarrow \varepsilon, \tau)$ un schéma récursif, I_0 une interprétation, d un élément de D et R une règle de calcul. Le calcul de π pour le couple (I, d) et la règle R est la suite $t_0, t_1, \ldots$ caractérisée par les conditions suivantes :

1) $t_0 = r[d/x]$

et 2) pour tout i $\geqslant 0$,

a) ou bien $t_i \in D$ et t_i est le **résultat** du calcul,

b) ou bien $t_i \overset{R}{\Rightarrow} t_{i+1}$.

Dans le cas particulier où $\tau = \varphi(x)$, on parle encore de calcul de φ(d) pour la règle R. On définit la fonction Φ_R, calculée par application de la règle R en posant, pour tout d, $\Phi_R(d) = b$ si et seulement si le calcul de φ(d) s'arrête avec le résultat b.

Exercice 28

Reprenons la déclaration récursive interprétée :

$$\varphi(x) \Leftarrow \underline{si}\ x > 100\ \underline{alors}\ x - 10\ \underline{sinon}\ \varphi(\varphi(x + 11))\ \underline{fsi}$$

Déterminer, pour les règles d'appel par valeur, d'appel par nom et de substitution pleine, les calculs de $\varphi(100)$. □

4.5 Relations entre fonctions calculées et plus petit point fixe

Au terme de cette étude on peut se poser les questions suivantes. Quelles relations existe-t-il entre :

i) les fonctions calculées par application de diverses règles,
ii) la fonction Φ définie par une déclaration récursive,
iii) et le plus petit point fixe FIX de la fonctionnelle associée ?

Dans la suite, on sera amené à distinguer le cas où la fonction inconnue est monadique (cf. exemple 1) et celui où elle est polyadique.

a) Cas d'une fonction inconnue monadique

Remarquons que, dans ce cas, l'ensemble des variables est réduit à un élément. Nous allons introduire le concept de **règle sûre.** Intuitivement, une règle R est sûre si, lorsque les occurrences de φ sélectionnées par R dans un terme sont indéfinies, τ est lui même indéfini indépendamment des autres occurrences de φ. Pour préciser cette définition, adoptons quelques conventions. Ajoutons d'abord une valeur non définie (notée ω) au domaine D et posons alors $D^+ = D \cup \{\omega\}$. Si f est une fonction non définie en d, on pose $f(d) = \omega$. En particulier, la fonction nulle part définie Ω vérifie $\Omega(d) = \omega$ pour tout d de D^n. Adjoignons maintenant à l'ensemble des symboles fonctionnels, l'ensemble des fonctions de D^n dans D. Toute interprétation I_0 peut être prolongée de manière canonique en posant $I_0[f] = f$ pour tout f de $\mathscr{F}(D^n, D)$. Si t est un terme, nous pouvons considérer le terme

$$t' = t[\Omega/\underline{\varphi}, f/\varphi]$$

obtenu en remplaçant les occurrences de φ soit par Ω, soit par f. Une telle expression peut être simplifiée en utilisant les règles décrites précédemment auxquelles il convient d'ajouter les règles suivantes :

$$\text{Simpl}(f(a_1, \ldots, a_n)) = \omega \quad \text{s'il existe i tel que } a_i = \omega$$
$$\text{Simpl}(\underline{si}\ \omega\ \underline{alors}\ a\ \underline{sinon}\ b\ \underline{fsi}) = \omega$$
$$\text{Simpl}(\underline{si}\ \textbf{vrai}\ \underline{alors}\ a\ \underline{sinon}\ \omega\ \underline{fsi}) = a$$
$$\text{Simpl}(\underline{si}\ \textbf{faux}\ \underline{alors}\ \omega\ \underline{sinon}\ b\ \underline{fsi}) = b$$

Exemple 16

Soit t le terme $\underline{si}\ p(3, 2)\ \underline{alors}\ \underline{\varphi}(\varphi(2))\ \underline{sinon}\ h(\varphi(3), 4)\ \underline{fsi}$ et f une fonction de $\mathbb{N}$ dans $\mathbb{N}$ telle que $f(3) = 1$.
$t' = t[\Omega/\underline{\varphi}, f/\varphi] = \underline{si}\ p(3, 2)\ \underline{alors}\ f(\Omega(2))\ \underline{sinon}\ h(f(3), 4)\ \underline{fsi}$
Si $p(x, y)$ est interprété par $x > y$ et h par l'addition

$$\text{Simpl}(t') = \text{Simpl}(f(\Omega(2))) = \text{Simpl}(f(\omega)) = \omega$$

Si $p(x, y)$ est interprété par $x \leqslant y$ et h par l'addition

$$\text{Simpl}(t') = \text{Simpl}(h(f(3), 4)) = f(3) + 4 = 5\,. \qquad \square$$

Nous pouvons maintenant donner la définition suivante d'une règle sûre :

Définition 8

Une **règle R est sûre** si, pour tout terme t sans variable contenant des occurrences de φ et pour toute fonction $f : D^n \to D$, les occurrences de φ

sélectionnées par R sont telles que

$$\text{Simpl}(t[\Omega/\underline{\varphi}, f/\varphi]) = \omega .$$

Notons que, si R est une règle sûre,

$$\text{Simpl}(t[\Omega/\underline{\varphi}, F/\varphi]) = \text{Simpl}(t[\Omega/\varphi]) .$$

Exercice 29

a) Vérifier que, dans le cas d'une variable, l'appel par valeur, l'appel par nom et la substitution pleine sont des règles sûres.

b) Considérons la règle PD sélectionnant la plus à droite parmi les occurrences internes. Vérifier que la règle PD n'est pas une règle sûre. □

La partie b) de l'exercice 29 montre clairement qu'une règle R est sûre seulement si, parmi les occurrences de φ qu'elle sélectionne, il y en a qui sont dans les prédicats des conditionnelles. Cette situation résulte du fait que Si b ALORS x SINON y peut être défini même si x ou y est indéfini.

Alors, dans le cas d'une fonction inconnue monadique, les réponses aux questions posées au début du paragraphe peuvent être résumées par le théorème suivant :

Théorème 13 Soient $\langle \varphi(x) \Leftarrow \varepsilon, I_0 \rangle$ une déclaration récursive interprétée à une variable, Φ la fonction définie par interprétation, FIX le plus petit point fixe de la fonctionnelle associée et, pour toute règle R, Φ_R la fonction calculée par application de R. Alors

1 $\Phi_V = \Phi = \text{FIX}$. Autrement dit, la fonction calculée par appel par valeur coïncide avec le plus petit point fixe.

2 Pour toute règle R, $\Phi_R \sqsubseteq \text{FIX}$. En particulier, si Φ_R est définie en d, $\Phi_R(d) = \text{FIX}(d)$.

3 Pour toute règle sûre R, $\Phi_R = \text{FIX}$. □

Démontrons successivement chacun de ces résultats :

Démonstration de 1 $\Phi_V = \Phi$: ce premier résultat est assez naturel. Pour interpréter un schéma on effectue toujours au préalable l'évaluation des paramètres (définition 2, conditions b) et c)) et l'évaluation du test (condition d)). Une démonstration rigoureuse est laissée au lecteur. Il s'agit de montrer plus généralement que $\text{Res}_V(\tau)\,(d) = I^{\varepsilon}(\tau)\,(d)$ où $\text{Res}_V(\tau)$ désigne le résultat, s'il existe, du calcul de τ pour la donnée d et la règle V. C.Q.F.D.

Exercice 30

Soit PD la règle introduite à l'exercice 29. Montrer que, en général, $\Phi_{PD} \neq \Phi$. □

Démonstration de 2 $\Phi_R \sqsubseteq \text{FIX}$: Soient pour d donnée, $t_0^R, t_1^R, \ldots$ le calcul de $\varphi(d)$ déterminé par R et $t_0^P, t_1^P, \ldots$ le calcul de $\varphi(d)$ déterminé par la règle P (substitution pleine).

Posons $F_i(d) = \text{Simpl}(t_i^R[\Omega/\varphi])$ et $G_i(d) = \text{Simpl}(t_i^P[\Omega/\varphi])$. On peut montrer que

i) $F_i \sqsubseteq G_i$ pour tout i,

ii) G_i est égale à la i-ème approximation du plus petit point fixe FIX (cf. chapitre 2) ; en d'autres termes, $\text{FIX} = \bigcup_{i \geqslant 0} G_i$,

iii) $\bigcup_{i \geqslant 0} F_i = \Phi_R$, $\bigcup_{i \geqslant 0} G_i = \Phi_P$.

Il suit immédiatement que $\Phi_R = \bigcup_{i \geqslant 0} F_i \sqsubseteq \bigcup_{i \geqslant 0} G_i = \text{FIX}$. C.Q.F.D.

Démonstration de 3 Si R est une règle sûre $\Phi_R = \text{FIX}$.

Nous allons, pour démontrer ce résultat, considérer de nouveau les calculs de $\varphi(d)$ déterminés par la règle R et la règle de substitution pleine P. Soit $d \in D$ fixé. Il s'agit de montrer que la suite $(t_m^R)_{m \geqslant 0}$ déterminée par R ne prend pas trop de retard par rapport à la suite $(t_n^P)_{n \geqslant 0}$ déterminée par la règle P.

Nous avons besoin d'associer à chaque calcul t_m une autre suite u_m. Alors que la suite (t_m) est obtenue en simplifiant systématiquement après substitution, la suite (u_m) est construite en négligeant systématiquement ces simplifications. Le passage de u_m à u_{m+1} ne peut cependant être exécuté qu'en utilisant le terme simplifié t_m.

Montrons-le sur un exemple.

Exemple 17

Soient la définition récursive interprétée

$$\varphi(x) \Leftarrow \underline{si}\ x\ \underline{pair}\ \underline{alors}\ \varphi(x+2)\ \underline{sinon}\ \varphi(x-2)\ \underline{fsi}$$

et le terme $t_0 = \varphi(3)$. On obtient, par la règle d'appel par valeur les suites $(u_m)_{m \geqslant 0}$ et $(t_m)_{m \geqslant 0}$ suivantes :

$$\begin{aligned}
u_0 &= t_0 = \varphi(3)\,, \\
u_1 &= \underline{si}\ 3\ \underline{pair}\ \underline{alors}\ \varphi(3+2)\ \underline{sinon}\ \underset{\sim\sim\sim\sim}{\varphi(3-2)}\ \underline{fsi} \\
t_1 &= \varphi(1)\,, \\
u_2 &= \underline{si}\ 3\ \underline{pair}\ \underline{alors}\ \varphi(3+2) \\
&\qquad \underline{sinon}\ \underline{si}\ (3-2)\ \underline{pair}\ \underline{alors}\ \varphi(3-2+2)\ \underline{sinon}\ \underset{\sim\sim\sim\sim}{\varphi(3-2-2)}\ \underline{fsi} \\
t_2 &= \varphi(-1)\,, \\
&\text{etc...}
\end{aligned}$$

On voit que u_m contient en général des occurrences parasites et qu'il est nécessaire d'établir une correspondance entre les occurrences de t_m et les occurrences utiles de u_m soulignées dans l'exemple. u_{m+1} est alors simplement obtenu en substituant dans u_m les occurrences de φ associées aux occurrences sélectionnées par R dans t_m. Pour tout $m \geqslant 0$, on a évidemment $t_m = \text{Simpl}(u_m)$.

L'intérêt de travailler sur la suite (u_m) est que l'ensemble $\mathscr{T}$ des termes déduits de $\varphi(d)$ par une suite quelconque de substitutions peut être muni d'un ordre $\leqslant$ défini comme la plus petite relation d'ordre contenant les couples $(\varphi(d), \varepsilon(d))$ et qui soit stable par composition par des symboles fonctionnels.

$(\mathcal{T}, \leqslant)$ est un treillis. Autrement dit, tout couple t, t' de $\mathcal{T}$ admet un minimum et un maximum. De plus, si t, t' sont des termes tels que $t \leqslant t'$ et si f est une fonction de D^n dans D, $\mathrm{Simpl}(t[f/\varphi])$ est moins défini que $\mathrm{Simpl}(t'[f/\varphi])$.

Venons-en à la démonstration : Soient $d \in D$ et $n \geqslant 0$. On montre qu'il existe un entier $m \geqslant 0$ et un terme ε' tel que

$$(1) \qquad u_n^P \leqslant u_m^R[\varphi/\underline{\varphi}, \varepsilon'/\varphi] .$$

En effet, la suite (u_m^R) est croissante, donc aussi la suite $\min(u_n^P, u_m^R)$. Puisque u_n^P admet un nombre fini de minorants, il existe $m \geqslant 0$ tel que

$$(2) \qquad \mathrm{Min}(u_n^P, u_m^R) = \mathrm{Min}(u_n^P, u_{m+1}^R) .$$

On montre, par récurrence, qu'il existe alors un terme ε' vérifiant la relation (1). Cette partie de la démonstration est indépendante du fait que R est une règle sûre.

En remplaçant φ par Ω dans (1) et en simplifiant, il vient

$$(3) \qquad \mathrm{Simpl}(u_n^P[\Omega/\varphi]) \sqsubseteq \mathrm{Simpl}(u_m^R[\Omega/\underline{\varphi}, \varepsilon'[\Omega/\varphi]/\varphi]) .$$

Soit f l'interprétation de $\varepsilon'[\Omega/\varphi]$ induite par I_0. Puisque $t_n^P = \mathrm{Simpl}(u_n^P)$ et $t_m^R = \mathrm{Simpl}(u_m^R)$, on obtient

$$(4) \qquad \mathrm{Simpl}(t_n^P[\Omega/\varphi]) \sqsubseteq \mathrm{Simpl}(t_m^R[\Omega/\varphi, f/\varphi]) .$$

Puisque R est une règle sûre

$$(5) \qquad \mathrm{Simpl}(t_m^R[\Omega/\varphi, f/\varphi]) = \mathrm{Simpl}(t_m^R[\Omega/\varphi])$$

et avec les notations de la démonstration précédente,

$$(6) \qquad G_n(d) = \mathrm{Simpl}(t_n^P[\Omega/\varphi]) \sqsubseteq \mathrm{Simpl}(t_m^R[\Omega/\varphi]) = F_m(d) .$$

Donc

$$(7) \qquad G_n \sqsubseteq \bigcup_{m \geqslant 0} F_m = \Phi_R ;$$

et puisque l'assertion (7) est vraie pour tout n,

$$(8) \qquad \mathrm{FIX} = \bigcup_{n \geqslant 0} G_n \sqsubseteq \Phi_R . \quad \text{C.Q.F.D.}$$

Exercice 31

Considérons la déclaration interprétée

$$\varphi(x) \Leftarrow \underline{si}\ x > 100\ \underline{alors}\ x - 10\ \underline{sinon}\ \varphi(\varphi(x + 11))\ \underline{fsi}$$

la règle N de l'appel par nom et la donnée $d = 89$.

Calculer les premiers termes des suites (t_n^N), (t_n^P), (u_n^N), (u_n^P) dans ce cas.

Vérifier qu'il existe un entier $m \geqslant 0$ et un terme ε' tel que

$$u_2^P \leqslant u_m^N[\varphi/\underline{\varphi}, \varepsilon'/\varphi] . \qquad \square$$

b) Cas où la fonction inconnue est polyadique

Lorsque $\mathscr{X}$ n'est pas réduit à une variable, les parties 2 et 3 du théorème 2 peuvent être mises en défaut. Considérons par exemple la définition récursive suivante :

Exemple 18

$$\varphi(x, y) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ 0\ \underline{sinon}\ \varphi(x - 1, \varphi(x, y))\ \underline{fsi}$$

Il n'est pas difficile de voir que $\Phi(0, y) = \Phi_R(0, y) = \mathrm{FIX}(0, y) = 0$. Les résultats sont différents pour $x > 0$. En effet, considérons les calculs par valeur et par nom de $\varphi(1, 1)$:

$$\underline{\varphi}(1, 1) \overset{V}{\Rightarrow} \varphi(0, \underline{\varphi}(1, 1)) \overset{V}{\Rightarrow} \varphi(0, \varphi(0, \underline{\varphi}(1, 1))) \overset{V}{\Rightarrow} \cdots,$$

tandis que

$$\underline{\varphi}(1, 1) \overset{N}{\Rightarrow} \underline{\varphi}(0, \varphi(0, 1)) \overset{N}{\Rightarrow} 0 .$$

De même

$$\underline{\varphi}(1, 1) \overset{P}{\Rightarrow} \underline{\varphi}(0, \underline{\varphi}(1, 1)) \overset{P}{\Rightarrow} 0 .$$

Ainsi $\Phi_V(x, y)$ et $\Phi(x, y)$ sont non définis pour $x > 0$, tandis que $\Phi_N(x, y)$ et $\Phi_P(x, y)$ sont égaux à 0. Par le théorème 12, $\mathrm{FIX}(x, y) = \Phi(x, y)$ et donc $\mathrm{FIX}(x, y)$ est non défini pour $x > 0$. Il est facile de retrouver directement ce résultat en calculant FIX par approximations successives. Soit τ la fonctionnelle associée à la définition précédente. On trouve successivement

$$\tau(\Omega)(x, y) = \mathrm{SI}\ x = 0\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \omega$$

$$\tau^2(\Omega)(x, y) = \mathrm{SI}\ x = 0\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \tau(\Omega)(x - 1, \tau(\Omega)(x, y)) .$$

Mais pour $x > 0$, $\tau(\Omega)(x, y)$ est non définie, donc aussi

$$\tau(\Omega)(x - 1, \tau(\Omega)(x, y)) . \quad \text{Ainsi} \quad \tau(\Omega) = \tau^2(\Omega) = \mathrm{FIX} . \qquad \square$$

Cette situation paradoxale n'est pas très satisfaisante pour l'informaticien, pas plus que ne le serait une situation dans laquelle Φ_V serait distincte de FIX. Ainsi, nous sommes conduits à associer un deuxième type de fonctionnelle aux définitions récursives. Notons cependant que le théorème 13 reste vrai dans le cas général si l'on remplace dans les assertions 2 et 3 FIX par Φ_P. En effet, dans ce cas, on a en général $\mathrm{FIX} \sqsubseteq \Phi_P$.

c) Un nouveau type de fonctionnelle.

La remarque clé, dans l'exemple précédent, est, qu'en mathématiques classiques $F(G, H)$ n'est définie en x **que si** G et H sont définies en x et **si** F est définie en $(G(x), H(x))$. Une déclaration plus correcte serait

$$\varphi(x, y) \Leftarrow \underline{si}\ x = 0\ \underline{et}\ y\ \textit{défini}\ \underline{alors}\ 0\ \underline{sinon}\ \varphi(x - 1, \varphi(x, y))\ \underline{fsi} .$$

Pour mieux rendre compte de l'appel par nom, étendons le domaine d'interprétation en ajoutant à D le symbole ω (non défini) déjà introduit précédemment. Soit $D^+ = D \cup \{ \omega \}$. D^+ est ordonné par la relation $x \sqsubseteq y \Leftrightarrow x = \omega$ ou $x = y$.

On peut prolonger toute fonction f de D^m dans D en une application $\bar{f}$ de $(D^+)^m$ dans D^+ définie par la relation

$$\bar{f}(x) \underset{\text{def}}{=} \text{SI } x \in D^m \text{ et f définie en x ALORS } f(x) \text{ SINON } \omega .$$

Définition 9

Une application f de $(D^+)^m$ est **stricte** si $f(x_1, \ldots, x_m) = \omega$ dès qu'il existe $i \in [1, m]$ avec $x_i = \omega$. □

Ainsi toute fonction f de D^m dans D induit une application stricte $\bar{f}$ de $(D^+)^m$ dans D^+. En revanche, SI ... ALORS ... SINON induit une application non stricte sur $B^+ \times (D^+)^2$ (où B = { **vrai, faux** }). Il est naturel, en effet, de poser, pour $(x, y) \in (D^+)^2$

SI ω ALORS x SINON y $= \omega$
SI **vrai** ALORS x SINON y $= x$
SI **faux** ALORS x SINON y $= y$.

Notons que, dans les paragraphes précédents, SI ... ALORS ... SINON n'est pas une application, SI b ALORS x SINON y pouvant avoir un sens même si x ou y est non défini.

Indiquons dans la définition suivante l'équivalent des définitions 2, 3 et 7.

Définition 10

Soit $\langle \varphi(x) \Leftarrow \varepsilon, I_0 \rangle$ une définition récursive interprétée sur un domaine D. I_0 induit sur D^+ une interprétation $\tilde{I}_0$ définie par $\tilde{I}_0[g] = \overline{I_0[g]}$. On peut d'autre part associer à chaque terme τ une fonctionnelle $\tilde{I}[\![\tau]\!]$ sur l'espace $\mathscr{F}((D^+)^n, D^+)$ vérifiant les conditions a', ..., e' de la définition 3. On notera $\widetilde{FIX}$ le plus petit point fixe de $\tilde{I}[\![\varepsilon]\!]$.

L'interprétation $\tilde{I}_0$ induit une interprétation $\tilde{I}^\varepsilon$ des termes par des applications de $(D^+)^n$ dans D^+, vérifiant les propriétés a) à e) de la définition 2. On posera $\tilde{\Phi} = \tilde{I}^\varepsilon[\varphi(x)]$.

De même, nous pouvons définir le calcul de φ pour un point $d \in (D^+)^n$ et une règle R. Le résultat de ce calcul sera noté $\tilde{\Phi}_R(d)$. □

Remarque Pour $d \in D^n$, $\tilde{\Phi}_R(d) = \Phi_R(d)$. Par contre, pour $d \notin D^n$, $\tilde{\Phi}_R(d)$ peut être distinct de ω et donc $\tilde{\Phi}_R$ et $\overline{\Phi_R}$ sont en général distinctes.

Exercice 32

Déterminer la fonctionnelle sur $\mathscr{F}((D^+)^2, D^+)$ associée à la déclaration récursive

$$\varphi(x, y) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ 0\ \underline{sinon}\ \varphi(x - 1, \varphi(x, y))\ \underline{fsi} .$$

Calculer $\widetilde{FIX}$ et $\tilde{\Phi}_N$. Vérifier que $\widetilde{FIX} = \tilde{\Phi}_N \sqsupseteq \tilde{\Phi}_V$.

Vérifier que $\widetilde{\text{FIX}}$ n'est pas une application stricte. En particulier, montrer que $\widetilde{\text{FIX}} \neq \overline{\text{FIX}}$. □

Comme le prouve l'exercice 32, le plus petit point fixe associé à un programme récursif n'est pas nécessairement strict. Il nous reste à étendre la notion règle sûre.

Définition 11

Une règle R est fortement sûre si pour tout terme t simplifié, sans variable, contenant des occurrences de φ et toute application $\mathrm{f} : (\mathrm{D}^+)^{\mathrm{n}} \to \mathrm{D}^+$ non nécessairement stricte, les occurrences de φ sélectionnées par R sont telles que

$$\mathrm{Simpl}(t[\Omega/\underline{\varphi}, \mathrm{f}/\varphi]) = \mathrm{Simpl}(t[\Omega/\varphi]) \,.$$ □

Remarquons que toute règle fortement sûre est une règle sûre mais que la réciproque est fausse comme le montre l'exercice suivant.

Exercice 33

Montrer que les règles d'appel par nom et de substitution pleine sont fortement sûres. Montrer que la règle d'appel par valeur n'est pas fortement sûre. □

Nous pouvons maintenant écrire des équivalents des théorèmes 12 et 13.

Théorème 14 $\tilde{\Phi} = \widetilde{\text{FIX}}$ □

Théorème 15

1. $\tilde{\Phi}_{\mathrm{P}} = \tilde{\Phi} = \widetilde{\text{FIX}}$.
2. Pour toute règle R, $\tilde{\Phi}_{\mathrm{R}} \sqsubseteq \widetilde{\text{FIX}}$.
3. Pour toute règle R fortement sûre, $\tilde{\Phi}_{\mathrm{R}} = \widetilde{\text{FIX}}$. □

En particulier, il résulte de 3 et de l'exercice 33 que la règle d'appel par nom est fortement sûre et donc que $\tilde{\Phi}_{\mathrm{N}} = \widetilde{\text{FIX}}$.

4.6 Relation entre les règles d'appel par valeur et d'appel par nom

Nous voulons montrer dans ce paragraphe que, lorsque l'appel par nom donne un résultat différent de l'appel par valeur, il y a en quelque sorte des paramètres inutiles et que l'on peut écrire une définition récursive équivalente avec passage des paramètres par valeur. Etudions tout d'abord quelques exemples.

Exemple 19

Considérons la définition récursive

$$\varphi(x, y) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ 0\ \underline{sinon}\ \varphi(x - 1, \varphi(x, y)) + 1\ \underline{fsi}$$

On constate immédiatement que la valeur de $\Phi_{\mathrm{N}}(\mathrm{x}, \mathrm{y})$ ne dépend pas du tout de y. On peut donc supprimer y de cette définition. La fonction Ψ_{V} associée à la définition

$$\psi(x) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ 0\ \underline{sinon}\ \psi(x - 1) + 1\ \underline{fsi}$$

vérifie effectivement

$$\forall x, y, \Psi_V(x) = \Phi_N(x, y) . \qquad \square$$

Les exemples suivants peuvent sembler quelque peu triviaux puisque le premier ne comporte aucune récursivité et que le terme de droite du second contient une seule occurrence de φ et que, de ce fait, toutes les règles se confondent. Cependant, si l'on effectue un appel dans lequel les paramètres effectifs ne sont plus des valeurs simples mais des termes quelconques, il redevient essentiel de savoir si l'on évalue d'abord les paramètres ou si l'on effectue d'abord la substitution.

Soit, par exemple, la déclaration $\varphi(x) \Leftarrow 1$. Il est clair que $\Phi_V(x) = \Phi_N(x) = 1$. Par contre, si l'on considère maintenant une deuxième déclaration $\psi(x) \Leftarrow \psi(x)$, l'évaluation de l'appel $\varphi(\psi(x))$ diffère selon que l'on utilise la règle d'appel par valeur ou la règle d'appel par nom. Cette question mériterait une étude détaillée que nous ne pouvons entreprendre ici.

L'exemple 20 montre qu'il peut être nécessaire de se ramener à plusieurs équations.

Exemple 20

Considérons la déclaration

$$\varphi(x, y, z) \Leftarrow \underline{si}\ p(x)\ \underline{alors}\ g(y)\ \underline{sinon}\ h(z)\ \underline{fsi}$$

et le système

$$\psi_1(x, y) \Leftarrow \underline{si}\ p(x)\ \underline{alors}\ g(y)\ \underline{sinon}\ \omega\ \underline{fsi}$$
$$\psi_2(x, z) \Leftarrow \underline{si}\ \neg p(x)\ \underline{alors}\ h(z)\ \underline{sinon}\ \omega\ \underline{fsi}\ .$$

Cette fois, la valeur de $\Phi_N(x, y, z)$ dépend soit de x, y, soit de x, z. Plus précisément, pour chaque x, une seule des valeurs $\Psi_V^1(x, y)$ et $\Psi_V^2(x, z)$ est définie. On peut donc calculer $\Phi_N(x, y, z)$ en calculant parallèlement $\Psi_V^1(x, y)$ et $\Psi_V^2(x, z)$. Au plus, un de ces calculs se terminera, donnant la valeur de $\Phi_N(x, y, z)$. Nous utilisons, pour traduire cette situation, la notion de borne supérieure. Considérons une famille $(x_s)_{s \in S}$ d'éléments de D^+ tels que, pour tout couple (s, t), $x_s = \omega$ $\underline{ou}$ $x_t = \omega$ $\underline{ou}$ $x_s = x_t$. Cette famille admet une borne supérieure que l'on note $\bigcup_{s \in S} x_s$.

Si $S = \{ 1, ..., n \}$, on écrit la borne supérieure $\bigcup_{1 \leqslant i \leqslant n} x_i$ sous la forme

$$x_1 \vee x_2 \vee \ldots \vee x_n .$$

Ainsi, dans le deuxième exemple, $\Phi_N(x, y, z)$ est égal à $\Psi_V^1(x, y) \vee \Psi_V^2(x, z)$. Cette dernière notation exprime assez bien l'indéterminisme introduit pour conserver uniquement les paramètres utiles. $\square$

Présentons enfin un troisième exemple mettant en évidence la nécessité d'introduire l'indéterminisme jusque dans les définitions récursives elles-mêmes.

Exemple 21

On considère la définition récursive

$$(1) \qquad \varphi(x_1, x_2, \ldots, x_n) \Leftarrow \underline{si}\ p(x_1)\ \underline{alors}\ x_1\ \underline{sinon}\ \varphi(x_2, \ldots, x_n, h(x_1))\ \underline{fsi}$$

Il est clair que Φ_N dépend, soit de x_1 seulement, soit de x_1 et x_2, soit plus généralement des i premières variables $x_1, \ldots, x_i$. Il existe donc n fonctions $\Phi^1, \Phi^2, \ldots, \Phi^n$ telles que l'on puisse écrire

$$\Phi_N(x_1, \ldots, x_n) = \Phi^1(x_1) \vee \Phi^2(x_1, x_2) \vee \ldots \vee \Phi^n(x_1, \ldots, x_n) .$$

Nous voulons montrer que la famille $(\Phi^i)_{1 \leqslant i \leqslant n}$ est définie par un système récursif. Remplaçons pour cela φ par $\varphi^1 \vee \dots \vee \varphi^n$ dans (1). Il vient, après développement

$$\begin{aligned}\varphi^1(x_1) \vee \dots \vee \varphi^n(x_1, \dots, x_n) \Leftarrow\\ &\underline{si}\ p(x_1)\ \underline{alors}\ x_1\ \underline{sinon}\ \omega\ \underline{fsi}\\ \vee\ &\underline{si}\ \neg p(x_1)\ \underline{alors}\ \varphi^1(x_2)\ \underline{sinon}\ \omega\ \underline{fsi}\\ \vee\ &\underline{si}\ \neg p(x_1)\ \underline{alors}\ \varphi^{n-1}(x_2, \dots, x_n) \vee \varphi^n(x_2, \dots, x_n, h(x_1))\ \underline{sinon}\ \omega\ \underline{fsi}\end{aligned}$$

En séparant les variables et en faisant en sorte que les φ^i aient des « domaines » deux à deux disjoints, on obtient successivement les équations

$$\begin{aligned}&\varphi^1(x_1) \Leftarrow \underline{si}\ p(x_1)\ \underline{alors}\ x_1\ \underline{sinon}\ \omega\ \underline{fsi}\\ &\varphi^2(x_1, x_2) \Leftarrow \underline{si}\ \neg p(x_1)\ \underline{alors}\ \varphi^1(x_2)\ \underline{sinon}\ \omega\ \underline{fsi}\\ &\varphi^n(x_1, \dots, x_n)\\ &\qquad \Leftarrow \underline{si}\ \neg p(x_1)\ \underline{alors}\ \varphi^{n-1}(x_2, \dots, x_n) \vee \varphi^n(x_2, \dots, h(x_1))\ \underline{sinon}\ \omega\ \underline{fsi}\,.\end{aligned}$$

Il est clair que chaque Φ^i peut être calculée en appliquant la règle de l'appel par valeur (mais en utilisant uniquement les variables $x_1, \dots, x_i$). Ainsi

$$\Phi_N(x_1, \dots, x_n) = \Phi_V^1(x_1) \vee \Phi_V^2(x_1, x_2) \vee \dots \vee \Phi_V^n(x_1, \dots, x_n)\,. \qquad \square$$

Exercice 34

a) Calculer les domaines de définitions de $\Phi_V^1, \dots, \Phi_V^n$ de l'exemple 21.

b) Donner des expressions explicites de ces fonctions (Φ_V^n peut être exprimée comme la borne supérieure d'une famille infinie de fonctions). □

Nous pouvons maintenant énoncer le résultat général suivant, démontré par de ROEVER [ROE, 75]. Commençons par introduire la notation suivante :

Si $x = (x_1, \dots, x_n)$ est un n-uplet et si $s = (s_1, \dots, s_m)$ est un m-uplet d'entiers tels que $1 \leqslant s_1 < s_2 < \cdots < s_m \leqslant n$, on note x_s le m-uplet $(x_{s_1}, \dots, x_{s_m})$.

Théorème 16

Etant donnée une définition récursive $\varphi(x) \Leftarrow \varepsilon$, on peut déterminer un ensemble S de uplets de [1, n] et un système de définitions mutuellement récursives

$$\{\ \varphi^s(x_s) \Leftarrow \varepsilon_s\ \}_{s \in S}$$

tels que, pour tout $x \in D^n$

$$\Phi_N(x) = \bigcup_{s \in S} \Phi_V^s(x_s)\,. \qquad \square$$

Commentaire. Pour chaque $s \in S$, x_s représente une combinaison de paramètres nécessaires et suffisants pour le calcul de Φ_N en un point de D^n. Le résultat de DE ROEVER exprime qu'on peut trouver un système de définitions récursives tel que, pour tout $x \in D^n$, les valeurs $y_s = \Phi_V^s(x_s)$ sont compatibles entre elles et tel que $\Phi_N(x)$ soit la borne supérieure de y_s.

4.7 Complexité des calculs. Une règle optimale

Nous avons vu sur des exemples que l'appel par valeur était optimal dans le cas d'une variable. Dans le cas général, il arrive que l'appel par valeur ne permette pas d'obtenir le résultat espéré. A l'inverse, l'appel par nom nécessite de nombreuses substitutions.

Exemple 22

Considérons la déclaration $\varphi(x) \Leftarrow \underline{si}\ x > 0\ \underline{alors}\ x - 1\ \underline{sinon}\ \varphi(\varphi(x + 2))\ \underline{fsi}.$
L'appel par nom génère le calcul :

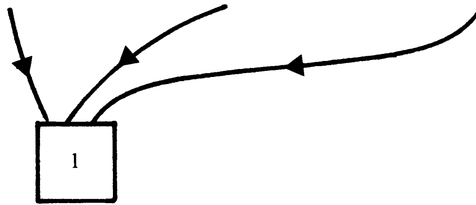
$$\underline{\varphi}(0) \to \underline{\varphi}(\varphi(2)) \to \underline{si}\ \underline{\varphi}(2) > 0\ \underline{alors}\ \varphi(2) - 1\ \underline{sinon}\ \varphi(\varphi(\varphi(2) + 2))\ \underline{fsi}$$
$$\to \underline{\varphi}(2) - 1 \to 0\ .$$ □

Dans la deuxième substitution ci-dessus, le terme $\varphi(2)$ a été reproduit à trois exemplaires et de ce fait calculé deux fois ; pour éviter ces duplications, VUILLEMIN propose la règle du report énoncé ci-après.

Règle du report (R) : R sélectionne dans un terme τ l'occurrence de φ la plus à gauche, ainsi que toutes les occurrences admettant mêmes arguments que celle-ci.

Ainsi, dans l'exemple, les trois occurrences de $\varphi(2)$ sont substituées simultanément. La règle du report apparaît moins efficace que l'appel par nom si ces trois substitutions sont comptées indépendamment. Il n'est pas difficile cependant d'implémenter cette règle de telle sorte qu'un seul calcul soit nécessaire. L'idée est d'éviter les recopies en faisant pointer les diverses occurrences de φ vers une unique copie.

Exemple : $\underline{si}\ \underline{\varphi(2)} > \quad 0\ \underline{alors}\ \underline{\varphi(2)} - \quad 1\ \underline{sinon}\ \varphi(\varphi(\underline{\varphi(2)} + 2)).$



Remarquons que, dans le cas d'une variable, un calcul régi par la règle du report contient les mêmes substitutions que le calcul régi par la règle d'appel par valeur, mais dans un ordre différent. Dans ce sens, la règle d'appel par valeur peut être considérée comme également optimale dans le cas d'une variable.

Exercice 35

Considérons le programme récursif suivant permettant de calculer la fonction d'Ackermann

$$A(x, y) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ y + 1\ \underline{sinon}\ \underline{si}\ y = 0\ \underline{alors}\ A(x - 1, 1)$$
$$\underline{sinon}\ A(x - 1, A(x, y - 1))\ \underline{fsi}\ \underline{fsi}\ .$$

Calculer $A(2, 1)$ en appliquant successivement les règles d'appel par valeur, d'appel par nom, du report et la règle de substitution pleine. Calculer, dans chaque cas, le nombre de substitutions effectuées. □

4.8 Conclusion

La puissance des schémas récursifs réside principalement dans le fait qu'ils sont en même temps un moyen commode de poser les problèmes (voir § 2.2) et un outil de calcul effectif des solutions. Nous avons essentiellement étudié ici l'approche point fixe des schémas récursifs, telle qu'elle a été développée depuis 1969 par SCOTT et De BAKKER, puis par de nombreux autres cher-

cheurs. Nous n'avons pas abordé les problèmes de relations entre schémas de programmes avec branchement et schémas récursifs ainsi que les problèmes d'équivalence entre schémas récursifs.

5 CALCUL RELATIONNEL RÉCURSIF

5.1 Présentation

Jusqu'à présent, nous avons rencontré trois types de schémas de programme :

— *les schémas avec branchements* qui sont rudimentaires quant à leurs structures de contrôle et assez éloignés de la pratique actuelle de la programmation ;

— *les schémas itératifs* qui formalisent les structures de contrôle des langages de programmation ;

— *les schémas récursifs* qui rendent compte des procédures récursives.

Ici, on introduit un outil formel, le calcul relationnel, qui contient ces divers types de schémas et qui permet d'étudier les preuves de correction de programme (§ 4.5). Outre les caractéristiques énoncées, ce cadre permet de comparer des programmes et introduit **l'indéterminisme** pour lequel on peut trouver plusieurs justifications.

D'une part, certaines instructions de contrôle telles que l'instruction conditionnelle, l'itération, etc. contiennent, elles-mêmes, une forme d'indéterminisme. Dans le cas des conditionnelles, par exemple, le choix de la branche de sortie nécessite une interprétation, mais notre but étant de travailler sur un système formel hors des interprétations, l'indéterminisme s'introduit donc naturellement. D'autre part l'indéterminisme permet de simplifier la théorie puisqu'il est inutile de vérifier que les constructions du système conservent le déterminisme. L'introduction de l'indéterminisme conduit à ne plus considérer les programmes comme des fonctions (éventuellement partiellement définies) des initialisations vers les résultats, mais comme des **relations binaires** entre ces valeurs.

Pour plus de simplicité, nous limitons l'exposé au cas des schémas relationnels à une variable qui correspondent aux schémas de Ianov (§ 2.5) ; l'unique variable est alors un vecteur d'état. Pour les mêmes raisons de simplicité, nous n'étudierons, ni ne spécifierons, les opérations de base de notre langage de programmation, mais seulement les constructions fondamentales et leur interprétation relationnelle.

Un schéma relationnel spécifie un calcul ou une suite de calculs, on peut distinguer plusieurs points de vue :

— le schéma doit être interprété et alors il spécifie des suites de calculs du domaine d'interprétation (approche opérationnelle) (§ 5.4),

— le schéma est un système d'équation dont les solutions sont des calculs (approche calculatoire) (§ 5.5) ;

— le schéma est la notation d'une relation (approche dénotationnelle) (§ 5.6).

5.2 Opérations sur les relations et constructions de programmes

Le calcul relationnel est un système algébrique, qui généralise les treillis distributifs ; les opérations de base sont celles portant sur les relations, c'est-à-dire celles, définies sur les parties de l'ensemble $D \times D$, auxquelles on ajoute les opérations de composition et de produit itéré.

Dans la suite, on dit qu'un programme π spécifie une relation R si, pour tout $(x, y) \in D \times D$ on a $x \, R \, y$ si et seulement si y est un résultat possible du programme π pour la donnée x. Notons qu'une même relation peut être spécifiée par plusieurs programmes.

a) Composition de relations

Soient deux relations R_1 et R_2 ; la composée de R_1 et R_2 est notée $R_1 \, ; R_2$ comme la composition séquentielle des programmes. Elle est définie par :

$$R_1 \; ; R_2 = \{ (x, y) \in D \times D \mid \exists z \in D \, (x \, R_1 \, z \textbf{ et } z \, R_2 \, y) \} .$$

Si π_1 et π_2 sont deux programmes spécifiant R_1 et R_2, alors un programme spécifiant $R_1 \; ; R_2$ est le programme $\pi_1 \; ; \pi_2$ obtenu par composition sérielle à partir des deux programmes π_1 et π_2.

Exemple 23

Soient $D = \mathbb{N}$, $R_1 = \{ (x, x + 1) \mid x \in \mathbb{N} \}$ et $R_2 = \{ (x, 2\,x) \mid x \in \mathbb{N} \}$

alors
$$R_1 \; ; R_1 = \{ (x, x + 2) \mid x \in \mathbb{N} \}$$
$$R_1 \; ; R_2 = \{ (x, 2\,x + 1) \mid x \in \mathbb{N} \}$$
$$R_2 \; ; R_1 = \{ (x, 2\,x + 2) \mid x \in \mathbb{N} \} .$$
□

b) Unions de relations

Soient R_1 et R_2 deux relations ; la réunion, notée $R_1 \cup R_2$ est définie par

$$R_1 \cup R_2 = \{ (x, y) \in D \times D \mid x \, R_1 \, y \textbf{ ou } x \, R_2 \, y \} .$$

Si π_1 et π_2 sont deux programmes spécifiant R_1 et R_2 alors un programme spécifiant $R_1 \cup R_2$ est un programme que l'on pourrait noter **choix** (π_1, π_2) ; cela signifie que l'on choisit, de manière indéterministe, entre effectuer π_1 ou effectuer π_2.

c) Relations associées à un prédicat

Un prédicat p est, par définition, une fonction de D dans { **vrai, faux** }. Un prédicat p est dit total si c'est une fonction partout définie sur D.

On associe à un prédicat p, une relation que nous notons aussi p, cette relation est définie par

$$p = \{ (x, x) \in D \times D \mid p(x) = \textbf{vrai} \} .$$

On associe à p une relation notée $\overline{p}$ que l'on appelle la négation de p, elle est ainsi définie :

$$\overline{p} = \{ (x, x) \in D \times D \mid p(x) = \textbf{faux} \} .$$

Exemple 24

Si $D = \mathbb{N}$ et si 'zéro'(x) est le prédicat $x = 0$ on a alors

$$x \text{ 'zéro' } y \text{ si et seulement si } x = y = 0$$
$$x\ \overline{\text{'zéro'}}\ y \text{ si et seulement si } x = y \textbf{ et } x \neq 0\ .$$ □

Remarquons que $p\ ;\ q$ est la relation associée au prédicat p **et** q, tandis que $p \cup q$ est la relation associée au prédicat p **ou** q.

Exercice 36

1) Si p est un prédicat et R est une relation, vérifier que

$$x(p\ ;\ R)\ y \quad \text{si et seulement si} \quad p(x) \textbf{ et } x\ R\ y\ .$$

2) Si q est un prédicat, vérifier que

$$x(p\ ;\ R\ ;\ q)\ y \quad \text{si et seulement si} \quad p(x) \textbf{ et } x\ R\ y \textbf{ et } q(y)\ .$$ □

d) Description de l'instruction *si... alors... sinon... fsi*

Si π_1 spécifie la relation R_1 et si π_2 spécifie la relation R_2, alors le programme *si* p *alors* π_1 *sinon* π_2 *fsi* spécifie la relation entre x et y définie par

$$(p(x) \textbf{ et } x\ R_1\ y) \textbf{ ou } (\textbf{non } p(x) \textbf{ et } x\ R_2\ y)\ .$$

Compte tenu de l'exercice 27 ceci est équivalent à :

$$x(p\ ;\ R_1)\ y \textbf{ ou } x(\overline{p}\ ;\ R_2)\ y$$

c'est-à-dire

$$x[(p\ ;\ R_1) \cup (\overline{p}\ ;\ R_2)]\ y$$

ainsi le programme *si* p *alors* π_1 *sinon* π_2 *fsi* spécifie la relation $(p\ ;\ R_1) \cup (\overline{p}\ ;\ R_2)$.

e) Intersection de relations

Soient R_1 et R_2 deux relations, l'intersection de R_1 et R_2 est notée $R_1 \cap R_2$, elle est définie par

$$R_1 \cap R_2 = \{ (x, y) \in D \times D \mid x\ R_1\ y \textbf{ et } x\ R_2\ y \}\ .$$

Elle n'a pas d'interprétation au niveau de la construction des programmes, c'est essentiellement une opération qui permet de construire de nouveaux prédicats à partir des résultats d'un calcul (voir l'exercice 37).

Exercice 37

Montrer que si p et q sont des prédicats $p\ ;\ q = p \cap q$. □

f) Inverse d'une relation

L'inverse d'une relation R notée R^{-1} est par définition

$$R^{-1} = \{ (x, y) \in D \times D \mid y\ R\ x \}\ .$$

L'opération R^{-1} correspond intuitivement à la recherche d'une initialisation ayant conduit à un résultat connu.

g) Relations constantes

Introduisons trois relations constantes. Ce sont la relation vide, la relation identité ou d'égalité, la relation universelle ou pleine :

— la **relation vide** est notée $\Omega = \varnothing$; pour aucun couple $(x, y) \in D \times D$ on a $x \, \Omega \, y$; Ω est spécifiée par un programme qui ne termine jamais ;

— la **relation d'identité** (ou d'égalité) est notée $E = \{ (x, x) \in D \times D \mid x \in D \}$ (elle est aussi notée I par certains auteurs). La relation E est spécifiée par un programme qui ne fait rien, par exemple le programme vide, c'est-à-dire sans instruction (à ne pas confondre avec le programme qui spécifie la relation vide Ω) ;

— la **relation universelle** ou pleine notée

$$U = \{ (x, y) \in D \times D \mid x \in D, y \in D \} = D \times D .$$

Exercice 38

a) Montrer que $R \; ; E = E \; ; R = R$.

b) E est un prédicat. Que vaut le prédicat $\overline{E}$?

c) Montrer que <u>*si*</u> E <u>*alors*</u> π <u>*sinon*</u> π <u>*fsi*</u> spécifie la même relation que π. □

Exercice 39

Montrer que $R \cup \Omega = R$, $R \cap \Omega = \Omega$, $R \; ; \Omega = \Omega$, $p \; ; \overline{p} = \Omega$. □

Exercice 40

Montrer à partir de l'exercice précédent et en utilisant les propriétés des opérations introduites, que le programme

<u>*si*</u> p <u>*alors*</u> <u>*si*</u> p <u>*alors*</u> π <u>*sinon*</u> π' <u>*fsi*</u> <u>*sinon*</u> π'' <u>*fsi*</u>

spécifie la même relation que le programme

<u>*si*</u> p <u>*alors*</u> π <u>*sinon*</u> π'' <u>*fsi*</u> . □

Exercice 41

1) Montrer que $(R \; ; R^{-1}) \cap E = R \; ; U \cap E$.

2) Vérifier que $(R \; ; R^{-1}) \cap E$ est un prédicat.

3) Donner une interprétation informatique de ce prédicat, c'est-à-dire à l'aide d'une propriété du programme qui spécifie R. □

h) Inclusion de deux relations

L'inclusion de deux relations est à comprendre comme l'inclusion des parties de $D \times D$.

Soient π_1 et π_2 spécifiant respectivement R_1 et R_2. Si R_1 est inclus dans R_2 et si π_1 peut fournir y à partir de x alors il est de même de π_2 ; si π_1 et π_2 sont déterministes cela signifie que si π_1 termine pour x alors π_2 termine pour x et fournit le même résultat que π_1. On retrouve une nouvelle forme d'isologie qui n'est pas symétrique (§ 2.6).

Exercice 42

Vérifier que

$$R_1 \cup R_2 = R_1 \Leftrightarrow R_1 \cap R_2 = R_2 \Leftrightarrow R_2 \subseteq R_1$$
$$R_1 ; (R_2 \cup R_3) = (R_1 ; R_2) \cup (R_1 ; R_3)$$
$$R_1 \subseteq R_2 \Rightarrow R_1 ; R \subseteq R_2 ; R \quad \text{et} \quad R ; R_1 \subseteq R ; R_2$$
$$\Omega \subseteq R \quad \text{et} \quad R \subseteq U .$$

Pour tout prédicat $\Omega \subseteq p$ et $p \subseteq E$. □

Exercice 43

a) Montrer que $R_1 ; (R_2 \cap R_3) \subseteq (R_1 ; R_2) \cap (R_1 ; R_3)$.
b) Montrer sur un contre-exemple que l'inclusion réciproque n'est pas vérifiée. □

Exercice 44

Donner des propriétés informatiques du programme spécifiant R sachant que R vérifie :

a) $R ; R^{-1} \subseteq E$
b) $E \subseteq R ; R^{-1}$. □

i) Itéré d'une relation

Soit R une relation, l'itéré de R noté R^* est par définition

$$R^* = I \cup R \cup R ; R \cup \ldots \cup \underbrace{R ; \ldots ; R}_{n \text{ fois}} \cup \ldots = \bigcup_{n \geqslant 0} R^n$$

où par convention $R^0 = E$ et $R^n = R^{n-1} ; R$.

Exercice 45

a) Montrer que « R transitive » est équivalent à $R ; R \subseteq R$.
b) En déduire que, si R est réflexive et transitive, $R^* = R$. □

L'opération d'itération va nous servir pour décrire les instructions <u>tant que</u>... <u>faire</u>... <u>fait</u>.

j) Description de l'instruction <u>tant que</u>... <u>faire</u>... <u>fait</u>

Soient un prédicat p et un programme π spécifiant une relation R. Le programme <u>tant que</u> p <u>faire</u> π <u>fait</u> consiste en l'exécution de π tant que p est vérifiée à l'entrée ; ce programme spécifie une relation entre x et y ainsi définie : il existe une suite $x_0, \ldots, x_n$ telle que $x = x_0$, $x_n = y$, $x_i (p ; R) x_{i+1}$ si $i < n$ et **non** p(y).

Autrement dit, il existe n tel que $x (p ; R)^n y$ et **non** p(y). Cela correspond à $x \bigcup_{n \geqslant 0} (p ; R)^n y$ et **non** p(y), ce qui est équivalent à

$$x [(p ; R)^* ; \bar{p}] y .$$

Le programme *tant que* p *faire* π *fait* spécifie la relation $(p\ ;\ R)^*\ ;\ \overline{p}$ que nous notons $p * R$ (on peut remarquer l'analogie de telles expressions avec les traces de programme (§ 2.6)).

Nous convenons que la priorité des opérateurs est, dans l'ordre décroissant, la suivante

$$[*\ ;\ \cap\ \cup].$$

Exercice 46

Justifier la convention : « la relation $R\ ;\ (\overline{p}\ ;\ R)^*\ ;\ p$ est spécifiée par le programme *repeter* π *jusqu'à* p ». □

Exercice 47

a) Montrer que

$$p * R = p\ ;\ R\ ;\ p * R \cup \overline{p}\,.$$

b) En déduire que *tant que* p *faire* π *fait* spécifie la même relation que

si p *alors* π ; *tant que* p *faire* π *fait* *fsi* . □

Au niveau des prédicats nous avons déjà assimilé le prédicat à la relation qui lui est associée ; nous ferons de même au niveau des programmes en assimilant le programme à la relation spécifiée par lui ; en toute rigueur, il ne faut pas confondre ces deux notions.

Signalons que nous n'avons pas encore envisagé deux types de constructions :

— **les appels de procédures** (cela viendra plus loin comme un outil essentiel) ;

— **les branchements** qui peuvent être simulés par des appels de procédure.

5.3 Approche axiomatique

Les propriétés algébriques des opérations sur les relations peuvent être présentées de manière axiomatique, ce qui offre les avantages suivants :

— On est ramené à un petit nombre de propriétés de base,

— la démonstration d'une propriété peut être mécanisée, offrant ainsi une ébauche d'automatisation de tels processus. Cela peut être une première étape vers des logiciels de preuves de propriétés de programmes.

Mais une telle approche peut être moins intuitive et rendre les démonstrations indigestes.

Les différents axiomes que l'on peut proposer pour formaliser le calcul sur les relations se divisent en trois classes : celle des treillis distributifs, celle des algèbres relationnelles et celle des algèbres régulières. Nous épargnerons au lecteur l'énoncé fastidieux des axiomes et des propriétés immédiates ou moins immédiates qui en découlent et le renverrons pour cela à quelques ouvrages ou articles spécialisés.

a) Axiomes des treillis distributifs

C'est certainement la partie de l'axiomatique la plus connue, elle remonte à BOOLE et De MORGAN vers le milieu du 19e siècle et le nom du premier lui est souvent

associé. Il s'agit de formaliser les propriétés des relations comme éléments de l'ensemble $\mathscr{P}(D \times D)$ des parties de $D \times D$. On pourra consulter à ce sujet le cours d'Algèbre de McLANE et BIRKHOFF [McL, 70] (ou pour plus de renseignements Lattice Theory de BIRKHOFF [BIR, 67]).

Exemples d'axiome

Idempotence de la réunion : $X = X \cup X$.
Consistance : *si* $X \subseteq Y$ et $Y \subseteq X$ *alors* $X = Y$.

b) Axiomes des algèbres relationnelles

TARSKI, le premier, dégagea ces axiomes. Ils formalisent la composition et l'inversion ainsi que leurs rapports avec les opérations des treillis distributifs. On trouvera les axiomes de TARSKI dans un article de de BAKKER et de ROEVER [BAK, 73] et une version légèrement différente dans Lattice Theory de BIRKHOFF déjà cité.

Exemple d'axiomes

1) Distributivité de l'inversion par rapport à la composition :

$$(X ; Y)^{-1} = Y^{-1} ; X^{-1} .$$

2) Règle sur Ω : si $(X ; Y) \cap Z = \Omega$ alors $(Z ; Y^{-1}) \cap X^{-1} = \Omega$.

c) Axiomes des algèbres régulières

Ces axiomes formalisent les propriétés de l'itération. Ils ont été étudiés surtout par CONWAY, qui a résumé ses résultats dans « Regular Algebras and Finite Machines » [CON, 71] à partir de l'étude des comportements des automates finis, des langages réguliers et particulièrement du théorème de KLEENE. CONWAY montre qu'il n'existe pas d'axiomatique finie d'où l'on puisse déduire toutes les propriétés de l'itération. Par conséquent, pour obtenir une axiomatique complète, il faudra soit poser une liste infinie d'axiomes, soit définir une règle d'inférence : c'est cette dernière voie que nous suivons. Cette règle qui est un avatar de la règle de récurrence sur les entiers aura l'avantage de s'appliquer à toutes les définitions récursives, et pas seulement à celle de l'itération ($R^* = E \cup R ; R^*$).

Exemple d'axiome de Conway

L'axiome qu'il nomme « union-étoile » s'écrit :

$$(X \cup Y)^* = (X^* ; Y)^* ; X^* .$$

Outre les axiomes que nous venons de suggérer, il nous faut, pour formaliser les propriétés des programmes, des axiomes sur les prédicats.

d) Axiomes sur les prédicats

Ils sont au nombre de trois et sont particulièrement simples

$$p \subseteq E , \quad \bar{p} \subseteq E , \quad p \cap \bar{p} = \Omega .$$

5.4 Approche opérationnelle

L'outil relationnel, qui déjà ne contient pas de branchements, serait vraiment pauvre s'il ne possédait pas une possibilité de définir récursivement des relations. Ce concept correspond à celui de procédure récursive dans la programmation. Comme on le fait au paragraphe 4 pour les schémas récursifs,

on construit les termes relationnels en utilisant les opérations ; et $\cup$, les constantes Ω, U, E, des symboles de relations constantes à interpréter et des symboles de relations inconnues notés $R_1, R_2, \ldots, R_n$.

a) Spécification d'une relation par un couple déclaration-programme

Poussant plus loin la comparaison, on peut très bien admettre que la spécification d'une relation par un programme (disons principal) fasse référence à des relations spécifiées ou définies récursivement ailleurs, dans une partie déclaration.

Définition 10

Un **schéma relationnel** est un couple $\pi = (\Delta, T)$ dans lequel T est un terme relationnel et Δ un ensemble de déclarations du type

$$\{ R_1 \Leftarrow T_1, \ldots, R_i \Leftarrow T_i, \ldots, R_n \Leftarrow T_n \}$$

où les T_i sont des termes relationnels et où $R_i \Leftarrow T_i$ se lit « R_i est récursivement défini par T_i ». □

Un terme S sera parfois noté $S(R_1, \ldots, R_n)$ pour préciser les occurrences des R_j figurant dans S ; la notation $S(S_1, \ldots, S_n)$ désigne alors le nouveau terme obtenu en remplaçant $R_1, \ldots, R_n$ par les termes $S_1, \ldots, S_n$.

Exemple 25

$$(\{ R_1 \Leftarrow (p \; ; A \; ; R_1) \cup \bar{p} \}, R_1)$$

$$(\{ R_1 \Leftarrow p \; ; R_2 \cup \bar{p} \, , R_2 \Leftarrow p \; ; A \; ; R_2 \cup \bar{p} \; ; R_1 \}, R_1)$$

sont des schémas relationnels. □

Exercice 48

Définir à l'aide d'une grammaire la syntaxe des termes relationnels. □

Exercice 49

On suppose que l'action élémentaire A est l'affectation $\mathbf{x} := \mathbf{a}(\mathbf{x})$. Donner des schémas récursifs monadiques qui correspondent intuitivement aux deux schémas relationnels de l'exemple 25. □

Exercice 50

Comparer les deux schémas de l'exemple 25. □

b) Calculs des schémas relationnels

Comme précédemment, pour définir des calculs, on interprète. Ici on donne une interprétation des symboles de relations constantes sur un domaine. Par abus d'écriture, on identifie, dans la suite, un symbole relationnel et son interprétation.

Exemple 26

Examinons le schéma $(\{ R_1 \Leftarrow p \; ; A \; ; R_1 \cup \bar{p} \}, R_1)$. Choisissons $\mathbb{N}^2$ comme domaine, et interprétons A par l'affectation $(x_1, x_2) \leftarrow (x_1 - 1, x_1 * x_2)$

et p par le prédicat $(x_1, x_2) \neq (0, x_2)$. Evaluons l'une des valeurs associées à (2, 1) par la relation ainsi définie.

Donc à (2, 1) appliquons R_1, ce qui revient (réécriture) à appliquer $p ; A ; R_1 \cup \bar{p}$. Choisissons l'une des branches du choix défini par l'union, disons la première : cela conduit à appliquer $p ; A ; R_1$. Dans la composition, appliquons d'abord p, nous obtenons (2, 1) car $2 \neq 0$ puis appliquant A on obtient (1, 2). Il reste à appliquer R_1, etc. Chaque étape est caractérisée par

— la valeur de (x_1, x_2) à ce moment
— l'expression de la relation qu'il reste à évaluer.

Nous y ajouterons la liste des différentes « opérations » déjà effectuées, c'est en quelque sorte la « trace » du calcul ; cette trace est utilisée au paragraphe 5 du chapitre 4.

Un calcul n'est réussi que si l'on arrive à E (c'est-à-dire plus rien) comme expression à évaluer. Ainsi on obtient :

Trace	Valeur	Expression à évaluer
Λ	(2, 1)	R_1
Λ	(2, 1)	$p ; A ; R_1 \cup \bar{p}$
Λ	(2, 1)	$p ; A ; R_1$
p	(2, 1)	$A ; R_1$
pA	(1, 2)	R_1
pA	(1, 2)	$p ; A ; R_1 \cup \bar{p}$
pA	(1, 2)	$p ; A ; R_1$
pAp	(1, 2)	$A ; R_1$
pAp	(0, 2)	R_1
pApA	(0, 2)	$p ; A ; R_1 \cup \bar{p}$
pApA	(0, 2)	$\bar{p}$
$pApA\bar{p}$	(0, 2)	E

Remarquons, que si à la troisième étape on avait choisi la deuxième branche de l'alternative on aurait obtenu (2, 1), $\bar{p}$ et on n'aurait pas pu obtenir un calcul réussi. □

Définition 11

Soit (Δ, T) un schéma relationnel interprété. Un couple (α, x, S) (α', x', S') est un **pas de calcul** s'il vérifie l'une des propriétés suivantes :

1) $S = A ; S'$ et A est une notation de relation constante (interprétée) telle que xAx' et $\alpha' = \alpha A$.

2) $S = A$, $S' = E$, xAx' et $\alpha' = \alpha A$.

3) $x = x'$ et $\alpha = \alpha'$ et soit S est une réunion contenant S' soit $S = R_i ; S''$ et $S' = T_i ; S''$ soit $S = R_i$ et $S' = T_i$. □

Définition 12

Un **calcul,** de donnée x_1, est une suite $(\alpha_1, x_1, S_1), \ldots, (\alpha_n, x_n, S_n)$ telle que S_1 est T, α_1 est vide et (α_i, x_i, S_i) $(\alpha_{i+1}, x_{i+1}, S_{i+1})$ est un pas de calcul.

Le calcul est réussi si $S_n = E$. S est la relation spécifiée par (Δ, T) si, pour chaque couple (x, y), xSy si et seulement s'il existe un calcul réussi

$$(\alpha_1, x_1, S_1), \ldots, (\alpha_n, x_n, S_n) \quad \text{tel que} \quad x_1 = x \quad \text{et} \quad x_n = y\,. \qquad \square$$

Notons qu'on a choisi la règle sémantique la plus simple, qui consiste à remplacer un appel d'une procédure par le texte de la procédure. Comme on applique cette règle à partir de la variable, cela correspond, pour les schémas récursifs, à l'évaluation de l'intérieur, c'est-à-dire à l'appel par valeur. On sait que, dans le cas monadique, c'est-à-dire qui nous concerne, cela donne les mêmes résultats que l'appel par nom.

c) Relation spécifiée par un programme et plus petit point fixe

En s'appuyant sur des méthodes analogues à celles décrites au paragraphe 3.4 on démontre que la relation que l'on vient de définir est la même que celle qui est définie en remplaçant chaque occurrence des R_i par la plus petite solution (au sens de l'inclusion) du système d'équations

$$R_1 \underset{\text{def}}{=} T_1, \ldots, R_n \underset{\text{def}}{=} T_n\,.$$

On sait que dans le cas où $n = 1$, elle est donnée par $\bigcup_{k\in} T_1^k(\Omega)$. On montre ainsi la coïncidence entre les sémantiques opérationnelle et du plus petit point fixe.

Exercice 51

Soit $(\{ R_1 \Leftarrow p \,;A\,; R_1 \cup \bar{p} \}, R_1)$ le schéma relationnel interprété de l'exemple 25. Calculer la relation spécifiée par les calculs et celle définie par plus petit point fixe. □

5.5 Approche dénotationnelle : le μ-calcul

La sémantique dénotationnelle déduit la signification des relations définies récursivement, à partir de leur notation. Il est utile pour cela que la notation de la relation ne fasse plus appel à une partie déclaration.

Chaque occurrence de R_i dans le programme principal T est remplacée par une expression qui s'écrit $\mu_i\, R_1, \ldots, R_n[T_1, \ldots, T_n]$ dénotant la 1-ième composante du plus petit point fixe du système $R_1 \underset{\text{def}}{=} T_1, \ldots, R_n \underset{\text{def}}{=} T_n$.

Exemple 27

Ainsi les programmes définis à l'exemple 25 s'écrivent :

$$\mu R.[p\,;A\,;R \cup \bar{p}]$$
$$\mu_1\, R_1\, R_2.[p\,;R_2 \cup \bar{p}\,, p\,;A\,;R_2 \cup \bar{p}\,;R_1]\,. \qquad \square$$

De la même manière que précédemment les propriétés des μ-termes, c'est-à-dire des notations, sont régies par une axiomatique. Ici les propriétés à déduire sont, outre celle du paragraphe 5.3, celle du plus petit point fixe pour chaque terme de la forme $\mu R.P$(abréviation de $\mu R[P]$). Nous nous limitons à ceux qui ne comportent qu'une variable pour ne pas compliquer inutilement l'exposé (pour le cas général, le lecteur pourra consulter de ROEVER [ROE, 74] ou HITCHCOCK et PARK [HIT, 72]).

a) *Axiome du point fixe*

$$T(\mu R.T[R]) \subseteq \mu R.T[R] .$$

Intuitivement cet axiome signifie que $\mu R.T[R]$ est l'une des solutions de l'équation $T(R) \subseteq R$.

b) *Règle de Scott : forme simplifiée*

En comparant $R \Leftarrow T$ à une déclaration de procédure, pour démontrer une propriété sur $\mu R.T$ on procède par récurrence sur les appels de R dans T. Ce que l'on appelle propriété dans le μ-calcul est une suite d'inégalités entre termes. Une propriété Φ portant sur la variable R est notée $\Phi(R)$. La règle d'inférence de Scott dans le cadre du μ-calcul s'énonce ainsi : si $\Phi(R)$ est une propriété du μ-calcul, si $\Phi(\Omega)$ est vrai et si de $\Phi(R)$ on déduit $\Phi(T(R))$ alors $\Phi(\mu R.T(R))$ est vrai. Ceci peut s'écrire comme une règle d'inférence

$$\frac{\Phi(\Omega),\ \Phi(R)\ \models\ \Phi(T(R))}{\Phi(\mu R.T(R))}$$

Nous allons utiliser la règle de Scott pour montrer deux propriétés importantes :

Proposition 4

$\mu R.T(R)$ est un point fixe de T, c'est-à-dire $\mu R.T(R) = T(\mu R.T(R))$. □

Démonstration : Il suffit d'après l'axiome du point fixe de montrer que $\mu R.T(R) \subseteq T(\mu R.T(R))$; d'après la règle de Scott il faut montrer $\Omega \subseteq P(\Omega)$ (c'est évident) et que de $R \subseteq T(R)$ on déduit $T(R) \subseteq T(T(R))$; c'est une conséquence du lemme suivant qui se démontre par récurrence sur la longueur des termes.

Lemme : Tout terme du calcul relationnel est croissant autrement dit de $S \subseteq S'$. On déduit $T(S) \subseteq T(S')$.

Proposition 5 (Propriété de PARK)

La plus petite solution de l'inéquation $T(S) \subseteq S$ est le terme $\mu R.T(R)$ autrement dit de $T(S) \subseteq S$ on déduit $\mu R.T(R) \subseteq S$. □

Démonstration : Appliquons la règle de Scott à la propriété $R \subseteq S$; on a $\Omega \subseteq S$. De $R \subseteq S$ on déduit d'après la croissance des termes $T(R) \subseteq T(S)$ et l'hypothèse $T(S) \subseteq S$ donne par transitivité $T(R) \subseteq S$.

c) *Trois exemples d'application de la règle de Scott*

Exemple 28 (où l'on montre que la solution d'une équation ne dépend pas du nom de l'inconnue).

Montrons que $\mu R.T(R) = \mu S.T(S)$. Par symétrie, il suffit de montrer que $\mu R.T(R) \subseteq \mu S.T(S)$.

- $\Omega \subseteq \mu S.T(S)$ est immédiat ;
- si $R \subseteq \mu S.T(S)$ alors $T(R) \subseteq T(\mu S.T(S))$ (croissance des termes). D'où $T(R) \subseteq \mu S.T(S)$ par l'axiome du point fixe. On en déduit le résultat par la règle de Scott.

Exemple 29 (une propriété d'idempotence).

Nous nous proposons maintenant de montrer que la fonction définie par le schéma de programme récursif

$$F(x) \Leftarrow \underline{si}\ p(x)\ \underline{alors}\ x\ \underline{sinon}\ F(F(a(x)))\ fsi$$

est idempotente, autrement dit que $F \circ F = F$.

Traduisons cet énoncé en termes de μ-calcul en remarquant que $F \circ F \circ a$ s'écrit $a\ ;\ F\ ;\ F$ et que la conditionnelle se traduit par une union.

La fonction définie par le schéma est donc représentée par le terme

$$T(F) = p \cup \bar{p}\ ;\ a\ ;\ F\ ;\ F\ .$$

Il faut montrer que son plus petit point fixe est idempotent autrement dit que :

$$(1) \qquad (\mu F.T(F))\ ;\ (\mu F\ ;\ T(F)) = \mu F.T(F)\ .$$

Posons :

$$Q(F) = F\ ;\ \mu G.T(G)\ .$$

Le résultat (1) s'écrit encore :

$$Q(\mu F.T(F)) = \mu F.T(F)\ .$$

Nous allons prouver cette égalité par application de la règle de Scott ; il suffit de montrer :

$$(2) \qquad Q(\Omega) = \Omega$$

$$(3) \qquad \text{de } Q(F) = F \quad \text{on déduit} \quad Q(T(F)) = T(F)\ .$$

La preuve de (2) est immédiate en effet :

$$\Omega\ ;\ \mu G.T(G) = \Omega\ .$$

Pour montrer (3), supposons que $Q(F) = F$ autrement dit que

$$F\ ;\ \mu G.T(G) = F$$

et démontrons que $Q(T(F)) = T(F)$.

En effet :

$$Q(T(F)) = (p \cup \bar{p}\ ;\ a\ ;\ F\ ;\ F)\ ;\ \mu G.T(G)$$

$$= p\ ;\ \mu G.T(G) \cup \bar{p}\ ;\ a\ ;\ F\ ;\ F\ ;\ \mu G.T(G)\ .$$

$$p\ ;\ \mu G.T(G) = p\ ;\ (p \cup \bar{p}\ ;\ a\ ;\ \mu G.T(G)\ ;\ \mu G.T(G))$$

$$= p \quad \text{car} \quad p\ ;\ \bar{p} = \Omega\ .$$

Donc

$$Q(T(F)) = p \cup \bar{p}\ ;\ a\ ;\ F\ ;\ F\ ;\ \mu G.T(G)$$

en utilisant l'hypothèse de récurrence on a :

$$Q(T(F)) = p \cup \bar{p}\ ;\ a\ ;\ F\ ;\ F = T(F)\ .$$ □

Exercice 52

Considérons les deux fonctions définies par les équations :

$$F(x) \Leftarrow \underline{si}\ p(x)\ \underline{alors}\ x\ \underline{sinon}\ F(a(x))\ \underline{fsi}$$
$$G(x) \Leftarrow \underline{si}\ p(x)\ \underline{alors}\ b(x)\ \underline{sinon}\ G(a(x))\ \underline{fsi}\,.$$

1) Donner les termes qui leur correspondent.

2) Prouver que $b \circ F = G$. □

Exemple 30

Nous allons prouver la propriété suivante :
Si T_1 et T_2 sont deux termes tels que, pour un entier i :

a) $T_1(\Omega) = T_2(\Omega)$.
b) $T_1^i(T_2) = T_2(T_1)$

alors ces deux programmes définissent la même relation, dans le sens suivant :

$$\mu R.T_1(R) = \mu R.T_2(R)$$

(cette propriété a déjà été étudiée dans l'exercice 22 du chapitre 2).

Inclusion : $\mu R.T_2(R) \subseteq \mu R.T_1(R)$

Lemme $T_2(\mu R.T_1(R)) \subseteq \mu R.T_1(R)$.

On applique la règle de Scott à l'inclusion $T_2(S) \subseteq \mu R.T_1(R)$. En effet on a :

$$\Omega \subseteq \mu R.T_1(R)$$

donc :

$$T_2(\Omega) = T_1(\Omega) \subseteq T_1(\mu R.T_1(R)) = \mu R.T_1(R)\,.$$

Supposons que $T_2(S) \subseteq \mu R.T_1(R)$; alors

$$\begin{aligned} T_2(T_1(S)) &= T_1^i(T_2(S)) && \text{(hypothèse b)} \\ &\subseteq T_1^i(\mu R.T_1(R)) && \text{(hypothèse de récurrence et croissance)} \\ &= \mu R.T_1(R) && \text{(propriété du point fixe)}\,. \end{aligned}$$

L'inclusion $\mu R.T_2(R) \subseteq \mu R.T_1(R)$ est une conséquence de la propriété de Park, appliquée au lemme.

Inclusion : $\mu R.T_1(R) \subseteq \mu R.T_2(R)$

Nous prendrons pour hypothèse de récurrence la propriété :

$$\Phi(S) \begin{cases} 1.\ S \subseteq T_1^{i-1}(S) \\ 2.\ T_1(S) \subseteq T_2(S) \\ 3.\ S \subseteq \mu R.T_2(R) \end{cases}$$

$\Phi(\Omega)$ est immédiate ; montrons que $\Phi(S)$ entraîne

$$\Phi(T_1(S)) \begin{cases} 1'.\ T_1(S) \subseteq T_1^i(S) \\ 2'.\ T_1(T_1(S)) \subseteq T_2(T_1(S)) \\ 3.\ T_1(S) \subseteq \mu R.T_2(R)\,. \end{cases}$$

1'. est immédiat à partir de 1 et de la croissance de T_1.

$$\begin{aligned} 2'.\ T_1(T_1(S)) &\subseteq T_1^{i+1}(S) && \text{(hypothèse 1 de } \Phi(S) \text{ et croissance)} \\ &= T_1^i(T_1(S)) && \\ &\subseteq T_1^i(T_2(S)) && \text{(hypothèse 2 de } \Phi(S)) \\ &= T_2(T_1(S)) && \text{(hypothèse b)}\ . \end{aligned}$$

$$\begin{aligned} 3'.\ T_1(S) &\subseteq T_2(S) && \text{(hypothèse 2 de } \Phi(S)) \\ &\subseteq T_2(\mu R.T_2(R)) && \text{(hypothèse 3 de } \Phi(S)) \\ &= \mu R.T_2(R) && \text{(propriété du point fixe)} \end{aligned}$$

on a par conséquent

$$\Phi(\mu S.T_1(S)) \begin{cases} 1.\ \mu S.T_1(S) \subseteq T_1^{i-1}(\mu S.T_1(S)) \\ 2.\ T_1(\mu S.T_1(S)) \subseteq T_2(\mu S.T_2(S)) \\ 3.\ \mu S.T_1(S) \subseteq \mu R.T_2(R)\ . \end{cases}$$

L'inclusion 3 est le résultat demandé.

6 COMMENTAIRES BIBLIOGRAPHIQUES

Les schémas de programmes ont suscité de nombreuses études, dont nous ne pouvons donner ici un compte rendu exhaustif.

Concernant les schémas avec branchements, l'article le plus ancien, qui fait figure de pionnier, est celui de IANOV [IAN, 60] ; il décrit les schémas auxquels nous avons donné son nom ; sur les schémas avec branchements, l'article de LUCKHAM, PARK et PATERSON [LUC, 70] est un des plus importants. On trouve des présentations détaillées dans les ouvrages de GREIBACH [GRE, 75], d'ENGELFRIET [ENG, 74] (sur les schémas d'Ianov et ceux qui leur ressemblent) et MANNA [MAN, 74]. Le livre d'ENGELER [ENG, 73] est une approche intéressante de la décidabilité et de la calculabilité à l'aide des programmes sur les entiers.

L'étude des schémas itératifs tire son origine de la célèbre lettre de DIJKSTRA « Goto statement considered as harmfull » (« les aller à considérés comme dangereux ») [DIJ, 68] (voir aussi [DAH, 72]). On trouvera des résultats plus détaillés que ceux présentés ici dans ASHCROFT et MANNA [ASH, 71], PETERSON, KASAMI et TOKURA [KAS, 73], TAKUMI KASAI [TAK, 74] (pour les schémas sans variable), KOSARAJU R. S. [KOS, 74] et ARSAC [ARS, 74]. L'article de LEDGARD et MARCOTTY [LED, 75] est une synthèse sur le sujet et comporte une importante bibliographie. Il est amusant de constater que l'important théorème de BÖHM et JACOPINI [BOH, 66] est très antérieur à ces travaux.

En ce qui concerne les schémas récursifs, on peut lire le livre de MANNA [MAN, 74] qui présente dans un même chapitre les théories du plus petit point fixe, des schémas récursifs et des preuves de programmes récursifs. Dans un article, VUILLEMIN [VUI, 74] présente en détail la règle du report et justifie son caractère optimal. On trouve également une étude des règles sûres. VUILLEMIN, comme MANNA, s'est intéressé uniquement aux

fonctionnelles sur le domaine D^+. De ROEVER [ROE, 74] rapproche les règles d'appel par valeur et d'appel par nom. Certains aspects ont été ignorés ici ; le lecteur peut se reporter en particulier à l'ouvrage de GREIBACH [GRE, 75] ou à celui de NIVAT, COURCELLE et COUSINEAU [COU, 78]. Il faut également citer les travaux récents de COURCELLE [COU, 76] qui obtient des résultats de décidabilité et d'équivalence pour certaines classes de schémas récursifs. Notons enfin qu'une fonction définie récursivement peut être calculée autrement que par substitution. Il est possible de calculer les valeurs de Φ par récurrence sur la complexité des arguments. Ainsi n ! se calcule naturellement par récurrence sur n. BERRY [BER, 75] développe ces idées et obtient en particulier un programme itératif calculant la fonction d'Ackermann.

Le calcul relationnel récursif ou μ-calcul apparaît pour la première fois dans deux articles parus dans les comptes rendus du premier colloque Automata, Languages and Programming en 1972 ; ils sont dus à HITCHCOCK et PARK [HIT, 72] d'une part et à de BAKKER et de ROEVER [BAK, 72] d'autre part. De ROEVER a étudié le calcul relationnel à plusieurs types d'objets [ROE, 74]. PARK [PAR, 76] a situé la puissance du calcul relationnel exactement entre le calcul des prédicats du premier ordre et celui du second ordre. ENGELFRIET [ENG, 74] compare le μ-calcul aux autres schémas de programmes.

7 SOLUTION DES EXERCICES

Exercice 1

L'algorithme est décrit par la procédure récursive suivante proposée en langage de type algol ; '⊣' est le signe qui termine l'expression ; son arité n'est pas définie.

```
procédure parenthéser
    var sym : caractère
    début
      lire(sym) ; écrire(sym) ;
      si ar(sym) non défini alors erreur fsi ;
      si ar(sym) ≠ 0 alors écrire('(');
                           pour i de 1  jusqu'à ar(sym) − 1
                                        faire
                                        parenthéser ; écrire(',')
                                        fait ;
                           parenthéser ;
                           écrire(')')
                     fsi
    fin .
```

Exercice 2

a) x , $f(x)$, $f(f(x))$, ...

b) x , $f(x)$, $g(x)$, $f(f(x))$, $f(g(x))$, $g(f(x))$, $g(g(x))$, ...

c) x_1 , $f(x_2, x_1)$, $f(x_{44}, f(x_1, x_2))$, ... Il s'agit d'arbres binaires dont les nœuds sont étiquetés par f et les feuilles par x_i ($i \in \mathbb{N}$).

d) x , y , $f(g(y), f(g(x), y))$.

e) Il n'y a pas de schémas.

f) g , $f(g, g)$, $f(f(g, g), g)$.

Le dessin des arborescences associées est immédiat.

Exercice 3

$I[g(x, x, y)] = x + 2\,xy$

$I[h(g(x, x, y))] = 5(x + 2\,xy) = 5\,x + 10\,xy$

$I[g(x, f(x, y), y)] = x + 2(x + y)\,y$

$$I[f(g(x, f(x, y), y), h(g(x, x, y)))] = x + 2(x + y)\,y + 5\,x + 10\,xy$$
$$= 6\,x + 12\,xy + 2\,y^2\,.$$

Exercice 4

$\mathscr{X} = \{ x_1, x_2 \}$, $\mathscr{F} = \{ f, g, h \}$, $\underline{\text{ar}}(f) = 0$, $\underline{\text{ar}}(g) = 2$, $\underline{\text{ar}}(h) = 1$, $\mathscr{P} = \{ p \}$.

Exercice 5

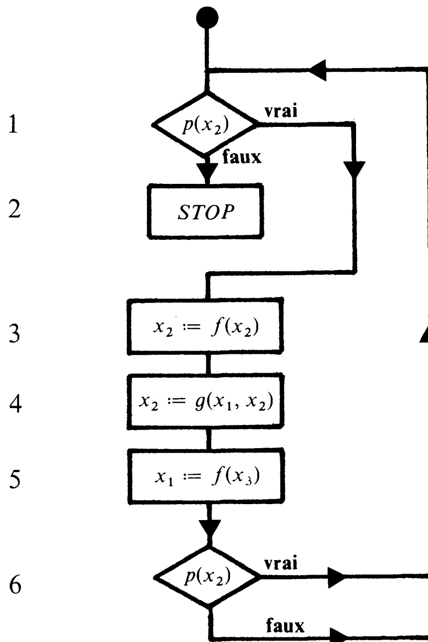


Figure 25.

Exercice 6

Le programme calcule la fonction carré : carré(x) = x^2.

Exercice 7

Pour toute interprétation I_0 telle que $I_0[p]$ soit identiquement faux, le programme ne termine jamais. Pour $I_0[h]$ (x) = x et $I_0[p]$ (x) = (x est pair), le programme termine si et seulement si x est pair.

Exercice 8

Pour le calcul du pgcd on peut prendre :

$$I_0[p]\,(x) = (x = 0)\ ;$$
$$I_0[f]\,(x) = x\ ;$$
$$I_0[g]\,(x, y) = x\ \underline{\text{mod}}\ y \quad \text{(reste de la division entière de x par y)}\,.$$

Pour obtenir un programme ne terminant jamais il suffit de prendre

$$I_0[p]\,(x) = \text{faux}\,.$$

Enfin avec $D = \mathbb{Z}$ et

$$I_0[p]\,(x) = (x = 0)\ ;$$
$$I_0[g]\,(x, y) = y - 2\ ;$$

le programme ne termine que si la deuxième composante de l'initialisation est paire et positive.

Exercice 9

$$(1, (\Lambda, \text{abcd})), (2, (\Lambda, \text{abcd})), (3, (\Lambda, \text{abcd})), (4, (\text{a}, \text{abcd}))\,,$$
$$(5, (\text{a}, \text{bcd})), (3, (\text{a}, \text{bcd})), (4, (\text{ba}, \text{bcd})), (5, (\text{ba}, \text{cd}))\,,$$
$$(3, (\text{ba}, \text{cd})), (4, (\text{cba}, \text{cd})), (5, (\text{cba}, \text{d})), (3, (\text{cba}, \text{d}))\,,$$
$$(4, (\text{dcba}, \text{d})), (5, (\text{dcba}, \Lambda)), (6, (\text{dcba}, \Lambda))\,.$$

Exercice 10

Il n'y a pas de calcul réussi ; en effet, toute séquence d'exécution est constituée de la seule instruction 1. La fonction $I[\pi]$ est la fonction nulle part définie.

Exercice 11

1) On a $(\sigma, x) \xrightarrow{I} (\sigma', x')$ si et seulement si l'une des conditions suivantes est vérifiée.

(a) L'instruction ε_σ de numéro σ est une affectation :

$$\varepsilon_\sigma = x_i := f(x_{j(1)}, \ldots, x_{j(m)})$$
$$x'_k = x_k \quad \text{pour} \quad k \neq i\,,$$
$$x'_i \in I_0[f]\,(x_{j(1)}, \ldots, x_{j(m)})\,,$$
$$\sigma' = \sigma + 1\,.$$

(b)
(c) } Mêmes conditions que dans le cas déterministe (voir 3.2.3).

La définition d'un calcul est la même que dans le cas déterministe ; cependant ici, pour un schéma π, $I[\pi]$ est une fonction partielle de D^n dans l'ensemble des parties de D^n.

2) Le programme calcule les sous-listes d'une liste.

Exercice 12

On se place dans $\mathrm{sch}(\{f, g, h\}, \{x_1, x_2\})$ et on pose

$$I'_0[p](u) = (\varphi(u) = 0)$$

où $\varphi(u)$ est l'interprétation du schéma fonctionnel obtenu en prenant les interprétations de l'exemple 3, en interprétant x_1 par 7 et x_2 par 2. Ce calcul associé est :

$$(1, (x_1, x_2)), \quad (2, (f, x_2)), \quad (3, (f, x_2)), \quad (4, (g(x_2, f), x_2)),$$
$$(5, (g(x_2, f), h(x_2))), \quad (3, (g(x_2, f), h(x_2))),$$
$$(4, (g(h(x_2), g(x_2, f)), h(x_2))),$$
$$(5, (g(h(x_2), g(x_2, f)), h(h(x_2)))),$$
$$(6, (g(h(x_2), g(x_2, f)), h(h(x_1)))).$$

Exercice 13

Un schéma possible est

1 $x := f(x)$
2 $q(x)$ 5 , 2
3 $x := g(x)$
4 $q(x)$ 5 , 2
5 *STOP* .

En prenant comme interprétation sur le domaine $\mathbb{N}^2$

$$I[f](x_1, x_2) = (1, x_2) ;$$
$$I[g](x_1, x_2) = (x_1 * x_2, x_2 - 1) ;$$
$$I[q](x_1, x_2) = (x_2 = 0).$$

Exercice 14

$$L(P) = (1\,h\,2\,k\,3\,q\,5\,h\,6\,k\,7\,q)^* \,(1\,h\,2\,k\,3\,\bar{q}\,4\,STOP \cup 1\,h\,2\,k\,3\,q\,5\,h\,6\,k\,7\,\bar{q}\,8\,STOP).$$

On en déduit facilement

$$\mathrm{Tr}(P) = (hkqhkq)^* \,(hk\bar{q} \cup hkqhk\bar{q}),$$

qui se simplifie en

$$\mathrm{Tr}(P) = hk(qhk)^* \,\bar{q}$$

ce qui est la trace du schéma de l'exemple (§ 2.5).

Exercice 15

Il est facile d'associer à tout schéma de Ianov π un automate fini A reconnaissant des mots de $(\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}})^*$:

— les états de A sont les instructions du schéma plus un état de rejet r,
— l'état initial est la première instruction,
— les états finals sont les instructions *STOP*,
— les transitions sont définies de la façon suivante :

si s est du type i*h*
 s.h est l'instruction de rang i + 1,
 s.x = r pour $x \neq h$;
si s est du type i*q*j , j′,
 s.q est l'instruction de rang j,
 s.$\overline{q}$ est l'instruction de rang j′,
 s.x = r pour $x \neq q$ ou $x \neq q'$,
enfin r.x = r pour tout x.

Le langage reconnu par cet automate fini est régulier. En prenant l'intersection de ce langage avec le complémentaire du langage régulter suivant

$$(\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}})^* . \bigcup_{p \in \mathscr{P}} (p(\mathscr{P} \cup \overline{\mathscr{P}})^* \, \overline{p} \cup \overline{p}(\mathscr{P} \cup \overline{\mathscr{P}})^* \, p).(\mathscr{F} \cup \mathscr{P} \cup \overline{\mathscr{P}})^*$$

on obtient la trace qui est donc un langage régulier.

Exercice 16

On transforme π_3 en π_1 par étapes en conservant l'isologie forte ; on utilise le fait qu'on peut permuter affectations et tests qui ne font pas intervenir les

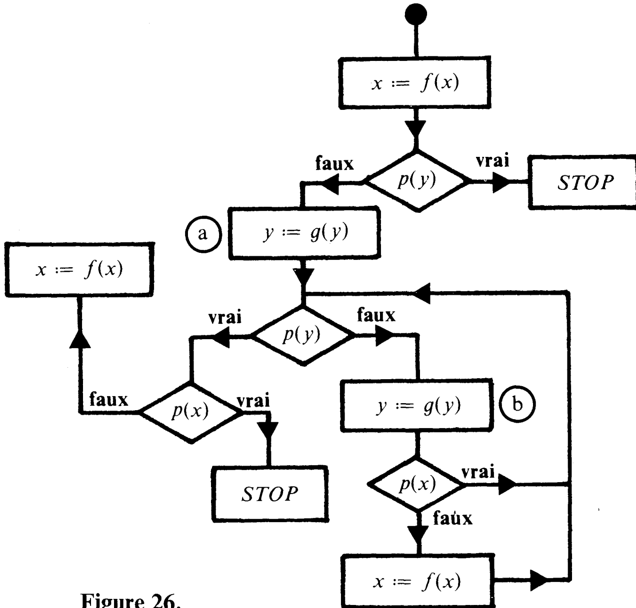


Figure 26.

mêmes variables. Dans π_3, à la sortie du premier test $p(y)$, on effectue dans tous les cas l'affectation $x := f(x)$; on peut sans inconvénient l'effectuer avant : on obtient un organigramme qui est trivialement fortement isologue à π_3 (figure 26).

De même l'affectation $y := g(y)$ étiquetée par ⓑ permute avec le test et l'affectation éventuelle qui suit, ce qui permet de se brancher ensuite, avant l'affectation étiquetée par ⓐ , tout en gardant un programme fortement isologue à π_3 (figure 27).

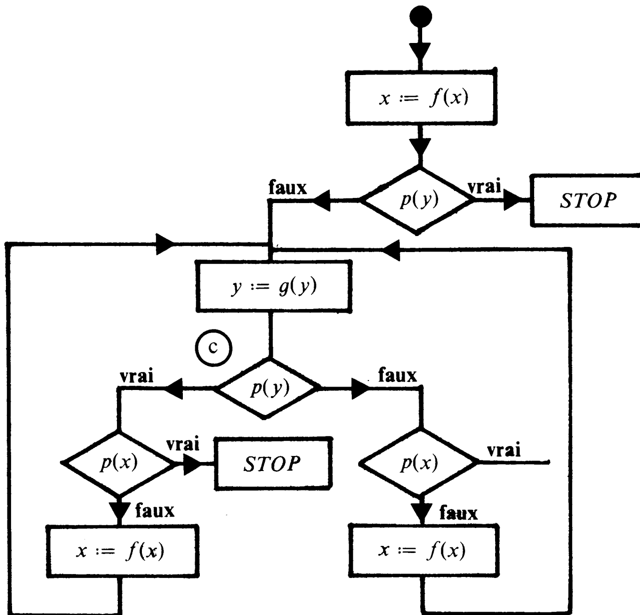


Figure 27.

Le test $p(y)$ étiqueté par ⓒ permute avec les tests $p(x)$ qui suivent (figure 28).

Le test étiqueté par ⓔ est inutile ; celui étiqueté par ⓕ peut être remplacé par un branchement en ⓓ ; on obtient alors π_1. Il est facile de voir que si l'on prend pour π_8 une interprétation libre telle que

$$\mathrm{I}[p]\,(y) = \textbf{faux}\,, \quad \mathrm{I}[p]\,(f(x)) = \textbf{vrai} \quad \text{et} \quad \mathrm{I}[p]\,(g(y)) = \textbf{vrai}$$

on a

$$\mathrm{I}[\pi_7]\,(x, y) = (f(f(x)), g(y))$$

et
$$\mathrm{I}[\pi_1]\,(x, y) = (f(x), g(y))\,.$$

Exercice 17

a) Il est immédiat que pour toute initialisation d on a $\mathrm{I}[\pi]\,(\mathrm{d}) = 1$ et si $\mathrm{I}[\rho]\,(\mathrm{d})$ est défini alors $\mathrm{I}[\rho]\,(\mathrm{d}) = 1$; il reste à montrer que $\mathrm{I}[\pi]\,(\mathrm{d})$ est tou-

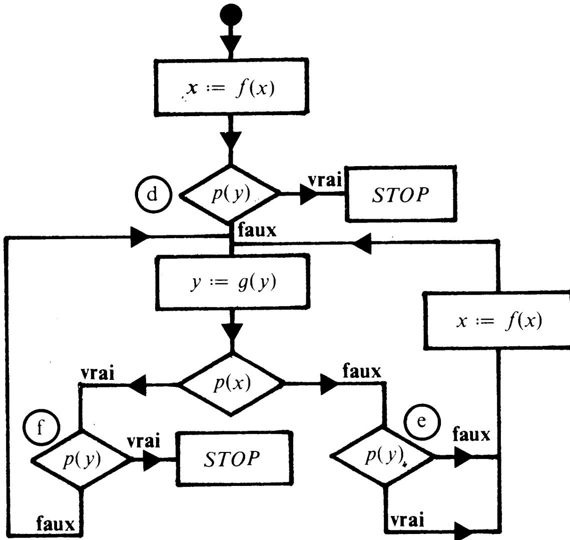


Figure 28.

jours définie en employant des méthodes décrites au chapitre 4 (§ 2). En effet, entre deux passages successifs en 2 x, décroît strictement et d'autre part x est toujours supérieur ou égal à 1, il atteint donc la valeur 1 en un nombre fini de pas de calcul.

b) Si $I[\pi]$ (d) $= 1$ et si $I[\rho]$ (d) est défini, $I[\rho]$ (d) vaut 1, mais le problème reste ouvert de savoir si $I[\rho]$ (d) est défini. En effet x ne décroît pas nécessairement entre deux passages par l'instruction 2.

Exercice 18

Considérons les deux schémas de programme suivants (figure 29)

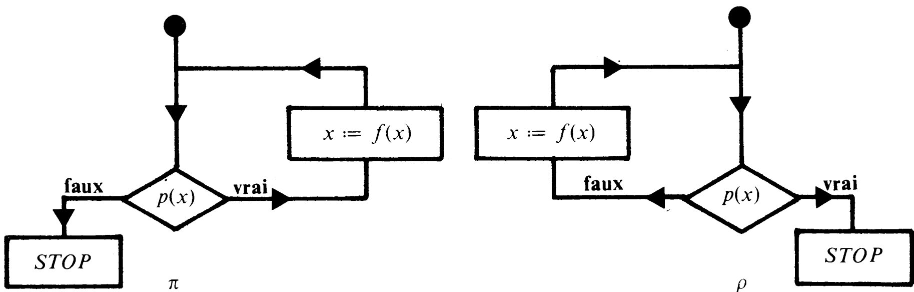


Figure 29.

de façon évidente $\Lambda(\pi) = \Lambda(\rho) = f^*$ et pourtant π n'est pas équivalent à ρ.

Exercice 19

Si x = y = z = t, $I[\pi]\,(\overline{xyzt}) = 0000$ sinon $I[\pi]\,(\overline{xyzt}) = 6\,174$.

On voit d'abord que $I[a\,;b]\,(6\,174) = 6\,174$. On peut vérifier ensuite par programme ou par un raisonnement par cas que, pour tous les nombres à quatre chiffres $I[\pi]\,(\overline{xyzt}) = 6\,174$.

Exercice 20

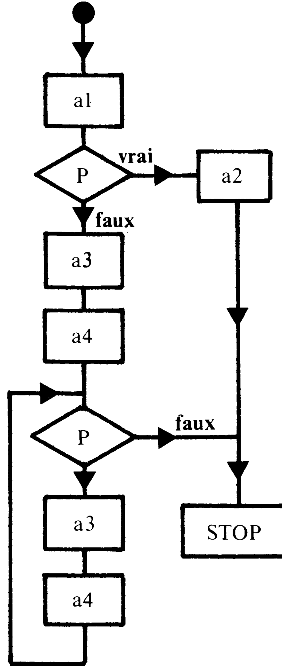


Figure 30.

On retrouve la forme générale du schéma de l'exemple 1 (figure 30).

Exercice 21

x := *premier caractère* ; *l* := 0 ;
<u>*tant que*</u> (*x* ≠ # <u>*et*</u> <u>*non*</u> (*x incorrect*)) <u>*faire*</u> *l* := *l* + 1 ;
x := *caractère suivant*
<u>*fait*</u> ;
<u>*si*</u> *x incorrect* <u>*alors*</u> *imprimer erreur*
<u>*sinon*</u> *imprimer l* <u>*fsi*</u>.

On remarque qu'un test supplémentaire *x incorrect* a été ajouté.

Exercice 22

Si $\pi = a$ alors :

$$\text{res}(\text{cal}(a, \text{d}), \text{d}) = \text{res}(a, \text{d}) = \text{I}[a]\,(\text{res}(\Lambda, \text{d})) = \text{I}[a]\,(\text{d})\,.$$

Dans le cas de la composition sérielle, en appliquant deux fois la définition de cal, on a :

$$\begin{aligned}
&\mathrm{res}(\mathrm{cal}(\alpha\,;\,a,\ \mathrm{d}),\ \mathrm{d})\\
&= \mathrm{res}(\mathrm{cal}(\alpha,\ \mathrm{d})\ \mathrm{cal}(a,\ \mathrm{res}(\mathrm{cal}(\alpha,\ \mathrm{d}),\ \mathrm{d})),\ d)\,,\\
&= \mathrm{res}(\mathrm{cal}(\alpha,\ \mathrm{d})\ a,\ \mathrm{d})\,,\\
&= \mathrm{I}[a]\ (\mathrm{res}(\mathrm{cal}(\alpha,\ \mathrm{d}),\ \mathrm{d})) \quad \text{en appliquant la définition de res}\,,\\
&= \mathrm{I}[a]\ (\mathrm{I}[\alpha]\ (\mathrm{d})) \quad \text{par hypothèse de récurrence}\,,\\
&= \mathrm{I}[\alpha\,;\,a]\ (\mathrm{d}) \quad \text{par définition de I}\,.
\end{aligned}$$

Pour la suite on utilise un résultat facile à démontrer par récurrence sur la longueur des calculs :

Lemme : si γ et δ sont des calculs, $\mathrm{res}(\gamma\delta,\ \mathrm{d}) = \mathrm{res}(\delta,\ \mathrm{res}(\gamma,\ \mathrm{d}))$.

Dans le cas de la conditionnelle supposons $\mathrm{I}[p]\ (\mathrm{d}) =$ **vrai** (le cas $\mathrm{I}[p]\ (\mathrm{d}) =$ **faux** se traite de façon analogue) :

$$\begin{aligned}
&\mathrm{res}(\mathrm{cal}(\underline{si}\ p\ \underline{alors}\ \alpha\ \underline{sinon}\ \beta\ \underline{fsi},\ \mathrm{d}),\ \mathrm{d}) = \mathrm{res}(\mathrm{p}\ \mathrm{cal}(\alpha,\ \mathrm{d}),\ \mathrm{d})\,,\\
&= \mathrm{res}(\mathrm{p},\ \mathrm{res}(\mathrm{cal}(\alpha,\ \mathrm{d}),\ \mathrm{d})) \quad \text{d'après le lemme}\,,\\
&= \mathrm{res}(\Lambda,\ \mathrm{res}(\mathrm{cal}(\alpha,\ \mathrm{d}),\ \mathrm{d}) \quad \text{d'après la définition de res}\,,\\
&= \mathrm{res}(\mathrm{cal}(\alpha,\ \mathrm{d}),\ \mathrm{d}) \quad \text{d'après le lemme}\,,\\
&= \mathrm{I}[\alpha]\ (\mathrm{d}) \quad \text{d'après l'hypothèse de récurrence}\,,\\
&= \mathrm{I}[\underline{si}\ p\ \underline{alors}\ \alpha\ \underline{sinon}\ \beta\ \underline{fsi}]\ (\mathrm{d})\,.
\end{aligned}$$

Le cas *tant que*, plus technique, ne sera pas traité ici.

Exercice 23

Toutes ces relations sont définies par des égalités d'ensembles ou de fonctions définies à partir des programmes, ce sont donc trivialement des relations d'équivalences.

Exercice 24

La correspondance est même stricte ; les cycles des BJ_1-organigrammes sont exactement de la même forme que ceux des $\mathscr{D}$-organigrammes.

L'organigramme de la figure 14 est un BJ_2-organigramme où α_1 est l'instruction vide ; il a déjà été dit que ce n'est pas un $\mathscr{D}$-schéma. De plus il est impossible de le réduire à un $\mathscr{D}$-schéma.

Exercice 25

Traitons seulement le cas des itérations

$$\begin{aligned}
&\mathrm{I}[\underline{faire}\,p \to \alpha\,;\,q \to \beta\ \underline{fait}]\ (\mathrm{d}) =\\
&\qquad \text{SI } \mathrm{I}_0[p]\ (\mathrm{d}) \text{ ALORS}\\
&\qquad\qquad \text{SI } \mathrm{I}_0[q]\ (\mathrm{I}[\alpha]\ (\mathrm{d})) \text{ ALORS } \mathrm{I}\ [\alpha\,;\,\beta\ \underline{faire}\ p \to \alpha\,;\,p \to \beta\ \underline{fait}]\ (\mathrm{d})\\
&\qquad\qquad\qquad\qquad \text{SINON } \mathrm{I}[\alpha]\ (\mathrm{d})\\
&\qquad\quad \text{SINON d}\,.
\end{aligned}$$

Trois cas sont à envisager :

— $I_0[p]$ (d) = faux : cal(*faire* $p \to \alpha ; q \to \beta$ *fait*, d) = Λ ;
— $I_0[q]$ (I[α] (d)) = faux : cal(*faire* $p \to \alpha ; q \to \beta$ *fait*, d) = cal(α, d) ;
— $I_0[q]$ (I[α] (d)) = vrai : cal(*faire* $p \to \alpha ; q \to \beta$ *fait*, d) =
cal(α, d) cal(β, res(cal(α, d), d))
cal(*faire* $p \to \alpha ; q \to \beta$ *fait*, res(cal($\alpha ; \beta$, d), d)).

Exercice 26

Le schéma de programme

faire $p \to \alpha ; q \to \beta$ *fait*

a les mêmes calculs que le schéma

faire si p *alors* α ; *si* q *alors* β *sinon exit* (1) *fsi sinon exit* (1) *fsi fait*

De même plus généralement

faire $p_1 \to \alpha_1 ; \ldots ; p_n \to \alpha_n$ *fait*

a les mêmes calculs que

faire si p_1 *alors* α_1 *si* p_2 *alors* α_2 ... *sinon exit* (1) *fsi sinon exit* (1) *fsi fait*.

Il est immédiat qu'il ne peut y avoir équivalence car les RE_1-schémas permettent de simuler tous les BJ_n-schémas quel que soit n.

Exercice 27

Les justifications renvoient aux alinéas de la définition 2.

1) $I^\varepsilon[\varphi(x)]$ (0) = I^ε[*si* $p(x)$ *alors* a *sinon* $h(x, \varphi(g(x)))$ *fsi*] (0)
d'après a) et c) ;
= SI 0 = 0 ALORS 1 SINON $I^\varepsilon[h(x, \varphi(g(x)))]$ (0)
d'après d) et e) ;
= 1.

2) Soit n > 0
$I^\varepsilon[\varphi(x)]$ (n) = SI n = 0 ALORS 1 SINON $I^\varepsilon[h(x, \varphi(g(x)))]$ (n)
= $I^\varepsilon[h(x, \varphi(g(x)))]$ (n).

3) $I^\varepsilon[h(x, \varphi(g(x)))]$ (n) = $I_0[h]$ ($I^\varepsilon[x]$ (n), $I^\varepsilon[\varphi(x)]$ ($I^\varepsilon[g(x)]$ (n)))
= n * $I^\varepsilon[\varphi(x)]$ (n − 1).
en effet si $\varphi(x) \Leftarrow \varepsilon$ est une déclaration récursive :
$I^\varepsilon[\varphi(\tau)]$ (n) = $I^\varepsilon[\varepsilon]$ ($I^\varepsilon[\tau]$ (n)) d'après c),
= $I^\varepsilon[\varphi(x)]$ ($I^\varepsilon[\tau]$ (n)) d'après c) et a).

Il résulte immédiatement de 1), 2), 3) que pour n $\geqslant$ 0 :

$I^\varepsilon[\varphi(x)]$ (n) = n !

Exercice 28

Etudions uniquement la règle de substitution pleine.
Posons $t_1 = \underline{si}\ 111 > 100\ \underline{alors}\ 101\ \underline{sinon}\ \varphi(\varphi(122))\ \underline{fsi}$.
Alors

$$\begin{aligned}\varphi(100) \overset{P}{\Rightarrow} \varphi(\varphi(111)) \overset{P}{\Rightarrow} & \ \mathrm{Simpl}[\underline{si}\ t_1 > 100\ \underline{alors}\ t_1 - 10\ \underline{sinon}\ \varphi(\varphi(t_1 + 11))\ \underline{fsi}] \\ = & \ \mathrm{Simpl}[\underline{si}\ 101 > 100\ \underline{alors}\ 101 - 10\ \underline{sinon}\ \ldots] \\ & \ (\text{car } \mathrm{Simpl}[t_1] = 101), \\ = & \ 91 .\end{aligned}$$

Exercice 29

a) Vérifions que $\mathrm{Simpl}(t[\Omega/\underline{\varphi}, f/\varphi]) = \omega$ pour tout terme t contenant φ, toute fonction f, où les occurrences $\underline{\varphi}$ sont choisies par l'une des règles V, N, P.

Si $t = g(t_1, \ldots, t_n)$ soit i le premier indice tel que t_i contienne une occurrence $\underline{\varphi}$. Par récurrence

$$\mathrm{Simpl}(t_i[\Omega/\underline{\varphi}, f/\varphi]) = \omega .$$

Donc
$$\begin{aligned}\mathrm{Simpl}(t[\Omega/\underline{\varphi}, f(\varphi)]) &= \mathrm{I}_0[g]\ (\ldots, \mathrm{Simpl}(t_i[\Omega/\underline{\varphi}, f/\varphi]), \ldots) \\ &= \mathrm{I}_0[g]\ (\ldots, \omega, \ldots) = \omega.\end{aligned}$$

Si $t = \varphi(t')$ et si la règle sélectionne le premier φ

$$\mathrm{Simpl}(t[\Omega/\underline{\varphi}, f/\varphi]) = \Omega(a) = \omega \qquad \text{pour un } a .$$

Si la règle sélectionne une occurrence de φ dans t'

$$\mathrm{Simpl}(t[\Omega/\underline{\varphi}, f/\varphi]) = \omega \qquad \text{par récurrence} .$$

Si $t = \underline{\mathrm{si}}\ b\ \underline{\mathrm{alors}}\ t_1\ \underline{\mathrm{sinon}}\ t_2\ \underline{\mathrm{fsi}}$; b contient une occurrence $\underline{\varphi}$, donc, par récurrence

$$\begin{aligned}\mathrm{Simpl}(t[\Omega/\underline{\varphi}, f/\varphi]) &= \mathrm{SI}\ \omega\ \mathrm{ALORS}\ a_1\ \mathrm{SINON}\ a_2 \\ &= \omega .\end{aligned}$$

Le cas des prédicats se traite comme celui des fonctions de base.

b) Soit $t = \underline{si}\ \varphi(0) = 0\ \underline{alors}\ 1\ \underline{sinon}\ \varphi(0)\ \underline{fsi}$ et f une application telle que $f(0) = 0$. La règle PD sélectionne la deuxième occurrence de φ et donc

$$\mathrm{Simpl}(t[\Omega/\underline{\varphi}, f/\varphi]) = 1 \neq \omega .$$

Exercice 30

a) Considérons la déclaration récursive suivante

$$\begin{aligned}\varphi(x) \Leftarrow \underline{si}\ x = 0\ & \underline{alors}\ 1 \\ & \underline{sinon}\ si\ \varphi(x-1) > 0\ \underline{alors}\ 1\ \underline{sinon}\ \varphi(x)\ \underline{fsi} \\ & \underline{fsi} .\end{aligned}$$

On prouve sans peine que $\Phi(\mathrm{n}) = 1$ pour tout $\mathrm{n} \geqslant 0$ alors que $\Phi_{\mathrm{PD}}(\mathrm{n}) = \omega$ pour tout $\mathrm{n} > 0$.

Exercice 31

Vérifions seulement qu'il existe $m \geqslant 0$ et ε' tel que

$$u_2^{\mathrm{P}} \leqslant u_{\mathrm{m}}^{\mathrm{N}}[\varphi/\underline{\varphi}, \varepsilon'/\varphi]$$

$$u_1^{\mathrm{P}} = \underline{\mathit{si}}\ 89 > 100\ \underline{\mathit{alors}}\ 89 - 10\ \underline{\mathit{sinon}}\ \underline{\varphi}(\underline{\varphi}(89 + 11))\ \underline{\mathit{fsi}}$$

$$\begin{aligned}
u_2^{\mathrm{P}} = {} & \underline{\mathit{si}}\ 89 > 100\ \underline{\mathit{alors}}\ 89 - 10 \\
& \quad \underline{\mathit{sinon}}\ \underline{\mathit{si}}\ \{\ \underline{\mathit{si}}\ 100 > 100\ \underline{\mathit{alors}}\ 100-10\ \underline{\mathit{sinon}}\ \varphi(\varphi(111))\ \} > 100 \\
& \qquad \underline{\mathit{alors}}\ \{\ \underline{\mathit{si}}\ 100 > 100\ \underline{\mathit{alors}}\ 100-10\ \underline{\mathit{sinon}}\ \varphi(\varphi(111))\ \} -10 \\
& \qquad \underline{\mathit{sinon}}\ \varphi(\varphi(\{\ \underline{\mathit{si}}\ 100 > 100\ \underline{\mathit{alors}}\ 100-10\ \underline{\mathit{sinon}}\ \varphi(\varphi(111))\ \} +11)) \\
& \qquad \underline{\mathit{fsi}} \\
& \underline{\mathit{fsi}}
\end{aligned}$$

$$u_1^{\mathrm{N}} = \underline{\mathit{si}}\ 89 > 100\ \underline{\mathit{alors}}\ 89-10\ \underline{\mathit{sinon}}\ \underline{\varphi}(\varphi(89+11))\ \underline{\mathit{fsi}}\ ;$$

$$\begin{aligned}
u_2^{\mathrm{N}} = {} & \underline{\mathit{si}}\ 89 > 100\ \underline{\mathit{alors}}\ 79\ \underline{\mathit{sinon}}\ \underline{\mathit{si}}\ \varphi(100) > 100\ \underline{\mathit{alors}}\ \varphi(100)-10 \\
& \qquad\qquad\qquad\qquad\qquad\qquad \underline{\mathit{sinon}}\ \varphi(\varphi(\varphi(100) + 11)) \\
& \qquad\qquad\qquad\qquad \underline{\mathit{fsi}} \\
& \underline{\mathit{fsi}}
\end{aligned}$$

$$\begin{aligned}
u_3^{\mathrm{N}} = {} & \underline{\mathit{si}}\ 89 > 100\ \underline{\mathit{alors}}\ 89-10 \\
& \qquad \underline{\mathit{sinon}}\ \underline{\mathit{si}}\ \{\ \underline{\mathit{si}}\ 100 > 100\ \underline{\mathit{alors}}\ 100-10\ \underline{\mathit{sinon}}\ \varphi(\varphi(111))\ \} > 100 \\
& \qquad\quad \underline{\mathit{alors}}\ \varphi(100)-10 \\
& \qquad\quad \underline{\mathit{sinon}}\ \varphi(\varphi(\varphi(100)+11)) \\
& \qquad\quad \underline{\mathit{fsi}} \\
& \underline{\mathit{fsi}}\ ;
\end{aligned}$$

$m = 3$ et $\varepsilon' = \varepsilon$ conviennent ici. En effet

$$u_2^{\mathrm{P}} \leqslant u_3^{\mathrm{N}}[\varphi/\underline{\varphi}, \varepsilon/\varphi]$$

Exercice 32

$\tilde{\mathrm{I}}[\![\varepsilon]\!] = \lambda \mathrm{f}.\lambda \mathrm{x}, \mathrm{y}.\mathrm{SI}\ \mathrm{x} = 0\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \mathrm{f}(\mathrm{x}-1, \mathrm{f}(\mathrm{x}, \mathrm{y}))$.

Calculons $\widetilde{\mathrm{FIX}} = \mu(\tilde{\mathrm{I}}[\![\varepsilon]\!])$ par approximations successives. Soit $(\mathrm{f}_{\mathrm{n}})_{\mathrm{n} \geqslant 0}$ la suite des approximations de $\widetilde{\mathrm{FIX}}$

$$\begin{aligned}
\mathrm{f}_0 &= \Omega \\
\mathrm{f}_1 &= \lambda \mathrm{x}, \mathrm{y}.\mathrm{SI}\ \mathrm{x} = 0\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \omega \\
\mathrm{f}_2 &= \lambda \mathrm{x}, \mathrm{y}.\mathrm{SI}\ \mathrm{x} = 0\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \mathrm{f}_1(\mathrm{x}-1, \mathrm{f}_1(\mathrm{x}, \mathrm{y})) \\
&= \lambda \mathrm{x}, \mathrm{y}.\mathrm{SI}\ \mathrm{x} = 0\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \mathrm{SI}\ \mathrm{x}-1 = 0\ \mathrm{ALORS}\ 0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{SINON}\ \omega \\
&= \lambda \mathrm{x}, \mathrm{y}.\mathrm{SI}\ \mathrm{x} \leqslant 1\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \omega\ .
\end{aligned}$$

Plus généralement, on vérifie que f_{n} est la fonction

$$\lambda \mathrm{x}, \mathrm{y}.\mathrm{SI}\ \mathrm{x} < \mathrm{n}\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \omega\ .$$

Donc $\widetilde{\mathrm{FIX}} = \lambda \mathrm{x}, \mathrm{y}.\mathrm{SI}\ \mathrm{x} \neq \omega\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \omega$.

$\widetilde{\mathrm{FIX}}$ n'est pas stricte puisque $\widetilde{\mathrm{FIX}}(x, \omega) = 0$ pour tout x distinct de ω.

Vérifions, par récurrence sur x, que $\tilde{\Phi}_N(x, y) = 0$. C'est trivial pour $x = 0$. Supposons l'assertion vérifiée pour tout $x' < x$ et considérons le calcul de $\varphi(x, y)$ déterminé par l'appel par nom

$$\varphi(x, y) \overset{N}{\Rightarrow} \varphi(x - 1, \varphi(x, y))$$

Par récurrence le calcul de $\varphi(x - 1, \varphi(x, y))$ se termine en 0. Donc $\tilde{\Phi}_N(x, y)$ est égal à 0.

Finalement $\tilde{\Phi}_N$ coincide avec $\widetilde{\mathrm{FIX}}$. Par contre on vérifie que $\tilde{\Phi}_V$ est la fonction

$$\lambda x, y . \mathrm{SI}\ x = 0\ \mathrm{ALORS}\ 0\ \mathrm{SINON}\ \omega$$

distincte de $\widetilde{\mathrm{FIX}}$.

Exercice 33

Pour les règles d'appel par nom et de substitution pleine il suffit de reprendre la correction de l'exercice 29 en notant que les fonctions de base sont des fonctions strictes et que dans le terme $\varphi(t_1, \ldots, t_n)$ les deux règles sélectionnent l'occurrence externe de φ.

La règle de l'appel par valeur n'est pas fortement sûre. Considérons le terme $t = \varphi(0, \varphi(0, 0))$ et une fonction f telle que $f(0, \omega) = 0$. Alors

$$\mathrm{Simpl}(t[\Omega/\underline{\varphi}, f/\varphi]) = \mathrm{Simpl}(f(0, \Omega(0, 0))) = 0\ ,$$

tandis que $\mathrm{Simpl}(t[\Omega/\underline{\varphi}]) = \mathrm{Simpl}(\Omega(0, \Omega(0, 0)) = \omega$.

Exercice 34

Désignons par $E_1, \ldots, E_n$ les domaines respectifs de $\Phi_V^1, \ldots, \Phi_V^n$. Il est clair que $E_1 = \{ x \mid p(x_1) \}$ et que, plus généralement, pour

$$i \in [2, n - 1], E_i = \{ x \mid p(x_i)\ \textbf{et non}\ p(x_j) \quad \text{pour} \quad j = 1 \ldots n - 1 \}\ .$$

D'autre part, le domaine E_n peut être défini récursivement par l'équation :

$$E_n = \{ x \mid \textbf{non}\ p(x_1)\ \textbf{et}\ ((x_2, \ldots, x_n) \in E_{n-1}\ \textbf{ou}\ (x_2, \ldots, x_n, h(x_1)) \in E_n) \}\ .$$

On en déduit que

$$x \in E_n \Leftrightarrow \textbf{non}\ p(x_j) \quad \text{pour} \quad j = 1, \ldots, n-1$$
$$\textbf{et}\ (p(x_n)\ \textbf{ou}\ \exists k > 0, \exists i \in [1, n]\ p(h^k(x_i)))\ .$$

Finalement, pour $i = 1, \ldots, n - 1$ $\Phi_V^i(x) = x_i$ si $x \in E_i$.

D'autre part $\Phi_V^n(x)$ est égal à x_n si $p(x_n)$ est vrai et à $h^k(x_i)$ si (k, i) est le plus petit couple tel que $p(h^k(x_i))$ soit vérifié.

Exercice 35

Le nombre de substitutions est de 14 pour la règle d'appel par valeur, 29 pour l'appel par nom, 14 pour la règle du report, 23 pour la règle de substitution pleine [VUI, 73].

Exercice 36

1) $xp\,;\,Ry \Leftrightarrow \exists z\ xpz$ **et** $z\,R\,y$.

Or xpz entraîne $x = z$ et $p(x)$, donc $xp\,;\,Ry$ si et seulement si $p(x)$ et $x\,R\,y$.

2) Le raisonnement est analogue.

Exercice 37

$$x(p\,;\,q)\,x \Leftrightarrow \exists z\ xpz \textbf{ et } zqx \textbf{ et } z = x \Leftrightarrow xpx \textbf{ et } xqx \Leftrightarrow x(p \cap q)\,x\,.$$

Exercice 38

a) $xR\,;\,Ey \Leftrightarrow \exists z\ x\,R\,z$ **et** $z\,E\,y \Leftrightarrow \exists z\ x\,R\,z$ **et** $z = y \Leftrightarrow x\,R\,y$.

De même $xE\,;\,Ry \Leftrightarrow x\,R\,y$.

b) $\overline{E}$ est le prédicat total c'est-à-dire identiquement vrai.

c) $E\,;\,R \cup \overline{E}\,;\,R' = R \cup \Omega\,;\,R' = R$.

Exercice 39

a) $x(R \cup \Omega)\,y \Leftrightarrow x\,R\,y$ **ou** $x\,\Omega\,y \Leftrightarrow x\,R\,y$ **ou faux** $\Leftrightarrow x\,R\,y$.
b) $x(R \cap \Omega)\,y \Leftrightarrow x\,R\,y$ **et** $x\,\Omega\,y \Leftrightarrow x\,R\,y$ **et faux** $\Leftrightarrow$ **faux** $\Leftrightarrow x\,\Omega\,y$.
c) $x(R\,;\,\Omega)\,y \Leftrightarrow \exists z\ x\,R\,z$ **et** $z\,\Omega\,y \Leftrightarrow \exists z\ x\,R\,z$ **et faux** $\Leftrightarrow$ **faux** $\Leftrightarrow x\,\Omega\,y$.
d) $x(p\,;\,\overline{p})\,y \Leftrightarrow p(x)$ **et** $\overline{p}(x) \Leftrightarrow$ **faux** $\Leftrightarrow x\,\Omega\,y$.

Exercice 90

Traduisons *si p alors* (*si p alors* π *sinon* π') *sinon* π'' *fsi* sous forme relationnelle :

$$p\,;\,(p\,;\,\pi \cup \overline{p}\,;\,\pi') \cup \overline{p}\,;\,\pi'' = p\,;\,p\,;\,\pi \cup p\,;\,\overline{p}\,;\,\pi' \cup \overline{p}\,;\,\pi''$$
$$= p\,;\,\pi \cup \Omega\,;\,\pi' \cup \overline{p}\,;\,\pi'' = p\,;\,\pi \cup \Omega \cup \overline{p}\,;\,\pi'' = p\,;\,\pi \cup \overline{p}\,;\,\pi''\,.$$

Le programme associé à cette dernière relation est

si p alors π *sinon* π'' *fsi* .

Exercice 41

1) $x(R\,;\,R^{-1} \cap E)\,y \Leftrightarrow \exists z\,(x\,R\,z$ **et** $z\,R^{-1}\,y)$ **et** $x = y$

$\Leftrightarrow \exists z(x\,R\,z$ **et** $z\,R^{-1}\,x)$ **et** $x = y \Leftrightarrow \exists z(x\,R\,z)$ **et** $x = y$

$\Leftrightarrow \exists z(x\,R\,z$ **et** $z\,U\,y)$ **et** $x = y$ (car $z\,U\,y$ est vrai)

$\Leftrightarrow x(R\,;\,U \cap E)\,y$.

2) On a $R\,;\,R^{-1} \cap E \subseteq E$ c'est donc un prédicat.

3) Comme on l'a écrit plus haut $x(R\,;\,R^{-1} \cap E)\,x$ est équivalent à $\exists z\ x\,R\,z$ ce qui signifie que x appartient au domaine de définition de R, autrement dit x est un point pour lequel R a au moins un résultat.

Exercice 42

La première ligne est un résultat d'algèbre de Boole bien connu.

$$\begin{aligned} & xR_1 ; (R_2 \cup R_3)\, y \Leftrightarrow \exists z\; x\, R\, z \textbf{ et } z(R_2 \cup R_3)\, y \\ & \Leftrightarrow \exists z\; x\, R_1\, z \textbf{ et } (z\, R_2\, y \textbf{ ou } z\, R_3\, y) \\ & \Leftrightarrow \exists z (x\, R_1\, z \textbf{ et } z\, R_2\, y) \textbf{ ou } (x\, R_1\, z \textbf{ et } z\, R_3\, y) \\ & \Leftrightarrow \exists z (x\, R_1\, z \textbf{ et } z\, R_2\, y) \textbf{ ou } \exists z (x\, R_1\, z \textbf{ et } z\, R_3\, y) \\ & \Leftrightarrow xR_1 ; R_2\, y \textbf{ ou } xR_1 ; R_3\, y \\ & \Leftrightarrow x(R_1 ; R_2 \cup R_1 ; R_3)\, y\, . \end{aligned}$$

Les autres assertions se prouvent de façon analogue.

Exercice 43

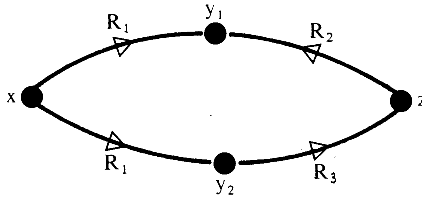
a) Rappelons que

$$\exists z (p(z) \textbf{ et } q(z)) \Rightarrow \exists z\; p(z) \textbf{ et } \exists z\; q(z)$$

mais que la réciproque n'est pas vraie ; c'est pourquoi on ne peut écrire ci-dessous qu'une implication (qui correspond à l'inclusion demandée) :

$$\begin{aligned} & x(R_1 ; (R_2 \cap R_3))\, y \Leftrightarrow \exists z ((x\, R_1\, z \textbf{ et } z\, R_2\, y) \textbf{ et } (x\, R_1\, z \textbf{ et } z\, R_3\, y)) \\ & \Rightarrow \exists z (x\, R_1\, z \textbf{ et } z R_2\, y) \textbf{ et } \exists z (x\, R_1\, z \textbf{ et } x\, R_3\, y) \Leftrightarrow x(R_1 ;\; R_2 \cap R_1 ; R_3)\, y\, . \end{aligned}$$

b) Le diagramme sagittal suivant donne un contre-exemple



On voit que $R_1 ; (R_2 \cap R_3) = \Omega$ alors que $R_1 ; R_2 \cap R_1 ; R_3 = \{\, (x, z)\, \}$.

Exercice 44

$R ; R^{-1} \subseteq E$ signifie qu'il existe au plus un y tel que $x\, R\, y$, autrement dit que R est déterministe.

$E \subseteq R ; R^{-1}$ signifie que pour chaque x il existe au moins un y tel que $x\, R\, y$, autrement dit que R « calcule » au moins une valeur pour chaque x.

Exercice 45

a) Sachant que $[\forall z (p(z) \Rightarrow q)] \Leftrightarrow [(\exists z\; p(z)) \Rightarrow q]$ on a immédiatement :

$$\begin{aligned} R \text{ transitive} & \Leftrightarrow \forall z (x\, R\, z \textbf{ et } z\, R\, y \Rightarrow x\, R\, y) \\ & \Leftrightarrow (\exists z\; x\, R\, z \textbf{ et } z\, R\, y) \Rightarrow x\, R\, y \\ & \Leftrightarrow (xR ; Ry \Rightarrow x\, R\, y) \\ & \Leftrightarrow R ; R \subseteq R\, . \end{aligned}$$

b) R réflexive signifie $E \subseteq R$ donc $E \cup R = R$; d'autre part par récurrence on a immédiatement $R^n \subseteq R$. Finalement

$$R \subseteq \bigcup_{n \geqslant 0} R^n \subseteq R .$$

Exercice 46

Si Q est la relation spécifiée par <u>*répéter*</u> π <u>*jusqu'à*</u> *p* alors x Q y si et seulement s'il existe une suite $x_0, \ldots, x_n$ telle que

$$x_0 = x \textbf{ et } x_0 \mathrel{R} x_1 \textbf{ et } \overline{p}(x_1) \textbf{ et } x_1 \mathrel{R} x_2 \textbf{ et} \ldots x_{n-1} \mathrel{R} x_n \textbf{ et } p(x_n) \textbf{ et } x_n = y$$

autrement dit : $x \mathrel{Q} y \Leftrightarrow x(R ; (\overline{p} ; R)^* ; p)\, y$.

Exercice 47

a) $p * R = (p ; R)^* ; \overline{p} = (p ; R ; (p ; R)^* \cup E) ; \overline{p} = p ; R ; ((p ; R)^* ; \overline{p}) \cup E ; \overline{p}$
$= p ; R ; p * R \cup \overline{p}$.

b) Il s'agit simplement d'associer aux deux programmes les relations.

Exercice 48

Les T_i sont les dérivés de ⟨ terme ⟩ dans la grammaire suivante :

$$\begin{aligned}\langle \text{terme} \rangle ::= {} & \langle \text{terme} \rangle ; \langle \text{terme} \rangle \mid \langle \text{terme} \rangle \cap \langle \text{terme} \rangle \mid (\langle \text{terme} \rangle) \\ & \mid \langle \text{terme} \rangle \cup \langle \text{terme} \rangle \mid \langle \text{terme} \rangle^* \mid \langle \text{terme} \rangle^{-1} \\ & \mid E \mid \Omega \mid U \mid A \mid \ldots \mid Z \mid a \mid \ldots \mid z \mid \bar{a} \mid \ldots \mid \bar{z} \mid \\ & \mid R_1 \mid \ldots \mid R_i \mid \ldots\end{aligned}$$

Exercice 49

$f_1(x) \Leftarrow$ <u>*si*</u> $p(x)$ <u>*alors*</u> $f_1(a(x))$ <u>*sinon*</u> x <u>*fsi*</u> ;
$f_1(x) \Leftarrow$ <u>*si*</u> $p(x)$ <u>*alors*</u> $f_2(x)$ <u>*sinon*</u> x <u>*fsi*</u>
$f_2(x) \Leftarrow$ <u>*si*</u> $p(x)$ <u>*alors*</u> $f_2(a(x))$ <u>*sinon*</u> $f_1(x)$ <u>*fsi*</u> .

Exercice 50

Les deux schémas spécifient la même relation.

Exercice 51

On trouve la relation $p ; A ; (p ; A)^* \, \overline{p}$.

Exercice 52

1) Pour F on obtient $\mu X(p \cup \overline{p} ; A ; X)$
et pour G $\mu Y(p ; B \cup \overline{p} ; A ; Y)$.

2) Il s'agit de montrer que

$$\mu X(p \cup \overline{p} ; A ; X) ; B = \mu Y(p ; B \cup \overline{p} ; A ; Y) .$$

a) On a

$$p\,;B \cup \bar{p}\,;A\,;\mu X(p \cup \bar{p}\,;A\,;X)\,;B = (p \cup \bar{p}\,;A\,;\mu X(p \cup \bar{p}\,;A\,;X))\,;B$$
$$= \mu X(p \cup \bar{p}\,;A\,;X)\,;B$$

d'où d'après la propriété de Park :

$$\mu Y(p\,;B \cup \bar{p}\,;A\,;Y) \subseteq \mu X(p \cup \bar{p}\,;A\,;X)\,;B\,.$$

b) Montrons par la règle de Scott que

$$\mu X(p \cup \bar{p}\,;A\,;X)\,;B \subseteq \mu Y(p\,;B \cup \bar{p}\,;A\,;Y)\,.$$

Posons $\Phi(X) \;=\; X\,;B \subseteq \mu Y(p\,;B \cup \bar{p}\,;A\,;Y)$.
$\Phi(\Omega)$ est vrai. Supposons $\Phi(X)$ alors

$$(p \cup \bar{p}\,;A\,;X)\,;B = p\,;B \cup \bar{p}\,;A\,;X\,;B$$
$$\subseteq p\,;B \cup \bar{p}\,;A\,;\mu Y(p\,;B \cup \bar{p}\,;A\,;Y)$$
$$= \mu Y(p\,;B \cup \bar{p}\,;A\,;Y)\,.$$

(on peut rapprocher cet exercice de l'exercice 21 du chapitre 2).

CHAPITRE 4

Vérification et conception de programmes

1 INTRODUCTION

L'outil informatique joue un rôle de plus en plus important dans l'activité humaine, ce qui a pour conséquence d'augmenter considérablement le coût de ses défaillances. Aussi les utilisateurs exigent de plus en plus de garanties de fiabilité des produits informatiques. Cette tendance, d'abord sensible pour le matériel, s'étend actuellement au logiciel. Cependant, alors que les progrès spectaculaires de la technologie permettent de réduire et de prévenir les risques d'erreurs physiques, la situation au niveau de la conception des programmes apparaît beaucoup plus critique.

Il y a peu de temps, la méthode de construction d'un programme était la suivante : le programmeur, tel un artiste, écrivait selon son inspiration ; puis, comme le programme devait réaliser une fonction (plus ou moins bien définie), il procédait à des essais sur machine et tentait de supprimer par des retouches délicates chaque erreur ainsi découverte. Il s'engageait alors dans un processus itératif, bien connu : chaque retouche pouvait entraîner de nouvelles erreurs provoquant à leur tour de nouvelles retouches et ainsi de suite. Et si, par bonheur il arrivait à un programme satisfaisant aux essais, ce programme était si différent du programme prévu que personne, sauf peut-être son créateur, n'était capable de le modifier ou même simplement de le comprendre. De plus, il est fort courant qu'un tel programme donne un jour des résultats erronés sur un cas non couvert par les tests.

C'est pourquoi, actuellement, tout travail informatique se décompose en une partie de conception (analyse et programmation) et une partie de mise au point (tests de recherche d'erreurs). Pour essayer de réduire, sinon d'annuler, le temps réservé à la deuxième partie, certains chercheurs ont développé des méthodes de justification de programmes plus sûres que les simples jeux d'essais.

Dans ce chapitre, en distinguant différents types de schémas que nous avons rencontrés au chapitre 3, nous présentons les principales méthodes de vérification d'algorithmes. Ainsi le paragraphe 2 concerne-t-il les programmes avec branchements, le paragraphe 3 les programmes itératifs et le paragraphe 6 les programmes récursifs. Nous justifions ces méthodes dans le cadre

du calcul relationnel au paragraphe 5. De plus, partant des méthodes présentées dans les paragraphes 2 et 3, nous envisageons au paragraphe 4 le problème de la construction de programmes corrects. La conclusion (§ 7) montre les limites de ces méthodes et indique quelques voies de recherches actuelles.

2 PREUVES DE PROGRAMMES AVEC BRANCHEMENTS

Dans la suite de ce paragraphe, nous appelons organigrammes la représentation graphique des programmes avec branchements. La notation générique des assertions utilise les lettres p, q éventuellement indicées.

2.1 Introduction

« No loop should be written down without providing a proof for terminaison and without stating the relation whose invariance will not be destroyed by the execution of the repeatable statement » DIJKSTRA.

Exemple 1 division euclidienne

Considérons l'organigramme de la figure 1 :

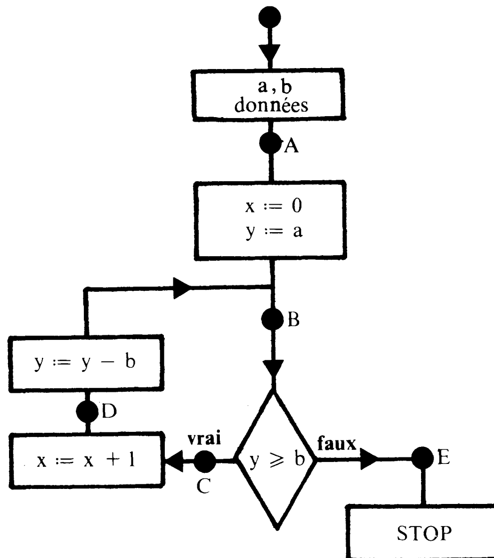


Figure 1 Division euclidienne.

Prouvons que ce programme admet comme résultat le quotient x et le reste y de la division euclidienne de a par b. Pour cela, on précise à chaque étape du calcul de données a et b les propriétés que vérifient les valeurs des variables x et y. Ces propriétés sont données sous forme de prédicats « étiquetant »

chaque arc de l'organigramme. Chaque prédicat doit être vrai à chaque « passage » par l'arc associé pour les valeurs courantes des variables. En A, on exprime les hypothèses faites sur les données :

$$p_A = a \in \mathbb{N} \ \underline{\text{et}}\ b \in \mathbb{N}\ .$$

En B, au premier passage, on a $a = y$, mais lors d'un passage ultérieur, les valeurs de x et y se sont modifiées. On cherche un prédicat qui reste vérifié à chaque passage au point B. Il s'agit d'imaginer ce prédicat en analysant les contraintes qu'il doit vérifier. On trouve ainsi que le prédicat $p_B = (a = b * x + y)$ convient.

En C, on a le même prédicat qu'en B avec en plus $y \geqslant b$; ainsi :

$$p_C = (a = b * x + y \ \underline{\text{et}}\ y \geqslant b)\ .$$

Avant D, x est incrémenté de 1, donc :

$$p_D = (a = b * (x - 1) + y \ \underline{\text{et}}\ y \geqslant b)\ .$$

En E, on a $p_E = (a = b * x + y \ \underline{\text{et}}\ y < b)$, c'est-à-dire le résultat cherché.

Après avoir découvert ces prédicats, il s'agit de prouver qu'ils sont vérifiés à chaque passage par l'arc associé. On constate qu'il suffit de prouver qu'il en est ainsi pour p_B, les autres cas s'en déduisant immédiatement.

— Au premier passage en B : $x = 0, a = y$, on a donc bien p_B.

— Si x et y sont les valeurs des variables en B et x', y' les valeurs correspondantes après un cycle B-C-D-B on a :

$$a = b * x + y \qquad \text{(on suppose } p_B \text{ en début de cycle)}$$
$$x' = x + 1$$
$$y' = y - b$$

dont on déduit :

$$a = b * (x' - 1) + y' + b = b * x' + y'\ .$$

On exprime cette propriété de l'assertion p_B en disant que c'est un **invariant** du cycle B-C-D-B.

Nous prouvons ainsi que, **si le programme termine,** il calcule le quotient x et le reste y de la division euclidienne de a par b (caractérisés par le prédicat p_E). Remarquons cependant que nous n'avons pas prouvé l'arrêt de l'exécution. Par exemple, si $b = 0$ et $a > 0$, p_B s'écrit :

$$a = 0 * x + y$$

et donc $a = y > b = 0$ à chaque passage en B : **le programme ne termine pas.**

D'autre part, il est facile de montrer que pour $b \geqslant 1$, l'arrêt est assuré : il suffit de constater qu'à chaque itération la valeur de y décroît strictement en prenant des valeurs entières et donc, qu'après un nombre fini d'itérations, on a $y < b$ et on sort de l'unique cycle du programme. □

Sur cet exemple apparaissent les propriétés caractérisant un **organigramme correct :**

i) Si les données vérifient une certaine assertion p′ (dans l'exemple précédent $p'(a, b) = (a \in \mathbb{N}$ et $b \in \mathbb{N}))$ et si le calcul s'arrête, alors il fournit le résultat souhaité, c'est-à-dire que les données et les valeurs finales des variables sont liées par une relation exprimée par une assertion q (dans l'exemple 1

$$q(x, y, a, b) = (a = b * x + y \ \underline{\text{et}} \ y < b)) .$$

ii) Si les données vérifient une certaine assertion p″, alors le calcul s'arrête (dans l'exemple 1, il suffit de prendre pour $p''(a, b) = (a \in \mathbb{N} \ \underline{\text{et}} \ b \in \mathbb{N}^*))$.

Un organigramme qui vérifie la propriété i) est dit **partiellement correct par rapport à** p′ et q ; s'il vérifie ii), on dit qu'il se **termine pour** p″. Un organigramme vérifiant à la fois i) et ii) (avec en entrée le prédicat p = p′ et p″) est dit **totalement correct par rapport à** p **et** q.

Pour démontrer la correction partielle, la méthode de FLOYD [FLO, 67] conduit à découper le programme en morceaux plus élémentaires et à introduire des assertions (ou prédicats) intermédiaires. Nous présentons cette méthode aux paragraphes 2.2 et 2.3. Le problème de l'arrêt, quant à lui, est étudié au paragraphe 2.4. Notons que nous traitons seulement le problème de l'arrêt d'un programme dans son ensemble ; nous ne nous intéressons pas aux « blocages » provoqués par des instructions non définies (division par zéro dans une expression, utilisation d'un indice de tableau extérieur à l'intervalle des bornes). Tous ces problèmes ne peuvent pas se résoudre de manière complètement algorithmique ; nous ne pouvons donc pas donner de recettes permettant de trouver les « points de coupures » et les assertions associées. Leur découverte résulte du bon sens, de l'expérience et de l'esprit de méthode de l'utilisateur. Nous donnons cependant une certaine méthodologie et des heuristiques permettant de guider cette recherche. Lorsque points et assertions sont bien choisis, le reste de la preuve est presque automatique.

Dans la suite, les organigrammes considérés sont des schémas interprétés. Pour faciliter l'exposé, le vecteur d'état est composé ici de deux parties :

$a = (a_1, ..., a_n)$ représente les données, c'est une constante,
$x = (x_1, ..., x_p)$ représente les variables.

L'initialisation des variables (§ 2.2) est faite explicitement par une affectation placée au début.

En général, l'interprétation est sous-entendue par le contexte. De plus, on accepte indifféremment des affectations globales à x et des affectations à certains sous-vecteurs de x.

2.2 Correction partielle et méthode des assertions de FLOYD

Un des premiers, FLOYD suggéra, pour prouver la correction des programmes, d'introduire des assertions (ou prédicats) que les valeurs des variables vérifient.

Exemple 2

Montrons que le programme de la figure 2 calcule le carré de a :

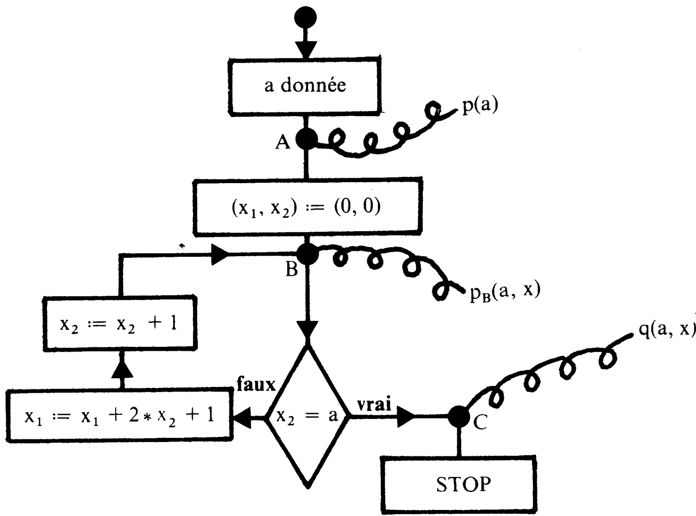


Figure 2 Calcul du carré de a.

Supposons $a \geqslant 0$; nous voulons montrer qu'à la fin de l'exécution, a et (x_1, x_2) vérifient l'assertion $\mathrm{q}(a, x) = (x_1 = a^2)$.

Essayons de construire une assertion $\mathrm{p_B}(a, x)$ qui soit vraie à chaque passage au point B :

— $\mathrm{p_B}$ doit être vraie au deuxième passage (s'il existe), donc pour $x_1 = 1$, et alors les valeurs des variables vérifient : $x_1 = 0, x_2 = 0$.

— $\mathrm{p_B}$ doit être vraie au deuxième passage (s'il existe), donc pour $x_1 = 1$, $x_2 = 1$ et $x_2 \leqslant a$.

— De même, au troisième passage (s'il existe) : $x_1 = 4, x_2 = 2$ et $x_2 \leqslant a$, etc...

On peut donc proposer $\mathrm{p_B}(a, x) = (x_1 = x_2^2 \ \underline{\mathrm{et}}\ x_2 \leqslant a)$.

$\mathrm{p_B}$ est bien un invariant du cycle car il est vrai au premier passage et vrai d'une itération sur l'autre (ce qui résulte immédiatement de

$$(x_2 + 1)^2 = x_2^2 + 2 * x_2 + 1)\,.$$

Si l'instruction STOP est atteinte, on peut donc affirmer que les variables vérifient :

$$\mathrm{q}(a, x) = (x_1 = x_2^2 \ \underline{\mathrm{et}}\ x_2 = a)$$

c'est-à-dire $\mathrm{q}(a, x) = (x_1 = a^2 \ \underline{\mathrm{et}}\ x_2 = a)$.

Ainsi, cet organigramme est partiellement correct vis-à-vis du prédicat d'entrée $a \geqslant 0$ et l'assertion de sortie $x_1 = a^2$. □

Afin de démontrer précisément des propriétés telles que : « $\mathrm{p_B}$ est un invariant », nous sommes conduits à formaliser les notions intuitives introduites jusqu'ici. On marque certains arcs de l'organigramme considéré par des

points de coupure auxquels sont associés des prédicats portant sur la donnée a et la valeur courante de x. En particulier, l'arc arrivant à la première instruction et l'arc arrivant à l'instruction STOP sont marqués par un point (resp. prédicat) d'entrée et de sortie. On appelle **étiquetage** cet ensemble de couples (points de coupure, assertion). Soient A et B deux points de coupure d'un étiquetage d'un organigramme π et p_A et p_B les prédicats associés.

Intuitivement, un chemin α de π est dit compatible avec une valeur x (*) si, pour la valeur x à l'origine de α, le calcul suit le chemin α ; dans ces conditions $\alpha(x)$ est le résultat du calcul le long de α. Les définitions suivantes précisent ces notions (des définitions plus formelles sont proposées à l'alinéa c du § 2.3).

Définition 1

a) La *condition de cheminement* (ou condition de compatibilité) $C(a, x\,;\alpha)$ est une assertion dépendant de x et a telle que si x vérifie $C(a, x\,;\alpha)$ un calcul commençant en l'origine de α avec la valeur x suit nécessairement le chemin α.

b) Si la valeur x vérifie $C(a, x\ \alpha)$, $\alpha(x)$ est la valeur de la variable après le calcul correspondant au chemin α. □

Par exemple sur l'organigramme de la figure 2 :

$$C(a, (x_1, x_2)\,; \text{A-B}) = \underline{\text{vrai}} \qquad \text{et} \qquad (\text{A-B})\,(x_1, x_2) = (0, 0)$$
$$C(a, (x_1, x_2)\,; \text{B-C}) = (x_2 = a) \qquad (\text{B-C})\,(x_1, x_2) = (x_1, x_2)$$

en notant A-B (resp. B-C) le chemin reliant directement A à B (resp. B à C).

Exercice 1

Donner une définition formelle de $C(x\,;\alpha)$ et de $\alpha(x)$ en utilisant les notations du paragraphe 3.2. □

Cette première définition nous permet de préciser la notion de chemin correct : Dans la suite on s'autorise à abréger $C(a, x\,;\alpha)$ en $C(x\,;\alpha)$.

Définition 2

Un chemin α de A à B est **correct** pour les prédicats p_A et p_B si pour toute valeur x :

$$p_A(a, x)\ \underline{\text{et}}\ C(a, x\,;\alpha) \Rightarrow p_B(a, \alpha(x))$$

ce qu'on note simplement : $p_A \{ \alpha \} p_B$. □

Par exemple sur l'organigramme de la figure 2 en notant B-B le chemin associé à un cycle de B vers B, le cycle B-B est correct pour le prédicat :

$$p_B(a, x) = (x_1 = x_2^2\ \underline{\text{et}}\ x_2 \leqslant a)\,.$$

(*) Dans la suite, pour alléger les conventions typographiques, on note de la même façon l'identificateur x (lettre utilisée dans le programme ou l'organigramme) et la valeur de x qui est un élément de l'ensemble des données.

En effet :

$$C(x ; \text{B-B}) = (x_2 \neq a)$$

$$(\text{B-B})(x) = (x_1 + 2 * x_2 + 1, x_2 + 1)$$

$$(x_1 = x_2^2 \ \underline{\text{et}}\ x_2 \leqslant a \ \underline{\text{et}}\ x_2 \neq a) \Rightarrow (x_1 + 2 * x_2 + 1 = (x_2 + 1)^2 \ \underline{\text{et}}\ x_2 + 1 \leqslant a) .$$

Définition 3

Un chemin α entre deux points de coupure A et B est **direct** s'il ne passe par aucun autre point de coupure. □

Par exemple l'organigramme de la figure 2 comporte 3 chemins directs : A-B, B-C et le cycle B-B.

Définition 4

Un organigramme π muni d'un étiquetage est dit **localement correct** si pour tout couple de points de coupure A, B et tout chemin direct α de A vers B on a $p_A \{ \alpha \} p_B$.

Un organigramme π muni d'un étiquetage est dit **partiellement correct** par rapport aux assertions p et q si pour tout chemin α de la première instruction à l'instruction STOP on a $p \{ \alpha \} q$. □

Le théorème suivant relie simplement la correction locale et la correction partielle :

Théorème 1

Soit un organigramme π muni d'un étiquetage de prédicats d'entrée p et de sortie q. Si π est localement correct il est partiellement correct par rapport à p et q. □

La démonstration repose sur le fait que tout chemin α de la première à la dernière instruction est une suite de chemins directs $\alpha_1, \ldots, \alpha_n$ (α_i relie A_i à A_{i+1}) et il est clair que si pour $i = 1, \ldots, n$ on a :

$$p_{A_i}(a, x) \ \underline{\text{et}}\ C(a, x ; \alpha_i) \Rightarrow p_{A_{i+1}}(a, \alpha_i(x))$$

on peut en déduire :

$$p_{A_1}(a, x) \ \underline{\text{et}}\ C(a, x ; \alpha) \Rightarrow p_{A_{n+1}}(a, \alpha(x)) .$$

(La démonstration précise s'appuie évidemment sur les définitions de $C(a, x ; \alpha)$ et $\alpha(x)$ données au § 2.3, alinéa c.)

Remarquons que, dès qu'un organigramme comprend un cycle, le nombre des chemins n'est pas fini, ce qui ne facilite pas la preuve directe de la correction partielle. Cependant on peut toujours choisir un étiquetage de façon que chaque cycle comporte au moins un point de coupure ; le nombre des chemins directs est alors fini, d'où l'intérêt du théorème 1.

Une réciproque du théorème 1 garantissant l'existence d'assertions intermédiaires si le programme est partiellement correct sera présentée au paragraphe 5. Dans la suite nous nous efforçons d'indiquer une certaine méthodologie et des heuristiques pour établir la correction partielle d'un programme.

2.3 Méthodologie pour vérifier la correction partielle

a) Règles heuristiques

Pas 1 : *choix des points de coupure*

On peut, bien sûr, décider d'étiqueter tous les arcs de l'organigramme, mais en pratique, il est plus simple de se limiter à un ensemble de « point-clés ». Les points d'entrée et de sortie en font partie évidemment. Une première règle est essentielle pour obtenir un nombre fini de chemins directs :

Règle 1 : couper chaque cycle.

La règle suivante permet souvent de faciliter la recherche des assertions :

Règle 2 : étiqueter chaque cycle avant un test de sortie.

Ainsi dans l'exemple 2, l'application de ces règles conduit aux trois points de coupure A, B, C (figure 2).

Pas 2 : *choix des assertions aux points de coupure*

Il s'agit d'associer à tout point de coupure A une assertion p_A telle que chaque fois que le calcul atteint le point A avec la valeur x, alors $p_A(a, x)$ est vrai. Intéressons-nous au cas des points de coupure situés sur un cycle (ce sont les seuls pour lesquels la découverte de l'assertion n'est pas évidente). Avec les notations de la figure 3, on constate en suivant les différents chemins que p_A doit vérifier les conditions suivantes :

$$p_E(a, x) \Rightarrow p_A(a, x) \quad \text{(chemin direct E-A)}$$

$$p_A(a, x) \ \underline{\text{et}}\ t(a, x) \Rightarrow p_S(a, x) \quad \text{(chemin direct A-S).}$$

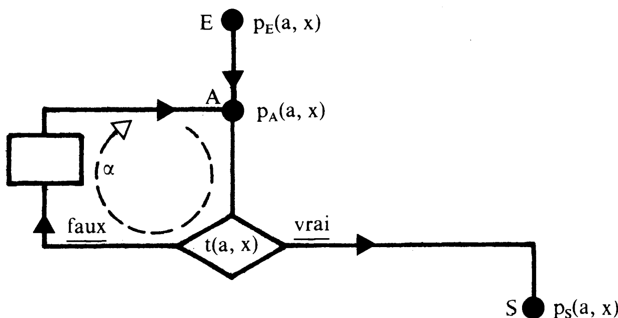


Figure 3 Cycle coupé par un point A.

Il reste à exprimer une condition analogue pour le cycle de A à A. A chaque nouveau parcours du cycle A-A, la valeur de x est modifiée par les affectations du cycle ; en notant x et x' deux valeurs successives en A, la condition sur le cycle s'écrit :

$$(\mathrm{p_A}(a, x) \ \underline{\text{et}} \ \underline{\text{non}} \ \mathrm{t}(a, x)) \Rightarrow \mathrm{p_A}(a, x')$$

ou encore plus généralement si α est le cycle de A à A :

$$(\mathrm{p_A}(a, x) \ \underline{\text{et}} \ \mathrm{C}(a, x ; \alpha)) \Rightarrow \mathrm{p_A}(a, \alpha(x)) \quad \text{(cycle } \alpha \text{ de A à A).}$$

On exprime souvent cette condition en disant que $\mathrm{p_A}$ est un **invariant** du cycle α ; cette condition conduit à l'énoncé de la règle suivante (notations de la figure 3) :

> Règle 3 : Pour trouver un invariant du cycle α de A à A, on note x_n la valeur de x après la n-ième itération et on évalue x_n en fonction de x_0. On cherche à construire une assertion sur x_n indépendante de n ; cette assertion est choisie comme assertion $\mathrm{p_A}$.

Exemple 3 : Recherche d'invariant à l'aide de la règle 3

Reprenons l'organigramme de la figure 2 ; en A, à la n-ième étape on a :

$$x_n = (x_{1,n}, x_{2,n})$$
$$x_0 = (0, 0)$$

Sur le cycle, on trouve :

$$x_{1,n} = x_{1,n-1} + 2 * x_{2,n-1} + 1$$
$$x_{2,n} = x_{2,n-1} + 1$$

d'où

(1) $\quad x_{2,n} = n$

et

(2) $\quad x_{1,n}=x_{1,n-1}+2*(n-1)+1=x_{1,n-1}+2*n-1=\sum_{i=1}^{n}(2*i-1)=n^2 .$

De (1) et (2), on déduit une relation indépendante de n :

$$x_{1,n} = (x_{2,n})^2 .$$

La règle 3 nous incite à choisir le prédicat

$$p_{\mathrm{B}}(a, x) = (x_1 = x_2^2)$$

qui est moins précis que celui que nous avons utilisé dans l'exemple 2 mais qui suffit pour prouver la correction partielle. □

Reprenons l'étude générale du cycle de la figure 3. La condition sur le chemin direct A-S peut inspirer la règle intuitive suivante :

> Règle 4 : choisir pour p_A un « élargissement » de p_S tel que :
>
> $p_A \; \underline{\text{et}} \; t \Rightarrow p_s$.

Exemple 4 : Application de la règle 4.

Revenons encore à l'organigramme de la figure 2 ; supposons connue l'assertion en C :

$$p_C(a, x) = (x_1 = a^2) .$$

Le prédicat p_B doit vérifier :

$$p_B(a, x) \; \underline{\text{et}} \; x_2 = a \Rightarrow x_1 = a^2 .$$

Il est assez naturel « d'élargir » p_C en y introduisant x_2 qui ne figure pas dans p_C :

$$x_1 = x_2^2 \; \underline{\text{et}} \; x_2 = a$$

ce qui incite à prendre pour p_B :

$$x_1 = x_2^2 .$$ □

En fait, la règle 3 permet de trouver un invariant en étudiant directement le cycle (et la valeur des variables à l'entrée de ce cycle) : c'est une étude descendante ; la règle 4, par contre, suppose connue l'assertion de sortie du cycle et s'en sert pour retrouver l'invariant : c'est une étude ascendante. Ces deux règles peuvent évidemment être appliquées conjointement.

Exemple 5

Considérons l'organigramme de la figure 4.

Ce programme, s'il est correct, calcule la racine carrée entière x_1 de a.

Cherchons p_B en utilisant exclusivement la règle 3 ; nous trouvons immédiatement :

$$x_{1,n} = x_{1,0} + n = n$$
$$x_{3,n} = x_{3,0} + 2\,n = 2\,n + 1$$
$$x_{2,n} = x_{2,n-1} + x_{3,n} = x_{2,0} + \sum_{i=1}^{n} (2\,i + 1) = 1 + \sum_{i=1}^{n} (2\,i + 1) = (n + 1)^2 .$$

Ainsi, l'invariant de cycle peut être :

$$p_B(a, x) = (x_2 = (x_1 + 1)^2 \; \underline{\text{et}} \; x_3 = 2\,x_1 + 1) .$$

Il vérifie bien :

— p_A { A-B } p_B
— p_B { B-B } p_B.

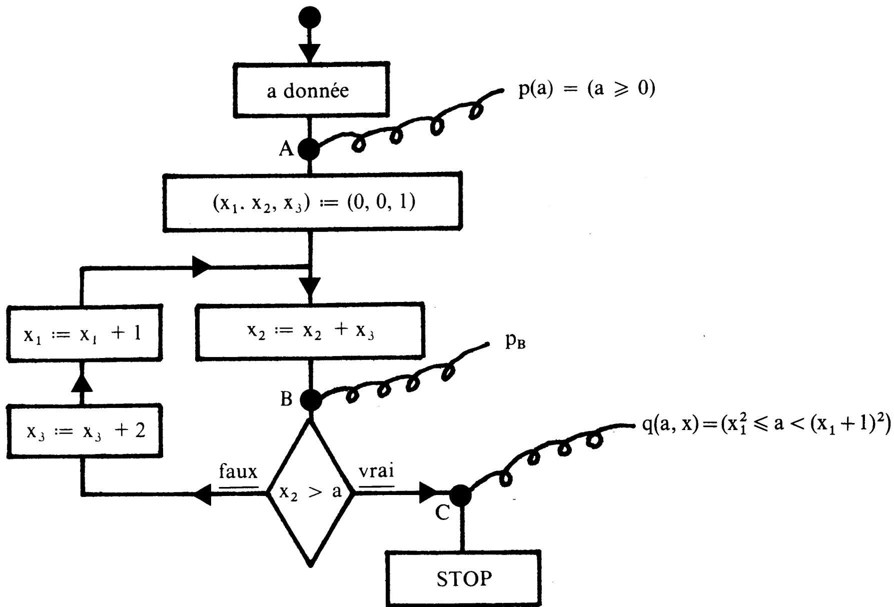


Figure 4 Racine carrée entière.

Mais il ne vérifie pas :

$$p_B \{ \text{B-C} \} q$$

car on ne peut prouver :

$$x_2 = (x_1 + 1)^2 \ \underline{\text{et}}\ x_3 = 2\,x_1 + 1 \ \underline{\text{et}}\ x_2 > a \Rightarrow x_1^2 \leqslant a < (x_1 + 1)^2 .$$

(Intuitivement, ceci provient du fait que l'on n'exprime pas que l'on sort du cycle pour la première valeur de x_2 qui vérifie $x_2 > a$.)

L'utilisation de la règle 4 permet alors de compléter p_B en ajoutant : $x_1^2 \leqslant a$.

Finalement, en remarquant que $x_3 = 2\,x_1 + 1$ n'a aucune utilité pour la correction partielle, on obtient p'_B :

$$p'_B = (x_2 = (x_1 + 1)^2 \ \underline{\text{et}}\ x_1^2 \leqslant a) .$$

On vérifie que p'_B est invariant et on montre que $p'_B \{ \text{B-C} \} q$, ce qui entraîne la correction partielle de l'organigramme considéré. □

Pas 3 : *Vérification des conditions*

L'étiquetage étant déterminé, il suffit de prouver la correction locale pour pouvoir en déduire la correction partielle (théorème 1), c'est-à-dire de montrer, pour tout chemin direct α d'origine A, d'extrémité B, que $p_A \{ \alpha \} p_B$.

b) Exemples

Exemple 6

Démontrons la correction partielle de l'organigramme de l'exemple 2 (figure 2).

Soit α (resp. β, γ) le chemin A-B (resp. le cycle B-B, le chemin B-C). Il s'agit de prouver :

i) $(p(a) \; \underline{\text{et}} \; C(x\,;\alpha) \Rightarrow p_B(a, \alpha(x))$

$C(x\,;\alpha) = \underline{\text{vrai}}$ (de A il n'y a aucune condition à remplir pour atteindre B).
$\alpha(x) = (0, 0)$ (de A à B il n'y a eu que l'initialisation de x_1 et x_2).

Il est immédiat que :

$$(a \geqslant 0 \; \underline{\text{et}} \; \underline{\text{vrai}} \Rightarrow 0 = 0^2 \; \underline{\text{et}} \; 0 \leqslant a)\,.$$

ii) $(p_B(a, x)$ et $C(x\,;\beta) \Rightarrow p_B(a, \beta(x)))$

$C(x\,;\beta) = (x_2 \neq a)$ (condition de parcours du cycle).
$\beta(x) = (x_1 + 2\,x_2 + 1, x_2 + 1)$.

On vérifie que pour x quelconque :

$$x_1 = x_2^2 \; \underline{\text{et}} \; x_2 \leqslant a \; \underline{\text{et}} \; x_2 \neq a \Rightarrow x_1 + 2\,x_2 + 1 = (x_2 + 1)^2 \; \underline{\text{et}} \; x_2 + 1 \leqslant a\,.$$

iii) $(p_B(a, x) \; \underline{\text{et}} \; C(x\,;\gamma)) \Rightarrow q(a, \gamma(x))$

$C(x\,;\gamma) = (x_2 = a)$ (condition de sortie du cycle).
$\gamma(x) = x$.

Il est clair que :

$$(x_1 = x_2^2 \; \underline{\text{et}} \; x_2 \leqslant a \; \underline{\text{et}} \; x_2 = a \Rightarrow x_1 = a^2)\,. \qquad \square$$

Exemple 7

Considérons l'organigramme suivant (figure 5) qui exprime l'algorithme d'Euclide pour le calcul du pgcd de deux entiers naturels a_1, a_2 ; $reste(x_1, x_2)$ est le reste de la division euclidienne de x_1 par x_2.

Appliquons la méthode proposée :

<u>Pas 1</u> : choisissons les points de coupure A, B, C (figure 5).

<u>Pas 2</u> : p_A et p_C sont connus :

$$p_A = a_1 \geqslant 0 \; \underline{\text{et}} \; a_2 \geqslant 0$$
$$p_C = (x_1 = pgcd(a_1, a_2))\,.$$

En appliquant la règle 4, p_B doit vérifier :

$$(p_B(a, x) \; \underline{\text{et}} \; x_2 = 0) \Rightarrow x_1 = pgcd(a_1, a_2)\,.$$

Déduire p_B par « élargissement » revient à y faire figurer x_1 ainsi que x_2 ; on peut imaginer pour p_B :

$$p_B(a, x) = (pgcd(x_1, x_2) = pgcd(a_1, a_2))\,.$$

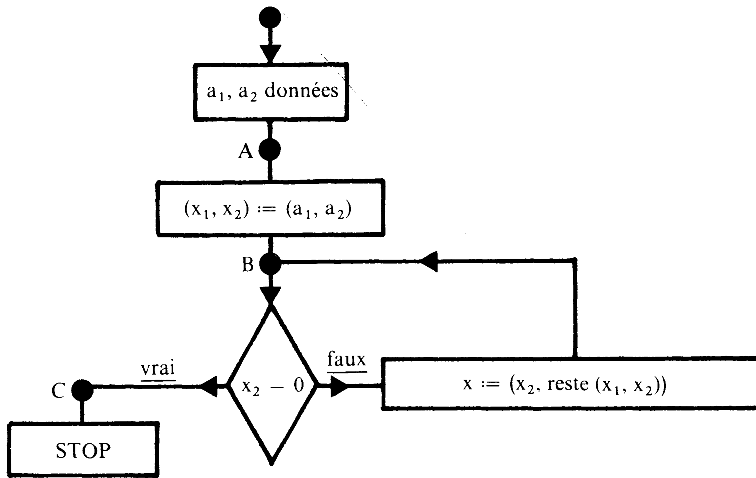


Figure 5 Algorithme d'Euclide.

Il n'est pas inutile de se rappeler la propriété suivante du pgcd :

(1) pour tout $l \geqslant 0$ $pgcd(l, 0) = l$.

Pour trouver p_B, on peut aussi appliquer la règle 3 :

(2) $$(x_{1,0}, x_{2,0}) = (a_1, a_2)$$

(3) $$(x_{1,n}, x_{2,n}) = (x_{2,n-1}, \text{reste}(x_{1,n-1}\ x_{2,n-1}))\ .$$

Pour éliminer n, on peut ici encore utiliser une propriété du pgcd :

(4) pour $l \neq 0$ et $k \geqslant 0$ $pgcd(l, reste\,(k, l)) = pgcd(k, l)$

On déduit de (3) :

$$pgcd(x_{1,n}, x_{2,n}) = pgcd(x_{1,n-1}, x_{2,n-1})\ ,$$

puis, par récurrence et d'après (2) :

$$pgcd(x_{1,n}, x_{2,n}) = pgcd(a_1, a_2)\ .$$

On retrouve ainsi l'assertion trouvée par application de la règle 4.

Pas 3

- Le long de A-B :

$$C(x\ ;\ \text{A-B}) = \underline{\text{vrai}}$$
$$(\text{A-B})\,(x) = (a_1, a_2)\ .$$

On a bien :

$$(a_1 \geqslant 0\ \underline{\text{et}}\ a_2 \geqslant 0\ \underline{\text{et}}\ \underline{\text{vrai}}) \Rightarrow pgcd(a_1, a_2) = pgcd(a_1, a_2)\ .$$

• Le long du cycle B-B :

$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ $\underline{\text{et}}$ $x_2 \neq 0 \Rightarrow$
$pgcd(x_2, reste(x_1, x_2)) = pgcd(a_1, a_2)$,

ce qui se déduit de la propriété (4).

• Le long de B-C :

$$pgcd(x_1, x_2) = pgcd(a_1, a_2) \; \underline{\text{et}} \; x_2 = 0 \Rightarrow x_1 = pgcd(a_1, a_2)$$

qui se déduit de (1).

On peut remarquer qu'on n'a pas utilisé ici l'hypothèse $a_1 \geqslant 0$ et $a_2 \geqslant 0$, cependant si l'on convient que *pgcd* est défini sur $\mathbb{N}^2$ et *reste* sur

$$\mathbb{N} \times (\mathbb{N} - \{0\}),$$

il serait utile, en toute rigueur, de vérifier que toutes les opérations sont définies, en précisant chaque fois $x_1 \geqslant 0$, $x_2 \geqslant 0$, ...

L'organigramme de la figure 5 suppose que *reste* est une opération élémentaire. Si on ne dispose que de la soustraction, on est amené à calculer explicitement le reste comme dans l'exemple 1. On peut obtenir l'organigramme de la figure 6 ; nous allons prouver sa correction partielle par rapport à :

$$\mathrm{p_A}(a) = (a_1 \geqslant 0 \; \underline{\text{et}} \; a_2 \geqslant 0)$$

et $$\mathrm{p_S}(a, x) = (x_1 = pgcd(a_1, a_2)) .$$

Pas 1 : Choix des points de coupures (voir figure 6).

Pas 2 : Choix des assertions :
La règle 4 permet d'imaginer :

$$\mathrm{p_B}(a, x) = (\mathrm{pgcd}(x_1, x_2) = \mathrm{pgcd}(a_1, a_2)) .$$

On utilise ici évidemment la propriété :

(1) pour tout l $\mathrm{pgcd}(l, l) = l$.

La règle 3 permet de trouver :

$$\mathrm{p_C}(a, x) = \mathrm{p_D}(a, x) = (pgcd(x_1, x_2) = pgcd(a_1, a_2))$$

en utilisant la propriété :

(2) pour $k > l$ $pgcd(k, l) = pgcd(k - l, l)$.

Pas 3 : indiquons sommairement les démonstrations à faire :

• Le long de A-B :

$$(a_1 \geqslant 0 \; \underline{\text{et}} \; a_2 \geqslant 0 \; \underline{\text{et}} \; \underline{\text{vrai}}) \Rightarrow (a_1, a_2) = (a_1, a_2) .$$

• Le long de B-C :

$$(pgcd(x_1, x_2) = pgcd(a_1, a_2) \; \underline{\text{et}} \; x_1 \neq x_2) \Rightarrow pgcd(x_1, x_2) = pgcd(a_1, a_2) .$$

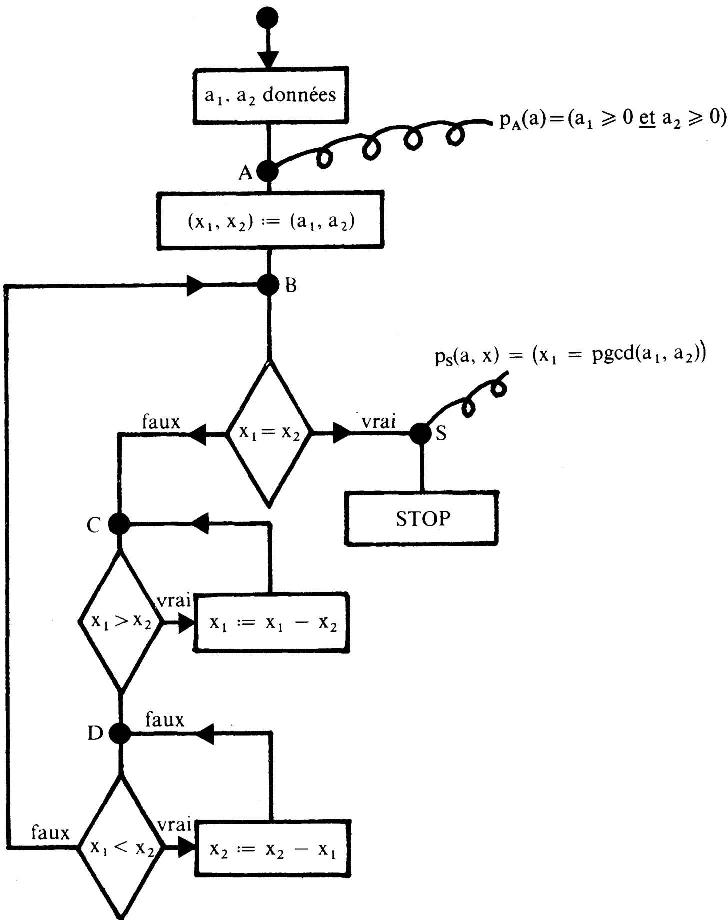


Figure 6 Calcul du pgcd par soustractions.

- Le long de B-S :

$$(pgcd(x_1, x_2) = pgcd(a_1, a_2) \ \underline{\text{et}}\ x_1 = x_2) \Rightarrow x_1 = pgcd(a_1, a_2)$$

en utilisant (1).

- Le long de C-D :

$$(pgcd(x_1, x_2) = pgcd(a_1, a_2) \ \underline{\text{et}}\ x_1 \leqslant x_2) \Rightarrow pgcd(x_1, x_2) = pgcd(a_1, a_2)\,.$$

- Le long de C-C :

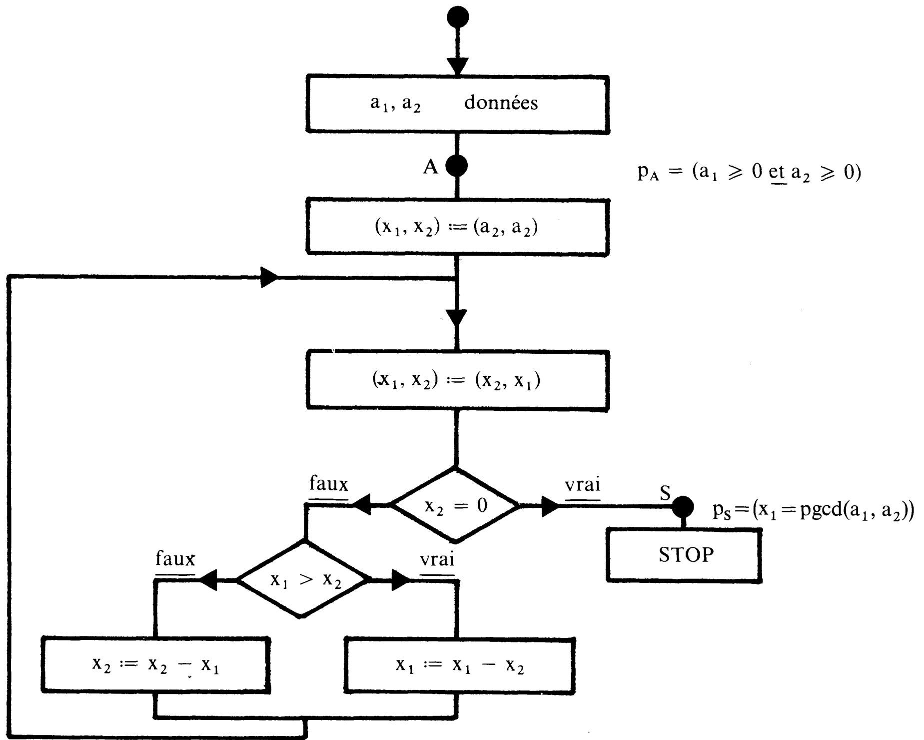
$$(pgcd(x_1, x_2) = pgcd(a_1, a_2) \ \underline{\text{et}}\ x_1 > x_2) \Rightarrow pgcd(x_1 - x_2, x_2) = pgcd(a_1, a_2)$$

en utilisant (2).

- Les chemins D-D et D-B se traitent de façon analogue. □

Exercice 2

Démontrer la condition partielle de l'organigramme de la figure 7. □



c) Evaluation des conditions de compatibilité $C(x\ ;\ \alpha)$ *et de la valeur de la variable à l'issue d'un chemin* $\alpha(x)$

Lorsqu'un chemin α contient de nombreuses instructions, la détermination de $C(x\ ;\alpha)$ et de $\alpha(x)$ peut être compliquée. Donnons donc des définitions complètes de $C(x\ ;\alpha)$ et de $\alpha(x)$ (ces définitions pourront servir également à une démonstration précise du théorème 1). Procédons simplement par récurrence sur la longueur de α ; en effet, α est une suite d'instructions comportant des affectations de forme générale $x := f(a, x)$ et des tests de forme $t(a, x)$ ou $\bar{t}(a, x)$ selon que le chemin se poursuit par la branche vrai ou la branche faux du test. Convenons enfin de noter Λ le chemin vide. Quatre cas sont à envisager pour les définitions de $C(x\ ;\alpha)$ et $\alpha(x)$:

$\alpha = \Lambda$

$$\begin{cases} C(x\ ;\ \Lambda) = \underline{\text{vrai}} \\ \Lambda(x) \quad\ = x \end{cases}$$

$\alpha = \beta t(a, x)$

$$\begin{cases} C(x\ ;\ \alpha) = C(x\ ;\ \beta)\ \underline{\text{et}}\ t(a, \beta(x)) \\ \quad \alpha(x) = \beta(x) \end{cases}$$

$$\alpha = \beta\bar{t}(a, x)$$

$$\begin{cases} C(x\ ;\alpha) = C(x\ ;\beta)\ \underline{\text{et}}\ \underline{\text{non}}\ t(a, \beta(x)) \\ \alpha(x) = \beta(x) \end{cases}$$

$$\alpha = \beta(x := f(a, x))$$

$$\begin{cases} C(x\ ;\alpha) = C(x\ ;\beta) \\ \alpha(x) = f(a, \beta(x))\,. \end{cases}$$

Ainsi l'évaluation de $C(x\ ;\alpha)$ et de $\alpha(x)$ se fait en progressant depuis l'origine A du chemin α jusqu'à son extrémité B. On pourrait aussi exprimer des définitions analogues en remontant de B vers A [MAN, 74].

Exemple 8 : Détermination de $C(x\ ;\alpha)$ et de $\alpha(x)$

La figure 8 représente un chemin α de A vers B ; par convention nous avons noté α_i le sous-chemin réduit aux i premières instructions de α $(\alpha_0 = \Lambda,\ \ \alpha_6 = \alpha)$.

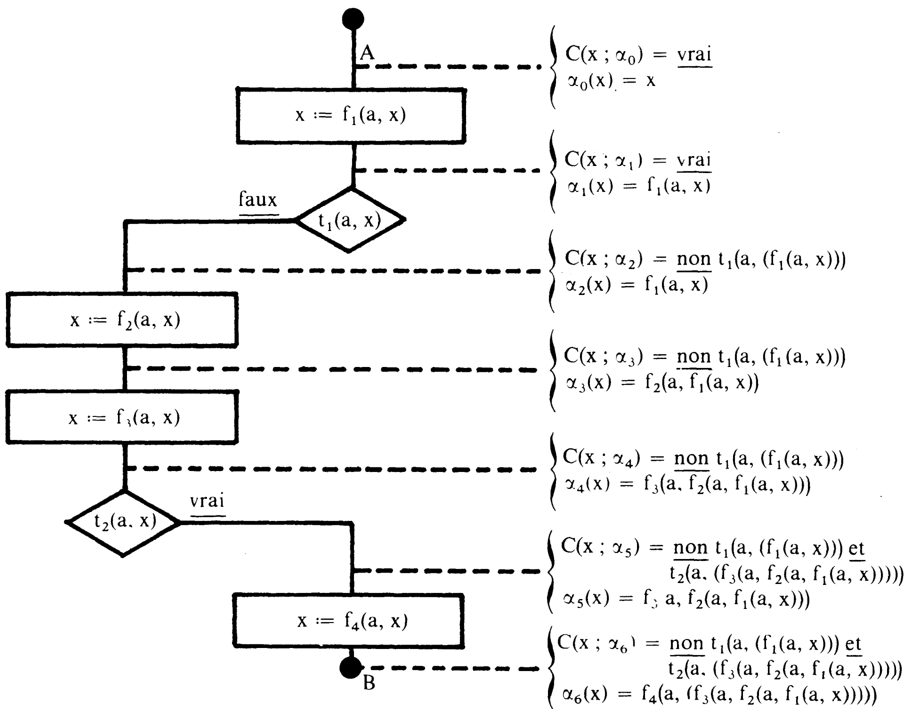


Figure 8 Calcul de $C(x\ ;\ A - B)$ et de $(A - B)\,(x)$.

Ainsi pour le chemin α de A à B :

$$C(x\ ;\alpha) = \underline{\text{non}}\ t_1(a, (f_1(a, x_0)))\ \underline{\text{et}}\ t_2(a, (f_3(a, f_2(a, f_1(a, x)))))$$

$$\alpha(x) = f_4(a, (f_3(a, f_2(a, f_1(a, x)))))\,. \qquad \square$$

Exercice 3

Retrouver à l'aide de cette méthode, les valeurs calculées le long des chemins et les conditions de cheminement de l'organigramme de l'exemple 2. □

d) De nouveaux exemples

Exemple 9 : Classement des éléments d'un tableau par rapport à une valeur

Considérons l'organigramme suivant (figure 9) ; il est supposé transformer un tableau $b[1 : n]$ donné en un tableau $t[1 : n]$ et fournir un entier j tels que :

(1) $t[1 : j] \leqslant b[1] \leqslant t[j+1 : n]$ et t perm b,

en convenant que $t[k : l] \leqslant a$ est une abréviation de

$$\forall p \quad k \leqslant p \leqslant l \Rightarrow t[p] \leqslant a$$

et que t perm b signifie que le tableau t est une permutation du tableau b (nous supposons ici que perm est un prédicat de base).

Sur l'organigramme, nous avons utilisé l'opération *échange* : par définition *échange*(i, j, t) est le tableau déduit de t en transposant $t[i]$ et $t[j]$.

Enfin sur cet exemple les données a forment le vecteur :

$$a = (n, b[1 : n]) = (n, b[1], b[2], \ldots, b[n])$$

et les variables sont représentées par :

$$x = (i, j, t[1 : n]) = (i, j, t[1], \ldots, t[n]) .$$

Prouvons la correction de cet organigramme.

Pas 1 : Recherche des points de coupure.

En appliquant les règles 1 et 2, on obtient comme points de coupure A, B, C, D et E. Pour décrire plus facilement les chemins directs nous étiquetons aussi d'autres arcs du graphe (e_1, e_2, e_3, e_4).

Avant d'aborder le choix des assertions (pas 2) nous allons insérer ici un pas 1′ qui a pour but le calcul systématique des conditions de compatibilité et des valeurs des variables à l'issue d'un chemin direct.

Pas 1′ : Conditions de compatibilité et valeurs des variables à l'issue d'un chemin direct.

Détaillons le calcul le long de deux chemins ; les résultats complets sont résumés par la figure 10.

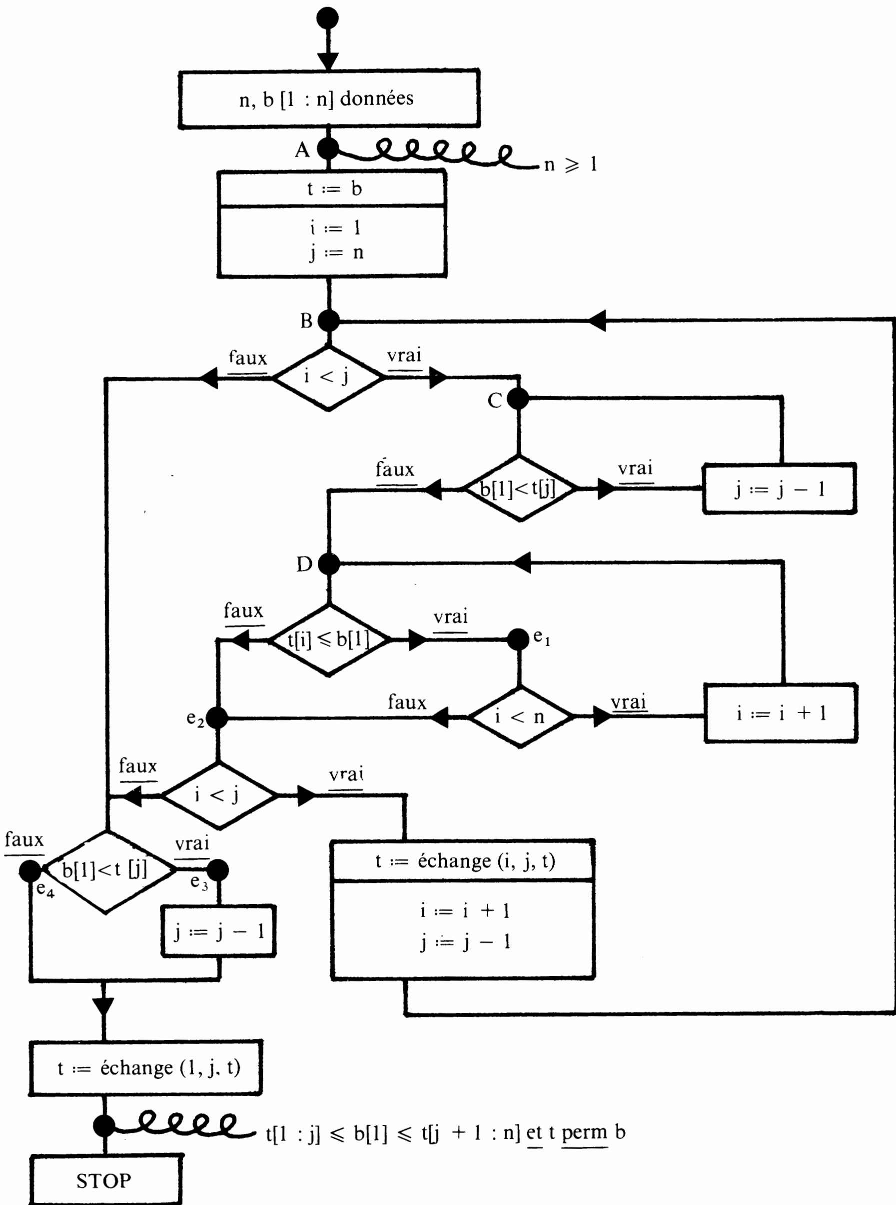
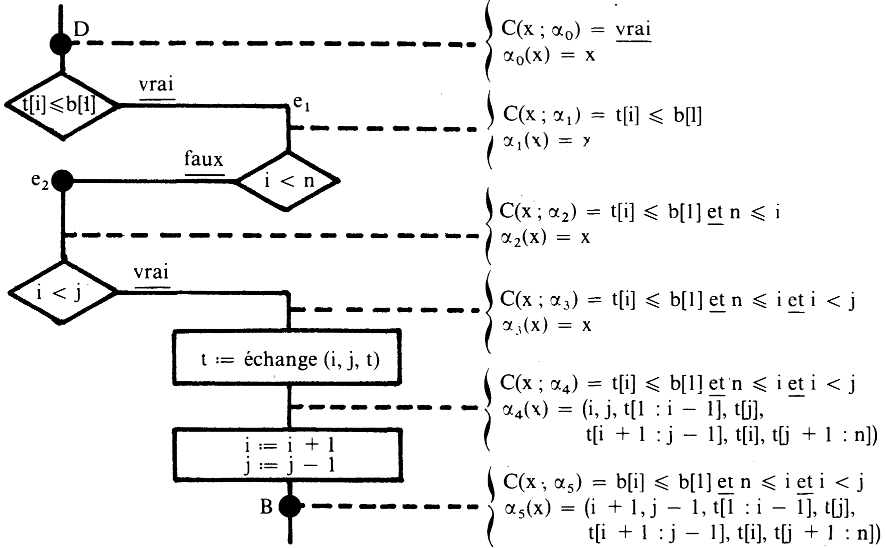


Figure 9 Classement des éléments de t par rapport à b [1].

Chemin $\alpha = D - e_1 - e_2 - B$

α_i est la suite des i premières instructions de α :



Chemin $\alpha = D - e_2 - e_3 - E$

D

t[i] ≤ b[1]

faux

e_2

i < j

faux

b[1] < t[j]

vrai

e_3

j := j − 1

t := échange (1, j, t)

E

$C(x\,;\alpha_0) = \underline{\text{vrai}}$
$\alpha_0(x) = x$

$C(x\,;\alpha_1) = b[1] < t[i]$
$\alpha_1(x) = x$

$C(x\,;\alpha_2) = b[1] < t[i]\ \underline{\text{et}}\ j \leq i$
$\alpha_2(x) = x$

$C(x\,;\alpha_3) = b[1] < t[i]\ \underline{\text{et}}\ j \leq i\ \underline{\text{et}}\ b[1] \leq t[j]$
$\alpha_3(x) = x$

$C(x\,;\alpha_4) = C(x\,;\alpha_3)$
$\alpha_4(x) = (i, j - 1, t[1 : n])$

$C(x\,;\alpha_5) = C(x\,;\alpha_3)$
$\alpha_5(x) = (i, j - 1, t[j - 1], t[2 : j - 2], t[1], t[j : n])$

Chemin α	$C(x ; \alpha)$	$\alpha(x)$ $(x = (i, j, t[1 : n])$
A-B	vrai	$(1, n, b[1 : n])$
B-C	$i < j$	$\underline{x}$
C-C	$b[1] < t[j]$	$(i, j - 1, t[1 : n])$
C-D	$t[j] \leqslant b[1]$	$\underline{x}$
D-e_1-D	$t[i] \leqslant b[1]$ et $i < n$	$(i + 1, j, t[1 : n])$
D-e_1-e_2-B	$t[i] \leqslant b[1]$ et $n \leqslant i$ et $i < j$	$(i + 1, j - 1, t[1 : i - 1], t[j], t[i + 1 : j - 1], t[i], t[j + 1 : n])$
D-e_2-B	$b[1] < t[i]$ et $i < j$	$(i + 1, j - 1, t[1 : i - 1], t[j], t[i + 1 : j - 1], t[i], t[j + 1 : n])$
De_1-e_2-e_3-E	$t[i] \leqslant b[1]$ et $n \leqslant i$ et $j \leqslant i$ et $b[1] \leqslant t[j]$	$(i, j - 1, t[j - 1], t[2 : j - 2], t[1], t[j : n])$
D-e_2-e_3-E	$b[1] < t[i]$ et $j \leqslant i$ et $b[1] < t[j]$	$(i, j - 1, t[j - 1], t[2 : j - 2], t[1], t[j : n])$
D-e_1-e_2-e_4-E	$t[i] \leqslant b[1]$ et $n \leqslant i$ et $j \leqslant i$ et $t[j] \leqslant b[1]$	$(i, j, t[j], t[2 : j - 1], t[1], t[j + 1 : n])$
D-e_2-e_4-E	$b[1] < t[i]$ et $j \leqslant i$ et $t[j] \leqslant b[1]$	$(i, j, t[j], t[2 : j - 1], t[1], t[j + 1 : n])$
B-e_3-E	$j \leqslant i$ et $b[1] < t[j]$	$(i, j - 1, t[j - 1], t[2 : j - 2], t[1], t[j : n])$
B-e_4-E	$j \leqslant i$ et $t[j] \leqslant b[1]$	$(i, j, t[j], t[2 : j - 1], t[1], t[j + 1 : n])$

Figure 10 Calcul de $C(x ; \alpha)$ et de $\alpha(x)$ pour les chemins directs de l'organigramme de la figure 9.

Exercice 4

Définir formellement le prédicat t perm b. □

Exercice 5

Terminer la preuve de la correction partielle du programme de la figure 9.

Exercice 6 □

Prouver la correction partielle du programme de la figure 11 relativement aux prédicats d'entrée p et de sortie q définis par :

$$p = a_2 > 0$$

$$q = (\exists k)\, a_1 = a_2 * k + x_1 \text{ et } x_1 < a_2$$ □

(ce programme calcule donc le reste x_1 de la division de a_1 par a_2).

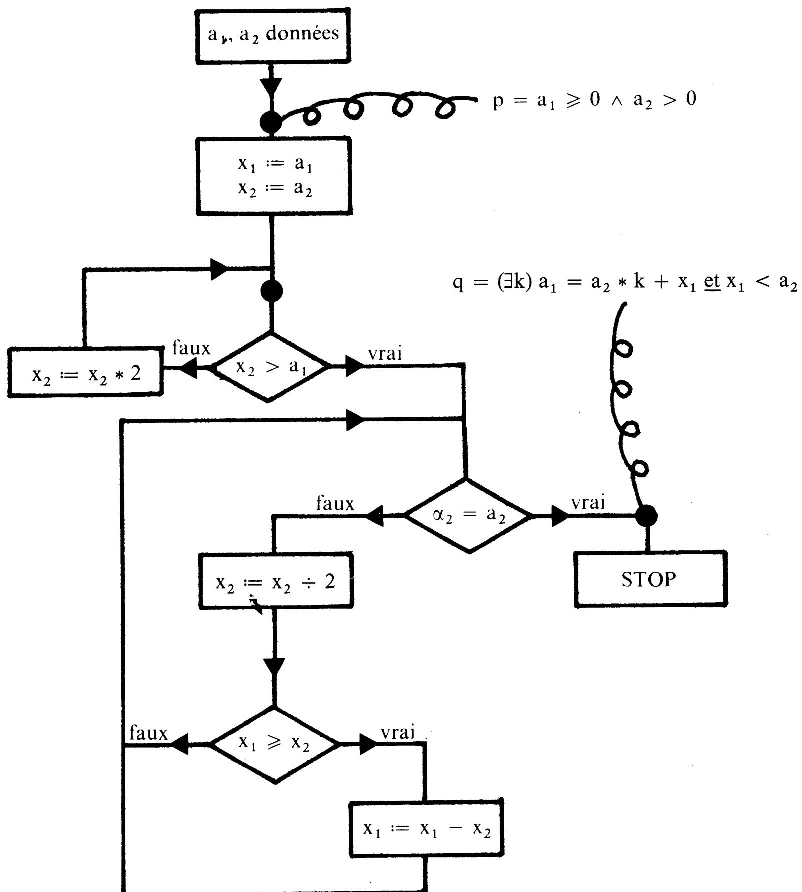


Figure 11 Calcul rapide du reste (sur des nombres représentés en binaire).

Exercice 7

Prouver la correction partielle du programme de la figure 12 relativement aux prédicats p et q (voir § 2.5, exemple 9). □

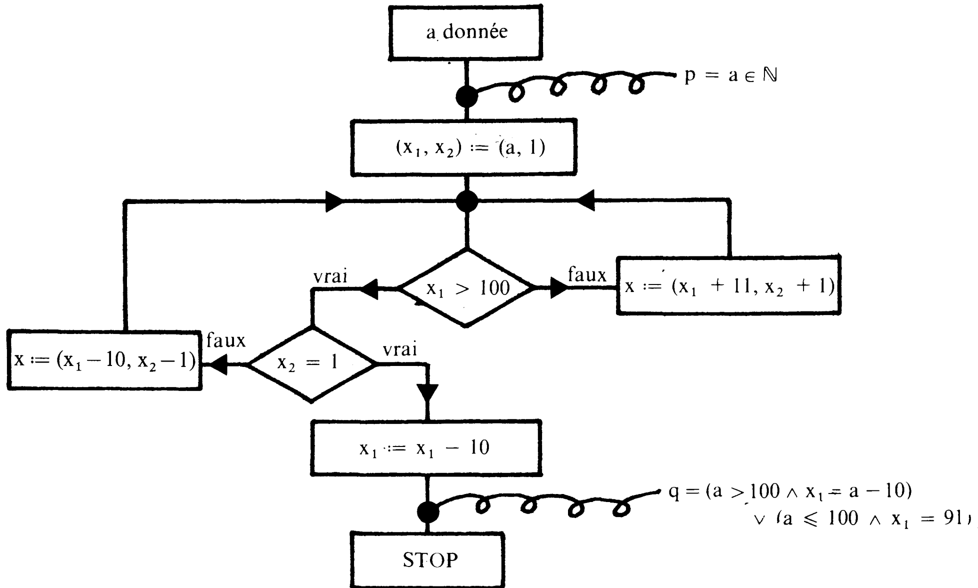


Figure 12 Calcul de la fonction 91.

2.4 Problème de la terminaison

Souvent, on montre qu'un algorithme se termine indépendamment de sa correction partielle. La démonstration, évidente dans le cas de cycles « pour i de 1 à n faire », demande plus de réflexion dans le cas général. Intuitivement, il s'agit de s'assurer qu'un calcul termine. Pour cela, il suffit d'exhiber pour chaque cycle, une « quantité » qui décroît à chaque itération, mais qui ne peut décroître indéfiniment. Ainsi, pour l'exemple 1, le reste y décroît à chaque itération (si $b > 0$) en restant positif ou nul ; or il n'existe pas de suite infinie d'entiers naturels strictement décroissante. On en déduit que pour toute donnée, le cycle ne peut être parcouru qu'un nombre fini de fois : le calcul se termine.

On formalise cette idée en introduisant la notion **d'ensemble bien fondé :** c'est un ensemble F muni d'une relation d'ordre $\preccurlyeq$ (*) pour laquelle toute suite décroissante est stationnaire. Pour prouver la terminaison d'un organigramme, il **suffit** alors :

1) de choisir un ensemble de points de coupures tel que chaque cycle soit coupé au moins une fois,

(*) Dans la suite $x \prec y$ signifie $x \preccurlyeq y$ et $x \neq y$.

2) d'associer à chaque point de coupure A une fonction f_A qui associe aux données et aux valeurs des variables x un élément de F,

3) de prouver, pour tout chemin direct α (d'origine A et d'extrémité **B**), que :

$$C(x\,;\alpha) \Rightarrow f_B(a, \alpha(x)) \prec f_A(a, x)\,.$$

Il est clair en effet que la condition 3 et la définition d'ensemble bien fondé interdisent de parcourir une infinité de fois tout chemin de l'organigramme (et en particulier tout cycle).

Exemple 10

On peut ainsi montrer qu'une partie de belote se termine : à chaque pli, quatre cartes sont abattues et le jeu ne peut avoir lieu que s'il reste des cartes en jeu. Considérons la variante du jeu de dames dans laquelle on arrête la partie à l'apparition de la première dame ; si l'on autorise le mouvement des pièces en avant et en arrière, la partie risque de ne jamais se terminer ; mais si l'on impose d'aller en avant, même si l'on autorise des prises en arrière, la partie se termine nécessairement dans le sens suivant : prenons en effet pour F l'ensemble $\mathbb{N} \times \mathbb{N}$ muni de la relation d'ordre $(a, b) \preccurlyeq (c, d)$ définie par : $a < c$ <u>ou</u> $(a = c$ <u>et</u> $b \leqslant d)$.

A chaque configuration du jeu, on associe le couple (x, y) où x représente le nombre de pièces en jeu et y est la somme (étendue à l'ensemble des pions en jeu) de la distance du pion à la ligne la plus avant du jeu. A chaque coup, soit un pion avance et donc y diminue (x reste constant), soit il y a prise d'un pion et donc le nombre total de pions diminue. □

Cependant la condition 3 ci-dessus est trop forte : elle exprime la terminaison du programme sans tenir compte des initialisations et cela est souvent trop restrictif : ainsi, dans le cas de l'exemple 1, il a fallu tenir compte de l'hypothèse $b > 0$.

On est ainsi conduit à introduire en tout point de coupure A une assertion q_A caractéristique de la valeur des variables en A. La formule à prouver en 3 devient alors :

$$(q_A(a, x) \ \underline{\text{et}} \ C(x\,;\alpha) \Rightarrow f_B(a, \alpha(x)) \prec f_A(a, x))\,.$$

Exemple 11

Montrons la terminaison du programme de la figure 4 (racine carrée entière). Posons $F = \{ x \in \mathbb{Z} \mid x \geqslant -1 \}$ et munissons-le de l'ordre habituel sur les entiers.

Sur le cycle $\alpha =$ **B-B** on a :

$$C(x\,;\alpha) = x_2 \leqslant a \quad \text{et} \quad \alpha(x) = (x_1 + 1, x_2 + x_3 + 2, x_3 + 2)\,.$$

Posons $f_B(a, x) = a - x_2$ (déduit du test de sortie).

On ne peut démontrer directement l'implication :

$$x_2 \leqslant a \Rightarrow a - (x_2 + x_3 + 2) \prec a - x_2\,,$$

qui revient à : $x_2 \leqslant a \Rightarrow -x_3 - 2 < 0$.

Imposons donc une condition supplémentaire sur x_3 au point B, par exemple $q_B(a, x) = x_3 \geqslant 0$. On obtient bien alors :

$$x_3 \geqslant 0 \ \underline{\text{et}}\ x_2 \leqslant a \Rightarrow -x_3 - 2 < 0 .$$

Il reste à prouver qu'au point B, $q_B(a, x)$ est vérifié si la donnée a vérifie $p(a)$, ce qui est immédiat. □

Remarquons sur cet exemple que les assertions nécessaires à la résolution du problème de terminaison ne sont pas les mêmes que les assertions utilisées pour la correction partielle : on ne s'intéresse pas aux mêmes caractéristiques de l'organigramme. Nous sommes conduits à énoncer :

Théorème 3

Soit un organigramme π muni d'un étiquetage localement correct. Supposons qu'on peut associer à tout point de coupure A une fonction f_A de D^{n+p} dans un ensemble bien fondé $(F, \preccurlyeq)$ de façon que :

(1) $f_A(a, x)$ est défini si $q_A(a, x)$ (q_A est l'assertion associée au point de coupure A).

(2) pour tout chemin direct α de A vers B

$$q_A(a, x)\ \underline{\text{et}}\ C(x\,;\alpha) \Rightarrow f_B(a, \alpha(x)) \prec f_A(a, x) .$$

Alors π se termine pour toute donnée vérifiant l'assertion d'entrée. □

Ceci résulte immédiatement de la propriété d'un ensemble bien fondé et du fait qu'un cycle est formé d'un nombre fini de chemins directs.

Une variante de ce théorème consiste à s'intéresser plus aux cycles qu'aux chemins directs, ce qui est naturel : si le nombre d'itérations pour chaque cycle est borné, il en est de même de l'algorithme.

Théorème 4

Soit un organigramme π muni d'un étiquetage localement correct coupant tout cycle. Supposons qu'on peut associer à tout point de coupure A (muni de l'assertion q_A) appartenant à un cycle, une fonction f_A de D^{n+p} dans un ensemble bien fondé $(F, \preccurlyeq)$ vérifiant :

(1) $f_A(a, x)$ est défini si $q_A(a, x)$

(2) pour tout cycle direct α d'origine et d'extrémité A :

$$q_A(a, x)\ \underline{\text{et}}\ C(x\,;\alpha) \Rightarrow f_A(a, \alpha(x)) \prec f_A(a, x) .$$

Alors π se termine pour toute donnée vérifiant l'assertion d'entrée. □

Remarquons que les démonstrations de type (1) et (2) à faire en appliquant le théorème 4 sont souvent moins nombreuses que celles utilisées par le théorème 3 car les cycles sont considérés globalement. Pour utiliser ce théorème, on peut munir chaque cycle d'un **compteur** qui permet de définir la fonction f comme nous le verrons plus loin.

2.5 Méthodologie pour démontrer la terminaison

a) *Méthode générale* : Elle résulte immédiatement des théorèmes précédents.

• Pas 1 : Choisir un ensemble bien fondé $(F, \preccurlyeq)$.

• Pas 2 : Déterminer un ensemble de points de coupure tels que chaque cycle soit coupé au moins une fois.

• Pas 3 : Associer à chaque point de coupure A une fonction f_A de D^{n+p} dans F et un prédicat q_A.

Les recherches de F, f_A, q_A sont en général conjointes. Indiquons simplement que dans le cas où le test de fin de cycle est de la forme $x_k = x_j$ ou $(x_k \geqslant x_j)$, les fonctions et prédicats attachés au point précédant ce test peuvent être de la forme :

- $f_A(a, x) = x_k - x_j$
- $q_A(a, x) = (x_k > x_j \ \underline{\text{ou}} \ x_k < x_j)$.

• Pas 4 : Vérifier la correction locale de l'étiquetage ainsi défini.

• Pas 5 : Vérifier en tout point A :

$$(q_A(a, x) \Rightarrow f_A(a, x) \text{ défini}) .$$

• Pas 6 : Vérifier pour tout couple de points de coupure A, B et tout chemin direct α de A à B :

$$(q_A(a, x) \ \underline{\text{et}} \ C(x ; \alpha) \Rightarrow f_B(a, \alpha(x)) \prec f_A(a, x))$$

ou (variante utilisant le théorème 4) :

• Pas 6′ : Vérifier pour tout cycle α de A à A :

$$(q_A(a, x) \ \underline{\text{et}} \ C(x ; \alpha) \Rightarrow f_A(a, \alpha(x)) \prec f_A(a, x)) .$$

Exemple 12

Démontrons l'arrêt de l'organigramme de la figure 2 (calcul du carré).

• Pas 1 : Prenons $F = \mathbb{N}$ muni de l'ordre usuel.

• Pas 2 : Les points de coupure sont A, B, C comme dans la correction partielle.

• Pas 3 : Prenons

$$q_A(a, x) = a \geqslant 0$$
$$q_B(a, x) = a \geqslant x_2 \quad f_B(a, x) = a - x_2$$
$$q_C(a, x) = \underline{\text{vrai}}$$

• Pas 4 : On vérifie : sur α_1 = A-B :

$$(q_A(a, x) \ \underline{\text{et}} \ C(x ; \alpha_1)) \Rightarrow q_B(a, \alpha_1(x))$$

car $(a \geqslant 0 \ \underline{\text{et}} \ \underline{\text{vrai}} \Rightarrow a \geqslant 0)$;

sur α_2 = B-B :

$$(q_B(a, x) \ \underline{\text{et}} \ C(x ; \alpha_2)) \Rightarrow q_B(a, \alpha_2(x))$$

car $(a \geqslant x_2 \ \underline{\text{et}} \ x_2 \neq a \Rightarrow a \geqslant x_2 + 1)$;

sur α_3 = B-C :

$$q_B(a, x) \ \underline{\text{et}}\ C(x ; \alpha_3) \Rightarrow q_C(a, \alpha_3(x))$$

car $\qquad a \geqslant x_2 \ \underline{\text{et}}\ x_2 = a \Rightarrow \underline{\text{vrai}}$

- Pas 5 : Il est immédiat que

$$(q_B(a, x) \Rightarrow f_B(a, x) \in F)$$

car $\qquad (a \geqslant x_2 \Rightarrow a - x_2 \geqslant -1)$.

- Pas 6′ : Il suffit de prouver

$$(q_B(a, x) \ \underline{\text{et}}\ C(x ; \alpha_2) \Rightarrow f_B(a, \alpha_2(x)) \prec f_B(a, x))$$

c'est-à-dire $(a \geqslant x_2 \ \underline{\text{et}}\ x_2 \neq a \Rightarrow a - x_2 - 1 < a - x_2)$. □

Exercice 8

Prouver l'arrêt de l'organigramme de la figure 6. □

b) Méthode des compteurs : Un compteur est une variable auxiliaire associée à un cycle, incrémentée (ou décrémentée) de 1 à chaque passage dans le cycle. Présentons leur utilisation sur un exemple.

Exemple 13

Reprenons l'organigramme du calcul du pgcd par soustractions (figure 6) et introduisons trois compteurs i_1, i_2, i_3

- Pas 1 : F = ℕ muni de l'ordre habituel.
- Pas 2 : Les points de coupure sont A, B, C, D, S.
- Pas 3 : $f_B(a, x) = i_3$; $f_C(a, x) = i_1$; $f_D(a, x) = i_2$.

i_1, i_2, i_3 étant des compteurs, il est évident que leur valeur décroît strictement à chaque passage dans le cycle associé. Il suffit donc de prouver que les ensembles des valeurs prises sont bornés inférieurement, d'où la nécessité d'un choix judicieux des invariants :

— $x_1 > 0$ et $x_2 > 0$ doit être vrai en B, C et D.

— A chaque décrémentation de i_1, x_1 est au moins décrémenté de 1 ; les valeurs initiales étant les mêmes, on doit avoir $x_1 \leqslant i_1$.

De même $x_2 \leqslant i_2$, ce qui entraîne la minoration de i_1 et i_2.

— Il est impossible de parcourir un cycle B-B sans passer par l'un des cycles C-C ou D-D : toute décrémentation de i_3 correspond à une décrémentation d'une valeur au moins égale de $i_1 + i_2$, or à l'origine on a $i_3 = i_1 + i_2$, donc dans tous les cas $i_3 \geqslant i_1 + i_2$ (d'où la dernière minoration cherchée).

On peut donc prendre pour assertions :

$q_A(a) = a_1 > 0 \ \underline{\text{et}}\ a_2 > 0$

$q_B(a, x) = q_C(a, x) = x_1 > 0 \ \underline{\text{et}}\ x_2 > 0 \ \underline{\text{et}}\ x_1 \leqslant i_1 \ \underline{\text{et}}\ x_2 \leqslant i_2 \ \underline{\text{et}}\ i_3 \geqslant i_1 + i_2$.

$q_D(a, x) = q_B(a, x) \ \underline{\text{et}}\ (x_1 \geqslant x_2 \Rightarrow i_3 > i_1 + i_2)$

et, par exemple, $q_S(a, x) = \underline{\text{vrai}}$

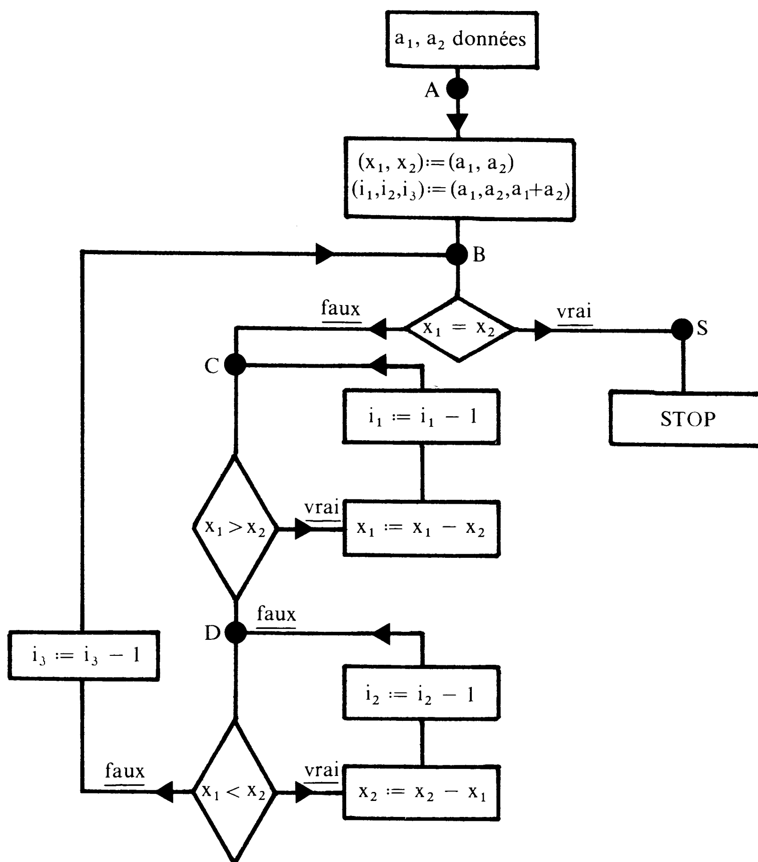


Figure 13 Calcul du pgcd avec utilisation de compteurs.

- Pas 4 :

— $q_A(a) \Rightarrow q_B(a, x)$ (provient de l'initialisation de x et des compteurs).

— Sur le cycle C-C, il est immédiat que :

$$q_C(a, x) \ \underline{\text{et}}\ x_1 > x_2 \Rightarrow x_1 - x_2 > 0 \ \underline{\text{et}}\ x_2 > 0 \ \underline{\text{et}}\ x_1 - x_2 \leqslant i_1 - 1 \ \underline{\text{et}}\ x_2 \leqslant i_2 \ \underline{\text{et}}\ i_3 \geqslant i_1 - 1 + i_2 .$$

— On montre de façon analogue que q_D est un invariant du cycle D-D.

— Sur le chemin direct D-B il faut prouver :

$$(q_D(a, x) \ \underline{\text{et}}\ x_2 \leqslant x_1) \Rightarrow (x_1 > 0 \ \underline{\text{et}}\ x_2 > 0 \ \underline{\text{et}}\ x_1 \leqslant i_1 \ \underline{\text{et}}\ x_2 \leqslant i_2 \ \underline{\text{et}}\ i_3 - 1 \geqslant i_1 + i_2)$$

ou encore en développant la partie gauche :

$$(x_1 > 0 \ \underline{\text{et}}\ x_2 > 0 \ \underline{\text{et}}\ x_1 \leqslant i_1 \ \underline{\text{et}}\ x_2 \leqslant i_2 \ \underline{\text{et}}\ x_2 \leqslant x_1 \ \underline{\text{et}}\ i_3 > i_1 + i_2) \Rightarrow (x_1 > 0 \ \underline{\text{et}}\ x_2 > 0 \ \underline{\text{et}}\ x_1 \leqslant i_1 \ \underline{\text{et}}\ x_2 \leqslant i_2 \ \underline{\text{et}}\ i_3 - 1 \geqslant i_1 + i_2)$$

ce qui est immédiat.

— Sur les autres chemins directs B-C, B-S, C-D les vérifications sont évidentes.

• Pas 5 : Il résulte immédiatement de la forme des invariants que les compteurs i_1, i_2, i_3 sont positifs en tous les points de coupure.

• Pas 6′ : Chaque compteur a été introduit de façon que, d'une itération à l'autre, sa valeur décroît strictement ; ce qui termine la preuve de la terminaison de ce programme. □

Remarques

• Lorsque l'on introduit des compteurs, le pas 6 (ou 6′) est immédiat, celui qui demande le plus de soin est le pas 4 (ou éventuellement 5) où l'on prouve que l'ensemble des valeurs prises par les compteurs est minoré (ou majoré).

• L'introduction de compteurs permet de diminuer l'importance du choix des points de coupure puisqu'on est, en général, amené à raisonner globalement sur un cycle et qu'en tout point l'invariant doit être vérifié.

Exercice 9

Prouver la terminaison du programme introduit à l'exercice 6. □

Exercice 10

Prouver la terminaison du programme introduit à l'exercice 7. □

Exercice 11 [KNU, 68a]

Prouver la correction totale du programme de la figure 14 calculant $pgcd(a_1, a_2)$.

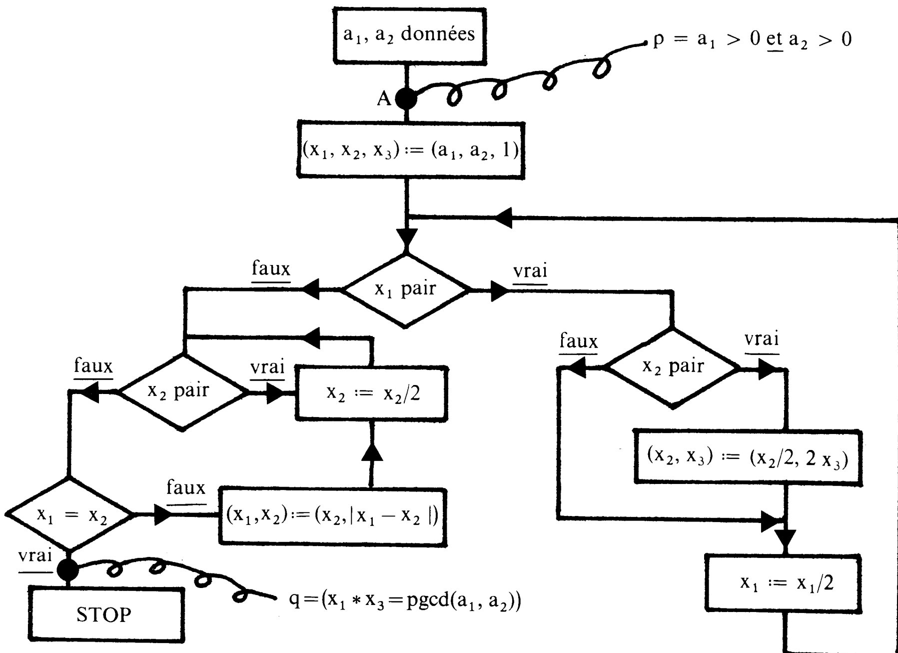


Figure 14 Calcul du pgcd utilisant une représentation binaire de x_2. □

Exercice 12 (recherche dichotomique)

Considérons l'organigramme décrit par la figure 15.

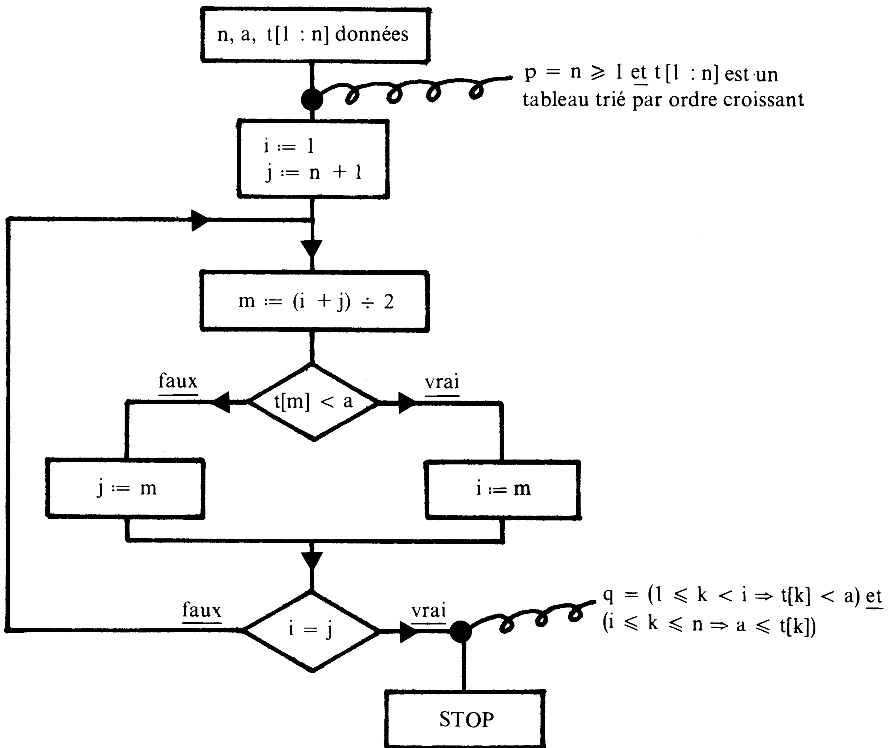


Figure 15 Recherche dichotomique.

$(i + j) \div 2$ représente le quotient entier de $i + j$ par 2.

— Montrer la correction partielle.

— Ce programme est-il totalement correct ? Le modifier éventuellement. □

Exercice 13

On considère l'organigramme de la figure 16 (où p est un prédicat sur $\mathbb{N}$) :

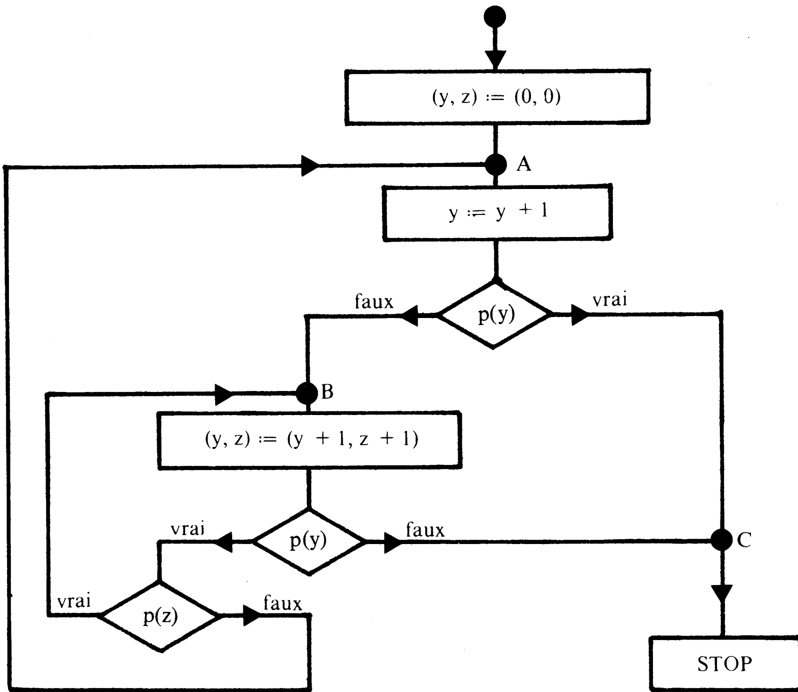


Figure 16.

Convenons que S désigne l'ensemble des sommes des k premiers entiers :

$$S = \left\{ \frac{k(k+1)}{2} \,\middle|\, k \in \mathbb{N} \right\}.$$

On définit l'application s de S dans S par :

$$\forall k \in \mathbb{N} \quad s\left(\frac{k(k+1)}{2}\right) = \frac{(k+1)(k+2)}{2}.$$

Enfin, pour $n \in \mathbb{N}$, pred(n) est le plus grand élément de S inférieur ou égal à n :

$$\text{pred}(n) = \text{Max}\,\{ x \in S \mid x \leqslant n \}.$$

1) Avec ces notations, montrer que l'organigramme ci-dessus est partiellement correct avec les prédicats suivants :

p_A : $\text{pred}(y) = \text{pred}(z)$ $\underline{\text{et}}$ $[n < y \Rightarrow (\text{p}(n) = \underline{\text{faux}} \Leftrightarrow n \in S)]$
$\underline{\text{et}}$ $z \leqslant y$ $\underline{\text{et}}$ $z \in S$ $\underline{\text{et}}$ $y + 1 \in S$

$p_B : y - \text{pred}(y) = z - \text{pred}(z)\ \underline{\text{et}}\ \text{pred}(y) = s(\text{pred}(z))$
$\underline{\text{et}}\ [n \leqslant y \Rightarrow (p(n) = \underline{\text{faux}} \Leftrightarrow n \in S)]\ \underline{\text{et}}\ z < y$
$p_C : p(y) \Rightarrow (y \in S\ \underline{\text{et}}\ z \in S\ \underline{\text{et}}\ y = s(z))$.

2) Montrer que la non-terminaison est équivalente à

$$\forall n \in \mathbb{N} \quad p(n) = \underline{si}\ n \in S\ \underline{alors}\ \underline{faux}\ \underline{sinon}\ \underline{vrai}\ \underline{fsi}\ .$$

3) Reprendre l'étude faite au chapitre 3 (§ 2.6, exemple 5) qui propose un schéma de programme terminant pour toute interprétation finie et pouvant ne pas terminer pour une interprétation infinie. □

** *Exercice* 14

Etudier l'arrêt du programme de la figure 17.

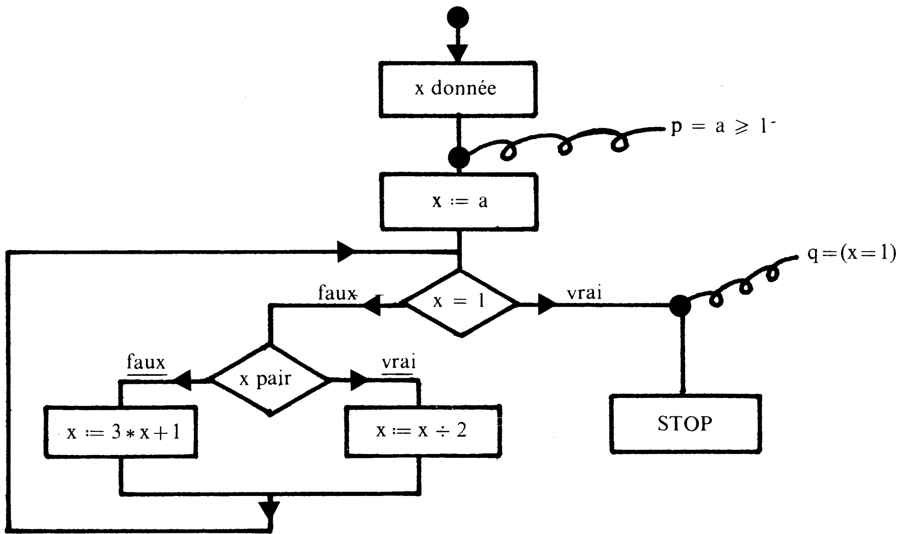


Figure 17.

□

3 PREUVES DE PROGRAMMES ITÉRATIFS

3.1 Introduction

Dans les méthodes présentées au paragraphe 2, la notion de schéma de programme était fondamentale : c'était sur l'organigramme que nous définissions les points de coupure auxquels étaient attachés prédicats et fonctions. Dans ce paragraphe, nous nous intéressons à des programmes de type itératif (c'est-à-dire des D-programmes en utilisant la terminologie du chapitre 3).

Rappelons que x désigne le vecteur d'état et que les affectations sont faites soit globalement, soit composante par composante.

Les programmes des figures 2, 4 et 6 du paragraphe 2 peuvent chacun s'écrire (en prenant pour domaine l'ensemble des entiers) :

π_1
$x := (0, 0)$;
<u>tant que</u> $x_2 \neq a$ <u>faire</u> $x_1 := x_1 + 2 * x_2 + 1 ; x_2 := x_2 + 1$ <u>fait</u>.

π_2
$x := (0, 0, 1) ; x_2 := x_2 + x_3$;
<u>tant que</u> $x_2 \leqslant a$ <u>faire</u> $x := (x_1 + 1, x_2, x_3 + 2) ; x_2 := x_2 + x_3$ <u>fait</u>.

π_3
$x := a$;
<u>tant que</u> $x_1 \neq x_2$ <u>faire</u>

<u>tant que</u> $x_1 > x_2$ <u>faire</u> $x_1 := x_1 - x_2$ <u>fait</u> ;
<u>tant que</u> $x_2 > x_1$ <u>faire</u> $x_2 := x_2 - x_1$ <u>fait</u>

<u>fait</u>.

HOARE [HOA, 69] a proposé une méthode de vérification de la correction partielle de tels programmes, définie de manière axiomatique. Nous présentons cette méthode au paragraphe 3.2, puis nous évoquons brièvement quelques techniques relatives à la terminaison au paragraphe 3.4.

3.2 Correction partielle et méthode de HOARE

a) Axiomes et règles d'inférences

La définition de la correction partielle est la même ici que dans la méthode des assertions de Floyd. La différence entre les deux approches réside en la manière d'associer à π les prédicats « intermédiaires » ; dans la méthode de Hoare, à chaque instruction ou suite d'instructions α du langage, on associe un ensemble de couples de prédicats (p_1, p_2) tels que si $p_1(a, x)$ est vrai avant l'exécution de α alors $p_2(a, \alpha(x))$ est vrai après cette exécution, ce que nous notons : $p_1 \{ \alpha \} p_2$.

Cette association de prédicats à une instruction d'un programme ne se fait plus à l'aide d'étiquetage prédicatif comme dans la méthode de Floyd, mais, de manière plus systématique, grâce à des axiomes et des règles d'inférences d'un certain système formel que nous allons préciser maintenant.

L'alphabet du système formel associé au langage précédent est composé à la fois de symboles logiques (<u>et</u>, <u>ou</u>, <u>non</u>, $\Rightarrow$, ...) et d'éléments du langage. Les formules sont soit des prédicats, soit de la forme $p \{ E \} q$ où p et q sont des prédicats et E une instruction du langage. Les axiomes portent sur les instructions élémentaires du langage (dans notre exemple, les seuls axiomes se rapportent aux affectations entre variables). Les règles d'inférence rendent compte des divers modes de composition des instructions élémentaires (séquences, conditionnelles, cycles, ...). Les seules règles considérées dans

notre exemple sont de la forme :

$$\frac{p_1, p_2, \ldots, p_n}{q}$$

où les p_i et q sont des formules. Elles se lisent « de $p_1, \ldots, p_n$ on déduit q ».

Une démonstration pour ce système formel est une liste $p_1, p_2, \ldots, p_m$ où p_i est soit un axiome, soit se déduit de $p_{j_1}, \ldots, p_{j_i}$ (avec $j_k < i$ pour $1 \leqslant k \leqslant i$) par une règle d'inférence.

Précisons maintenant les axiomes et règles d'inférence dans le cas du langage introduit ci-dessus :

— axiome d'affectation :

(AFF) : pour toute affectation $x := t$ et pour tout prédicat p :

$$p(a, t) \{ x := t \} p(a, x) \; ;$$

— règles d'inférence :

(CND) : conditionnelle

$$\frac{p \; \underline{\text{et}} \; t \{ E \} q, \; p \; \underline{\text{et}} \; \underline{\text{non}} \; t \{ F \} q}{p \{ \underline{si} \; t \; \underline{alors} \; E \; \underline{sinon} \; F \; \underline{fsi} \} q}$$

(ITE) : itération

$$\frac{p \; \underline{\text{et}} \; t \{ E \} p}{p \{ \underline{tant\ que} \; t \; \underline{faire} \; E \; \underline{fait} \} \; p \; \underline{\text{et}} \; \underline{\text{non}} \; t}$$

(SEQ) : composition séquentielle

$$\frac{p \{ E \} q, \; q \{ F \} r}{p \{ E \; ; F \} r} \; .$$

(IMP) : règles relatives à l'implication

- $$\frac{p \{ E \} q, \; q \Rightarrow r}{p \{ E \} r}$$
- $$\frac{r \Rightarrow p, \; p \{ E \} q}{r \{ E \} q} \; .$$

Ces diverses règles et axiomes expriment la « transformation » qu'opère sur un prédicat une instruction du langage, c'est donc une manière de définir la sémantique de ce langage comme nous le précisons au paragraphe 7 du chapitre 5. Cependant pour faire une preuve de correction, il faut pouvoir utiliser les propriétés des prédicats manipulés, par exemple pouvoir démontrer des formules telles que p $\underline{\text{et}}$ q $\Rightarrow$ r.

Pour cette raison, en plus de l'axiome (AFF) introduit ci-dessus, nous prendrons comme axiomes tous les théorèmes logiques du calcul des prédicats, augmentés éventuellement d'axiomes précisant les propriétés du

domaine sur lequel on travaille. Par exemple, si le domaine est ℕ, l'un des axiomes est :

$$a = b * x + y \ \underline{\text{et}}\ b < y \Rightarrow a = b * (x + 1) + (y - b).$$

Exercice 15 : Vérifier que l'axiome d'affectation est mis en défaut lorsqu'on autorise des variables indicées. (On pourra considérer l'affectation $a[a[2]] := 1$ et le prédicat final $a[a[2]] = 1$). □

On peut alors définir formellement la notion de correction partielle :

Définition 5

Un programme π est dit partiellement correct par rapport à deux prédicats p et q si, en appliquant les règles d'inférence et en utilisant l'axiome (AFF), les axiomes logiques et ceux liés au domaine, on peut prouver que p { π } q. □

Cette définition est analogue à la définition 1 du paragraphe 2.2, la différence essentielle réside dans la formalisation qui précise les règles de démonstration.

b) Recherche de prédicats et preuve de correction partielle

Exemple 14

Reprenons le programme π_1 :

$$\begin{array}{l} x := (0, 0) ; \\ \underline{\textit{tant que}}\ x_2 \neq a\ \underline{\textit{faire}}\ x_1 := x_1 + 2 * x_2 + 1 ; \\ \qquad\qquad\qquad\qquad x_2 := x_2 + 1 \\ \qquad\qquad\quad \underline{\textit{fait}} \end{array}$$

et proposons-nous de prouver sa correction partielle relativement aux prédicats

$$\mathrm{p} = a \geqslant 0 \text{ et } \mathrm{q} = (x_1 = a^2).$$

Il s'agit donc de prouver :

$$(1) \qquad (a \geqslant 0)\ \{\ \pi_1\ \}\ (x_1 = \mathrm{a}^2).$$

Or, d'après l'axiome (AFF) sur les affectations on obtient la formule :

$$(a \geqslant 0)\ \underline{\text{et}}\ (0 = 0)\ \underline{\text{et}}\ (0 = 0)\ \{\ x := (0, 0)\ \}\ (a \geqslant 0)\ \underline{\text{et}}\ (x_1 = 0)\ \underline{\text{et}}\ (x_2 = 0).$$

En utilisant le théorème évident $(a \geqslant 0) \Rightarrow (a \geqslant 0)\ \underline{\text{et}}\ (0 = 0)\ \underline{\text{et}}\ (0 = 0)$ et la règle d'implication, on a

$$(a \geqslant 0)\ \{\ x := (0, 0)\ \}\ (a \geqslant 0)\ \underline{\text{et}}\ (x_1 = 0)\ \underline{\text{et}}\ (x_2 = 0).$$

Dorénavant nous passerons souvent rapidement sur l'usage de la règle d'implication.

Ainsi, en utilisant (SEQ), pour prouver (1) il suffit de montrer :

(2) $(a \geqslant 0)$ et $(x_1 = 0)$ et $(x_2 = 0)$

$\{$ *tant que* $x_2 \neq a$ *faire* $x_1 := x_1 + 2 * x_2 + 1 \; ; \; x_2 := x_2 + 1$ *fait* $\}$ $(x_1 = a^2)$.

Pour démontrer (2), considérons les instructions internes à ce cycle.

On sait (axiome (AFF)) que pour tout prédicat p :

$$p(a, (x_1, x_2 + 1)) \{ x_2 := x_2 + 1 \} \, p(a, (x_1, x_2))$$

et $$p(a, (x_1 + 2 * x_2 + 1, x_2 + 1)) \{ x_1 := x_1 + 2 * x_2 + 1 \} \, p(a, (x_1, x_2 + 1))$$

c'est-à-dire avec (SEQ) :

(3) $$p(a, (x_1 + 2 * x_2 + 1, x_2 + 1)) \{ x_1 := x_1 + 2 * x_2 + 1 \; ; \; x_2 := x_2 + 1 \} \; p(a, (x_1, x_2)) .$$

Il nous faut choisir le prédicat p de telle sorte que l'on ait :

$$p(a, x_1, x_2) \text{ et } (x_2 \neq a) \Rightarrow p(a, x_1 + 2\,x_2 + 1, x_2 + 1) .$$

Le prédicat $x_1 = x_2^2$ convient.

Alors, en utilisant (ITE) on déduit de (3) :

(4) $p(a, x) \{$ *tant que* $x_2 \neq a$ *faire* $x_1 := x_1 + 2 * x_2 + 1 := x_2 := x_2 + 1$ *fait* $\}$
$p(a, x)$ et $(x_2 = a)$

d'où l'on peut déduire (2) à l'aide de (IMP) à condition de démontrer

(5) $(a \geqslant 0)$ et $(x_2 = 0)$ et $(x_2 = 0) \Rightarrow p(a, x)$

(6) $p(a, x)$ et $(x_2 = a) \Rightarrow x_1 = a^2$

c'est-à-dire :

(5) $(a \geqslant 0)$ et $(x_1 = 0)$ et $(x_2 = 0)$ et $(x_2 \neq a) \Rightarrow x_1 = x_2^2$

(6) $(x_1 = x_2^2)$ et $(x_2 = a) \Rightarrow x_1 = a^2$

qui sont vrais tous les deux. □

Exemple 15

Considérons maintenant le programme π_2 :

$x := (0, 0, 1) \; ; \quad x_2 := x_2 + x_3 \; ;$
tant que $x_2 \leqslant a$ *faire* $x := (x_1 + 1, x_2, x_3 + 2) ;$
$x_2 := x_2 + x_3$
fait

dont on veut démontrer la correction partielle par rapport aux prédicats :

$$p = (a \geqslant 0)$$
$$q = x_1^2 \leqslant a < (x_1 + 1)^2 .$$

L'essentiel de la démonstration de la correction partielle revient à trouver un invariant de cycle r tel que :

(1) $\quad r\ \underline{et}\ (x_2 \leqslant a)\ \{\ x := (x_1 + 1, x_2, x_3 + 2)\ ;\ x_2 := x_2 + x_3\ \}\ r$,

car alors on peut en déduire (règle (ITE)) :

(2) $\quad r\ \{\ \underline{\textit{tant que}}\ x_2 \leqslant a\ \underline{\textit{faire}}\ x := (x_1+1, x_2, x_3+2)\ ;\ x_2 := x_2+x_3\ \underline{\textit{fait}}\ \}\ r\ \underline{et}\ (x_2 > a)$.

Pour utiliser cet invariant dans la preuve de correction partielle de π_2, il faut avoir :

(3) $\quad p\ \{\ x := (0, 0, 1)\ ;\ x_2 := x_2+x_3\ \}\ r$ et

(4) $\quad r\ \underline{et}\ (x_2 > a) \Rightarrow q$

(lien avec les prédicats d'entrée et de sortie), car il est alors immédiat que de (2), (3) et (4) on déduit :

$$p\ \{\ \pi_2\ \}\ q \text{ d'après les règles (SEQ) et (IMP)} .$$

On remarque de plus que (3) se déduit, à l'aide de (IMP), (SEQ) et (AFF), de :

(5) $\quad p \Rightarrow r(a, (0, 1, 1))$.

En résumé, pour prouver la correction partielle de π_2, il suffit de trouver un prédicat r vérifiant (1), (4) et (5).

Explicitons (1) :

En utilisant (AFF) et (SEQ), pour tout prédicat r :

$$r(a, (x_1+1, x_2+x_3+2, x_3+2))\ \{\ x := (x_1+1, x_2, x_3+2)\ ;\ x_2 := x_2+x_3\ \}\ r(a, x) .$$

Ainsi (1) est vrai si l'on peut prouver :

(6) $\quad r(a, x)\ \underline{et}\ (x_2 \leqslant a) \Rightarrow r(a, (x_1 + 1, x_2 + x_3 + 2, x_3 + 2))$.

En utilisant les définitions de p et q, on est donc conduit à chercher un prédicat r rendant vraies les trois formules :

$$\begin{cases} (4) & r(a, x)\ \underline{et}\ (x_2 > a) \Rightarrow x_1^2 \leqslant a < (x_1+1)^2 \\ (5) & (a \geqslant 0) \Rightarrow r(a, (0, 1, 1)) \\ (6) & r(a, x)\ \underline{et}\ x_2 \leqslant a \Rightarrow r(a, (x_1+1, x_2+x_3+2, x_3+2)) . \end{cases}$$

On peut chercher à deviner r à partir de (4) et poser :

$$r(a, x) : x_1^2 \leqslant a\ \underline{et}\ x_2 = (x_1+1)^2 ,$$

ce qui donne :

— $r(a, (0, 1, 1)) = (0 \leqslant a\ \underline{et}\ 1 = 1^2)$

— $r(a, x)\ \underline{et}\ x_2 \leqslant a = x_1^2 \leqslant a\ \underline{et}\ x_2 = (x_1+1)^2\ \underline{et}\ x_2 \leqslant a$

— $r(a, (x_1+1, x_2+x_3+2, x_3+2)) =$

$$(x_1+1)^2 \leqslant a\ \underline{et}\ x_2+x_3+2 = (x_1+2)^2 .$$

On vérifie alors (4) et (5), mais la démonstration de (6) nécessite l'existence d'une relation entre x_3 et x_1.

Il faudra prouver entre autres que

$$x_1^2 \leqslant a \ \underline{\text{et}}\ x_2 = (x_1+1)^2 \ \underline{\text{et}}\ x_2 \leqslant a \Rightarrow x_2+x_3+2 = (x_1+2)^2$$

ou encore que

$$x_1^2 + 2\,x_1 + 1 + x_3 + 2 = x_1^2 + 4\,x_1 + 4$$

c'est-à-dire $x_3 = 2\,x_1 + 1$, propriété qu'il faut donc ajouter à r.

Ainsi, avec une intuition bien développée, une certaine obstination et une « bonne étoile », on peut arriver à l'expression suivante de r :

$$\mathrm{r}(a, x) = (x_1^2 \leqslant \mathrm{a} \ \underline{\text{et}}\ x_2 = (x_1+1)^2 \ \underline{\text{et}}\ x_3 = 2\,x_1+1)\,,$$

pour laquelle on peut vérifier immédiatement que (4), (5) et (6) sont des théorèmes. On peut ainsi en conclure la correction partielle de π_2. □

Remarque

Ainsi, la méthode de Hoare apparaît comme une axiomatisation des pas 1 et 3 de la méthode de Floyd (§ 2.3) qui est obtenue en définissant un système formel dans lequel se font les démonstrations. Cependant, le pas 3 reste sensiblement le même dans les deux approches : il faut toujours découvrir les prédicats intermédiaires et aucun algorithme ne permet de le faire. On remarque ici encore que deux démarches sont possibles :

— Méthode ascendante, dans laquelle on part des propriétés des instructions élémentaires du programme et on cherche à synthétiser les prédicats entourant les cycles (exemple 14).

— Méthode descendante, dans laquelle on détermine les prédicats internes à l'aide des prédicats d'entrée et de sortie (exemple 15).

Ainsi cette méthode n'apparaît pas encore satisfaisante : malgré l'introduction d'un système formel dans lequel sont définies précisément les règles à utiliser lors d'une démonstration (ce qui pourrait donc permettre de l'automatiser), on se heurte au problème de l'invention des prédicats intermédiaires, qui ne peut pas se résoudre algorithmiquement. C'est ce problème fondamental que nous étudions au paragraphe suivant. □

Exercice 16 : Reprendre l'exemple 14 en remplaçant le test $x_2 \neq a$ du cycle tant que par $x_2 < a$. □

c) Programmes munis d'assertions

Jusqu'à présent, nous avons cherché à prouver a posteriori la correction d'un programme. Remarquons bien que cette démarche n'est ni très efficace, ni très raisonnable :

— Le manque d'efficacité provient essentiellement du problème lié à la découverte des invariants de cycles et autres prédicats intermédiaires.

— Le manque de cohérence est lié au problème de la construction du programme : bien que les méthodes de programmation ne soient pas encore très

satisfaisantes, il est quand même rare que le programmeur, pour résoudre un problème, écrive au hasard une suite d'instructions, puis cherche à prouver l'adéquation du programme obtenu avec le problème initial et espère ainsi, après un nombre limité de corrections, arriver au « bon » programme.

En réalité, au moment où le programme se construit, on peut penser que son auteur a « dans sa tête », explicitement ou non, une idée sur le rôle des différentes instructions alignées.

C'est ce rôle (cette sémantique) que traduisent les différentes assertions utilisées lors de la preuve. Par exemple, en reprenant le programme de l'exemple 14, on veut obtenir a^2, en l'approchant par la suite des valeurs $0^2, 1^2, 2^2 \ldots$ jusqu'à a^2 : l'invariant du cycle est bien $x_1 = x_2^2$.

En résumé, il est beaucoup plus naturel de penser au problème de la correction partielle au moment même où on construit le programme, en faisant l'effort d'expliciter alors les invariants de cycle. Ces invariants et les assertions d'entrée et de sortie permettent de déduire les autres assertions internes de manière automatique à l'aide des axiomes et règles d'inférence.

On peut utiliser alors la méthode de Hoare en insérant dès l'écriture du programme, les assertions internes sous forme de commentaires placés à des emplacements conventionnels, éventuellement on n'écrira que les invariants de cycle.

La preuve de correction partielle est faite alors simplement en appliquant, pour ces prédicats, les axiomes et règles d'inférences associés au langage. En particulier, on distingue, beaucoup mieux que dans les exemples précédents, l'étape de recherche de prédicats et celle de vérification.

Exemple 16

Reprenons le programme π_3 calculant le *pgcd* de deux nombres a_1 et a_2, il est muni d'assertions qui expriment que dans chaque cycle le pgcd de x_1 et x_2 est celui de a_1 et a_2 ; les invariants sont placés en tête d'itération

$$
\begin{array}{l}
x := a\,; \\
\underline{\textit{tant que}}\ x_1 \neq x_2\ \underline{\textit{faire}}\ \underline{\textit{co}}\ pgcd\,(x_1, x_2) = pgcd(a_1, a_2)\ \underline{\textit{co}} \\
\qquad \underline{\textit{tant que}}\ x_1 > x_2\ \underline{\textit{faire}}\ \underline{\textit{co}}\ pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{\textit{co}} \\
\qquad\qquad\qquad x_1 := x_1 - x_2\ \underline{\textit{fait}}\ ; \\
\qquad \underline{\textit{tant que}}\ x_2 > x_1\ \underline{\textit{faire}}\ \underline{\textit{co}}\ pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{\textit{co}} \\
\qquad\qquad\qquad x_2 := x_1 - x_2\ \underline{\textit{fait}} \\
\quad \underline{\textit{fait}}
\end{array}
$$

Proposons-nous de vérifier la correction partielle de π_3 par rapport aux prédicats :

$$p = (a_1 > 0)\ \underline{\text{et}}\ (a_2 > 0)$$

$$q = (x_1 = x_2 = \text{pgcd}(a_1, a_2))\,.$$

La démonstration formelle de la correction partielle de l'itération

$\underline{tant\ que}\ x_1 < x_2\ \underline{faire}\ x_2 := x_2 - x_1\ \underline{fait}$

s'écrit :

(1) $pgcd(x_1, x_2 - x_1) = pgcd(a_1, a_2)\ \{\ x_2 := x_2 - x_1\ \}\ pgcd(x_1, x_2) = pgcd(a_1, a_2)$ d'après (AFF)

(2) $pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{et}\ x_1 < x_2 \Rightarrow pgcd(x_1, x_2 - x_1) = pgcd(a_1, a_2)$ (d'après les propriétés du pgcd)

(3) $pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \{\ \underline{tant\ que}\ x_1 < x_2\ \underline{faire}\ x_2 := x_2 - x_1\ \underline{fait}\ \}$
$pgcd(x_1, x_2) = pgcd(a_1, a_2) \wedge x_1 \geqslant x_2$
de (1), (2) avec (ITE) et (IMP)

De manière analogue à la démonstration précédente, on peut prouver que

(4) $pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \{\ \underline{tant\ que}\ x_1 > x_2\ \underline{faire}\ x_1 := x_1 - x_2\ \underline{fait}\ \}$
$pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{et}\ x_1 \leqslant x_2$

(5) $pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{et}\ x_1 \leqslant x_2 \Rightarrow pgcd(x_1, x_2) = pgcd(a_1, a_2)$

(6) $pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \{\ \underline{tant\ que}\ x_1 > x_2\ \underline{faire}\ x_1 := x_1 - x_2\ \underline{fait}\ ;$
$\underline{tant\ que}\ x_1 < x_2\ \underline{faire}\ x_2 := x_2 - x_1\ \underline{fait}\ \}\ pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{et}\ x_1 \geqslant x_2$
(en appliquant (SEQ) à (3) et (4))

(7) $pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{et}\ x_1 \neq x_2 \Rightarrow pgcd(x_1, x_2) = pgcd(a_1, a_2)$

(8) $pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \{\ \underline{tant\ que}\ x_1 \neq x_2\ \underline{faire}\ \ldots\ \underline{fait}\ \}$
$pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{et}\ x_1 = x_2$
(en appliquant (IMP) à (6) et (7) puis (SEQ))

(9) $p \Rightarrow p\ \underline{et}\ pgcd(a_1, a_2) = pgcd(a_1, a_2)$

(10) $p\ \{\ x := a\ \}\ pgcd(x_1, x_2) = pgcd(a_1, a_2)$ (AFF), (9) et (IMP)

(11) $p\ \{\ \pi_3\ \}\ pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{et}\ x_1 = x_2$
(SEQ) appliqué à (10) et (8)

(12) $x_1 = x_2 \Rightarrow pgcd(x_1, x_2) = x_1 = x_2$

(13) $p\ \{\ \pi_3\ \}\ q$ (11) et (12).

Ainsi, moyennant la démonstration des formules logiques (2), (5), (7), (9) et (12) la correction partielle de π_3 est prouvée. □

Exercice 17

Démontrer la correction partielle du programme

$\pi : x := a\ ;\ \alpha : \underline{si}\ x_1 < x_2\ \underline{alors}\ x := (x_2, x_1)\ \underline{fsi}\ ;$

$\beta : \underline{tant\ que}\ x_1 > x_2\ \underline{faire}\ \underline{co}\ pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{co}$
$x_1 := x_1 - x_2\ ;$

$\gamma : \underline{tant\ que}\ x_1 < x_2\ \underline{faire}\ \underline{co}\ pgcd(x_1, x_2) = pgcd(a_1, a_2)\ \underline{co}$
$\underline{fait}$ $x_2 := x_2 - x_1\ \underline{fait}$

relativement aux prédicats de l'exemple 16 ci-dessus. □

Les étiquettes ajoutées au programme ont uniquement pour but de structurer la preuve. □

Nous avons déjà remarqué que, lorsque les invariants d'un programme sont explicités, le reste de la preuve est automatique. IGARASHI, LONDON et LUCKHAM [IGA, 73] ont conçu un programme (VCG : générateur de conditions de vérifications) qui vérifie si un programme muni d'assertions (les invariants essentiellement) est partiellement correct ; la vérification des formules logiques fait, quant à elle, appel à un prouveur de théorème qui demande une collaboration du programmeur (système interactif).

Exemple 17 : Montrons, sur le programme π_3 muni d'assertions (exemple 16), comment on peut générer les conditions de vérification permettant de montrer la correction partielle. Intuitivement, on part du résultat (but) que l'on veut obtenir, considéré comme la conclusion d'une règle d'inférence dont on cherche les premisses (sous-buts) et ainsi de suite pour chaque sous-but jusqu'à ce qu'on arrive à des formules logiques ou à des axiomes. Nous présentons cette recherche dans le tableau suivant

N° du sous-but	Assertion	Servant à la démonstration de	A l'aide de la règle
(but)	$(a_1 > 0)$ $\underline{\text{et}}$ $(a_2 > 0)$ $\{ \pi_3 \}$ $x_1 = x_2 = pgcd(a_1, a_2)$		
(1)	$(a_1 > 0)$ $\underline{\text{et}}$ $(a_2 > 0)$ $\{ x := a \}$ $pgcd(x_1, x_2) = pgcd(a_1, a_2)$	(but)	(SEQ)
(2)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ $\{ \underline{tant\ que}\ x_1 \neq x_2\ \underline{faire}$ $\ldots\ \underline{fait} \}$ $x_1 = x_2 = pgcd(a_1, a_2)$	(but)	(SEQ)
(3) axiome (AFF)	$pgcd(a_1, a_2) = pgcd(a_1, a_2)$ $\{ x := a \}$ $pgcd(x_1, x_2) = pgcd(a_1, a_2)$	(1)	(IMP)
(4)	$(a_1 > 0)$ $\underline{\text{et}}$ $(a_2 > 0) \Rightarrow pgcd(a_1, a_2) = pgcd(a_1, a_2)$	(1)	(IMP)
(5)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ $\{ \underline{tant\ que}\ x_1 \neq x_2\ \underline{faire}$ $\ldots\ \underline{fait} \}$ $pgcd(x_1, x_2) = pgcd(a_1, a_2)$ $\underline{\text{et}}$ $x_1 = x_2$	(2)	(IMP)
(6)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ $\underline{\text{et}}$ $x_1 = x_2 \Rightarrow x_1 = x_2 = \text{pgcd}(a_1, a_2)$	(2)	(IMP)
(7)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ $\underline{\text{et}}$ $x_1 \neq x_2$ $\{ \underline{tant\ que}\ x_1 > x_2$ $\underline{faire} \ldots \underline{fait}$; $\underline{tant\ que}\ x_1 < x_2\ \underline{faire} \ldots \underline{fait} \}$ $pgcd(x_1, x_2) = pgcd(a_1, a_2)$	(5)	(ITE)
(8)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ $\underline{\text{et}}$ $x_1 \neq x_2$ $\{ \underline{tant\ que}\ x_1 > x_2$ $\underline{faire}\ x_1 := x_1 - x_2\ \underline{fait} \}$ $pgcd(x_1, x_2) = pgcd(a_1, a_2)$	(7)	(SEQ)

N° du sous-but	Assertion	Servant à la démonstration de	A l'aide de la règle
(9)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ { <u>*tant que*</u> $x_1 < x_2$ <u>*faire*</u> $x_2 := x_2 - x_1$ <u>*fait*</u> } $pgcd(x_1, x_2) = pgcd(a_1, a_2)$	(7)	(SEQ)
(10)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ <u>et</u> $x_1 \neq x_2$ <u>et</u> $x_1 > x_2$ { $x_1 := x_1 - x_2$ } $pgcd(x_1, x_2) = pgcd(a_1, a_2)$	(8)	(ITE)
(11) axiome (AFF)	$pgcd(x_1 - x_2, x_2) = pgcd(a_1, a_2)$ { $x_1 := x_1 - x_2$ } $pgcd(x_1, x_2) = pgcd(a_1, a_2)$	(10)	(IMP)
(12)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ <u>et</u> $x_1 \neq x_2$ <u>et</u> $x_1 > x_2 \Rightarrow pgcd(x_1 - x_2, x_2) = pgcd(a_1, a_2)$	(10)	(IMP)
(13)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ <u>et</u> $x_1 < x_2$ { $x_2 := x_2 - x_1$ } $\text{pgcd}(x_1, x_2) = \text{pgcd}(a_1, a_2)$	(9)	(ITE)
(14) axiome (AFF)	$pgcd(x_1, x_2 - x_1) = pgcd(a_1, a_2)$ { $x_2 := x_2 - x_1$ } $pgcd(x_1, x_2) = pgcd(a_1, a_2)$	(13)	(IMP)
(15)	$pgcd(x_1, x_2) = pgcd(a_1, a_2)$ <u>et</u> $x_1 < x_2 \Rightarrow pgcd(x_1, x_2 - x_1) = pgcd(a_1, a_2)$	(13)	(IMP)

On peut schématiser l'arbre de recherche des hypothèses utilisées dans les règles d'inférence par la figure 18. □

Remarques

1) Un des reproches adressés à l'introduction d'assertions dans le texte même des programmes est que, finalement, on écrit deux fois le même programme alors que les assertions sont suffisantes, à elles seules, pour caractériser l'algorithme construit. Malheureusement, il n'existe pas encore de procédé automatique (« compilateur d'assertions ») permettant, en toute généralité, de transformer un ensemble d'assertions en un programme : c'est un des problèmes de la synthèse de programme. Plus modestement, on ne sait pas encore définir une bonne méthodologie permettant au programmeur d'effectuer systématiquement cette transformation. Synthèse de programme et méthodologie de la programmation font actuellement l'objet de nombreuses recherches et nous aborderons brièvement ces problèmes (§ 4). Il semble clair qu'il vaut mieux construire directement un programme correct plutôt que de le « mettre au point » a posteriori ; cependant en attendant mieux, il nous semble que l'adjonction de prédicats, sous forme de commentaires dans le texte du

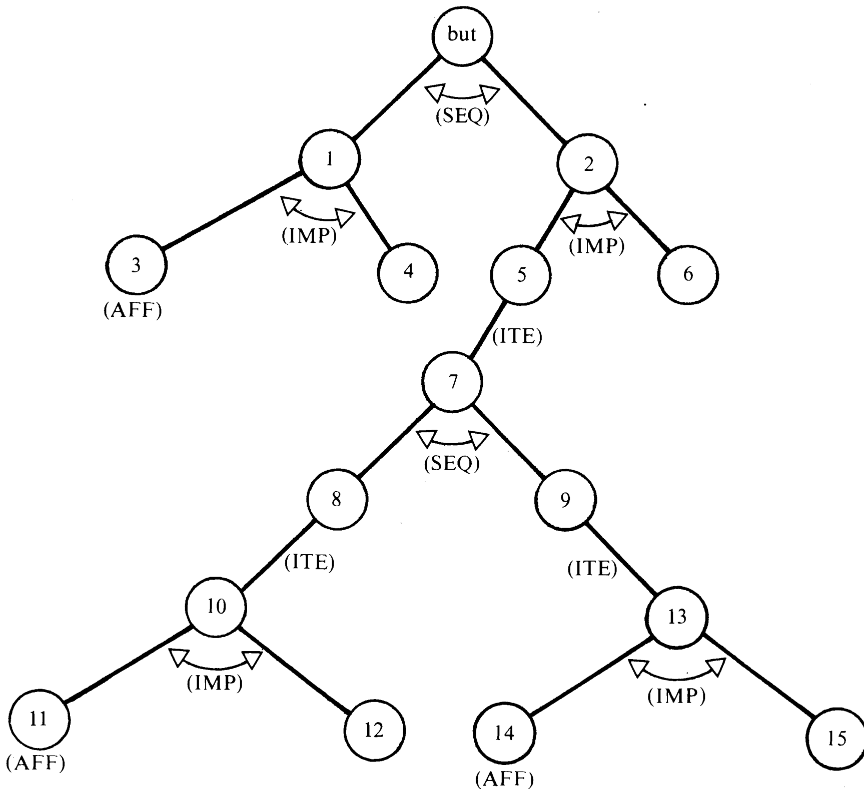


Figure 18 Arbre de recherche de la preuve du pgcd.

programme, est un bon moyen pour inciter le programmeur à plus de précision et de rigueur.

2) La méthode de Hoare s'applique à des langages plus sophistiqués que celui présenté ci-dessus, elle permet par exemple de rendre compte des notions de procédures [HOA, 70], de saut [CLI, 72] de coroutines [CLI, 73] et de programmes concurrents [BRI, 73], mais reste généralement bornée à la correction partielle.

3) Enfin nous avons déjà signalé que l'association de prédicats à chaque type d'instruction du langage permet une définition de sa sémantique ; nous approfondirons cette idée au chapitre 5, paragraphe 7. □

Dans l'immédiat nous présentons un exemple de construction simultanée de programme et d'assertions (§ 3.3), puis nous donnons quelques idées sur la correction totale dans le cadre de la méthode de Hoare (§ 3.4).

3.3 Exemple

Si α est un mot sur un alphabet convenons de noter $|\alpha|$ la longueur de α et $\alpha\langle i, j\rangle$ le sous-mot formé par $a_i\, a_{i+1} \ldots a_{i+j-1}$ (où a_i est la i-ième lettre de α).

Soient α et β deux mots donnés sur l'alphabet A, on cherche l'ensemble des entiers i tels que $\beta \langle 1, i \rangle \alpha$ soit facteur gauche de β, autrement dit l'ensemble F défini par :

$$F = \{ i \mid \beta \langle i + 1, |\alpha| \rangle = \alpha \} .$$

On peut envisager de construire F par approximations successives. Notons F_j le sous-ensemble des éléments de F inférieurs à j :

(1) $$F_j = \{ i \mid \beta \langle i + 1, |\alpha| \rangle = \alpha \ \underline{\text{et}}\ 0 \leqslant i \leqslant j \} .$$

Tout entier i appartenant à F vérifie

$$i + |\alpha| \leqslant |\beta| .$$

Autrement dit

$$F = F_{|\beta| - |\alpha|} .$$

De plus

$$F_{-1} = \varnothing .$$

Il reste à construire F_j à partir de F_{j-1} :

$$F_j = \underline{si}\ \beta \langle j + 1, |\alpha| \rangle = \alpha\ \underline{alors}\ F_{j-1} \cup \{ j \}\ \underline{sinon}\ F_{j-1}\ \underline{fsi} .$$

Cette étude se transcrit en un premier programme, où F est une variable à valeur dans les parties de $\mathbb{N}$ et dont la valeur finale est le résultat. L'invariant de l'itération est la propriété (1) :

```
ρ : début j := 1 ; F := ∅ ;
        tant que j ⩽ |β| − |α| faire co F = { i | β⟨i+1, |α|⟩ = α et 0 ⩽ i ⩽ j } co
                                   j := j+1 ;
                                   si β⟨j+1, |α|⟩ = α alors F := F ∪ {j} fsi
                                fait
```

Le calcul de la condition

(2) $$\beta \langle j + 1, |\alpha| \rangle = \alpha$$

peut être développé pour se ramener à des comparaisons de lettres de α et de β. On peut, par exemple, rechercher le plus grand $k \in [0, |\alpha| + 1]$ tel que :

(3) $$i < k \Rightarrow \alpha \langle 1, i \rangle = \beta \langle j + 1, i \rangle .$$

La condition (2) s'écrit alors plus simplement :

$$k > |\alpha| .$$

L'entier k est facile à calculer en comparant successivement les lettres de α et de $\beta \langle j + 1, |\alpha| \rangle$. Voici un programme correspondant au calcul de k :

θ : *début* $k := 1$;
  tant que $k \leqslant |\alpha|$ *et* $a_k = b_{j+k}$
    faire *co* $i < k \Rightarrow \alpha\langle 1, i\rangle = \beta\langle j+1, i\rangle$ *co*
      $k := k + 1$
    fait
fin

L'invariant de l'itération est la propriété 3.

Finalement, nous pouvons écrire le programme complet sous la forme suivante (le lecteur pourra facilement donner une version effective de ce programme dans un langage de programmation en représentant par exemple les mots par des tableaux et en décrivant F par des impressions)

ρ : $j := -1$; $F := \varnothing$;
χ : *tant que* $j \leqslant |\beta| - |\alpha|$ *faire* *co* $F = \{ i \mid \alpha = \beta\langle i+1, |\alpha|\rangle$ *et* $0 \leqslant i \leqslant j \}$ *co*
  $j := j + 1$;
  θ : $k := 1$;
  tant que $k \leqslant |\alpha|$ *et* $a_k = b_{j+k}$
    faire *co* $i < k \Rightarrow \alpha\langle 1, i\rangle = \beta\langle j+1, i\rangle$ *co*
    $k := k+1$
    fait
  si $k > |\alpha|$ *alors* $F := F \cup \{j\}$ *fsi*
  fait

grâce à la technique de construction du programme, nous connaissons ici les principaux prédicats à utiliser pour la preuve de ce programme, ce qui évite tout effort d'imagination pour démontrer la correction partielle.

Exercice 18 : Prouver le programme de l'exemple ci-dessus. □

3.4 Correction totale

Les techniques présentées au paragraphe 2.3. (Introduction d'un ensemble bien fondé E et de fonctions d'évaluation à valeurs dans E, introduction de compteurs) ont été adaptées au cadre introduit par HOARE. Nous présentons maintenant ces techniques respectivement en a) et b).

a) Ensemble bien fondé et fonction d'évaluation

Les idées à la base de cette méthode de preuve directe de la correction totale introduite par MANNA et PNUELI [MAN, 74] sont :

— Utiliser le même étiquetage pour la terminaison et la correction partielle, cet étiquetage étant exprimé par un système formel analogue à celui de HOARE.

— Faire porter les prédicats de fin d'instruction (q dans p { S } q) à la fois sur les valeurs des variables avant l'exécution et après l'exécution (appelés respectivement x et x'), ce qui permet d'introduire dans ces prédicats des formules de la forme $u(x) \succ u(x')$.

Le nouveau système formel contient des règles de la forme :

— (AFF′) affectation : pour tout prédicat p et toute affectation $x := E$:

$$\mathrm{p}(a, E) \{ x := E \} \mathrm{p}(a, x') \ \underline{\text{et}}\ x' = E .$$

— (CND′) conditionnelle :

$$\frac{\mathrm{p}(a, x)\ \underline{\text{et}}\ \mathrm{t}(a, x) \{ E_1 \} \mathrm{q}(\mathrm{a}, \mathrm{x}, \mathrm{x}'),\ \mathrm{p}(a, x)\ \underline{\text{et}}\ \underline{\text{non}}\ \mathrm{t}(\mathrm{a}, \mathrm{x}) \{ E_2 \} \mathrm{q}(a, x, x')}{\mathrm{p}(a, x) \{ \underline{si}\ t\ \underline{alors}\ E_1\ \underline{sinon}\ E_2\ \underline{fsi} \} \mathrm{q}(a, x, x')}$$

La règle la plus fortement modifiée concerne évidemment les itérations, puisque ce sont les seules instructions posant le problème de la terminaison.

— (ITE′) itérations :

$$\frac{\begin{array}{l} \mathrm{p}(a, x)\ \underline{\text{et}}\ t(a, x) \{ E \} \mathrm{q}(a, x, x')\ \underline{\text{et}}\ (\mathrm{u}(\mathrm{x}') \prec \mathrm{u}(\mathrm{x})), \\ (\forall x, y) (\mathrm{q}(a, x, y)\ \underline{\text{et}}\ t(a, y) \Rightarrow \mathrm{p}(a, y)), \\ (\forall x, x', x'') (\mathrm{q}(a, x, x')\ \underline{\text{et}}\ \mathrm{q}(a, x', x'') \Rightarrow \mathrm{q}(a, x', x'')), \\ \qquad (\forall x) (\mathrm{p}(a, x)\ \underline{\text{et}}\ \underline{\text{non}}\ t(a, x) \Rightarrow \mathrm{q}(a, x, x)) \end{array}}{\mathrm{p}(a, x) \{ \underline{tant\ que}\ t\ \underline{faire}\ E\ \underline{fait} \} \mathrm{q}(a, x, x')\ \underline{\text{et}}\ \underline{\text{non}}\ \mathrm{t}(a, x')}$$

où u est une fonction d'évaluation à valeurs dans un ensemble bien fondé.

Exercice 19

Prouver la terminaison du programme de l'exemple 16 (§ 3.2).

b) Compteurs

L'introduction de compteurs pour prouver la terminaison a été reprise par LUCKHAM et SUZUKI [LUC, 75], qui ramènent ainsi la preuve de l'arrêt à une preuve de correction partielle du programme augmenté de compteurs. Ils ont pu ainsi utiliser le système VCG pour prouver la correction totale.

L'idée fondamentale consiste à modifier toute itération

$$\underline{tant\ que}\ t\ \underline{faire}\ E\ \underline{fait}$$

en ajoutant un « compteur » : variable C prenant des valeurs entières :

$$C := C_0 ;\ \underline{tant\ que}\ t\ \underline{faire}\ C := f(a, C) ; E\ \underline{fait}$$

où f vérifie $(\forall C)\ (f(a, C) \leqslant C + 1)$ et C_0 peut dépendre de a.

Si l'on peut ajouter, à l'invariant p de cycle, une assertion de la forme $C \leqslant \mathrm{g}(a)$ (c'est-à-dire que l'ensemble des valeurs de C est borné par une fonction de a), alors la correction partielle du nouveau programme π', ainsi obtenu, entraîne nécessairement la terminaison du programme initial.

4 CONSTRUIRE DES PROGRAMMES CORRECTS

> « If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with. »
>
> (DIJKSTRA)

4.1 Introduction

Dans les deux paragraphes précédents, nous avons vu comment prouver la correction de programmes déjà construits ; cette démarche n'est pas très satisfaisante pour le programmeur : on lui demande non seulement d'écrire un programme mais encore de prouver qu'il est correct. S'il découvre alors des erreurs, rien dans les méthodes précédentes ne lui permet de corriger ces erreurs ou a fortiori d'écrire directement un programme correct. C'est pourtant bien à cette dernière solution qu'il faudrait parvenir. Certes de nombreuses méthodes ont été proposées ces dernières années pour mettre fin à une programmation empirique, elles permettent d'éviter certaines erreurs mais ne garantissent absolument pas la correction de l'algorithme qu'elles ont permis de construire. Le lecteur trouvera de nombreuses références bibliographiques dans [DIJ, 72], [FRA, 75]. Parler de correction suppose que l'on sait prouver cette correction ; aussi la réalisation automatique de programmes corrects est-elle apparue comme une application possible de la démonstration de théorèmes fondée sur le principe de résolution de Robinson [CHA, 73]. La construction d'un programme peut être assimilée à la recherche d'une fonction f qui à une donnée x fait correspondre un ensemble de résultats $z = f(x)$. Il suffit de formuler cette recherche comme celle d'un théorème à démontrer dans le calcul des prédicats du 1er ordre :

$$(\forall x)\ (\exists z)\ Q(x, z)$$

où Q est un prédicat qui est vrai si z est bien le résultat cherché. Le programme est alors dérivé de la démonstration. Des résultats ont été obtenus dans des cas très simples [CHA, 73], [MAN, 74] ; mais pratiquement cette méthode est peu utilisable : elle suppose une description de tout l'environnement du problème en énoncés du calcul des prédicats qui est source d'erreurs. Et comme cet environnement devient vite trop riche, le nombre d'essais pour la démonstration du théorème croît de façon démesurée et la preuve ne peut être obtenue en un temps raisonnable. De plus le principe de résolution n'a pas répondu aux espoirs qu'il avait suscités.

D'autres recherches essaient de s'inspirer des méthodes de preuves développées précédemment pour prouver des programmes, étape par étape, en les construisant. Il nous a semblé important de présenter deux approches de la construction de programmes corrects se rattachant à ce courant de recherches qui sont, pensons-nous, une application importante des techniques précé-

dentes. Nous ne prétendons évidemment pas épuiser cette question, un livre n'y suffirait pas, ni même citer tous les travaux s'y rapportant ; nous renvoyons le lecteur à la bibliographie [ARS, 76], [NEW, 75] pour les approches les plus connues.

4.2 Construction d'un programme par raffinements successifs

a) *Idée générale*

Dès qu'on est en présence d'un problème un peu complexe, on ne sait pas exprimer en une seule fois un algorithme permettant de résoudre ce problème. Un des procédés les plus utilisés pour vaincre la complexité d'un problème est de le décomposer en sous-problèmes plus simples. On est conduit à écrire l'algorithme général à l'aide de modules dont on se contente d'indiquer le rôle (résolution de tel sous-problème) et pour lesquels on construit ensuite des algorithmes de résolution. Ces algorithmes peuvent à leur tour faire appel à d'autres modules et ainsi de suite jusqu'à ce que le niveau du langage d'expression d'algorithmes ou de programmation soit atteint.

Il faut donc regarder comment se posent les problèmes de correction au cours d'une telle démarche et comment on peut les résoudre. Nous le faisons sur un algorithme de numérotation des sommets d'un graphe, puis nous tirons quelques conclusions plus générales.

b) *Problème du tri topologique*

Ce problème peut s'énoncer sous la forme suivante :

Etant donnés un ensemble fini F et une relation binaire Γ sur F telle que le graphe (F, Γ) soit sans circuit, on demande de trouver une relation d'ordre total Σ compatible avec Γ ; autrement dit, de trouver une relation Σ qui vérifie :

$$\forall x, y \in F \qquad x \,\Gamma\, y \Rightarrow x \,\Sigma\, y\,.$$

La figure 19 illustre cet énoncé sur un exemple

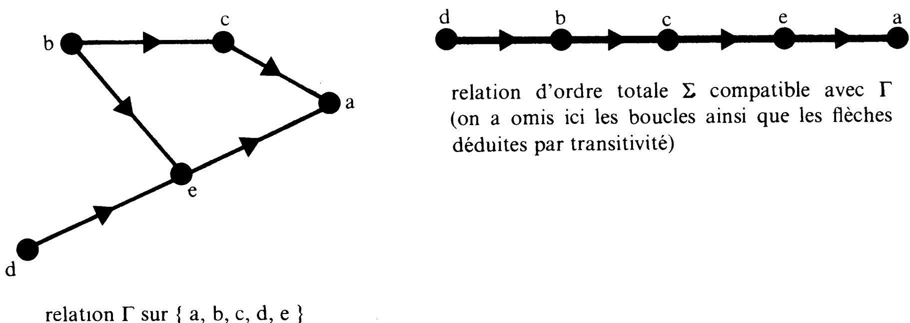


Figure 19 Problème du tri topologique.

Présentons une résolution par raffinements successifs de ce problème :

Le résultat et sa représentation

Supposons ici que F ait n éléments. Une façon particulièrement concise de représenter une relation d'ordre total Σ sur F consiste à la représenter sous la forme d'une suite $x_1, \ldots, x_n$ d'éléments distincts de F, de façon que :

$$i \leqslant j \Leftrightarrow x_i \, \Sigma \, x_j \, .$$

Le résultat du problème peut alors être défini comme une fonction injective r, de domaine de définition $D(r) = [1 : n]$ et de domaine d'arrivée $A(r) = F$, vérifiant la propriété :

$$\forall i, j \in \mathrm{D}(r) \quad r(i) \, \Gamma \, r(j) \Rightarrow i \leqslant j \, .$$

Plus précisément notre problème peut s'énoncer sous la forme :

E : <u>*Résultat*</u> : *suite r telle que*
$D(r) = [1 : n]$ <u>*et*</u> $A(r) = F$ <u>*et*</u> *r est injective* <u>*et*</u>
$\forall i, j \in D(r) \;\; r(i) \, \Gamma \, r(j) \Rightarrow i \leqslant j$.

On peut alors avoir l'idée de construire r par étapes. A la p-ième étape, r est définie sur p éléments ($0 \leqslant p \leqslant n$) et vérifie donc la propriété I suivante :

I : $D(r) \subset [1 : n]$ <u>*et*</u> *card* $D(r) = p$ <u>*et*</u> $A(r) \subset F$ <u>*et*</u>
r est injective <u>*et*</u> $\forall i, j \in D(r) \;\; r(i) \, \Gamma \, r(j) \Rightarrow i \leqslant j$.

L'objectif de chaque étape est d'étendre r en ajoutant un élément à son domaine de définition. On est ainsi conduit à l'algorithme S_1 suivant :

S_1 : $r \leftarrow \varnothing$
  <u>*tant que*</u> *card* $D(r) < n$
   <u>*faire*</u>
   [<u>*Résultat*</u> : *couple* (i, x) *tel que*
   $i \notin D(r)$ <u>*et*</u> $x \notin A(r)$ <u>*et*</u> $\forall j \in D(r)$ $(x \, \Gamma \, r(j) \Rightarrow i \leqslant j$ <u>*et*</u> $r(j) \, \Gamma x \Rightarrow j \leqslant i)$] ;
   $r \leftarrow r \cup (i, x)$
   <u>*fait*</u>

dans lequel la notation $r \leftarrow r \cup (i, x)$ signifie que r est prolongée en i de telle manière que $r(i) = x$ (en particulier elle implique : $D(r) \leftarrow D(r) \cup \{ i \}$ et $A(r) \leftarrow A(r) \cup \{ x \}$).

On obtient ainsi un algorithme inachevé (ou encore un squelette d'algorithme) dans lequel la partie entre crochets représente un nouvel énoncé à remplacer par une construction algorithmique de (i, x). Avant de poursuivre, il faut vérifier que :

— L'itération préserve l'invariant I,
— L'itération se termine.

Ceci est sans difficulté ici, sous réserve naturellement que la partie placée entre crochets soit correcte.

Une première idée de solution

Pour déterminer (i, x), une idée simple consiste à construire r en progressant de 1 à chaque étape sur l'indice i. Ceci nous assure automatiquement :

$$i \notin D(r) \ \underline{et} \ \forall j \in D(r) \ \ r(j) \ \Gamma \ x \Rightarrow j \leqslant i \, .$$

Si l'on remarque de plus qu'à chaque étape, i est le cardinal de $D(\Omega)$ l'algorithme prend la forme S_2 suivante :

S_2 : $r \leftarrow \varnothing$
  <u>pour</u> i <u>de</u> 1 <u>jusqu'à</u> n
   <u>faire</u>
   [<u>Résultat</u> : x tel que
   $x \notin A(r)$ <u>et</u> $\forall j \in D(r) \ \ x \ \Gamma \ r(j) \Rightarrow i \leqslant j$] ;
   $r \leftarrow r \cup (i, x)$
   <u>fait</u>

En remarquant que, par construction de r, l'inégalité $i \leqslant j$ est fausse on peut remplacer le prédicat

$$x \, \Gamma \, r(j) \Rightarrow i \leqslant j$$

par le prédicat

$$\underline{non} \ x \ \Gamma \ r(j)$$

et il est donc légitime de remplacer :

$$\forall j \in D(r) \ \ x \ \Gamma \ r(j) \Rightarrow i \leqslant j$$

par le prédicat équivalent :

$$\Gamma(x) \cap A(r) = \varnothing \, .$$

L'algorithme S_2 prend alors la forme S_2' suivante :

S_2' : $r \leftarrow \varnothing$
  <u>pour</u> i <u>de</u> 1 <u>jusqu'à</u> n
   <u>faire</u>
   [<u>Résultat</u> : x tel que
   $x \notin A(r)$ <u>et</u> $\Gamma(x) \cap A(r) = \varnothing$] ;
   $r \leftarrow r \cup (i, x)$
   <u>fait</u>

Ici encore il n'y a aucune difficulté à prouver que si r est un résultat de S_2' (ou ce qui revient au même de S_2), c'est une solution du problème posé. Ceci peut se démontrer directement en utilisant la définition de l'invariant I, mais il est possible également d'effectuer cette démonstration par étapes en utilisant la correction de S_1 et en montrant que tout résultat de S_2' est un résultat de S_1. Notons que cette deuxième manière de procéder est bien adaptée à la démarche suivie pour construire S_2' : il suffit de justifier chaque transformation d'énoncés.

Il reste donc, pour prouver que S_2' est bien une solution du problème considéré, à montrer l'existence d'un résultat r pour toute donnée (F, Γ). Commençons tout d'abord par exécuter cet algorithme pour une donnée particulière, par exemple celle de la figure 19. Pour cela donnons une suite possible de valeurs de r :

$$\begin{array}{l} D(r_0) = \varnothing \\ D(r_1) = [1 : 1] ; r_1(1) = c \\ D(r_2) = [1 : 2] ; r_2(1) = c, r_2(2) = e \\ D(r_3) = [1 : 3] ; r_3(1) = c, r_3(2) = e, r_3(3) = a \\ D(r_4) = [1 : 4] ; r_4(1) = c, r_4(2) = e, r_4(3) = a, r_4(4) = ? \end{array}$$

Ici il n'est plus possible de choisir un élément de F satisfaisant aux conditions. Ainsi r_3 est une solution correcte pour le graphe partiel restreint à { c, e, a } mais il n'est plus possible de continuer. Intuitivement, ceci provient de ce que r_1, r_2, r_3 construits par l'algorithme S_2' ne sont pas nécessairement inclus dans le résultat cherché, ce qui rend impossible le choix de certains éléments.

En bref, on peut dire que l'algorithme S_2' peut s'arrêter par un blocage, c'est-à-dire pour des valeurs de $i < n$: il est partiellement mais non totalement correct. Il faut donc remettre en cause cet algorithme. Notons que l'idée développée ici conduisait à une simplification du choix de i (valeurs successives) ; elle ne peut se réaliser que si, conjointement, on renforce les conditions sur x.

Une solution correcte au problème posé

En supposant que r_n est un résultat correct c'est-à-dire satisfaisant l'énoncé E et que r_i est formé des i premiers éléments de la suite r_n on obtient :

(1) $\quad j \leqslant i \Rightarrow j \in D(r_i)$.

Notons alors x le dernier élément de r_i, autrement dit $r_i(i)$. Nous avons vu, lors de la recherche précédente, que la propriété :

$$\forall j \in D(r_{i-1}) \quad r_{i-1}(j)\, \Gamma\, x \Rightarrow j \leqslant i$$

était automatiquement vérifiée. Le choix séquentiel des i rendait inutile cette condition, ce qui pouvait conduire à des blocages. Pour les éviter exigeons que :

$$\forall y \in F \qquad y\, \Gamma\, x \Rightarrow r_{i-1}^{-1}(y) \leqslant i - 1$$

c'est-à-dire encore, avec (1) :

$$\forall y \in F \qquad y\, \Gamma\, x \Rightarrow r_{i-1}^{-1}(y) \in D(r_{i-1})$$

et donc :

$$\forall y \in F \qquad y\, \Gamma\, x \Rightarrow y \in A(r_{i-1})$$

ou encore :

$$\Gamma^{-1}(x) \subset A(r_{i-1})$$

en convenant que $\Gamma^{-1}(x)$ est l'ensemble des prédécesseurs de x (voir figure 20).

Intuitivement il est clair que cette condition doit éviter les blocages rencontrés précédemment puisqu'à chaque étape un élément x ne peut être choisi que si tous ses prédécesseurs l'on déjà été. On est conduit, en utilisant ce qui précède à remplacer l'énoncé entre crochets de S_2' par l'énoncé suivant :

(2) [<u>*Résultat*</u> : *x tel que*
$x \notin A(r)$ <u>*et*</u> $\Gamma(x) \cap A(r) = \varnothing$ <u>*et*</u> $\Gamma^{-1}(x) \subset A(r)$].

Et, en remarquant que l'assertion $\Gamma(x) \cap A(r) = \varnothing$ est toujours vérifiée (voir exercice 20), on obtient l'algorithme S_3 suivant :

S_3 : $r \leftarrow \varnothing$
<u>*pour*</u> *i* <u>*de*</u> 1 <u>*jusqu'à*</u> *n*
<u>*faire*</u>
(3) [<u>*Résultat*</u> : *x tel que*
$x \notin A(r)$ <u>*et*</u> $\Gamma^{-1}(x) \subset A(r)$];
$r \leftarrow r \cup (i, x)$
<u>*fait*</u>

S_3 est partiellement correct, mais on peut également montrer (voir exercice 21) qu'il admet une solution r pour toute donnée (F, Γ) ce qui prouve la correction totale.

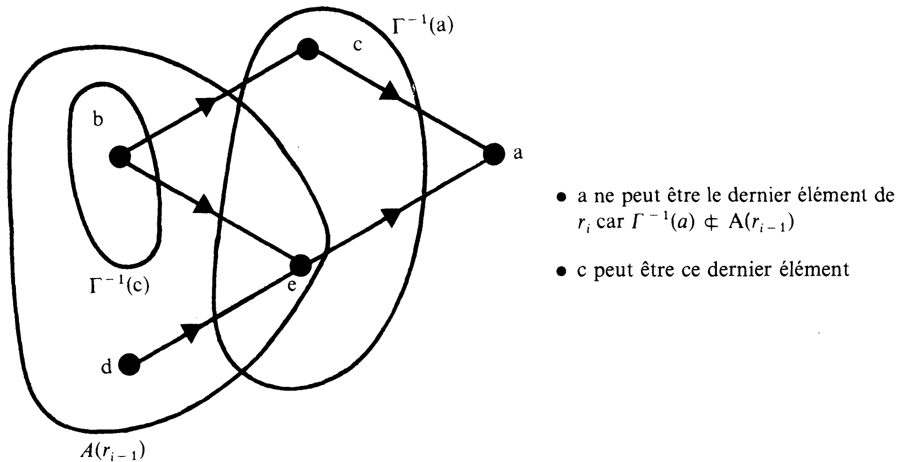


Figure 20 Etape de résolution du tri topologique pour l'exemple de la figure 19.

Exercice 20

Montrer par récurrence qu'avec les hypothèses et notations de l'énoncé (2), l'assertion $\Gamma(x) \cap A(r) = \varnothing$ est toujours vérifiée. □

Exercice 21

Montrer qu'il existe toujours x satisfaisant à l'assertion :

$$x \notin A(r) \ \underline{et}\ \Gamma^{-1}(x) \subset A(r) .$$ □

Algorithme et programme final

Pour aboutir à un algorithme, il reste à transformer l'énoncé (3) figurant dans S_3. Le choix de x sera plus facile si l'on construit à chaque étape l'ensemble :

$$CANDIDATS = (E - A(r)) \cap \{ z \mid \Gamma^{-1}(z) \subset A(r) \} .$$

L'énoncé (3) se réduit alors à

[<u>Résultat</u> : *x tel que*
$x \in CANDIDATS$] .

La construction itérative de CANDIDATS se déduit facilement de la définition de cet ensemble ce qui donne finalement l'algorithme suivant :

```
S4 : r ← ∅ ;
   CANDIDATS ← { z | Γ⁻¹(z) = ∅ } ;
   pour i de 1 jusqu'à n
     faire
     [Résultat : x tel que
      x ∈ CANDIDATS] ;
     r ← r ∪ (i, x) ;
     CANDIDATS ← (CANDIDATS − {x}) ∪ { z | x Γ z et Γ⁻¹(z) ⊂ A(r) } .
     fait
```

La vérification de cet algorithme consiste essentiellement à prouver que l'assertion suivante est un invariant (cf. exercice 22) :

$r(1), r(2), \ldots, r(i)$ *sont les i premiers éléments d'une suite résultat* <u>*et*</u>
$$CANDIDATS = (E - A(r)) \cap \{ z \mid \Gamma^{-1}(z) \subset A(r) \} .$$

A ce stade de la décomposition, il est nécessaire de préciser une représentation pour les objets F, $CANDIDATS$, r, ... qui sont manipulés. Il est intéressant de noter que ce choix a été retardé au maximum et que l'algorithme a été prouvé indépendamment des représentations. De plus le choix des représentations pourra être guidé par la nature des traitements qui est déjà connue (par exemple pour r on sait qu'on ne fait que des adjonctions, ...). Pour le graphe il est utile de connaître :

— pour chaque point x, l'ensemble de ses successeurs : soit $\Gamma(x)$ cet ensemble ;

— pour chaque point z, le nombre de ses prédécesseurs stricts qui ne sont pas inclus dans $A(r)$, soit $NP(z)$ ce nombre. NP est un tableau.

Ces choix nous conduisent à un nouveau raffinement de l'algorithme :

```
Acquisition de Γ ;
Initialisation de NP
r ← ∅ ;
CANDIDATS ← ∅ ;
```

```
pour chaque x ∈ F faire si NP(x) = 0 alors CANDIDATS ←
                                      CANDIDATS ∪ { x } fsi fait ;
pour i de 1 jusqu'à n
  faire
  [Résultat : x tel que
   x ∈ CANDIDATS] ;
  r ← r ∪ (i, x) ;
  CANDIDATS ← CANDIDATS − { x } ;
  pour chaque z ∈ Γ(x) faire
                 NP(z) ← NP(z) − 1 ;
                 si NP(z) = 0 alors CANDIDATS ←
                                   CANDIDATS ∪ { z } fsi
                 fait
  fait
```

Ici, comme après chaque nouveau raffinement, il faudrait vérifier que les descriptions qui ont été détaillées par rapport aux anciennes versions sont encore correctes.

Pour achever la programmation convenons de représenter simplement :

— les éléments de F par des entiers compris entre 1 et n.

— Γ par un tableau à deux dimensions S tel que $S[z, i]$ est le i-ième successeur de z et par un tableau NS : NS[z] est le nombre des successeurs de z.

— r par un tableau à une dimension R tel que $r(i) = x$ équivaut à $R[i]=x$.

— *CANDIDATS* par un tableau de booléens *CANDIDATS* tel que $z \in$ *CANDIDATS* équivaut à *CANDIDATS* $[z]$ = vrai.

Voici un programme ALGOL 68 obtenu à partir de la dernière version de l'algorithme en utilisant les conventions précédentes (les déclarations et lectures ne sont pas explicitées) :

```
début
# Déclarations et lectures : #
   # déclaration de n, S, NS, R, NP, CANDIDAT ;
     lecture de n, S, NS #
   ent x # premier candidat # ;
   ent z # point du graphe # ;
# Initialisations : #
   # initialisation de R inutile #
   # initialisation de NP à partir de S et NS #
   # initialisation de CANDIDAT : #
   pour i depuis 1 jusqu'à n faire CANDIDAT [i] := (NP[i] = 0) fait ;
# Itération #
   pour i depuis 1 jusqu'à n
      faire
      x := 1 ; tant que non CANDIDAT [x] faire x := x + 1 fait ;
      R[i] := x ;
      CANDIDAT [x] := faux ;
```

```
        pour j depuis 1 jusqu'à NS[x]
            faire
            z := S[x, j] ;
            NP[z] := NP[z] - 1 ;
            CANDIDAT [z] := CANDIDAT [z] ou NP[z] = 0
            fait
        fait
# Impression du résultat R : #
    imprimer (R)
fin
```

Exercice 22

Montrer que l'assertion

$r(1), r(2), \ldots, r(i)$ *sont les i premiers éléments d'une suite résultat* et
$CANDIDATS = (F - A(r)) \cap \{ z \mid \Gamma^{-1}(z) \subset A(r) \}$

est un invariant de l'algorithme S_4. □

c) Conclusion

Dans l'exemple que nous venons de développer, nous avons essayé de définir le résultat cherché par un module simple ne comprenant qu'une itération. Le corps de cette itération admet comme résultat intermédiaire le couple (i, x) qui n'est pas entièrement explicité mais seulement « spécifié » par les propriétés qu'il doit vérifier. Il faut ensuite « raffiner » c'est-à-dire écrire un algorithme pour construire ce résultat intermédiaire, et ainsi de suite jusqu'au programme. A chaque raffinement, on vérifie que la nouvelle version de l'algorithme est correcte si l'on admet la correction des modules qui restent à définir. Pour cela il faut adopter la même démarche qu'au paragraphe précédent sur les preuves de programme, c'est-à-dire vérifier que les cycles préservent le premier invariant défini et que ces cycles se terminent.

Nous venons implicitement de parler de raffinements sur les structures de contrôle du programme ; à chacun de ces raffinements correspondent un (ou plusieurs) raffinement(s) sur les structures de données utilisées dans l'algorithme (choix de représentation des objets introduits) et des choix de réalisation (progression d'une unité pour déterminer le couple (i, x), introduction de l'ensemble *CANDIDATS*).

Le passage d'une étape à la suivante correspond donc à des changements limités et bien définis, la preuve associée à ce passage sera donc facile à faire.

De plus les choix tardifs de représentation ne remettent pas en cause les preuves qui les précèdent, ce qui facilite la modification de ces choix, pour s'adapter à la plus ou moins grande richesse d'un langage de programmation au niveau des structures de données.

Problème : Ecrire par raffinements successifs un algorithme pour générer une configuration de 8 reines sur un échiquier de telle sorte qu'aucune reine ne

soit en prise avec une autre (c'est-à-dire de telle sorte qu'on ne trouve pas deux reines sur une même ligne, ni sur une même colonne, ni sur une même diagonale).

4.3 Construction d'un programme par transformations d'énoncés

A chaque étape du raffinement, la méthode précédente propose d'expliciter de nouvelles actions pour mieux définir et surtout construire un certain objet, en cela on peut dire qu'elle a un caractère assez dynamique. Tout en conservant ces caractères de modularité, de correction, de recul des choix de représentation, on peut souhaiter travailler de manière plus statique en opérant par transformations d'énoncés. C'est cette idée que nous essayons de mettre en œuvre dans les exemples qui suivent :

a) Premier exemple : Recherche associative dans un tableau trié.

Un énoncé informel de ce problème peut être : étant donné un tableau t de n entiers, trié par ordre croissant, et un entier a, trouver le premier indice m tel que $t[m] = a$ s'il existe, sinon donner à m la valeur 0.

Une formalisation peut conduire à l'énoncé suivant :

Résultat : *m tel que*
$\exists\, q \in [1 : n+1]\ \underline{et}\ (1 \leqslant i < q \Rightarrow t[i] < a)\ \underline{et}\ (q \leqslant i \leqslant n \Rightarrow t[i] \geqslant a)\ \underline{et}$
$m = \underline{si}\ q \leqslant n\ \underline{et}\ t[q] = a\ \underline{alors}\ q\ \underline{sinon}\ 0\ \underline{fsi}$

Dans cet énoncé m est bien défini à partir de q.

Il reste donc à chercher un énoncé algorithmique définissant q. q vérifie :

$$\forall i\,(1 \leqslant i < q \Rightarrow t(i) < a)\ \underline{et}\ \forall j\,(q \leqslant j \leqslant n \Rightarrow t(j) \geqslant a)\,.$$

Cet énoncé ne donne pas directement un moyen de construire q mais il comprend deux parties ; on peut alors penser lui substituer un énoncé plus général qui nous permettrait de scinder le problème de la recherche de q en deux sous-problèmes.

Si nous introduisons u et v tels que :

$$P(u, v) : \forall i\,(1 \leqslant i < u \Rightarrow t(i) < a)$$
$$\forall i\,(v \leqslant j \leqslant n \Rightarrow t(j) \geqslant a)$$

nous pouvons affirmer que $u \leqslant q \leqslant v$.

Rechercher u et v vérifiant P est un problème plus général que la recherche de q (si $u = v$ alors $q = u$). Il comporte une solution évidente $u = 1$, $v = n + 1$.

Essayons donc de construire une suite (u_k, v_k) vérifiant P, démarrant à $u_0 = 1$ et $v_0 = n + 1$, et convergeant vers la solution particulière $u = v$. Pour cela on scinde à chaque étape l'intervalle $[u_k : v_k]$ en deux, d'où la définition de la suite :

$$u_0 = 1, v_0 = n + 1$$
$$w_i = (u_i + v_i) \div 2$$
$$u_{i+1} = \underline{si}\ t(w_i) < a\ \underline{alors}\ w_i + 1\ \underline{sinon}\ u_i\ \underline{fsi}$$
$$v_{i+1} = \underline{si}\ t(w_i) < a\ \underline{alors}\ v_i\ \underline{sinon}\ w_i\ \underline{fsi}$$

On se convainc que $P(u_k, v_k)$ est vrai.
D'où l'énoncé algorithmique définissant q :

> <u>*Résultat*</u> : *q tel que*
> $u := 1;\quad v := n + 1;$
> <u>*tant que*</u> $u \neq v$ <u>*faire*</u>
> $w := (u + v) \div 2;$
>   <u>*si*</u> $t[w] < a$ <u>*alors*</u> $u := w + 1$ <u>*sinon*</u> $v := w$ <u>*fsi*</u>
>   <u>*fait*</u>;
> $q = u$.

$P(u, v)$ est l'invariant de ce cycle et il est clair que :

$$(u = v\ \underline{et}\ P(u, v)) \Rightarrow u \text{ vérifie la définition de } q .$$

Cet algorithme est donc partiellement correct.

Pour la correction totale, il suffit de remarquer que la suite $v - u$ décroît strictement à chaque itération ; donc l'algorithme s'arrête.

Pour bien se convaincre de la nécessité d'une démonstration complète, le lecteur est invité à examiner la situation proposée dans l'exercice suivant :

Exercice 23

Supposons que l'on ait défini q par :

> $u := 1;\ v := n;$
> <u>*tant que*</u> $u \neq v$ <u>*faire*</u>
> $w := (u + v) \div 2;$
>   <u>*si*</u> $t[w] < a$ <u>*alors*</u> $u := w$ <u>*sinon*</u> $v := w$ <u>*fsi*</u>
>   <u>*fait*</u>;
> $q = u$.

Etudier la correction partielle, puis la correction totale de cet algorithme. □

Exercice 24

Insertion dichotomique.

Etant donné un tableau $t[1 : n]$ trié, insérer une valeur a dans ce tableau de façon à obtenir un tableau $t'[0 : n]$ trié.

1) Proposer un énoncé formel (ED) de ce problème.

2) Déduire de (ED) un énoncé algorithmique correct (EA) en utilisant la même démarche que dans l'exemple précédent.

b) Deuxième exemple : « Le drapeau hollandais » (dû à DIJKSTRA)

On dispose de n cases dans lesquelles sont rangées n boules, de couleur bleue, blanche ou rouge (les trois couleurs du drapeau hollandais). Il s'agit de regrou-

per en tête toutes les boules rouges, en queue toutes les bleues, sachant que l'on ne dispose que des fonctions suivantes :

ech(i, j) qui réalise la permutation des contenus cases i et j,

$c(i)$ qui teste la couleur de la boule contenue dans la case i et dont le résultat est bleu, blanc ou rouge.

De plus, cette dernière fonction ne doit pas être utilisée plus de n fois.

Donnons un énoncé formel E_0 de notre problème :

E_0 : *Résultat* : *r et b tels que* :
$1 \leqslant i \leqslant r \Rightarrow c(i) = rouge$ *et*
$r < i < b \Rightarrow c(i) = blanc$ *et*
$b \leqslant i \leqslant n \Rightarrow c(i) = bleu$.

Cet énoncé ne permet pas de calculer directement les résultats (sauf par essais et erreurs).

Essayons donc, à nouveau, de définir r et b par un ou plusieurs énoncés plus larges que E_0 et de faire converger vers E_0 ces énoncés approchés.

Comme énoncé élargi considérons l'énoncé :

Résultat : r', r'', b', b'' *tels que*

$$p \begin{cases} (1 \leqslant i \leqslant r' \Rightarrow c(i) = rouge \text{ et} \\ r'' < i < b' \Rightarrow c(i) = blanc \text{ et} \\ b'' \leqslant i \leqslant n \Rightarrow c(i) = rouge) . \end{cases}$$

Cet énoncé admet une solution évidente :

$r' = r'' = 0$; $b' = 1$; $b'' = n + 1$ (en particulier on a bien $r'' < b'$) .

Si nous obtenons $r' = r''$ et $b' = b''$ nous pouvons alors poser $r = r'$ et $b = b'$. Ceci conduit à l'algorithme de construction de r et b suivant :

```
Résultat : r et b tels que :
r' := r'' := 0 ; b' := 1 ; b'' := n + 1 ;
tant que r' ≠ r'' ou b' ≠ b'' faire
                              [Classer les couleurs :
                              diminuer r'' − r' ou b'' − b'
                              en préservant l'invariant p]
                              fait ;
r = r' ; b = b' .
```

Le module « *Classer les couleurs* » sera explicité par la suite.

Remarquons que la condition d'arrêt est $r' = r''$ *et* $b' = b''$, or à l'initialisation nous avons déjà $r' = r''$; il nous faut donc essayer de conserver $r' = r''$. Un nouvel énoncé élargi peut donc être :

Résultat : r', b', b'' *tels que*

$$q \begin{cases} (1 \leqslant i \leqslant r' \Rightarrow c(i) = rouge \\ r' < i < b' \Rightarrow c(i) = blanc \\ b'' \leqslant i \leqslant n \Rightarrow c(i) = bleu) \end{cases}$$

et la définition de r et b devient :

> *Résultat* : *r et b tels que* :
> $r' := 0 ; b' := 1 ; b'' := n + 1 ;$
> *tant que* $b' \neq b''$ *faire*
> [*Classer les couleurs* : *diminuer* $b'' - b'$ *en préservant l'invariant* q]
> *fait* ;
> $r = r' ; b = b'$.

q devant être un invariant la correction partielle est évidente. Explicitons donc le module « Classer les couleurs ». De manière schématique, avant l'exécution de ce module nous sommes dans le cas représenté par la figure 21.

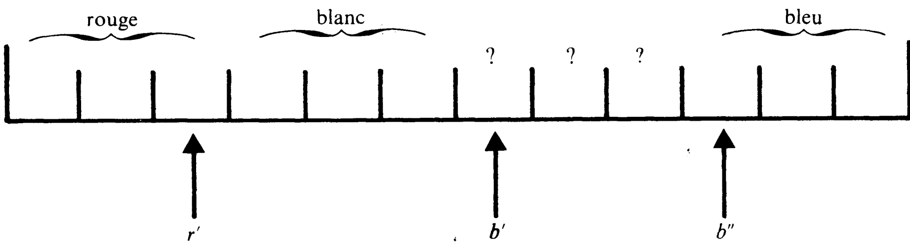


Figure 21 Etat des cases avant une étape de l'itération.

Examinons alors le contenu de b' :

Classer les couleurs : *cas* $c(b') = $ *rouge* : *mettre avec les rouges* ;
$c(b') = $ *blanc* : *mettre avec les blancs* ;
$c(b') = $ *bleu* : *mettre avec les bleus*
fcas

Détaillons « *mettre avec les rouges* » : avant exécution, nous avons

$$q \text{ et } c(b') = rouge$$

qui est vérifié ; il nous suffit donc de procéder à la suite d'instructions :

$$ech(b', r' + 1) ; r' := r' + 1 ; b' := b' + 1 ;$$

il est clair que si q *et* $c(b') = rouge$ est vrai avant exécution de cette séquence, q est vérifié après son exécution.

Dans les deux autres cas, il suffit de faire respectivement $b' := b' + 1$ et $(b'' := b'' - 1 ; ech(b', b''))$.

La correction totale résulte de la constatation suivante : à chaque itération, $b'' - b'$ décroît de 1 dans $\mathbb{N}$, or $q \Rightarrow b' \leqslant b''$ et donc l'algorithme s'arrête après n itérations (nous n'avons utilisé que n fois la fonction c).

L'algorithme obtenu est correct.

c) Conclusion

La stratégie que nous pouvons déduire de ces exemples est donc la suivante. Nous partons d'un énoncé donné le plus souvent sous la forme :

Calculer x vérifiant la propriété $q(a, x)$ sachant $p(a)$. De cet énoncé on déduit un algorithme A correct, c'est-à-dire qui vérifie $p(a) \{ A \} q(a, x)$ et qui termine.

Dans les cas les plus simples, on est amené à construire x vérifiant $q(a, x)$ à partir d'intermédiaires x', x'', etc... vérifiant $q'(a, x')$, $q''(a, x'')$, ..., $q^n(a, x^n)$. Ceci conduit à des algorithmes A', A'', ..., A^n tels que

$$p(a) \{ A^n \} q^n(a, x^n) \{ A^{n-1} \} \dots q'(a, x') \{ A' \} q(a, x) .$$

Mais il arrive fréquemment qu'on ne puisse définir le résultat directement en fonction des données et de quelques résultats intermédiaires. On essaie alors d'approcher ce résultat en construisant une (ou plusieurs) suite(s) dont les éléments approchent de plus en plus q ; les éléments x' de cette suite vérifient un prédicat $q'(a, x', k)$ approché, plus large, où k est une variable supplémentaire, et tel qu'il existe une propriété $p_1(k)$ vérifiant :

$$p_1(k) \ \underline{et} \ q'(a, x', k) \Rightarrow q(a, x') .$$

On peut donc résumer le processus de construction d'algorithme de la façon suivante :

— construire un algorithme *INIT* tel que

$$\exists k_0, \exists x_0 \ p(a) \{ INIT \} \ q'(a, x_0, k_0)$$

— construire le corps du cycle C tel que

$$\underline{non} \ p_1(k) \wedge q'(a, x, k) \{ C \} \ q'(a, x', k')$$

(l'invariant de ce cycle est donc q').

— assurer la convergence de l'itération, c'est-à-dire trouver un bel ordre sur le domaine de définition de k et s'assurer qu'à chaque étape k décroît strictement.

On obtient ainsi l'algorithme :

$$INIT ; \underline{tant\ que} \ \underline{non} \ p_1 \ \underline{faire} \ C \ \underline{fait} .$$

Ce processus peut évidemment être repris pour la construction de *INIT*, dans ce cas nous aurons une succession de cycles ; il peut reprendre pour la construction de C, ce qui donnera des cycles imbriqués.

Ces quelques règles reviennent à énoncer de façon formelle la vision intuitive qu'on a d'un algorithme lorsque celui-ci comporte des cycles, la principale difficulté étant d'exprimer les invariants de chacun de ces cycles.

4.4 Perspectives

Nous avons donné deux démarches possibles, pour la construction de programmes. Dans ce domaine beaucoup reste à faire.

Les travaux actuels portent sur la définition de langages de haut niveau mieux adaptés à la description d'algorithmes et sur la définition d'outils permettant de construire systématiquement des programmes corrects [ARS, 76], [SIN, 76], [ASH, 76], [DIJ, 76].

La preuve d'algorithmes se fait toujours en montrant que toute solution fournie par cet algorithme est bien solution de l'énoncé de départ. Or la démarche naturelle consiste à partir de l'énoncé du problème et à le transformer en des énoncés de plus en plus algorithmiques. PAIR [PAI, 75] indique une méthodologie de recherche et de preuve du passage d'un énoncé à un autre dans le sens où l'effectue le programmeur, c'est-à-dire de l'énoncé du problème vers les expressions algorithmiques obtenues aux différentes étapes de raffinement.

En tout cas, l'écriture automatique de programmes corrects n'est pas un problème résolu. Signalons toutefois que pour certains énoncés, ceux qui appartiennent à la classe des schémas de programmes récursifs, ce passage à un programme correct a été bien étudié. Comme nous l'avons vu au chapitre 2, la récursivité fournit un moyen agréable de passer de certains énoncés à un algorithme.

Enfin la correction d'un programme se définit par rapport à l'énoncé d'un problème, celle d'un module par rapport à des spécifications d'entrée et de sortie.

Notons aussi que l'étape importante de formalisation d'un problème à partir d'un énoncé plus ou moins bien défini a été jusqu'à présent assez peu abordée de manière générale alors que c'est une étape clé dans la plupart des applications pratiques. La construction méthodique de spécifications structurées, claires et adéquates de problèmes est également un domaine peu étudié.

5 APPLICATION DU CALCUL RELATIONNEL A LA THÉORIE DES PREUVES DE PROGRAMME

5.1 Introduction, rappel et but poursuivi

Nous utilisons ici la terminologie introduite au chapitre 3, paragraphe 5. Un programme est donc considéré comme spécification d'une relation entre les états d'entrée du programme et ses états de sortie. Notre but est une étude théorique de la correction partielle des programmes. Rappelons qu'un programme est partiellement correct par rapport à deux prédicats p et q si tout état d'entrée x vérifiant p et conduisant à l'arrêt du programme fournit un état de sortie y vérifiant q. Ici nous travaillons sur des programmes indéterministes et nous imposerons que tous les états de sortie vérifient q. Donc la correction partielle d'un programme répond à la définition suivante :

Définition 6 : Soit T un programme spécifiant une relation R. On dit que T est partiellement correct par rapport aux deux prédicats p et q si la condition suivante est vérifiée

$$\forall x \, \forall y \, (p(x) \; \underline{et} \; x \, R \, y \Rightarrow q(y)) \, . \qquad \square$$

En terme relationnel, cette condition s'écrit

$$p ; R \subseteq R ; q .$$

Ceci correspond aussi à la notation de HOARE $p \{ T \} q$ (cf. § 3.2).

Dans la suite de l'exposé, on confond pour des raisons d'allègement d'écriture un programme T et la relation R spécifiée par ce programme (voir chapitre 3, § 5).

Dans ce paragraphe on montre que la méthode de preuve (§ 2) est valide et qu'il est toujours possible d'étiqueter un programme de façon à prouver sa correction. Nous faisons ces démonstrations dans deux cas ; d'abord, dans le cas d'un simple cycle du type *tant que* ... *faire* ... *fait*, nous montrons l'existence d'un invariant de cycle ; puis, dans le cas d'un schéma de programme général, nous exhibons un étiquetage associé aux deux prédicats d'entrée et de sortie.

Les méthodes que nous allons esquisser ici se généralisent aux cas des schémas récursifs, le lecteur intéressé pourra se rapporter à l'article original [BAK, 72]. De même, on peut traiter dans le cadre de cette méthode certains aspects de la notion de correction totale. Nous renvoyons au même article pour ces développements qui ne seront pas traités ici ; voir aussi le paragraphe 3.4 et l'article de BASU et YEH [BAS, 75].

Les résultats énoncés ici ne doivent pas être considérés comme des procédés pratiques de construction d'invariants de cycle ou d'étiquetages. Par exemple, dans un cas où l'invariant de cycle est « *x est un entier pair* », la méthode décrite ici pourra conduire à une forme de ce prédicat du type

$$x = 0 \ \underline{ou} \ x = 2 \ \underline{ou} \ x = 4 \ \underline{ou} \ x = 6 \ \underline{ou} \ \dots .$$

Dans ce cas particulier, le passage d'une forme à l'autre est simple, mais dans des cas moins simples, l'emploi exclusif de la méthode théorique décrite ici peut conduire à des écritures de prédicats de longueur infinie et inexploitables pratiquement. Le résultat important est que ces objets existent et qu'une recherche empirique n'est pas vaine.

5.2 Correction partielle d'un cycle *tant que* ... *faire* ... *fait*

Soit R un programme. On a vu au chapitre 3, paragraphe 5.2 que la relation $(r ; R)^* ; \bar{r}$ notée $r * R$ est associée au programme *tant que* r *faire* R *fait*. Supposons que l'on veuille démontrer que ce programme est partiellement correct par rapport à deux prédicats p et q. On cherche donc à démontrer que

(1) $\quad p ; r * R \subseteq r * R ; q .$

La méthode pratique (règle 3 du paragraphe 2.3) consiste à chercher un invariant de cycle s tel que

(2) $\quad p \subseteq s \quad$ (l'invariant est vérifié par tout état initial compatible avec p)

(3) $s\,;r\,;R \subseteq R\,;s$ (l'invariant est vérifié après un cycle s'il est vérifié avant l'exécution de celui-ci)

(4) $s\,;\bar{r} \subseteq q$ (l'invariant et la condition de sortie du cycle impliquent q).

Remarquons que ces trois conditions s'écrivent avec les notations de HOARE sous la forme

$$\frac{p \Rightarrow s,\ s\ \underline{\text{et}}\ r\ \{\, R\,\}\ s,\ s\ \underline{\text{et}}\ \underline{\text{non}}\ r \Rightarrow q}{p\ \{\ \underline{\textit{tant que}}\ r\ \underline{\textit{faire}}\ R\ \underline{\textit{fait}}\ \}\ q}$$

cette règle est une conséquence simple des règles indiquées au paragraphe 3.2.

Montrons qu'effectivement, si on peut trouver un tel invariant, alors le programme <u>*tant que*</u> r <u>*faire*</u> R <u>*fait*</u> est partiellement correct par rapport aux prédicats p et q. Remarquons déjà que la condition (3) $s\,;r\,;R \subseteq R\,;s$ est équivalente à $s\,;r\,;R \subseteq r\,;R\,;s$.

En effet, en utilisant (3), on obtient :

$$s\,;r\,;R = r\,;s\,;r\,;R \subseteq r\,;R\,;s\,.$$

La réciproque est évidente.

On en déduit par récurrence que, pour tout n,

$$s\,;(r\,;R)^n \subseteq (r\,;R)^n\,;s$$

donc

(5) $$s\,;(r\,;R)^* \subseteq (r\,;R)^*\,;s\,.$$

Le résultat est alors une conséquence des inclusions et égalités suivantes :

$$p\,;(r * R) \subseteq s\,;(r * R) \quad \text{(de (2))}$$
$$s\,;(r * R) = s\,;(r\,;R)^*\,;\bar{r} \quad \text{(par définition)}$$
$$s\,;(r\,;R)^*\,;\bar{r} \subseteq (r\,;R)^*\,;s\,;\bar{r} \quad \text{(de (5))}$$
$$(r\,;R)^*\,;s\,;\bar{r} \subseteq (r\,;R)^*\,;\bar{r}\,;q \quad \text{(de (4))}$$
$$(r\,;R)^*\,;\bar{r}\,;q = (r * R)\,;q\,.$$

Ceci n'est pas nouveau et a déjà été étudié au paragraphe 2 avec d'autres notations. Mais dans ce paragraphe, il n'était pas question de démontrer une réciproque à cette proposition, c'est-à-dire trouver s connaissant p et q. C'est ce que nous allons faire après avoir introduit quelques opérations supplémentaires sur les prédicats et avoir démontré quelques lemmes techniques.

5.3 Opérations $\circ$ et $\rightarrow$ sur les prédicats

Comme on l'a rappelé dans l'introduction, un programme R est partiellement correct par rapport aux deux prédicats p et q si la formule suivante est satisfaite :

$$\forall x\ \forall y\ (p(x)\ \underline{\text{et}}\ x\ R\ y \Rightarrow q(y))\,.$$

Cette formule est équivalente à chacune des deux formules suivantes :

$$\forall y\ ((\exists x)\ (p(x)\ \underline{et}\ x\ R\ y) \Rightarrow q(y))$$
$$(\forall x)\ (p(x) \Rightarrow (\forall y)\ (x\ R\ y \Rightarrow q(y)))\ .$$

Définition 7 : On pose par définition

$$(p \circ R)\ (y) = \exists x\ (p(x)\ \underline{et}\ x\ R\ y)$$
$$(R \rightarrow q)\ (x) = \forall y\ (x\ R\ y \Rightarrow q(y))\ .$$ □

La justification de l'introduction de ces deux opérations sur les prédicats apparaît dès que l'on étudie leurs propriétés.

Propriétés des opérations $\circ$ *et* $\rightarrow$.

1) Soient R un programme et p un prédicat, alors R est partiellement correct par rapport à p et $p \circ R$ et par rapport à $p \rightarrow R$ et p. Ceci est évident eu égard aux définitions de $p \circ R$ et $p \rightarrow R$.

2) Soient R un programme et p un prédicat ; supposons que l'on impose aux états d'entrée de R de vérifier p. Il est intéressant de chercher les prédicats q tels que R soit partiellement correct par rapport à p et q et en particulier de chercher parmi ces prédicats q celui qui apporte le plus de renseignements possibles sur les états de sortie de R. Or $p \circ R$ est justement ce prédicat. En effet

$$(1) \qquad p\,;R \subseteq R\,;q \Leftrightarrow p \circ R \subseteq q\ .$$

Démonstration : Si $p \circ R \subseteq q$, on obtient $p\,;R \subseteq R\,;(p \circ R) \subseteq R\,;q$, d'où $p\,;R \subseteq R\,;q$.

Si $p\,;R \subseteq R\,;q$, soit y tel que $(p \circ R)\ (y)$.

Il existe un x tel que p(x) et $x\ R\ y$; donc y est un résultat de R pour une valeur d'entrée x vérifiant p et y vérifie q.

3) L'opération $\rightarrow$ possède une propriété analogue. En effet, si R est un programme et q un prédicat, quelle condition faut-il mettre en entrée pour que tous les états de sortie de R vérifient q ? Parmi ces conditions, quelle est celle qui est la moins contraignante ? Le prédicat $R \rightarrow q$ résout ce problème. En effet

$$(2) \qquad p\,;R \subseteq R\,;q \Leftrightarrow p \subseteq R \rightarrow q\ .$$

En résumé, si R est un programme, $p \circ R$ est la plus forte post-condition associée à la pré-condition p et $R \rightarrow q$ est la plus faible pré-condition associée à la post-condition q [DIJ, 76].

Exercice 25

Montrer que $p \circ R$ est la borne inférieure des prédicats q tels que

$$p\,;R \subseteq R\,;q\ .$$ □

Exercice 26

Démontrer que $p\,;R \subseteq R\,;q \Leftrightarrow p \subseteq R \rightarrow q$.

En déduire que $R \rightarrow q = \cup\ \{\,p \mid p\,;R \subseteq R\,;q\,\}$. □

Exercice 27

Démontrer les propriétés suivantes des opérations $\circ$ et $\rightarrow$:

— $p \circ (R_1 ; R_2) = (p \circ R_1) \circ R_2$ — $(R_1 ; R_2) \rightarrow q = R_1 \rightarrow (R_2 \rightarrow q)$
— $p \circ (R_1 \cup R_2) = (p \circ R_1) \cup (p \circ R_2)$ — $(R_1 \cup R_2) \rightarrow q = (R_1 \rightarrow q) \cap (R_2 \rightarrow q)$
— $R_1 \subseteq R_2 \Rightarrow p \circ R_1 \subseteq p \circ R_2$ — $R_1 \subseteq R_2 \Rightarrow R_1 \rightarrow q \supseteq R_2 \rightarrow q$
— $p \circ E = p$ — $E \rightarrow q = q$. □

Les propriétés de l'opération $\circ$ serviront dans la démonstration des théorèmes suivants.

Les opérations $\circ$ et $\rightarrow$ ne sont pas indépendantes des autres opérations sur les relations. En effet, on montre facilement que

$$p \circ R = R^{-1} ; p ; R \cap E$$

$$R \rightarrow p = \overline{\overline{p} \circ R^{-1}} .$$

Dans le cadre d'une théorie axiomatisée traitant de ce sujet, les prédicats $p \circ R$ et $R \rightarrow p$ pourraient être introduits par des définitions de ce type (voir le paragraphe 5.5 du chapitre 3, consacré au μ-calcul).

5.4 Complétude de la méthode de Floyd pour une itération

Les outils introduits sont maintenant suffisants pour démontrer la réciproque annoncée au paragraphe 2.

Théorème 5 : Soit R un programme. Le programme

<u>*tant que*</u> r <u>*faire*</u> R <u>*fait*</u>

est partiellement correct par rapport à p et q si et seulement s'il existe un prédicat s tel que

— $p \Rightarrow s$
— s <u>et</u> $\overline{r} \Rightarrow q$
— s est invariant de cycle pour R.

Autrement dit, en termes de relations, on obtient

$$p ; r * R \subseteq r * R ; q \Leftrightarrow \exists s \begin{cases} p \subseteq s \\ s ; r ; R \subseteq R ; s \\ s ; \overline{r} \subseteq q . \end{cases}$$ □

Démonstration : Le fait que cette condition soit suffisante a été vu au paragraphe 2. Il nous reste donc à montrer l'existence de s sachant que $p ; r * R \subseteq r * R ; q$. Si partant d'un état x on effectue n itérations avant de sortir du programme, la plus forte condition vérifiée par les valeurs de sortie est $p \circ (r ; R)^n$ (cf. la propriété 2 du paragraphe 3). Comme à priori on ne

connaît pas le nombre de cycles effectués avant de sortir, on est amené à prendre

$$s = \bigcup_{n=0}^{\infty} p \circ (r ; R)^n = p \circ (r ; R)^* .$$

Vérifions que s convient :

— $p = p \circ E \subseteq p \circ (r ; R)^* = s$.

— Pour vérifier que $s ; r ; R \subseteq R ; s$, il suffit de démontrer que $s ; r ; R \subseteq r ; R ; s$ et donc que $s \circ (r ; R) \subseteq s$, c'est-à-dire

$$(p \circ (r ; R)^*) \circ (r ; R) \subseteq p \circ (r ; R)^* .$$

Or, $$(p \circ (r ; R)^*) \circ (r ; R) = p \circ [(r ; R)^* ; r ; R]$$

(exercice 27) et de façon évidente

$$(r ; R)^* ; r ; R \subseteq (r ; R)^* .$$

D'où l'inclusion désirée.

Il reste à démontrer que $s ; \bar{r} \subseteq q$ ou, ce qui est équivalent, que $s ; \bar{r} \subseteq \bar{r} ; q$, c'est-à-dire $(p \circ (r ; R)^*) ; \bar{r} \subseteq \bar{r} ; q$.

Pour cela, il suffit de démontrer que $(p \circ (r ; R)^*) \circ \bar{r} \subseteq q$. Or :

$$(p \circ (r ; R)^*) \circ \bar{r} = p \circ [(r ; R)^* ; \bar{r}] = p \circ (r * R) .$$

Donc on cherche à prouver que $p \circ (r * R) \subseteq q$. Mais ceci est équivalent à $p ; (r * R) \subseteq r * R ; q$ qui est l'hypothèse de départ. C.Q.F.D.

Remarque : Comme on l'a déjà précisé dans l'introduction, ce genre de théorème ne doit pas être considéré comme un procédé effectif de construction d'invariants de cycles dans des exemples pratiques. Ce théorème prouve qu'une recherche empirique, au moyen des euristiques développées au paragraphe 2.3, d'invariants de cycles n'est pas vaine. Cependant, pour illustrer les résultats obtenus, on va utiliser, à titre d'exercice d'école, cette méthode sur un exemple.

Exemple 18 : Considérons le programme T suivant :

$$T : \underline{\textit{tant que}}\ y > 0\ \underline{\textit{faire}}\ x := x + 1 ; y := y - 1\ \underline{\textit{fait}} .$$

Supposons que l'on prenne pour p le prédicat

$$p(x, y) = (x = a\ \underline{et}\ y = b) .$$

L'invariant de cycle construit par la méthode théorique est $s = p \circ (r ; R)^*$ avec $r(x, y) = y > 0$ et $\# (x, y)\ R(x', y') \Leftrightarrow x' = x + 1\ \underline{et}\ y' = y - 1$.

Donc $s(x, y)$

$= \exists(x', y')\ (p(x', y')\ \underline{et}\ (x', y')\ (r ; R)^*\ (x, y))$
$= (\exists x', y')\ (x' = a\ \underline{et}\ y' = b\ \underline{et}\ (\exists n)\ ((x', y')\ (r ; R)^n)\ (x, y))$
$= (\exists n)\ ((a, b)\ (r ; R)^n\ (x, y))$
$= (\exists n)\ (b > 0\ \underline{et}\ b - 1 > 0\ \underline{et} \dots \underline{et}\ b - n + 1 > 0\ \underline{et}\ x = a + n\ \underline{et}\ y = b - n)$
$= (\exists n)\ (b - n + 1 > 0\ \underline{et}\ x = a + n\ \underline{et}\ y = b - n) .$

La théorie développée ci-dessus indique que T est partiellement correct par rapport à p et

$$s\,;\,\bar{r} = (\exists n)\,(b - n + 1 > 0 \;\underline{\text{et}}\; x = a + n \;\underline{\text{et}}\; y = b - n) \;\underline{\text{et}}\; y \leqslant 0\,.$$

Donc T est partiellement correct par rapport à $p(x, y) = (x = a \;\underline{\text{et}}\; y = b)$ et $q(x, y) = \exists n\,(n - 1 < b \leqslant n \;\underline{\text{et}}\; x = a + n \;\underline{\text{et}}\; y = b - n)$.

En particulier, si b est entier, on a $q(x, y) = (x = a + b \text{ et } y = 0)$.
De même, si b est entier $s(x, y) = (y \geqslant 0 \;\underline{\text{et}}\; x + y = a + b)$. □

5.5 Correction partielle d'un schéma avec branchements

a) Traduction de schéma avec branchements en schéma relationnel

Tout schéma de programme peut être traduit en un programme relationnel en remplaçant les branchements, séquentiels ou non, par des appels de procédure. Nous nous contentons d'expliciter cette traduction sur un exemple.

On commence pour cela par introduire des points de coupure en utilisant les mêmes euristiques qu'au paragraphe 2.3, puis on associe à chacun de ces points un nom de procédure. Le programme principal est le nom de la procédure étiquetant le point d'entrée et la déclaration d'une procédure se lit en suivant les chemins élémentaires partant du point de coupure considéré jusqu'à la rencontre d'un nouveau point de coupure.

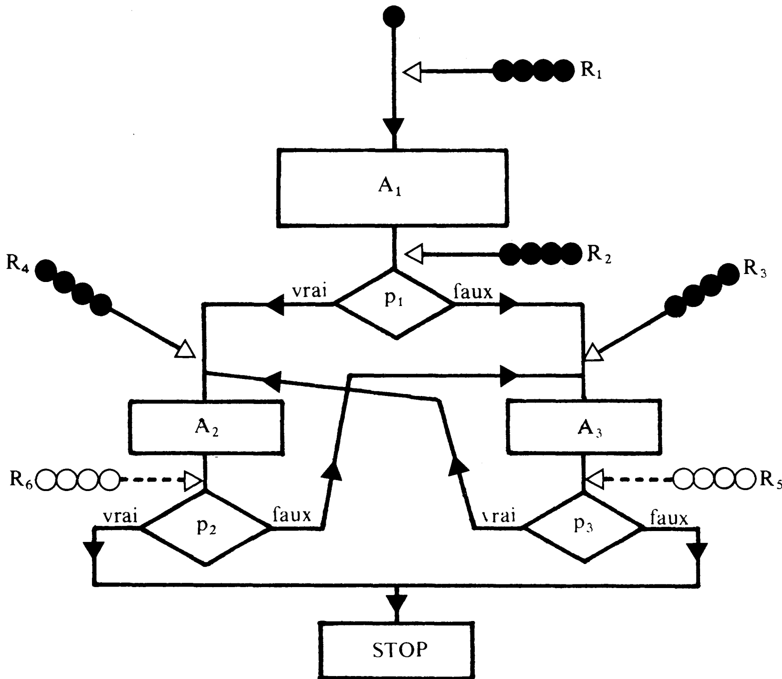


Figure 22 Un schéma avec branchement.

Le schéma avec branchements de la figure 22 est équivalent au schéma relationnel qui suit :

$$\pi = (\{ R_1 \Leftarrow A_1 ; R_2 ,\ R_2 \Leftarrow \overline{p}_1 ; R_3 \cup p_1 ; R_4 ,\ R_3 \Leftarrow A_3 ; p_3 ; R_4 \cup A_3 ; p_3 , R_4 \Leftarrow A_2 ; p_2 \cup A_2 ; \overline{p}_2 ; R_3 \} ,\ R_1) .$$

On peut, en fait, ramener les programmes récursifs provenant de la traduction d'un schéma avec branchements à une forme canonique où toutes les déclarations sont de la forme

$$R_i \Leftarrow A_{i1} ; R_1 \cup A_{i2} ; R_2 \cup \ldots \cup A_{in} ; R_n \cup A_{in+1}$$

où chaque A_{ij} désigne une relation constante élémentaire.

Pour obtenir cette forme particulière, on commence par introduire de nouveaux points de coupure, de façon qu'un chemin élémentaire joignant deux points de coupure porte au plus une action ou un test. Sur l'exemple ci-dessus, on obtient

$$\pi' = (\{ R_1 \Leftarrow A_1 ; R_2 , R_2 \Leftarrow \overline{p}_1 ; R_3 \cup p_1 ; R_4 , R_3 \Leftarrow A_3 ; R_5 , R_4 \Leftarrow A_2 ; R_6 , R_5 \Leftarrow p_3 ; R_4 \cup \overline{p}_3 , R_6 \Leftarrow \overline{p}_2 ; R_3 \cup p_2 \}, R_1) .$$

Et enfin on utilise les relations constantes Ω et E pour arriver à la forme canonique annoncée. Pour π', on obtient

$$\pi'' = (\{ R_1 \Leftarrow \Omega ; R_1 \cup A_1 ; R_2 \cup \Omega ; R_3 \cup \ldots \cup \Omega ; R_6 \cup \Omega$$
$$R_2 \Leftarrow \Omega ; R_1 \cup \Omega ; R_2 \cup \overline{p}_1 ; R_3 \cup \ldots \cup \Omega ; R_6 \cup \Omega$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$R_6 \Leftarrow \Omega ; R_1 \cup \Omega ; R_2 \cup \overline{p}_2 ; R_3 \cup \ldots \cup \Omega ; R_6 \cup p_2 \} , R_1)$$

(la relation E sert pour transformer $P \Leftarrow P'$ en $P \Leftarrow E ; P'$).

b) Théorème de complétude de la méthode de Floyd pour les schémas avec branchements

Le paragraphe 2.3 donne une méthode empirique pour vérifier qu'un schéma avec branchements est partiellement correct par rapport à deux prédicats p et q. De plus, le théorème 1 du paragraphe 2.2 montre que cette méthode est correcte. La lecture de ce paragraphe conduit à penser que la méthode mise en évidence est toujours applicable, mais on pouvait alors difficilement démontrer ce résultat. C'est ce que nous allons faire ici en prouvant le théorème suivant.

Théorème 6 : Soit $\pi = (\Delta, R_1)$ un schéma relationnel avec

$$\Delta = \{ R_i \Leftarrow A_{i1} ; R_1 \cup A_{i2} ; R_2 \cup \ldots \cup A_{in} ; R_n \cup A_{in+1} ; 1 \leqslant i \leqslant n \} .$$

π est partiellement correct par rapport à p et q si et seulement s'il existe des prédicats $p_1, p_2, \ldots, p_{n+1}$ tels que

— $p \subseteq p_1$,
— $(\forall i \in [1, n]) (\forall j \in [1, n+1]) (p_i ; A_{ij} \subseteq A_{ij} ; p_j)$,
— $p_{n+1} \subseteq q$ □

(les prédicats $p_1, \ldots, p_n$ sont exactement les prédicats qui étiquettent les points de coupure dans la méthode du paragraphe 2.3. La deuxième condition, en particulier, est exactement le pas 3 de cette méthode).

Démonstration

1) Le fait que cette condition soit suffisante résulte immédiatement du théorème 1 du paragraphe 2.2. On peut refaire une démonstration de ce résultat dans le cadre de cette théorie en raisonnant sur les calculs réussis de π (chapitre 3, § 5.4).

Un calcul réussi de π est de la forme

$$(E, x_0, R_1), (A_{1i_1}, x_1, R_{i_1}), (A_{1i_1}\, A_{i_1i_2}, x_2, R_{i_2}), \ldots ,$$
$$(A_{1i_1}\, A_{i_1i_2}, \ldots, A_{i_ki_{k+1}}, x_{k+1}, R_{i_{k+1}}), \ldots ,$$
$$(A_{1i_1}\, A_{i_1i_2}, \ldots, A_{i_mn+1}, x_{m+1}, E) ,$$

et, d'après la définition de ces calculs, on a, pour $0 \leqslant k \leqslant m + 1$,

$$x_k\, A_{i_ki_{k+1}}\, x_{k+1}$$

avec les conventions $i_0 = 1$ et $i_{m+1} = n + 1$.

Supposons que x_0 vérifie p ; comme $p \subseteq p_1$, x_0 vérifie aussi p_1. De plus, comme $p_i \,;\, A_{ij} \subseteq A_{ij} \,;\, p_j$ pour tout i et j, on en déduit que pour tout $k \in [0, q]$, on a

$$p_{i_k}(x_k) \text{ \underline{et} } x_k\, A_{i_ki_{k+1}}\, x_{k+1} \Rightarrow p_{i_{k+1}}(x_{k+1}) .$$

On en déduit immédiatement par récurrence $p_{i_{m+1}}(x_{m+1})$ c'est-à-dire $p_{n+1}(x_{m+1})$.

Or on a $p_{n+1} \subset q$. Donc x_{m+1} vérifie q. Donc tout état d'entrée vérifiant p et produisant un calcul réussi fournit des états de sortie vérifiant q et, par conséquent, π est partiellement correct vis-à-vis de p et q.

Remarque : On aurait pu faire cette démonstration en utilisant la règle d'induction de Scott (chapitre 3, § 5.5) sur la proposition

$$p_1 \,;\, R_1 \subseteq R_1 \,;\, p_{n+1} \text{ \underline{et} } p_2 \,;\, R_2 \subseteq R_2 \,;\, p_{n+1} \text{ \underline{et} } \ldots \text{ \underline{et} }$$
$$p_n \,;\, R_n \subset R_n \,;\, p_{n+1} .$$

2) Montrons que cette condition est nécessaire. On suppose que $p \,;\, R_1 \subseteq R_1 \,;\, q$. Posons pour tout $j \in [1, n + 1]$ $p_j = R_j \to q$ (avec la convention $R_{n+1} = E$, ce qui est conforme à la réalité puisque après les actions $A_{i_{n+1}}$, il ne reste plus d'action à effectuer). Vérifions que les prédicats p_j conviennent :

a) Par hypothèse, on a $p \,;\, R_1 \subseteq R_1 \,;\, q$. Donc d'après le paragraphe 5.3, on a $p \subseteq (R_1 \to q)$ soit $p \subseteq p_1$.

b) Il faut vérifier que $p_i \,;\, A_{ij} \subseteq A_{ij} \,;\, p_j$. Pour cela, il suffit de vérifier que $p_i \subseteq (A_{ij} \to p_j)$, c'est-à-dire

$$(R_i \to q) \subseteq (A_{ij} \to (R_j \to q)) .$$

D'après l'exercice 27, on a $A_{ij} \to (R_j \to q) = (A_{ij} \,;\, R_j) \to q$. D'autre part, $A_{ij} \,;\, R_j$ est un terme de la déclaration de R_i, donc $A_{ij} \,;\, R_j \subseteq R_i$. Donc $((A_{ij} \,;\, R_j) \to q) \supseteq (R_i \to q)$, d'où le résultat.

c) $p_{n+1} = R_{n+1} \to q = E \to q = q$. C.Q.F.D.

De même que le théorème 5, ce résultat est théorique et ne doit pas être considéré comme un procédé effectif de vérification de programme. Toutefois, à titre d'exercice, nous allons illustrer ce théorème.

Exemple 19 : Considérons le schéma de programme π de la figure 23.

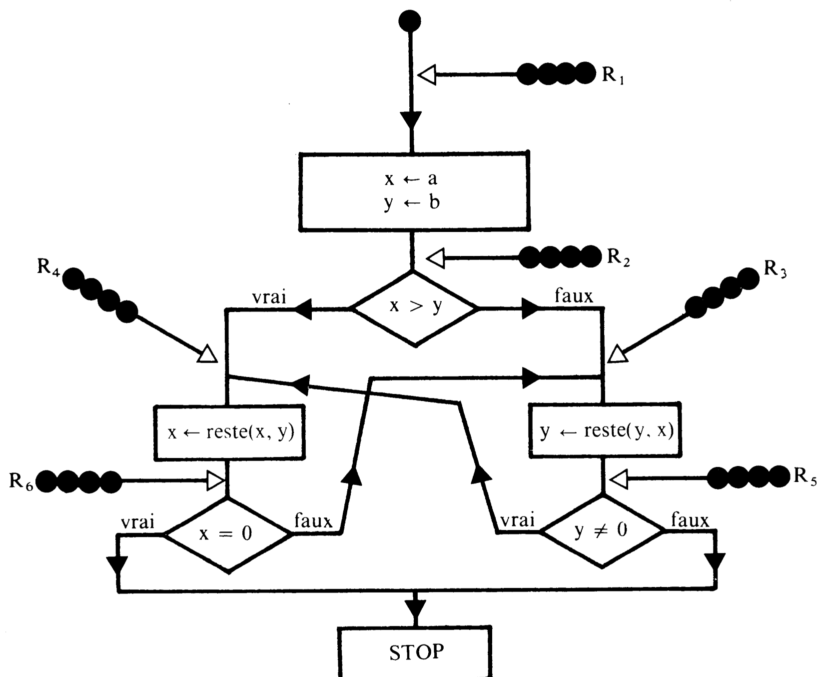


Figure 23 Un programme calculant le pgcd.

La fonction reste (z, t) donne le reste de la division euclidienne de z par t. Il est facile de voir que ce schéma calcule le pgcd de a et b, le résultat se trouvant soit dans x, soit dans y.

$(x, y)\, R_1(x', y') = (x = a \text{ \underline{et} } y = b \text{ \underline{et} } (x, y)\, R_2(x', y'))$

$(x, y)\, R_2(x', y') = ((x > y \text{ \underline{et} } (x, y)\, R_4(x', y')) \text{ \underline{ou} } (x \leqslant y \text{ \underline{et} } (x, y)\, R_3(x', y')))$

$(x, y)\, R_3(x', y') = (x, \text{reste}(y, x))\, R_5(x', y')$

$(x, y)\, R_4(x', y') = (\text{reste}(x, y), y)\, R_6(x', y')$

$(x, y)\, R_5(x', y') = ((y \neq 0 \text{ \underline{et} } (x, y)\, R_4(x', y')) \text{ \underline{ou} } (y = 0 \text{ \underline{et} } x = x' \text{ \underline{et} } y = y'))$

$(x, y)\, R_6(x', y') = ((x \neq 0 \text{ \underline{et} } (x, y)\, R_3(x', y')) \text{ \underline{ou} } (x = 0 \text{ \underline{et} } x = x' \text{ \underline{et} } y = y'))$.

On veut montrer que le schéma de programme π est partiellement correct vis-à-vis de $p = E$ et q tels que

$$q(x, y) = (x = \text{pgcd}(a, b) \ \underline{\text{et}}\ y = 0) \ \underline{\text{ou}}$$
$$(x = 0 \ \underline{\text{et}}\ y = \text{pgcd}(a, b)) .$$

Pour cela, on construit les prédicats p_i $(1 \leqslant i \leqslant 7)$ tels que

$$p_i(x, y) = (R_i \rightarrow q)(x, y)$$

(rappelons que $R_7 = E$). En utilisant les relations ci-dessus, on obtient facilement que les prédicats p_i vérifient les relations

$$p_1(x, y) = (x = a \ \underline{\text{et}}\ y = b \Rightarrow p_2(x, y))$$
$$p_2(x, y) = (x > y \Rightarrow p_4(x, y)) \ \underline{\text{et}}\ (x \leqslant y \Rightarrow p_3(x, y))$$
$$p_3(x, y) = p_5(x, \text{reste}(y, x))$$
$$p_4(x, y) = p_6(\text{reste}(x, y), y)$$
$$p_5(x, y) = (y \neq 0 \Rightarrow p_4(x, y)) \ \underline{\text{et}}\ (y = 0 \Rightarrow p_7(x, y))$$
$$p_6(x, y) = (x \neq 0 \Rightarrow p_3(x, y)) \ \underline{\text{et}}\ (y = 0 \Rightarrow p_7(x, y))$$
$$p_7(x, y) = q(x, y) = (x = \text{pgcd}(a, b) \ \underline{\text{et}}\ y = 0) \ \underline{\text{ou}}$$
$$(x = 0 \ \underline{\text{et}}\ y = \text{pgcd}(a, b)) .$$

Il faudrait maintenant résoudre ce système récursif définissant les prédicats p_i. Pour cela, on peut, par exemple, employer une méthode d'approximations successives. Comme on le voit ici, l'emploi du théorème 6 ne simplifie pas la recherche de l'étiquetage prédicatif. Remarquons que dans cet exemple très simple, il est évident, à toute personne ayant l'habitude des preuves de programme, que l'étiquetage prédicatif

$$p_2(x, y) = p_3(x, y) = p_4(x, y) = p_5(x, y) = p_6(x, y) =$$
$$(\text{pgcd}(x, y) = \text{pgcd}(a, b))$$

permet de prouver la correction partielle de ce programme par rapport aux deux prédicats considérés.

6 PREUVES DE PROGRAMMES RÉCURSIFS

6.1 Introduction

Nous allons maintenant étudier de manière moins formelle qu'au paragraphe précédent les problèmes de preuves concernant des programmes récursifs. Plus précisément, nous nous proposons d'étudier la correction des procédures récursives du type : $f(x\,;y) : \gamma$ où :

— x est la suite des **paramètres d'entrée ;** x est **passé par valeur ;** autrement dit, la valeur de chacun des identificateurs de x est élaborée une fois pour toutes lors de l'appel de la procédure et au retour de la procédure, la valeur de

chacun des paramètres effectifs est inchangée ; on se reportera au chapitre 5 pour une définition plus formelle de ce concept.

— y est la suite des paramètres auxquels seront affectés les résultats au retour de la procédure.

— f est le nom de la procédure.

— γ en est le **corps.**

On convient également qu'il n'y a **pas de variables globales,** ce qui signifie que dans γ tout identificateur qui n'est pas celui d'un paramètre est une variable locale.

6.2 Règle de preuve des appels de procédure

Considérons une procédure *Fact*$(x\,;y)$ calculant la fonction factorielle. *Fact* vérifie la condition suivante : si le paramètre x est un entier positif ou nul, le paramètre y vérifie au retour $y = x\,!$.

Soit maintenant un appel de *Fact* de la forme *Fact*$(3\;;n)$; 3 étant un entier positif, on trouve en substituant 3 à x et n à y

$$n = 3\,!$$

Plus généralement, pour pouvoir prouver la correction d'un appel de procédure, il faut connaître la condition que doivent vérifier les paramètres effectifs d'entrées substitués à x, et la relation vérifiée par les résultats et les entrées ; une procédure est donc caractérisée par un prédicat $\mathrm{p}(x)$ vérifié avant l'appel et par un prédicat de retour $\mathrm{q}(x, y)$. La preuve de l'appel se déroule alors ainsi : soit à prouver dans un programme l'appel $f(exp\,;z)$; si $\mathrm{p}(exp)$ est vérifié, alors après l'appel $\mathrm{q}(exp, z)$ est vérifié.

Ceci se traduit par la règle d'inférence (de HOARE)

$$(\mathrm{APP})\quad \frac{\mathrm{p}(x)\ \{\ f(x\,;y) : \gamma\ \}\ \mathrm{q}(x, y)}{\mathrm{p}(exp)\ \{\ f(exp\,;z)\ \}\ \mathrm{q}(exp, z)}\,.$$

Cette règle n'est valide que si z n'apparaît pas dans l'expression ; en effet, considérons *Fact* : elle est partiellement correcte pour les prédicats $\mathrm{p}(x) = (x = 0)$ et $\mathrm{q}(x, y) = (y = x\,!)$.

Si on appelle $\mathrm{Fact}(x\,;x)$ on n'a évidemment pas $x\,! = x$.

Pour simplifier l'énoncé de la règle, nous supposerons donc que les paramètres effectifs du résultat n'apparaissent pas dans les expressions paramètres effectifs d'entrée. On peut toujours se ramener à ce cas par une affectation préliminaire.

Note : La règle d'inférence donnée ci-dessus ne suffit pas pour rendre compte de l'appel de procédure ; il faut en plus un schéma d'axiomes indiquant que seules les valeurs des paramètres résultats sont modifiées : si r est un prédicat ne contenant pas d'occurrence libre de z, alors $\mathrm{r}\ \{\ \mathrm{f}(exp\,;z)\ \}\ \mathrm{r}$.

6.3 Etude d'un exemple

Reprenons pour cet exemple les notations introduites chapitre 2, paragraphe 2.1

— <u>tableau</u> $t[a : b]$ pour la déclaration du tableau t à une dimension dont l'indice est compris entre les valeurs de a et de b ;

— $t[i : j]$ pour désigner un sous-tableau de t correspondant aux éléments d'indice compris entre i et j ;

— $bs(t)$ et $bi(t)$ pour désigner respectivement les bornes supérieure et inférieure de t.

Nous ne définirons pas formellement la partie de preuve s'attachant aux déclarations.

Considérons la procédure récursive *tri* qui trie un tableau t_1 de façon dichotomique, le résultat étant rangé dans un tableau t_2 :

```
tri(t1 ; t2) :
  i := bi(t1) ; j := bs(t1) ;
  si i < j   alors m := (i + j) ÷ 2 ;
                   début tableau t3[i : m] ; tableau t4[m + 1 : j] ;
                         tri(t1[i : m] ; t3) ;
                         tri(t1[m + 1 : j] ; t4) ;
                         interclassement(t3, t4 ; t2)
                   fin
             sinon t2(i) := t1(i) .
  fsi ;
```

interclassement$(t_3, t_4 ; t_2)$ est une procédure qui interclasse les tableaux t_3 et t_4 et range le résultat dans t_2 ; plus précisément, cette procédure est caractérisée par les prédicats p et q suivants :

$$\mathrm{p}(t_3, t_4) = t_3 \ \underline{\text{trié}} \ \underline{\text{et}} \ t_4 \ \underline{\text{trié}}$$

$$\mathrm{q}(t_3, t_4 ; t_2) = t_2 \ \underline{\text{trié}} \ \underline{\text{et}} \ t_2 \ \underline{\text{équivalent à}} \text{ la réunion de } t_3 \text{ et } t_4 .$$

On souhaite que l'appel de la procédure tri avec l'assertion

$$\mathrm{p}_{tri} = bi(t) \leqslant bs(t)$$

conduise au résultat t_2 vérifiant :

$$\mathrm{q}_{tri} = t_2 \ \underline{\text{trié}} \ \underline{\text{et}} \ t_2 \ \underline{\text{équivalent à}} \ t_1 .$$

Nous allons démontrer cette correction par induction sur le niveau d'imbrication des appels internes. Lors de l'appel de tri(t_1, t_2), deux cas sont à envisager :

— soit $bi(t_1) = bs(t_1)$ et alors il n'y a pas de nouvel appel de *tri* et au retour $t_1 = t_2$, donc q_{tri} est vérifié,

— soit $bi(t_1) < bs(t_1)$, ce qui conduit à un ou plusieurs appels internes. Faisons l'hypothèse (de récurrence) que les appels internes se déroulent correctement, c'est-à-dire que :

- si $i \leqslant m$, alors après exécution de l'appel de $tri(t_1[i : m], t_3)$, t_3 est trié et est équivalent à $t_1[i : m]$,
- si $m + 1 \leqslant j$, alors après l'exécution de l'appel de $tri(t_1[m + 1 : j], t_4)$, t_4 est trié et est équivalent à $t_1[m + 1 : j]$.

Le reste de la preuve est alors celle d'un simple programme non récursif. Comme i est strictement inférieur à j, les conditions

$$i \leqslant m \text{ et } m + 1 \leqslant j$$

sont vérifiées à l'appel de *tri*, et, après le second appel interne de *tri*, nous avons :

t_4 <u>trié</u> <u>et</u> t_3 <u>trié</u> <u>et</u> t_3 <u>équivalent à</u> $t_1[i : m]$ <u>et</u> t_4 <u>équivalent à</u> $t_1[m + 1 : j]$

et donc après interclassement, $q_{tri}(t_1, t_2)$ est vérifié.

Ainsi, par récurrence sur le niveau des appels imbriqués, si l'appel de *tri* ne produit qu'un nombre fini d'appels internes, cette procédure est correcte : ceci est l'énoncé de la **correction partielle.**

Pour prouver complètement la correction de *tri*, il suffit de démontrer que le nombre d'appels emboîtés est fini : c'est un cas particulier de terminaison ; comme il n'y a pas de boucle ici, ce sera la seule possibilité d'engendrer des calculs infinis. Remarquons que l'appel de $tri(t_1 ; t_2)$ ne se fait que si

$$bi(t_1) \leqslant bs(t_1), \text{ donc } bs(t_1) - bi(t_1) \geqslant 0 .$$

Il suffit alors de constater que dans les appels internes cette quantité décroît strictement, il ne peut donc y avoir qu'un nombre fini d'appels imbriqués !

Donc *tri* est correct.

6.4 Correction partielle

Revenons au cas général en formalisant la méthode utilisée (qui n'est qu'un prolongement des techniques présentées aux §§ 2 et 3).

Soit $f(x ; y) : \gamma$ une déclaration de procédure de corps γ. Nous voulons démontrer que si $\mathrm{p}(x)$ est vérifié à l'entrée de la procédure, alors $\mathrm{q}(x, y)$ est vérifié à la sortie, ce que nous noterons

$$\mathrm{p}(x) \{ f(x ; y) : \gamma \} \mathrm{q}(x, y) \text{ (correction de la procédure) .}$$

Par récurrence sur le niveau d'imbrication, on peut justifier le théorème suivant.

Théorème 7 : Si l'on peut démontrer la correction du corps de procédure γ relativement au prédicat d'entrée $\mathrm{p}(x)$ et au prédicat de sortie $\mathrm{q}(x, y)$ sous l'hypothèse de la correction par rapport à p et q de tous les appels internes, alors

$$\mathrm{p}(x) \{ f(x ; y) : \gamma \} \mathrm{q}(x, y) . \qquad \square$$

Ce théorème peut être présenté avec le formalisme du paragraphe 3 par la règle d'inférence :

$$\frac{\mathrm{p}(x) \{ f(x ; y) \} \mathrm{q}(x, y) \mapsto \mathrm{p}(x) \{ \gamma \} \mathrm{q}(x, y)}{\mathrm{p}(x) \{ f(x, y) : \gamma \} \mathrm{q}(x, y)} .$$

Exemple 20

Considérons la déclaration de procédure suivante (fonction 91 de Mac Carthy) :

$F91(x\,;y)$: <u>*si*</u> $x > 100$ <u>*alors*</u> $y := x - 10$ <u>*sinon*</u> $F91(x + 11\,;z)\,;F91(z\,;y)$ <u>*fsi*</u>.

Démontrons la correction de cette procédure relativement aux prédicats :

$$\mathrm{p}(x) \quad = x \geqslant 0$$
$$\mathrm{q}(x, y) = (x > 100 \Rightarrow y = x - 10)\ \underline{\mathrm{et}}\ (x \leqslant 100 \Rightarrow y = 91)\,.$$

Dans le cas où $x > 100$, le résultat est immédiat.

Dans le cas $0 \leqslant x \leqslant 100$, supposons la correction des appels internes :

i) $x + 11 \geqslant 0\ \{\ F91(x + 11\,;z)\ \}\ (x > 89 \Rightarrow z = x + 1)$ <u>et</u> $(x \leqslant 89 \Rightarrow z = 91)$

ii) $z \geqslant 0\ \{\ F91(z\,;y)\ \}\ (z > 100 \Rightarrow y = z - 10)$ <u>et</u> $(z \leqslant 100 \Rightarrow y = 91)$.

Deux cas apparaissent après *i)*.

1) $x > 89$ donc $z = x + 1$ soit donc, d'après *ii)*, deux sous-cas :
   - 11) $x + 1 > 100$ c'est-à-dire $x > 99$ donc $x = 100$
     et donc $y = z - 10 = x + 1 - 10 = 91$
     donc $\mathrm{q}(x, y)$ est vérifié.
   - 12) $x + 1 \leqslant 100$ c'est-à-dire $x \leqslant 99$
     donc $y = 91$ et ainsi $\mathrm{q}(x, y)$ est vérifié.

2) $x \leqslant 89$ donc avec *i)* on en déduit $z = 91$
   donc avec *ii)* $y = 91$
   $\mathrm{q}(x, y)$ est vérifié.

La fonction $F91$ est donc partiellement correcte. □

Exercice 28

Prouver la correction partielle des deux procédures :

1) $F(x : y) : F(x - 1 : z)\,; y := 2 * z$
   avec p = <u>vrai</u>
   q = $(y = 0)$.

2) $PGCD(a, b\,;r)$: <u>*si*</u> $b = 0$ <u>*alors*</u> $r := a$
   <u>*sinon*</u> $PGCD(b, reste(a, b)\,;r)$ <u>*fsi*</u>

   où $reste(a, b)$ est le reste de la division entière de a par b
   avec $\mathrm{p}(a, b) = a \geqslant 0$ <u>et</u> $b \geqslant 0$
   avec $\mathrm{q}(a, b, r) = r$ est le *pgcd* de a et b. □

Exercice 29

Justifier la règle de preuve des procédures récursives par la règle d'induction de Scott (chapitre 2, § 4). □

6.5 Terminaison

L'appel d'une procédure récursive peut conduire à un calcul infini, parce qu'un cycle de son corps ne se termine pas ou parce que l'enchaînement des appels imbriqués est infini. Nous ne nous intéresserons ici qu'au second cas, le premier ayant été traité dans les paragraphes précédents.

De manière analogue au cas des programmes itératifs pour prouver la terminaison de la procédure récursive $f(x\,;y):\gamma$, on peut suivre la démarche suivante :

— choix d'un ensemble F muni d'un bel ordre $\preccurlyeq$,

— choix d'un prédicat p tel que p(x) soit vérifié pour l'appel initial $f(x\,;y)$ et tel que pour tout appel $f(u\,;v)$ de f dans le corps de procédure γ, p(u) soit vérifié,

— choix d'une fonction u qui, si p(x) est vérifié, associe à x un élément de F,

— vérification de l'inégalité u(t) < u(x) pour tous les appels $f(t\,;z)$ inclus dans γ.

Exemple 21 : Considérons la fonction d'Ackerman :

$$\begin{array}{l} A(x,y\,;z) : \underline{si}\ x = 0\ \underline{alors}\ z := y+1 \\ \qquad\qquad \underline{sinon}\ \underline{si}\ y = 0\ \underline{alors}\ A(x-1,1\,;z) \\ \qquad\qquad\qquad\qquad \underline{sinon}\ A(x,y-1\,;t)\,;A(x-1,t\,;z)\ \underline{fsi} \\ \qquad\qquad \underline{fsi} \end{array}$$

Posons ici $p(x,y) = (x \geqslant 0\ \underline{et}\ y \geqslant 0)$ et considérons l'ordre lexicographique sur l'ensemble $\mathbb{N} \times \mathbb{N}$: c'est un bel ordre.

Il y a trois appels internes :

$$A(x-1,1\,;z)$$
$$A(x,y-1\,;t)$$
$$A(x-1,t\,;z)\,.$$

Nous remarquons que dans les trois cas :

$$(x,y) \succ (x-1,1)$$
$$(x,y) \succ (x,y-1)$$
$$(x,y) \succ (x-1,t)\,.$$

Donc cette procédure termine. □

Exercice 30

Si l'on permute les deux paramètres d'entrée du second appel d'Ackerman, on obtient :

$$\begin{array}{l} A'(x,y\,;z) : \underline{si}\ x = 0\ \underline{alors}\ z := y+1 \\ \qquad\qquad \underline{sinon}\ \underline{si}\ y = 0\ \underline{alors}\ A'(x-1,1\,;z) \\ \qquad\qquad \underline{sinon}\ A'(y-1,x\,;t)\,;A'(x-1,t\,;z)\ \underline{fsi} \\ \qquad\qquad \underline{fsi}\,. \end{array}$$

A' se termine-t-elle avec les conditions d'entrée $x \geqslant 0$ et $y \geqslant 0$? □

Exercice 31 : Le problème des n reines (§ 4.2c).

Il s'agit de placer sur un échiquier de côté n, n reines de façon à ce qu'aucune reine ne soit en prise par rapport à une autre (c'est-à-dire ni sur une même ligne ni sur une même colonne, ni sur une même diagonale). Une procédure construisant un ensemble E de solutions peut s'écrire :

```
reine(échiquier, i, n ; E) :
    si i > n alors E := E ∪ { échiquier }
             sinon pour j de 1 à n faire
                   si la position (i, j) n'est pas en prise avec les
                      positions déjà occupées dans échiquier
                   alors nouveau := ajoutereine(échiquier, i, j) ;
                         reine(nouveau, i + 1, n ; E)

                   fsi fait
    fsi.
```

La fonction *ajoutereine* a pour résultat un nouvel échiquier obtenu à partir de l'échiquier paramètre en ajoutant une reine dans la case (i, j).

Le programme : *lire*(n) ; *échiquier* := *vide* ; E := *vide* ;

reine(*échiquier*, 1, n, E)

doit ranger dans l'ensemble E toutes les solutions.

Ecrire plus précisément la procédure *reine*, prouver sa terminaison, puis sa correction. □

7 PREUVES DE CORRECTION ET TESTS DE VALIDITÉ DES PROGRAMMES

7.1 Inconvénients des preuves a posteriori

Au paragraphe 1 nous avons évoqué l'insuffisance des méthodes empiriques, aussi bien pour construire que pour vérifier les programmes. Mais force est de constater que les méthodes mathématiques présentées dans les paragraphes précédents ne sont pas non plus tout à fait satisfaisantes.

Une première constatation concerne la grande simplicité des exemples que l'on a utilisés pour illustrer les différentes méthodes. Certes, la clarté de la présentation nécessite cette simplicité mais il faut remarquer que la complexité de la preuve croît très vite (plus qu'exponentiellement) avec la taille du programme ce qui limite pratiquement le champ d'application de ces méthodes. En effet une preuve mathématique n'a de sens que dans la mesure où l'on fait confiance aux déductions invoquées et il est bien connu que plus une démonstration faite par un homme est longue, plus elle a de chances d'être fausse. On peut cependant penser que la création de prouveurs automatiques de programmes permettra d'améliorer la fiabilité des démonstrations sans toutefois, dans l'état actuel de nos connaissances, éliminer les erreurs humaines introduites par le biais de l'interaction homme-machine.

D'un autre point de vue nous avons déjà signalé combien la notion de preuve a posteriori est peu satisfaisante puisqu'elle nécessite, à partir d'un texte de programme et de spécifications d'entrée et de sortie, l'invention d'assertions et d'invariants intermédiaires. Cette démarche revient en quelque sorte, à « réinventer le programme ».

Remarquons également qu'une preuve mathématique n'a de sens que relativement à une spécification formelle du problème. Mais peut-on être certain de l'adéquation de la spécification au problème ? Et à ce niveau aucune preuve n'est possible puisque, dans la plupart des cas l'énoncé du problème est informel. Notons que cette question ne se pose pas avec autant d'acuité dans les vérifications par jeux d'essai puisque alors il suffit de constater que les résultats satisfont l'énoncé.

Signalons enfin que les méthodes mathématiques rendent difficilement compte d'erreurs telles que débordement mémoire, erreur d'arrondi, etc...

7.2 Vers la conception de programmes fiables

La mise au point de méthodologies de programmation (§ 4) permettra certainement de répondre aux deux premières objections précédentes : la construction d'un programme par raffinements successifs permet de prouver à chaque étape la correction du fragment de programme obtenu. La preuve est ainsi divisée en preuves secondaires plus petites pour lesquelles la mise en œuvre de l'outil mathématique présente moins de difficultés. Cette construction progressive présente un autre avantage : la correction d'une erreur ne nécessite pas la refonte complète du programme mais seulement d'une partie bien délimitée.

Même dans le cadre d'une construction méthodique il est difficile d'obtenir une preuve parfaite en l'absence d'un langage de spécifications précis et facilement utilisable. En effet on ne peut pas demander à un programmeur d'utiliser le langage de la logique des prédicats pour énoncer les problèmes qu'il résout. En fait, lorsqu'elles sont exprimées, les spécifications d'un problème sont habituellement décrites dans une langue naturelle un peu formalisée ce qui diminue singulièrement la confiance que l'on peut avoir dans les constructions et les preuves mises en œuvre.

Comme il est difficile de prouver parfaitement un programme, les tests par jeux d'essais restent, malgré leur caractère incomplet, l'un des seuls moyens permettant d'accorder une certaine confiance à un programme. Ainsi preuves imparfaites et jeux d'essais se complètent et permettent d'obtenir un produit dont la fiabilité peut être précisée par le fournisseur, tout comme, dans l'industrie électronique, on précise la fiabilité des composants fabriqués. Bien évidemment cette fiabilité dépendra beaucoup des moyens mis en œuvre, c'est-à-dire finalement du budget accordé à la construction du programme.

Terminons ce chapitre en indiquant une technique de tests qui permet d'augmenter notablement la validité d'un programme.

7.3 Techniques d'évaluation symbolique

L'idée de base de l'évaluation symbolique est très simple : plutôt que d'exécuter un programme sur le plus grand nombre possible de données différentes

de manière à parcourir le plus grand nombre possible de chemins dans l'organigramme du programme, on exécute « symboliquement » le programme sur une classe de données. Ainsi on effectue des opérations algébriques plutôt qu'arithmétiques. Explicitons ceci sur le programme suivant :

lire(x) ; *lire*(n) ; $s := 0$; $i := 1$;
<u>*tant que*</u> $i \leqslant n$ <u>*faire*</u> $s := s + x ** i$; $i := i + 1$ <u>*fait*</u> ;
imprimer(s) .

Plutôt que de donner des valeurs numériques à x et n on leur associe des « symboles » α, β puis on « exécute symboliquement » le programme pour ces données. Il est clair que le déroulement de cette exécution nécessite des hypothèses sur les données : Après avoir initialisé s et i il est nécessaire de faire une hypothèse sur la valeur du test $i \leqslant n$. Selon la réponse on imprime s ou on exécute une nouvelle fois le corps du cycle puis on se pose à nouveau la question et ainsi de suite.

Finalement, selon les hypothèses sur β, on obtient différents résultats possibles :

$\beta < 1 : s = 0$
$\beta = 1 : s = x$
$\beta = 2 : s = x + x^2$
etc...

Cette démarche n'a bien entendu d'intérêt que dans la mesure où l'on dispose d'un interprète capable d'effectuer le calcul symbolique, c'est-à-dire le calcul algébrique formel : manipulation de formules, simplifications, mises en facteur, etc... On peut souhaiter également que cet interprète soit interactif pour permettre à l'utilisateur « d'explorer » à son gré le programme. En particulier, une telle exploration systématique nécessite la sauvegarde de l'état mémoire symbolique avant d'effectuer un test de manière à permettre l'exploration de la deuxième branche du test après celle de la première. Il est également nécessaire de construire pas à pas les conditions de cheminement dans le programme de façon à connaître à tout instant le maximum d'information sur les variables (et de retenir en particulier les hypothèses sur les données).

Le lecteur attentif notera que, Monsieur JOURDAIN de l'informatique, nous avons fait de l'évaluation symbolique au paragraphe 2.3c lorsque nous avons décrit une méthode pour évaluer le résultat d'un calcul le long d'un chemin ainsi que la condition de cheminement.

Bien que naturelle, la mise en œuvre de l'évaluation symbolique demande un travail important; citons deux exemples connus : EFFIGY [KIN, 76] et SELECT [BOY, 75]. Remarquons que cette technique présente une certaine parenté avec la méthode de preuve par assertions : alors qu'elle représente une classe de valeurs par une formule, la preuve par assertions représente cette classe par un prédicat et dans l'un et l'autre cas l'exécution d'une instruction provoque une transformation de cette représentation.

A mi-chemin entre preuve mathématique et jeux d'essais, l'évaluation symbolique apparaît comme un outil intéressant pour la mise au point.

8 COMMENTAIRES BIBLIOGRAPHIQUES

Le livre de MANNA [MAN, 74a] contient une présentation détaillée des outils de preuves introduits par FLOYD et HOARE ; ce livre propose également de nombreux exemples et exercices. On y trouve en particulier une présentation axiomatique de la correction totale qui étend le système formel proposé par HOARE en incluant des preuves de terminaison.

FLOYD [FLO, 67] et HOARE [HOA, 69] sont à l'origine des méthodes de preuves proposées ici aux paragraphes 2 et 3. Dans [MAN, 74b], on trouve des développements relatifs à la correction totale.

L'approche du paragraphe 4 qui consiste, par transformations d'énoncés, à construire des programmes corrects en évitant une preuve a posteriori a été beaucoup développée : sur un plan théorique, il faudrait citer de nombreuses études sur la synthèse de programmes ; indiquons seulement ici un des articles les plus classiques sur ce sujet [MAN, 71] ; sur un plan pratique ce point de vue a fait évoluer de façon considérable la pédagogie de la programmation et la conception des langages : programmation structurée, langages d'affectation unique, ... Le livre d'ARSAC [ARS, 77] est un excellent exemple de cette rénovation directement induite par les recherches les plus récentes.

Le calcul relationnel que nous avons introduit au paragraphe 5 donne un outil très efficace de justification théorique des méthodes de preuve. Le lecteur intéressé peut se reporter à l'article de de BAKKER et MERTENS [BAK, 75].

HOARE [HOA, 70] traite de l'axiomatisation des procédures et complète le point de vue présenté au paragraphe 6. GORELICK [GORE, 75] propose un système plus général ; il montre que ce système est complet.

Dans [BOY, 75] et [KIN, 76] sont présentés deux systèmes opérationnels, utilisant la technique de l'évaluation symbolique évoquée au paragraphe 7. Un autre système d'évaluation symbolique DISSECT est décrit dans [HOW, 77] ; on y étudie les limites de l'évaluation symbolique en classifiant les types d'erreurs qui peuvent être détectées par cette méthode.

Les comptes rendus de « international conference on reliable software » [SIG, 75] contiennent de nombreux articles sur la vérification des programmes.

LEROY, dans [LER, 75], étudie les méthodes mathématiques de preuves de programmes, de façon très intéressante, avec l'œil critique du praticien.

Signalons enfin que certains systèmes de preuves ont été implémentés en utilisant les idées développées dans ce chapitre ; on peut, par exemple, consulter à ce sujet [IGA, 73].

9 SOLUTION DES EXERCICES

Exercice 1

Rappelons qu'au chapitre 3, paragraphe 2.3, nous avons appelé calcul d'un programme π une suite :

$$(\sigma_1, x^1), \dots, (\sigma_m, x^m)$$

où $\sigma_1, \ldots, \sigma_m$ sont des numéros d'instruction et $x^1, \ldots, x^m$ des éléments de D^n vérifiant :

σ_1 est le numéro 1 de la première instruction
x^1 est l'initialisation notée ici a

pour $i = 1, \ldots, m - 1$:

$(\sigma_i, x^i) \xrightarrow{I} (\sigma_{i+1}, x^{i+1})$.

Convenons qu'un chemin α est représenté par une suite $\sigma_r \ldots \sigma_m$ de numéros et notons $y = (a, x)$ $(y \in D^{n+p})$

$C(a, x\,;\alpha) \Leftrightarrow$ il existe un calcul $(\sigma_r, y^r), \ldots, (\sigma_m, y^m)$
tel que :
les n premières composantes de y^r forment a
$\alpha = \sigma_r\,\sigma_{r+1} \ldots \sigma_m$
les p dernières composantes de y^r forment x .

Dans ces conditions nous avons vu (chapitre 3, § 2.3) qu'un tel calcul est unique et on peut définir $\alpha(x)$ par :

$$\alpha(x) = x^m$$

où x^m désigne les p dernières composantes de y^m.

Exercice 2

Indiquons sur l'organigramme (figure 24) les points de coupure et les prédicats.

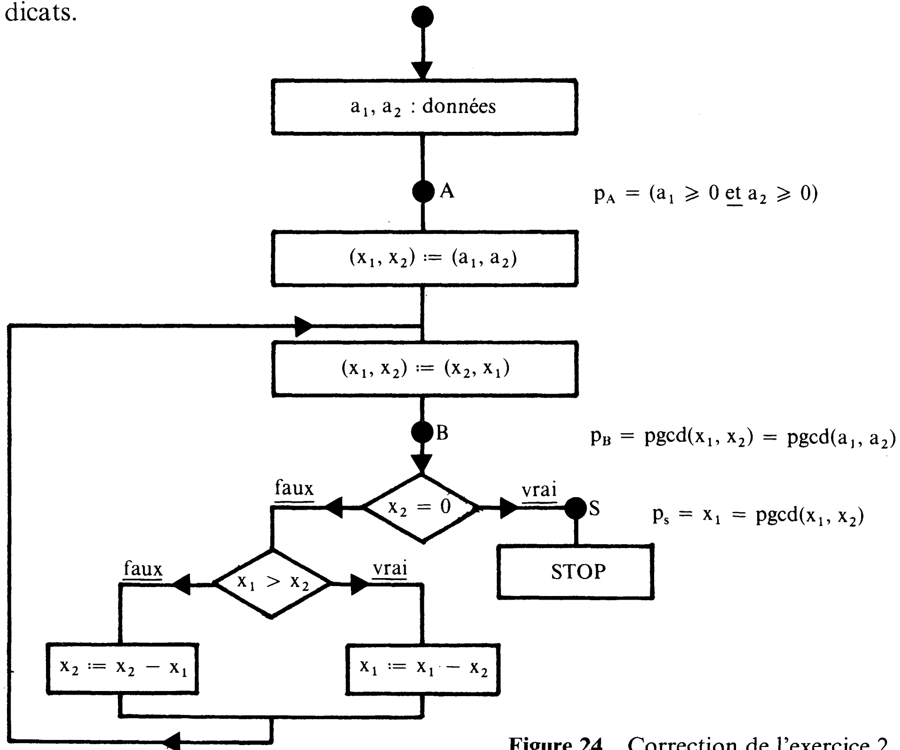


Figure 24 Correction de l'exercice 2.

Exercice 3

Avec les notations de la figure 2, nous notons à chaque étape la nouvelle condition et la nouvelle valeur de x :

— pour le chemin α de A vers B (figure 25),

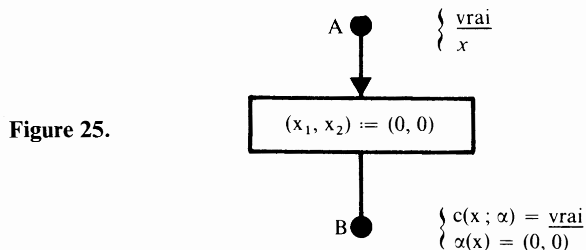


Figure 25.

— pour le circuit de B vers B (figure 26),

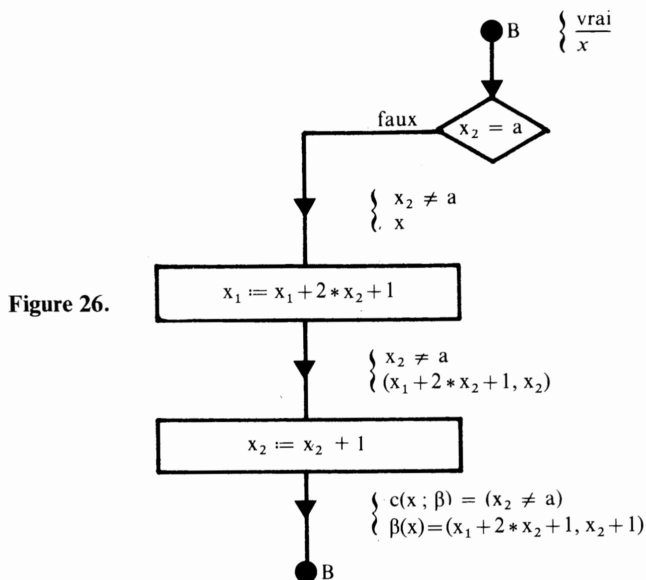


Figure 26.

— pour le chemin γ de B vers C (figure 27).

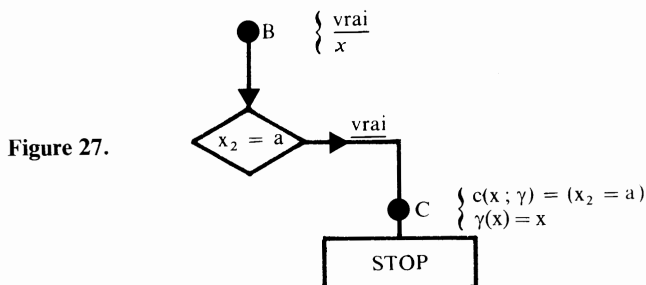


Figure 27.

Exercice 4

Un tableau t[1 : n] est une permutation d'un tableau b[1 : n] si et seulement s'il existe une bijection de [1 : n] vers [1 : n] telle que :

$$\forall i \in [1 : n] \quad t[i] = b[\varphi(i)] .$$

Remarquons que cette définition nous fait sortir du calcul des prédicats du premier ordre et donc que les preuves de programmes faisant intervenir des tableaux seront plus complexes que celles portant sur des variables simples.

Exercice 5

Indiquons les prédicats à vérifier aux points de coupure :

$$P_B = P_C = (t[1 : i] \leqslant b[1] \leqslant t[j + 1 : n] \ \underline{\text{et}}\ t\ \underline{\text{perm}}\ b\ \underline{\text{et}}\ i \leqslant j + 1)$$
$$P_D = \quad (t[j] \leqslant b[1]\ \underline{\text{et}}\ t[1 : i - 1] \leqslant b[1] \leqslant t[j + 1 : n]\ \underline{\text{et}}\ t\ \underline{\text{perm}}\ b)$$

Exercice 6

Indiquons simplement les prédicats qu'on peut placer aux points de coupure (figure 28).

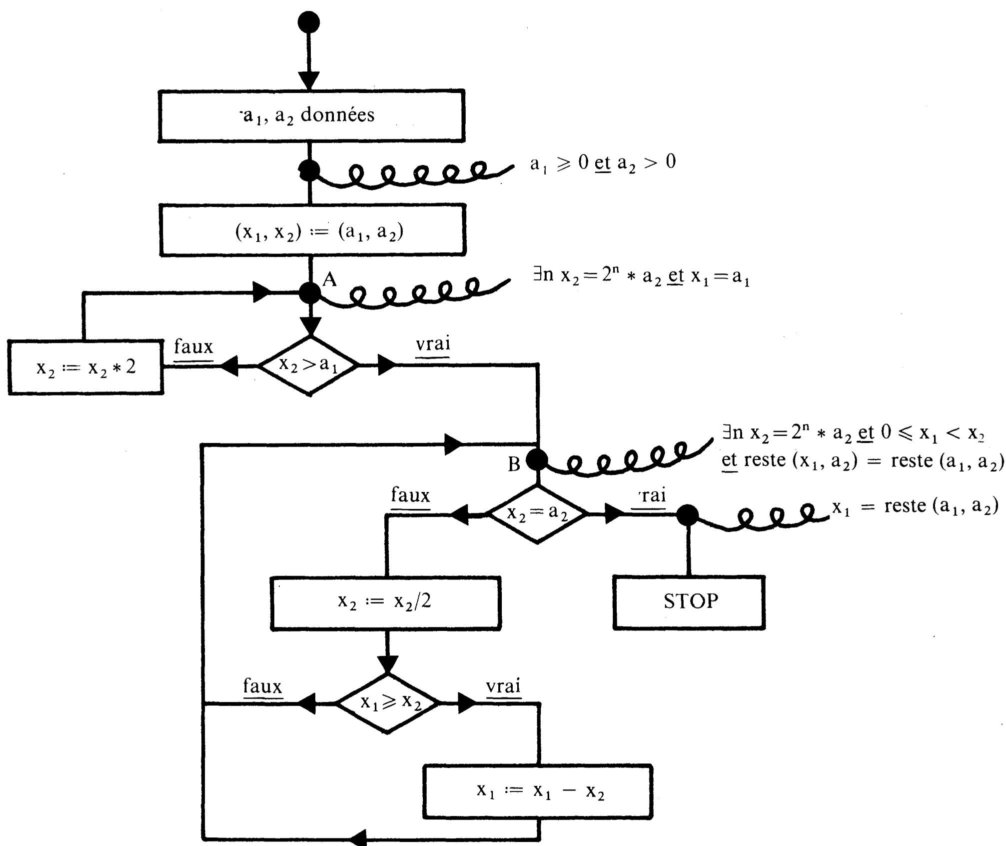


Figure 28.

Exercice 7

Indiquons les prédicats qu'on peut placer aux points de coupures (figure 29).

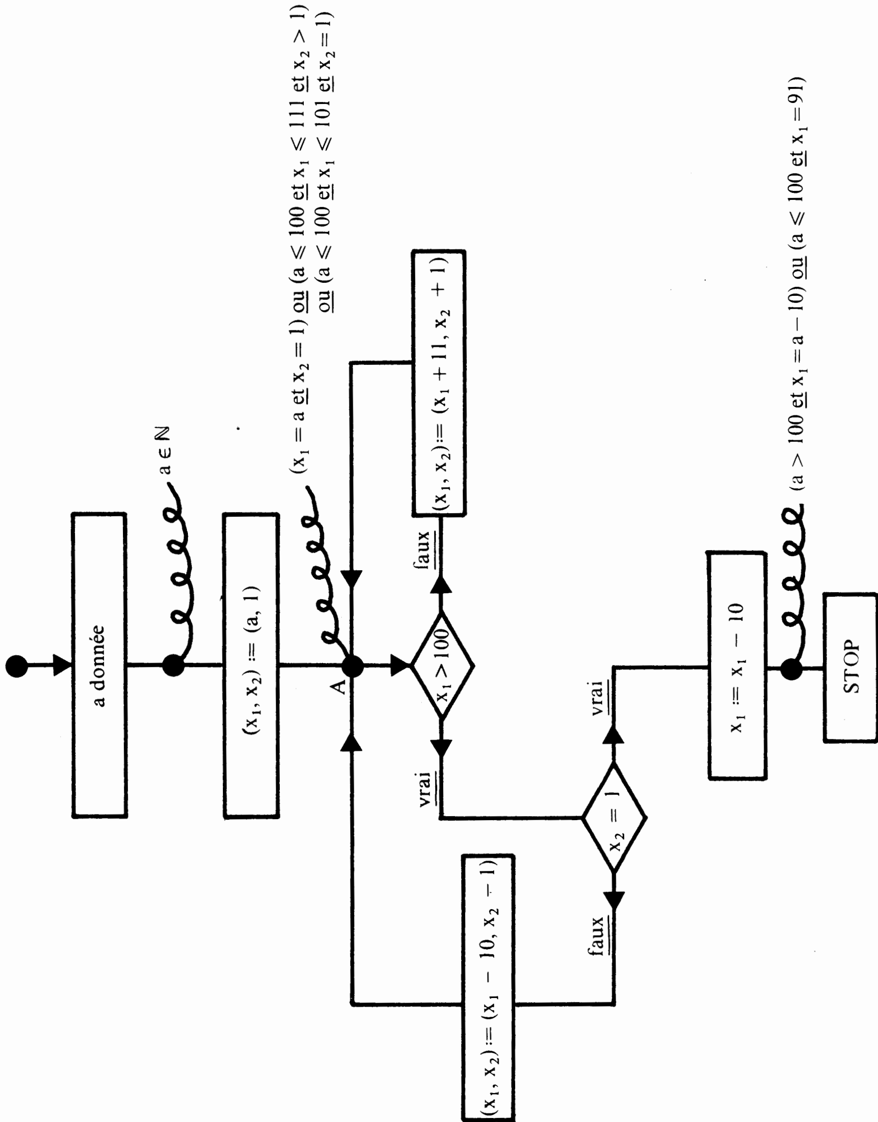


Figure 29.

Exercice 8

Reprendre les assertions utilisées pour la correction partielle (§ 2.3 exemple 7, figure 6) et les fonctions $f_B(x_1, x_2) = f_C(x_1, x_2) = f_D(x_1, x_2) = x_1 + x_2$ (à valeurs dans $\mathbb{N}$).

Exercice 9

On peut prouver séparément la terminaison des boucles de point de coupure A et B en reprenant les assertions de l'exercice 6 et les fonctions :

$$f_A(x_1, x_2) = a_1 - x_2$$
$$f_B(x_1, x_2) = x_2 - a_2$$

Exercice 10

Voir corrigé de l'exercice 7. Prendre au point de coupure A la fonction

$$f_A(x_1, x_2) = -2x_1 + 21x_2 + 2|a| + 222$$

la translation $2|a| + 222$ assure que $f_A(x_1, x_2) \in \mathbb{N}$ (qui, muni de l'ordre usuel, est bien fondé) les coefficients -2 et 21 peuvent être retrouvés en cherchant une fonction f_A de la forme $a_1 x_1 + a_2 x_2$.

Exercice 11

On peut prouver la correction partielle de l'organigramme en utilisant les prédicats proposés sur la figure 30.

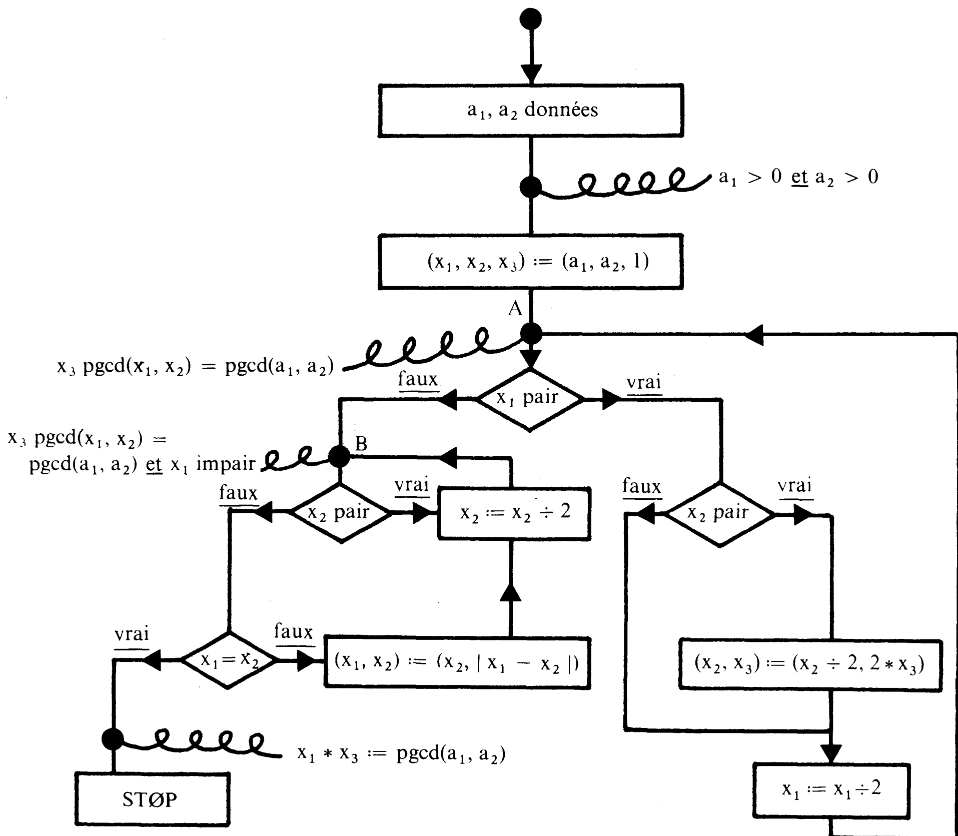


Figure 30.

$f_A(x_1, x_2, x_3) = x_1$

$f_B(x_1, x_2, x_3) = x_1 + 2\,x_2$ (ajouter $x_1 > 0$ et $x_2 > 0$ aux prédicats en A et B).

Exercice 12

On peut prouver la correction partielle en utilisant les prédicats étiquetant l'organigramme comme sur la figure 31.

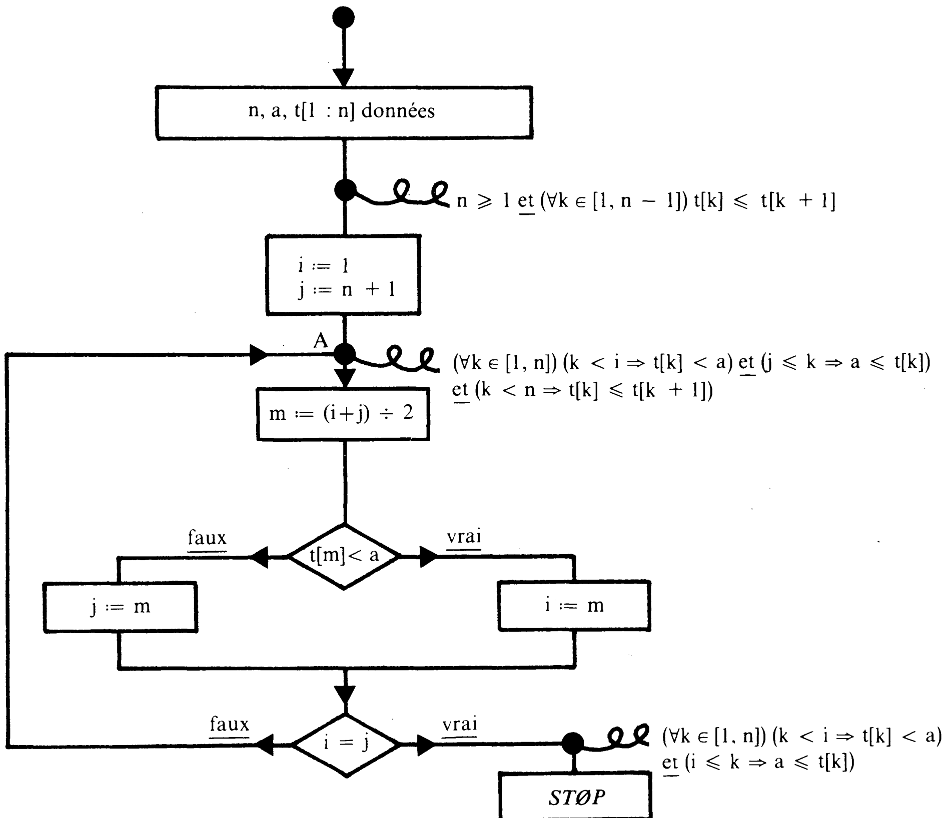


Figure 31.

La terminaison de cet organigramme n'est pas assurée : en effet si, lors des calculs, on a au point A :

$$j = i + 1 \text{ et } t[i] < a$$

en parcourant la boucle, on trouve $m := (i + j) \div 2$ $(= i)$ puis, comme le test est vérifié, $i := m$ $(= i)$; on se retrouve donc au point A sans avoir modifié les variables et le calcul ne s'arrête pas. Il aurait fallu faire progresser i de 1. Changeons l'organigramme en remplaçant $i := m$ par $i := m + 1$. La

correction partielle est toujours vérifiée avec le même prédicat en A. Pour vérifier la terminaison, placer la fonction :

$$f_A(i, j) = j - i .$$

Exercice 13

1) Pour faire la preuve le long des chemins directs de B vers B et de B vers A on peut utiliser le lemme suivant pour y, t entiers :

$$(\mathrm{pred}(y) = \mathrm{s}(\mathrm{pred}(z)) \ \underline{et}\ y - \mathrm{pred}(y) = z - \mathrm{pred}(z)) \Rightarrow y + 1 \notin S$$

[en effet si la partie gauche de l'implication est vraie on a les inégalités :

$$y + 1 - \mathrm{pred}(y) \leqslant z + 1 - \mathrm{pred}(z) \leqslant \mathrm{s}(\mathrm{pred}(z) - \mathrm{pred}(z) < \mathrm{s}(\mathrm{pred}(y)) - \\ - \mathrm{pred}(y)] .$$

2) A partir des prédicats P_A et P_B et du lemme précédent, on constate qu'à l'entrée de chacun des deux tests portant sur $P(y)$ on a pour le premier :

$$y \in S$$

et pour le second

$$y \notin S .$$

Donc, si p est défini par :

$$\mathrm{p}(n) = \underline{si}\ n \in S\ \underline{alors}\ \underline{faux}\ \underline{sinon}\ \underline{vrai} ,$$

on évite toujours la branche conduisant à C : il n'y a pas terminaison. Réciproquement supposons que le programme ne se termine pas, comme les valeurs de y sont strictement croissantes, pour tout $n \in \mathbb{N}$ on peut trouver un calcul tel qu'au point B, p_B soit vérifié avec $n \leqslant y$, d'où :

$$\mathrm{p}(n) = \underline{faux} \Leftrightarrow n \in S .$$

3) L'étude que nous venons de faire peut être reprise pour le schéma de programme de l'exemple 5 du chapitre 3. Dans ce schéma de programme, l'addition de 1 à y ou z est remplacée par la composition du symbole fonctionnel f ; la condition de non-terminaison s'écrit alors :

$$\forall n \in \mathbb{N} \quad \mathrm{p}(f^n(x)) = \underline{faux} \Leftrightarrow n \in \mathrm{S} .$$

Exercice 14

L'auteur se sent incapable de fournir une correction.

Exercice 15

Si l'on a initialement $a[1] = a[2] = 2$, on obtient, après l'affectation $a[a[2]] := 1$, la situation $a[1] = 2, a[2] = 1$. Le prédicat $a[a[2]] = 1$ n'est donc pas vérifié, cela met en défaut l'axiome d'affectation

$$1 = 1 \{ a[a[2]] := 1\} a[a[2]] = 1 .$$

Exercice 16

Notons π le morceau de programme :

$$x_1 := x_1 + 2 * x_2 + 1 \,;\, x_2 := x_2 + 1 \,.$$

On peut montrer :

$$x_1 = x_2^2 \; \underline{\text{et}} \; x_2 \leqslant a \; \underline{\text{et}} \; x_2 < a \; \{ \, \pi \, \} \; x_1 = x_2^2 \; \underline{\text{et}} \; x_2 \leqslant a$$

(règles sur l'affectation et la séquence).

On en déduit (grâce à la règle sur l'itération)

$$x_1 = x_2^2 \; \underline{\text{et}} \; x_2 \leqslant a \; \{ \; \underline{\textit{tant que}} \; x_2 < a \; \underline{\textit{faire}} \; \pi \; \underline{\textit{fait}} \; \} \; x_1 = x_2^2 \; \underline{\text{et}} \; x_2 \leqslant a \; \underline{\text{et}} \; a \leqslant x_2$$

d'où :

$$x_1 = x_2^2 \; \underline{\text{et}} \; x_2 \leqslant a \; \{ \; \underline{\textit{tant que}} \; x_2 < a \; \underline{\textit{faire}} \; \pi \; \underline{\textit{fait}} \; \} \; x_1 = a^2$$

et finalement :

$$a \geqslant 0 \; \{ \; x := (0, 0) \,;\, \underline{\textit{tant que}} \; x_2 < a \; \underline{\textit{faire}} \; \pi \; \textit{fait} \; \} \; x_1 = a^2 \,.$$

Exercice 17

Pour abréger les notations, nous notons $(x_1, x_2) \sim (a_1, a_2)$ à la place de

$$pgcd(x_1, x_2) = pgcd(a_1, a_2) \,.$$

On prouve successivement

(1) $(x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_1 < x_2 \; \{ \; x_2 := x_2 - x_1 \; \} \; (x_1, x_2) \sim (a_1, a_2)$ (AFF)

(2) $(x_1, x_2) \sim (a_1, a_2) \; \{ \, \gamma \, \} \; (x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_2 \leqslant x_1$ (ITE)

(3) $(x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_2 \leqslant x_1 \; \underline{\text{et}} \; x_2 < x_1 \; \{ \; x_1 := x_2 - x_1 \,;\, \gamma \; \} \; (x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_2 \leqslant x_1$ (SEQ)

(4) $(x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_2 \leqslant x_1 \; \{ \, \beta \, \} \; (x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_2 \leqslant x_1 \; \underline{\text{et}} \; x_1 \leqslant x_2$

(5) $(x_1, x_2) \sim (a_1, a_2) \; \{ \, \alpha \, \} \; (x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_2 \leqslant x_1 \,.$

(Pour déduire (5) de la règle (CND) on convient ici que $\underline{si}$ t $\underline{alors}$ $x := y$ $\underline{fsi}$ est une abréviation de $\underline{si}$ t $\underline{alors}$ $x := y$ $\underline{sinon}$ $x := x$ $\underline{fsi}$).

A partir de (5) on prouve :

(6) $\quad a_1 > 0 \; \underline{\text{et}} \; a_2 > 0 \; \{ \; x := a \,;\, \alpha \; \} \; (x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_2 \leqslant x_1 \,.$

Puis à partir de (6) et (4)

(7) $\quad a_1 > 0 \; \underline{\text{et}} \; a_2 > 0 \; \{ \, \pi \, \} \; (x_1, x_2) \sim (a_1, a_2) \; \underline{\text{et}} \; x_1 = x_2 \,.$

D'où

$$a_1 > 0 \; \underline{\text{et}} \; a_2 > 0 \; \{ \, \pi \, \} \; x_1 = x_2 = pgcd(a_1, a_2) \,.$$

Exercice 18

Il s'agit de montrer :

$$|\alpha| \geqslant 0 \text{ et } |\beta| \geqslant 0 \{ \rho \} F = \{ i \mid \beta \langle i+1, |\alpha| \rangle = \alpha \}$$

indiquons sommairement les principales étapes ; abrégeons les prédicats en posant :

$$\text{inv}(j) = (F = \{ i \mid \alpha = \beta \langle i+1, |\alpha| \rangle \ \underline{\text{et}}\ 0 \leqslant i \leqslant j \})$$

$$\text{p}(k,j) = (i < k \Rightarrow \alpha \langle 1, i \rangle = \beta \langle j+1, i \rangle)$$

(1) $\text{inv}(j-1)\ \underline{\text{et}}\ \text{p}(k,j)\ \underline{\text{et}}\ k \leqslant |\alpha| \text{ et } a_k = b_{j+k} \{ k := k+1 \}$
$\text{inv}(j-1)\ \underline{\text{et}}\ \text{p}(k,j)$

(2) $\text{inv}(j-1)\ \underline{\text{et}}\ \text{p}(k,j) \{ \underline{\textit{tant que}}\ k \leqslant |\alpha|\ \underline{\textit{et}}\ a_k = b_{j+k}$
$\underline{\textit{faire}}\ k := k+1\ \underline{\textit{fait}} \}\ \text{inv}(j-1)\ \underline{\text{et}}\ \text{p}(k,j)\ \underline{\text{et}}\ k > |\alpha|\ \underline{\text{ou}}\ a_k \neq b_{j+k}$

(3) $\text{inv}(j-1) \{ \theta \}\ \text{inv}(j-1)\ \underline{\text{et}}\ \text{p}(k,j)\ \underline{\text{et}}\ (k > |\alpha|\ \underline{\text{ou}}\ a_k \neq b_{j+k})$

(4) $\text{inv}(j-1)\ \underline{\text{et}}\ \text{p}(k,j)\ \underline{\text{et}}\ k > |\alpha| \{ F := F \cup \{ j \} \}\ \text{inv}(j)$

(5) $(\text{inv}(j-1) \text{ et } \text{p}(k,j)\ \underline{\text{et}}\ a_k \neq b_{j+k}\ \underline{\text{et}}\ k \leqslant |\alpha|) \Rightarrow \text{inv}(j)$.

De (4) et (5) on déduit (règle des conditionnelles étendue au cas d'absence de la partie *sinon*) :

(6) $\text{inv}(j-1)\ \underline{\text{et}}\ \text{p}(k,j)\ \underline{\text{et}}\ (k > |\alpha|\ \underline{\text{ou}}\ a_k \neq b_{j+k})$
$\{ \underline{\textit{si}}\ k > |\alpha|\ \underline{\textit{alors}}\ F := F \cup \{ j \}\ \underline{\textit{fsi}} \}\ \text{inv}(j)$.

De (3) et (6) on déduit :

(7) $\text{inv}(j-1) \{ \theta ; \underline{\textit{si}}\ k > |\alpha|\ \underline{\textit{alors}}\ F := F \cup \{ j \}\ \underline{\textit{fsi}} \}\ \text{inv}(j)$

(8) $\text{inv}(j) \{ j := j+1 \}\ \text{inv}(j-1)$

(9) $\text{inv}(j)\ \underline{\text{et}}\ j \leqslant |\beta| - |\alpha|$
$\{ j := j+1 ; \theta ; \underline{\textit{si}}\ k > |\alpha|\ \underline{\textit{alors}}\ F := F \cup \{ j \}\ \underline{\textit{fsi}} \}\ \text{inv}(j)$

(10) $\text{inv}(j) \{ \chi \}\ \text{inv}(j)\ \underline{\text{et}}\ j > |\beta| - |\alpha|$

(11) $|\alpha| \geqslant 0\ \underline{\text{et}}\ |\beta| \geqslant 0 \{ \rho \}$
$F = \{ i \mid \alpha = \beta \langle i+1, |\alpha| \rangle\ \underline{\text{et}}\ 0 \leqslant i \leqslant j \}\ \underline{\text{et}}\ j + |\alpha| > |\beta|$

(12) $|\alpha| \geqslant 0\ \underline{\text{et}}\ |\beta| \geqslant 0 \{ \rho \} F = \{ i \mid \alpha = \beta \langle i+1, |\alpha| \rangle \}$.

Exercice 19

L'idée consiste à choisir ici $u(x) = x_1 + x_2$. Adoptons comme dans la correction de l'exercice 17, la notation $(x_1, x_2) \sim (a_1, a_2)$.

Indiquons quelques étapes de la preuve

(1) $\begin{cases} (x_1, x_2) \sim (a_1, a_2) \text{ et } x_1 > 0 \text{ et } x_1 < x_2 \\ \{ x_2 := x_2 - x_1 \} \\ (x_1', x_2') \sim (a_1, a_2)\ \underline{\text{et}}\ x_1' > 0\ \underline{\text{et}}\ x_1' + x_2' < x_1 + x_2 \end{cases}$

(2) $\begin{cases} x_1' > 0\ \underline{\text{et}}\ (x_1', x_2') \sim (a_1, a_2)\ \underline{\text{et}}\ x_1' < x_2' \\ \quad \Rightarrow x_1' > 0\ \underline{\text{et}}\ (x_1', x_2') \sim (a_1, a_2) \end{cases}$

(3) $$\begin{cases} x'_1 > 0 \ \underline{\text{et}}\ (x'_1, x'_2) \sim (a_1, a_2)\ \underline{\text{et}}\ x''_1 > 0\ \underline{\text{et}}\ (x''_1, x''_2) \sim (a_1, a_2) \\ \qquad \Rightarrow x''_1 > 0\ \underline{\text{et}}\ (x''_1, x''_2) \sim (a_1, a_2) \end{cases}$$

(4) $$\begin{cases} x_1 > 0\ \underline{\text{et}}\ (x_1, x_2) \sim (a_1, a_2)\ \underline{\text{et}}\ \underline{\text{non}}\ x_1 < x_2 \\ \qquad \Rightarrow (x_1, x_2) \sim (a_1, a_2)\ \underline{\text{et}}\ x_1 > 0\,. \end{cases}$$

De (1), (2), (3), (4) on déduit grâce à (ITE')

(5) $$\begin{cases} x_1 > 0\ \underline{\text{et}}\ (x_1, x_2) \sim (a_1, a_2) \\ \{\ \underline{\textit{tant}}\ \underline{\textit{que}}\ x_1 < x_2\ \underline{\textit{faire}}\ x_2 := x_2 - x_1\ \underline{\textit{fait}}\ \} \\ x'_1 > 0\ \underline{\text{et}}\ (x_1, x_2) \sim (a_1, a_2)\ \underline{\text{et}}\ \underline{\text{non}}\ (x'_1 < x'_2)\,. \end{cases}$$

La complexité des règles entraîne une grande lourdeur dans les preuves ; on peut arriver à quelques allègements en déduisant des règles primitives de nouvelles règles (voir à ce sujet [MAN, 74]).

Exercice 20

$A(r)$ est vide au départ. Supposons que $\Gamma(x) \cap A(r) = \varnothing\ \forall x \in A(r)$ lorsque $A(r)$ contient i − 1 éléments ; choisissons alors x_0 tel que

$$x_0 \notin A(r)\ \underline{et}\ \Gamma^{-1}(x_0) \subset A(r)\,.$$

Supposons $\Gamma(x_0) \cap A(r) \neq \varnothing$ et soit

$$z \in \Gamma(x_0) \cap A(r) : z \in \Gamma(x_0) \Rightarrow x_0 \in \Gamma^{-1}(z)$$
$$z \neq x_0 \text{ et } z \in A(r) \Rightarrow \Gamma^{-1}(z) \subset A(r) \Rightarrow x_0 \in A(r)$$

ce qui est contraire aux hypothèses.
On a donc toujours $\Gamma(x) \cap A(r) = \varnothing$.

Exercice 21

Si ce n'était pas le cas, on aurait un circuit parmi les sommets n'appartenant pas à $A(r)$.

Exercice 22

Evident d'après la construction de *CANDIDATS* dans *S*4.

Exercice 23

Cet algorithme est encore partiellement correct, $P(u, v)$ est encore invariant du cycle. Mais il ne termine pas si $u = 2k$, $v = 2k + 1$, $t(u) < a$.

Exercice 24

On peut définir t' de la façon suivante :

t' trié $\underline{\text{et}}$ $\exists m\ \forall i, j$
$[(0 \leqslant i < m \Rightarrow t'(i) = t(i+1))\ \underline{et}\ t'(m) = a\ \underline{\text{et}}\ (m < j \leqslant n \Rightarrow t'(j) = t(j))]\,.$

Cet énoncé définit t' à partir de m. Voici une définition possible de m :

$$\forall i, j\ ((1 \leqslant i \leqslant m \Rightarrow t(i) < a)\ \underline{et}\ (m < j \leqslant n \Rightarrow t(j) > a))\,.$$

Pour trouver une définition algorithmique de m, on peut passer à l'énoncé élargi suivant, définissant u et v :

$$Q(u, v) = [\forall i, j\,(1 \leqslant i \leqslant u \Rightarrow t(i) < a)\, \underline{et}\, (v < j \leqslant n \Rightarrow t(j) \geqslant a)]\,.$$

Cet énoncé admet $u = 0$ et $v = n$. Comme solution évidente, on peut définir une suite de solutions :

$$\begin{aligned} u_0 &= 0 \quad v_0 = n \\ w_k &= (u_k + v_k) \div 2 \\ u_{k+1} &= \underline{si}\ t(w_k) < a\ \underline{alors}\ w_k\ \underline{sinon}\ u_k\ \underline{fsi} \\ v_{k+1} &= \underline{si}\ t(w_k) < a\ \underline{alors}\ v_k\ \underline{sinon}\ w_k - 1\ \underline{fsi} \end{aligned}$$

on vérifie que $Q(u_k, v_k) = \underline{\text{vrai}} \Rightarrow Q(u_{k+1}, v_{k+1}) = \underline{\text{vrai}}$.

Donc, si u_{k+1} et v_{k+1} sont définis, ils vérifient Q ; mais ils ne sont définis que si le test $t(w_k) > a$ l'est ; or a priori, $w_k \in [0, n]$ et donc $t(w_k)$ n'est défini que si $w_k \in [1, n]$; or on peut avoir $w_k = 0$ si $a < t(1)$ et il nous faut traiter ce cas différemment ; l'énoncé définissant m devient alors :

$$m = \underline{si}\ a < t(1)\ \underline{alors}\ 0\ \underline{sinon}\ q\ \underline{fsi}$$
$$q : \forall i, j\,(1 \leqslant i \leqslant q \Rightarrow t(i) < a\ \underline{et}\ q < j \leqslant n \Rightarrow t(j) > a)\,.$$

Il nous reste à donner une définition algorithmique de q sachant qu'elle n'est utilisée que si $a \geqslant t(1)$.

Nous recherchons à nouveau une suite (u_k, v_k) telle que $Q(u_k, v_k)$ soit vrai. Par hypothèse, $t(1) \geqslant a$ et cette suite peut être définie comme précédemment mais avec comme valeur initiale $u_0 = 1$ et $v_0 = n$; d'où la définition de q :

$$\begin{aligned} &q : u := 1 ;\ v := n ; \\ &\underline{tant\ que}\ u \neq v\ \underline{faire}\ (w = (u + v) \div 2 ; \\ &\qquad\qquad \underline{si}\ t(w) < a\ \underline{alors}\ u := w\ \underline{sinon}\ v := w - 1\ \underline{fsi}) ; \\ &q = u\,. \end{aligned}$$

L'invariant de la boucle est bien Q.

L'algorithme est-il borné ? Malheureusement, pour $u = 2\,k, v = 2\,k + 1$ l'algorithme boucle. Nous vérifions alors que si nous posons

$$w_k = (u_k + v_k + 1) \div 2$$

nous avons encore $v_k \leqslant w_k \leqslant u_k$ et cette fois, à chaque itération

$$u_k < v_k \Rightarrow v_{k+1} - u_{k+1} < v_k - u_k\,.$$

Avec cette modification l'algorithme est borné.

Récapitulons :

$$\begin{aligned} t' &: 0 \leqslant i < m \Rightarrow t'(i) = t(i + 1) \\ &\qquad t'(m) = a \\ &\quad m < i \leqslant n \Rightarrow t'(i) = t(i) \\ m &: \underline{si}\ t(1) < a\ \underline{alors}\ 0\ \underline{sinon}\ q\ \underline{fsi} \end{aligned}$$

$$
\begin{aligned}
&q : u = 1, v = n\,;\\
&\underline{\textit{tant que}}\ u \neq v\ \underline{\textit{faire}}\ (w := (u + v + 1) \div 2\,;\\
&\qquad\qquad\qquad\qquad \underline{\textit{si}}\ t(w) < a\ \underline{\textit{alors}}\ u := w\\
&\qquad\qquad\qquad\qquad\qquad \underline{\textit{sinon}}\ v := w - 1\ \underline{\textit{fsi}})\,;\\
&q = u\,.
\end{aligned}
$$

Exercice 25

Il suffit de montrer que $p \circ R = \cap \{ q \mid p\,;R \subseteq R\,;q \}$.
Or ceci résulte immédiatement de l'équivalence $p\,;R \subseteq R\,;q \Leftrightarrow p \circ R \subseteq q$.

Exercice 26

Montrons que $p\,;R \subseteq R\,;q \Leftrightarrow p \subseteq R \to q$.
D'après la définition de $R \to q$, on a $(R \to q)\,;R \subseteq R\,;q$.
Donc $p \subseteq R \to q \Rightarrow p\,;R \subseteq (R \to q)\,;R \subseteq R\,;q$.
Réciproquement, supposons que $p\,;R \subseteq R\,;q$. Soit x tel que $p(x)$; l'hypothèse se traduit par $\forall y\,(xRy \Rightarrow q(y))$ donc $(R \to q)\,(x)$.
De cette équivalence on déduit immédiatement que

$$R \to q = \cup \{ p \mid p\,;R \subseteq R\,;q \}\,.$$

Donc $R \to q$ est la borne supérieure des prédicats p tels que $p\,;R \subseteq R\,;q$.

Exercice 27

Détaillons seulement les preuves concernant l'opération $\circ$:

$$
\begin{aligned}
p \circ (S_1\,;S_2)\,(x) &\Leftrightarrow \exists y(p(y)\ \underline{\text{et}}\ (\exists z)\ yS_1\,z\ \underline{\text{et}}\ zS_2\,x)\\
&\Leftrightarrow \exists z\ \exists y(p(y)\ \underline{\text{et}}\ yS_1\,z\ \underline{\text{et}}\ zS_2\,x)\\
&\Leftrightarrow \exists z(p \circ S_1(z)\ \underline{\text{et}}\ zS_2\,x)\\
&\Leftrightarrow (p \circ S_1) \circ S_2(x)\\
p \circ (S_1 \vee S_2)\,(x) &\Leftrightarrow \exists y(p(y)\ \underline{\text{et}}\ (yS_1\,x\ \underline{\text{ou}}\ yS_2\,x))\\
&\Leftrightarrow \exists y(p(y)\ \underline{\text{et}}\ yS_1\,x)\ \underline{\text{ou}}\ \exists y(p(y)\ \underline{\text{et}}\ yS_2\,x)\\
&\Leftrightarrow (p \circ S_1\ \ p \circ S_2)\,(x)\,.
\end{aligned}
$$

Supposons $S_1 \subseteq S_2$:

$$p \circ S_1(x) \Leftrightarrow \exists y(p(y)\ \underline{\text{et}}\ yS_1\,x) \Rightarrow \exists y(p(y)\ \underline{\text{et}}\ yS_2\,x) \Rightarrow p \circ S_2(x)$$

Enfin :

$$p \circ E(x) \Leftrightarrow \exists y(p(y)\ \underline{\text{et}}\ yEx) \Leftrightarrow p(x)\,.$$

Exercice 28

1) Si l'on suppose que

$$\underline{\text{vrai}}\ \{ F(x - 1\,;z) \}\ z = 0\,,$$

alors

$$\underline{\text{vrai}}\ \{ F(x - 1\,;z)\,;y := 2 * z \}\ y = 0\,;$$

donc on peut déduire

$$\underline{\text{vrai}}\ \{ F(x\,;y) : F(x - 1\,;z)\,;y := 2 * z \}\ y = 0\,.$$

Evidemment il n'y a aucune chance que la procédure termine.

2) Faisons l'hypothèse de récurrence que l'appel interne se déroule correctement.

Distinguons alors deux cas :

- soit $b = 0$ alors le *pgcd* de a et de b est a
- soit $b \neq 0$ le prédicat d'entrée est vérifié pour l'appel de *reste* et évidemment aussi pour l'appel interne et donc, après l'appel interne, r est le *pgcd* de b et de *reste*(a, b) ; il reste alors à montrer que le *pgcd* de a, b est aussi celui de b, *reste*(a, b) ce qui est du domaine de l'arithmétique.

Exercice 29

Justification de la règle de preuve des procédures récursives par l'induction de SCOTT.

Soit une déclaration de procédure récursive du type $f(x\,;y) : \gamma$; pour tout x tel que $f(x\,;y)$ admette un résultat, notons $F_0(x)$ ce résultat. F_0 peut être définie par une déclaration récursive du type

$$F(x) \Leftarrow \tau(F)\,(x)\,.$$

En d'autres termes, F_0 est le plus petit point fixe d'une fonctionnelle τ ce que nous notons $F_0 = \mu(\tau)$.

Sur un exemple, vérifions la possibilité de définir F_0 comme plus petit point fixe : soit

$$fact(x\,;y) : \underline{si}\ x = 0\ \underline{alors}\ y := 1\ \underline{sinon}\ fact(x - 1\,;z)\,;y := x * z\ \underline{fsi}$$

on peut lui associer la déclaration

$$Fact(x) \Leftarrow \underline{si}\ x = 0\ \underline{alors}\ 1\ \underline{sinon}\ x * Fact(x - 1)\ \underline{fsi}\,.$$

La propriété de correction partielle d'une procédure $f(x : y)$ par rapport aux deux prédicats $\mathrm{p}(x)$ et $\mathrm{q}(x\,;y)$ se traduit par l'assertion :

$$\mathrm{P}(F_0) : (F_0(x) \neq \omega\ \underline{\mathrm{et}}\ \mathrm{p}(x)) \Rightarrow \mathrm{q}(x, F_0(x))\,.$$

De son côté l'assertion

$$\mathrm{p}(x)\ \{\ f(x\,;y)\ \}\ \mathrm{q}(x, y) \Rightarrow \mathrm{p}(x)\ \{\ \gamma\ \}\ \mathrm{q}(x, y)$$

peut être traduite par

$$\mathrm{P}(F) \Rightarrow \mathrm{P}(\tau(F))\,.$$

Pour justifier la règle d'induction présentée ici pour les procédures récursives, montrons que :

1) $\mathrm{P}(\Omega)$ est vérifiée (Ω désignant la fonction nulle part définie).

2) $\mathrm{P}(F)$ est une propriété admissible, c'est-à-dire qu'il existe deux familles (σ_i), (σ'_i) d'applications continues de $\mathscr{F}(\mathrm{D}, \mathrm{D})$ dans un ensemble ordonné $(\mathrm{B}, \leqslant)$ telles que la propriété P puisse s'écrire sous la forme

$$\mathrm{P}(F) : \forall \mathrm{i} \in \mathrm{I} \quad \sigma_i(F) \leqslant \sigma'_i(F)\,.$$

1) $P(\Omega)$ est naturellement vérifiée. Intuitivement cela correspond au fait qu'une procédure qui ne termine pas est partiellement correcte pour toute assertion.

2) Prenons pour B l'ensemble { vrai, faux } muni de l'ordre faux $\preccurlyeq$ vrai. Considérons par ailleurs les applications σ_x et σ'_x définies par

$$\sigma_x(F) = F(x) \neq \omega \ \underline{\text{et}}\ p(x)$$

et

$$\sigma'_x(F) = q(x, F(x))$$

si l'on pose $q(x, \omega) =$ faux pour tout x de D ; on vérifie aisément que σ_x et σ'_x sont des applications continues de $\mathscr{F}(D, D)$ dans B.

Maintenant, P(F) peut s'écrire :

$$\forall x \in D \quad \sigma_x(F) \preccurlyeq \sigma'_x(F)$$

et donc P est une propriété admissible. Nous pouvons donc appliquer la règle de SCOTT (voir chapitre 2, § 4) et écrire, puisque $P(\Omega)$ est toujours vérifiée :

$$\frac{P(F) \Rightarrow P(\tau(F))}{P(\mu(\tau))}.$$

Exercice 30

La première valeur des paramètres (première pour l'ordre lexicographique) conduisant à un calcul infini est (1, 3). Le lecteur pourra vérifier que si (x, y) conduit à un calcul infini, alors il existe des paramètres (x', y') avec lesquels la procédure A' est appelée une infinité de fois.

Exercice 31

Représentons l'échiquier par un tableau *ech* tel que $ech[i] = j$ si et seulement si une reine occupe la position (i, j). L'ensemble E des solutions sera imprimé au fur et à mesure. La procédure s'écrit alors :

```
reine(i, ech, n) : si i > n alors imprimer(ech)
                   sinon pour j de 1 a n faire
                         si la position (i, j) n'est pas en prise dans ech
                         alors
                              tableau nouveau[1 : n] ;
                              nouveau := ech ;
                              nouveau[i] := j ;
                              reine(i + 1, nouveau, n)
                         fsi
                   fait
       fsi
```

Prouvons la terminaison de *reine* :

— Soit $V = [0 : n]$ muni de l'ordre habituel ; V est bien fondé.
— Soit $p(i, ech, n) = (i \leqslant n + 1)$ un prédicat portant sur les paramètres

de reine ; p est vérifié pour l'appel initial et, si p est vérifié à l'entrée d'une procédure, p est vérifié pour ses appels internes.

— Soit $u(i, ech, n) = 1 + n - i$ une application des paramètres dans V qui est définie si p est vérifié.

On vérifie alors que pour chaque appel interne

$$u(i, ech, n) < u(i + 1, nouveau, n) .$$

Donc il n'y a pas d'appels infinis ; comme d'autre part il n'y a pas de cycle possible dans le corps de procédure, *reine* se termine effectivement.

Voici quelques indications pour la correction partielle : on considère les débuts de positions possibles P pour les reines :

$$P = (1, j_1), (2, j_2), \ldots, (k, j_k) \ (k \leqslant n)$$

on note $P \sqsubseteq P'$ si P' est un prolongement de P.

Il faut alors démontrer qui, si *ech* contient le début de position

$$P(1, j_1) , \ldots, (i - 1, j_{i-1}) ,$$

reine(i, ech, n) imprime toutes les solutions P' telles que $P \sqsubseteq P'$.

CHAPITRE 5

Sémantique d'un langage de programmation

1 BUTS D'UNE FORMALISATION DE LA SÉMANTIQUE D'UN LANGAGE

Ce qui se conçoit bien s'énonce clairement et les mots pour le dire arrivent aisément.

(BOILEAU, l'Art poétique)

1.1 Introduction

Au chapitre 3 nous avons présenté des **structures de contrôle** permettant d'écrire des algorithmes (schémas de programmes, schémas itératifs, schémas récursifs). Si ces différentes constructions sont bien adaptées à une étude des concepts de base de la programmation, si elles permettent de décrire des méthodes de preuves et de définir des notions d'équivalence, elles sont assez éloignées des langages de programmation actuels. Plus précisément nous avons négligé au chapitre 3 un aspect important de ces langages : c'est celui qui concerne la **structure d'information** du langage c'est-à-dire les propriétés des objets manipulés dans un programme. Par là même, nous avons peu ou pas abordé jusqu'à présent certaines constructions qui concernent les objets d'un langage et sont couramment utilisées par le programmeur : déclarations d'identificateur (et donc problèmes de portée, structure de bloc...), déclarations de procédures, définitions de type, etc...

En fait, l'approche que nous avons suivie a pour origine les difficultés qu'éprouvèrent au début des années 60 les informaticiens désireux de se doter de bons outils de définition de langages de programmation. Si les études sur l'aspect syntaxique aboutirent assez rapidement à la création de « bons » objets mathématiques (grammaires, arbres syntaxiques...) et à l'introduction de la notation de Backus [NAU, 63] qui fit rapidement l'unanimité, il n'en fut pas de même pour l'aspect sémantique. Ainsi en atteste le colloque organisé par l'IFIP (International Federation for Information Processing) en 1963

[STE, 66] qui permit de constater la diversité des approches proposées aussi bien que l'état embryonnaire des recherches.

Pour progresser deux principales voies s'offraient alors aux chercheurs :

— Tenter de formaliser la sémantique des langages de programmation de l'époque (ALGOL 60, PL/1...) quitte à obtenir des définitions lourdes et imparfaites que l'on pourrait affiner par la suite.

— Clarifier la situation en étudiant les concepts fondamentaux de la programmation et en en dégageant des structures de base (schémas itératifs, schémas récursifs, notion de type, domaines sémantiques, etc...).

Il semble qu'on approche actuellement du point de rencontre des deux chemins précédents puisque les chercheurs sont arrivés à des idées plus claires sur les concepts liés à la sémantique, idées que nous essayons de dégager dans la suite.

Dans ce chapitre, après avoir réfléchi sur les buts d'une définition précise de la sémantique d'un langage de programmation (§ 1.2, 1.3) nous donnons un aperçu des concepts fondamentaux permettant de les atteindre (§ 2). Ensuite nous examinons quatre manières d'envisager la sémantique sur un petit langage choisi pour sa simplicité (§ 3, 4, 5, 6, 7). Au paragraphe 8 enfin nous comparons brièvement les différentes approches précédemment évoquées. Le paragraphe 9 décrit un outil de définition d'une partie de la sémantique d'un langage évolué bien adapté en particulier à la construction de compilateurs : la méthode des attributs.

L'optique dans laquelle nous présentons ce chapitre nous a conduit à restreindre notre champ d'étude aux approches directement liées aux langages de programmations existants. C'est pourquoi plusieurs travaux importants ne sont pas abordés ici ; c'est le cas de la sémantique algébrique développée par NIVAT et son équipe et qui doit être l'objet d'un ouvrage à venir [COU, 78].

1.2 Le concept de sémantique d'un langage de programmation

D'après le grand Larousse, la sémantique d'un langage est l'étude des sens des mots de ce langage ainsi que de leurs variations. Les langages de programmation étant artificiels, on pourrait s'attendre à avoir une définition beaucoup plus précise du terme **sémantique.** Mais force est de constater que dans la société des informaticiens ce terme n'est pas perçu par tout le monde de la même manière.

Dans ce paragraphe nous pouvons nous contenter d'une idée intuitive de la notion de sémantique puisque le but des différentes méthodes de formalisation est d'en donner une définition précise. En première approximation nous convenons que la sémantique d'un langage de programmation associe un sens c'est-à-dire une valeur sémantique aux différents programmes de ce langage.

Le problème posé est donc double :

i) définir un ensemble de **valeurs sémantiques** (pour un langage donné) ;
ii) définir une application, appelée **fonction sémantique,** de l'ensemble des programmes (du langage) dans celui de ces valeurs.

L'étude de ces deux concepts passe par une réflexion sur les buts poursuivis, c'est ce que nous faisons maintenant.

1.3 Intérêts d'une formalisation de la sémantique

En général, la définition d'un langage de programmation évolué comporte trois aspects :

i) Définition formelle de la majeure partie de la syntaxe du langage à l'aide d'une grammaire exprimée par exemple à l'aide de la notation de Backus : ALGOL 60, SIMULA, PL/1, ... ou en utilisant éventuellement une méta-grammaire : ALGOL 68 [WIJ, 75].

ii) Définition moins formalisée de conditions syntaxiques supplémentaires (due essentiellement au fait qu'on se restreint en i) à des grammaires du type contexte libre, ceci afin d'alléger les définitions, plutôt qu'à des grammaires contextuelles).

iii) Définition peu formelle de la sémantique.

Les définitions ii) et iii) sont formulées en langue naturelle, en employant un style plus ou moins mathématique. L'introduction de méthodes permettant d'augmenter la précision de ces définitions serait d'un grand intérêt pour diverses catégories d'informaticiens :

a) *Aide à l'implantation*

Si la formalisation de la syntaxe permet d'automatiser certaines parties de la compilation (notamment l'analyse syntaxique) le manque de précision de la définition de la sémantique pose de nombreux problèmes à celui qui implante. Il importe donc de donner un sens complet et non ambigu à tout programme du langage, ce qui devrait permettre de contribuer à la résolution des difficiles problèmes de la construction automatique et de la preuve de correction de compilateurs. Plus simplement une bonne définition de la sémantique éviterait que, selon l'implantation, un programme donne pour les mêmes données des résultats différents. Considérons par exemple le programme FORTRAN suivant (dû à WAITE) :

```
COMMON M
M = 1
N = K(M, M + 3)
PRINT M, N
STØP
END
FUNCTION K(I, J)
CØMMØN L
I = I + 1
L = L + J
K = I + L
RETURN
END
```

Selon le mécanisme de passage des paramètres, les résultats sont différents. La plupart des compilateurs FORTRAN conduisent à $M = 6$ et $N = 12$ (passage par référence). Mais certains compilateurs qui définissent un mode optimisé de transmission des paramètres (valeurs – résultats) conduisent à $M = 2$, $N = 7$.

Exercice 1

Que peut-on dire de la terminaison du programme Algol 60 suivant ?

```
    début entier x ;
          procédure f(l), étiquette l ;
                   début si x = 1 alors début x := 0 ; f(m) fin ;
                         m : allera l
          x := 1   fin
          f(n)
n : fin
```

□

Remarquons au sujet des problèmes d'implantation que, même dans le cas d'une approche formelle, il n'est pas nécessaire que la définition soit complète. Par exemple la formalisation de la récursivité doit laisser le plus grand choix possible d'implantations de manière à permettre des optimisations. La difficulté est donc bien de donner une définition qui associe un **sens unique** à un programme sans pour autant trop contraindre celui qui procède à l'implantation.

b) Aide à la programmation

Une définition précise de la sémantique d'un langage doit permettre au programmeur de la maîtriser parfaitement et d'éviter ainsi la déplorable pratique actuelle qui consiste à aller vérifier ce que fait un langage par des passages sur machine ce qui, entre autres défauts, conduit à définir des programmes corrects relativement à une implantation (et non relativement à la définition du langage). Pour bien mettre en évidence la difficulté de donner un sens à toute instruction nous invitons le lecteur à répondre aux questions suivantes :

i) Quels sont, selon le mode de passage des paramètres, les résultats des appels de procédure suivants ?

— *perm*(i, $A[i]$)
— *perm*($A[i]$, i)

où perm est une procédure à deux paramètres a et b dont le corps est :

$$c := a\,;\, a := b\,;\, b := c\,.$$

ii) Quelle est la valeur de K après l'exécution du fragment de programme FORTRAN suivant ?

```
I, J, K = 0
DO 1 I = 1, 2
K = K + 1
```

```
      GOTO 1
      DO 1 J = 1, 2
      K = K + 1
    1 CONTINUE
```

Pour i), on peut vérifier que dans le cas de transmission par valeur-résultat ou par référence on obtient $A[\bar{i}] = \bar{i}\,; i = \bar{A}[\bar{i}]$ dans les deux appels si $\bar{u}$ désigne la valeur avant l'appel et u la valeur après l'appel d'une variable u quelconque. Dans le cas d'une transmission de paramètres par nom (comme en Algol 60) on obtient les mêmes résultats pour le premier appel mais le deuxième fournit $A[A[\bar{i}]] = \bar{i}\,; i = \bar{A}[\bar{i}]$. Le programme proposé en ii) a été essayé sur un compilateur, il a fourni, $K = 4$, le saut s'effectuant à l'intérieur d'une itération (ce qui n'est pas évident a priori). Signalons d'ailleurs que pour cette raison ce programme n'est pas acceptable.

Exercice 2

Quelle est la valeur de l'expression $8 > 6 > 4$ du langage PL/1 ? □

Enfin, les langages de programmation évoluent actuellement d'un mode impératif vers un mode plus déclaratif (langages à assignation unique [ASH, 76], langages de très haut niveau [KEN, 75], instructions gardées [DIJ, 75] et aussi langages d'intelligence artificielle [HEW, 72], [COL, 75], [ROU, 75]. Cette évolution nécessite une définition plus précise de la sémantique permettant de connaître le résultat de la ou des séquences d'exécution correspondant à une description donnée. En effet dans de tels langages un choix est laissé à l'exécution et ce n'est pas en effectuant des « exécutions à la main » que l'on peut connaître le sens d'un programme mais en partant des spécifications du problème qu'il est sensé résoudre.

Dans la suite nous ne nous préoccupons que de langages algorithmiques classiques. La maîtrise complète du langage manipulé est une condition nécessaire à la définition de méthodes de conception et de méthodes de preuves utilisables par le programmeur. On a d'ailleurs bien constaté au chapitre 4 que définir les transformations de prédicats associées aux instructions du langage revient à donner un sens à ces instructions. Nous verrons (§ 7) que certaines méthodes de formalisation de la sémantique reposent sur cette idée.

c) Aide à la conception de langages nouveaux

De même que la notation de Backus est un outil simple et efficace pour la description d'une syntaxe, une méthode générale de formalisation de la sémantique doit permettre de définir plus facilement et plus correctement de nouveaux langages de programmation, en évitant en particulier certaines contradictions. C'est l'un des buts principaux que se sont fixé certains auteurs tels que SCOTT et STRACHEY [SCO, 71] en faisant l'hypothèse raisonnable que l'introduction d'une « bonne » formalisation va de pair avec la mise en évidence de « bons » outils de construction de programme et donc avec la définition de « bons » langages. TENNENT [TEN, 77] a, par exemple, analysé des extensions de Pascal avec cette méthode.

Toujours en restant au niveau de la conception de nouveaux langages, une

définition précise doit permettre également une normalisation plus facile d'un langage ainsi qu'une comparaison aisée de différents langages.

d) Conclusion

« Parbleu ! dit le meunier, est bien fou du cerveau
Qui prétend contenter tout le monde et son père »
Le Meunier, son Fils et l'Ane.
JEAN DE LA FONTAINE

Il semble cependant que certains des objectifs précédents (a), b), c)) soient difficilement conciliables sinon contradictoires : peut-on donner une définition formelle qui soit à la fois suffisamment complète pour définir une implémentation, suffisamment simple et lisible pour être utilisable quotidiennement par le programmeur dans la construction de programmes corrects, suffisamment souple et puissante pour être adaptée à la conception de langages nouveaux ?

La diversité des buts poursuivis explique en partie le nombre d'approches différentes de la sémantique et souligne la relativité des jugements de valeur que l'on peut porter (une « bonne » définition pour celui qui implante peut être assez éloignée de l'idée que s'en fait un programmeur comme, de manière analogue, les sauts peuvent être de « bons » objets pour le premier et d'exécrables pour le deuxième). Il ne semble pas, à l'heure actuelle, qu'existe cette méthode universelle utilisable par les informaticiens de tous poils, aussi nous pensons qu'il faut se ranger au point de vue d'auteurs tels que HOARE et LAUER [HOA, 74] ou DONAHUE [DON, 76] qui préconisent de donner plusieurs définitions compatibles d'un même langage, chacune d'elles s'adressant à une catégorie particulière d'utilisateurs. Nous abordons ce point de vue au paragraphe 8.

2 PRINCIPAUX CONCEPTS UTILISÉS DANS LES MÉTHODES DE FORMALISATION

Schématiquement, un programme d'un langage de programmation classique décrit des **traitements** portant sur des **valeurs.** Pour préciser sa sémantique il est nécessaire d'introduire un cadre dans lequel on peut exprimer ces deux notions, c'est ce que nous allons faire maintenant.

2.1 Structure d'information

Dans le cas où les objets manipulés par le programme sont simples, les entiers par exemple, l'ensemble des valeurs accessibles à un instant donné de l'exécution d'un programme peut être ordonné en une suite finie de nombres : c'est le **vecteur d'état** introduit par McCARTHY [McC, 63] pour rendre compte de l'idée d'état de mémoire et utilisé au chapitre 3 dans le cadre des schémas de programmes. Cependant, les informations manipulées par un programme peuvent être beaucoup plus complexes que de simples nombres (tableaux, structures, listes, ...), une formalisation de la sémantique nécessite donc une

théorie des structures d'information [REM, 74], [SCO, 76], [FIN, 76]. Ici une telle structure permet de définir un cadre dans lequel on peut exprimer les propriétés des objets manipulés par un langage ; une information étant une abstraction de la notion d'état de mémoire à un moment de l'exécution d'un programme.

Cette notion doit permettre d'exprimer également le passage d'un état à un autre (ce qui formalise l'exécution d'une instruction). Notons qu'une telle étude a été menée par les auteurs de la méthode de Vienne [LUC, 68]. Nous présentons dans la suite (§ 4, 5 et 6) plusieurs formalisations du concept de structure d'information utilisant un cadre ensembliste. Une image partielle d'un état est constituée d'un prédicat liant les valeurs des variables à cet instant de l'élaboration du programme, ce point de vue est considéré au paragraphe 7.

2.2 Fonctions et calculs

La première idée pour définir le sens d'un programme est de lui **associer une fonction :** un programme fait en effet passer de certaines données à certains résultats (ou d'une information d'entrée à une information de sortie) ; on donne ainsi un point de vue fonctionnel d'un programme (§ 6).

Cependant ce type de définition rend mal ou difficilement compte de certaines notions liées à l'exécution d'un programme. Par exemple on assimile les programmes qui doivent boucler aux programmes indéfinis, c'est-à-dire ceux qui buttent sur une erreur (telle que la division par 0) ce qui n'est pas satisfaisant. Ainsi, plutôt que de retenir les états initiaux et finaux on peut considérer la suite d'états de mémoire par lesquels passe le calculateur pendant l'exécution d'un programme ; une telle suite s'appelle encore un calcul. Pour préciser cette notion on peut s'inspirer du chapitre 3 paragraphe 2.3 (ainsi dans la méthode de Vienne un état contient à la fois le programme qui s'exécute et le compteur ordinal) ou encore définir un calcul comme une suite de fonctions (ou modifications) élémentaires faisant passer d'un état à un autre (par exemple affectation, déclaration...). Parallèlement à une théorie des structures d'informations, une théorie des calculs est nécessaire, elle permet en particulier de rendre compte des calculs infinis.

2.3 Fonction sémantique

Parmi d'autres classifications, on peut grouper schématiquement en deux familles les différentes approches de la sémantique d'un langage de programmation auxquelles correspondent respectivement deux types de fonctions sémantiques selon que l'on définit la valeur associée à un couple (programme, donnée) ou simplement à un programme.

i) Le premier type correspond à un point de vue **interprétatif :** La fonction sémantique $\mathscr{S}$ associe à un couple (*programme*, donnée) un calcul qui peut posséder un résultat

$$(\textit{programme}, \text{donnée}) \xrightarrow{\mathscr{S}} \text{calcul}\,.$$

ii) Le deuxième type correspond à un point de vue **dénotationel** : La fonction sémantique $\mathscr{S}$ associe à un programme P un objet mathématique $\mathscr{S}[\![P]\!]$: sa « dénotation ». $\mathscr{S}[\![P]\!]$ peut être une fonction définie dans l'ensemble des données, à valeur dans l'ensemble des résultats (figure 1).

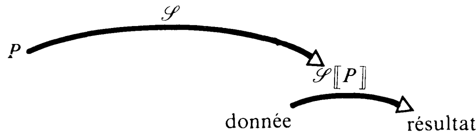


Figure 1.

Ce peut être également une fonction définie dans l'ensemble des données à valeur dans un ensemble de calculs (figure 2).

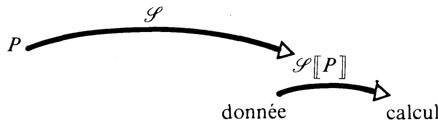


Figure 2.

Les méthodes de définition de ces fonctions sémantiques peuvent être très diverses, elles sont caractéristiques de la formalisation choisie. En particulier ces fonctions doivent être calculables, d'ailleurs on retrouvait dans les premières formalisations de la sémantique les différentes approches faites par les logiciens de la notion de calculabilité (POST, CHURCH, MARKOV, TURING). Citons simplement pour mémoire les formalisations fondées sur les algorithmes de MARKOV [MAR, 54], [BAK, 67] et le λ-calcul de CHURCH [CHU, 41], [CUR, 58], [LAN, 65].

Remarquons également la filiation entre les machines abstraites des méthodes interprétatives (§ 4) et les machines de TURING [TUR, 46], [KLE, 71].

2.4 Langage pivot

Un programme d'un langage de programmation habituel se présente sous la forme d'une chaîne de caractères et la découverte de son sens nécessite une certaine analyse :

Par exemple, pour définir le sens de la phrase ALGOL 60

$$\underline{\textit{début}}\ \underline{\textit{réel}}\ x\,;\ x := a + b\ \underline{\textit{fin}}$$

il faut en reconnaître les différents composants. En particulier l'élaboration de $x := a + b$ commence par celle que $a + b$ (qui peut se faire dans un ordre quelconque) ce qui n'apparaît pas sur la structure linéaire de la phrase. De même que pour comprendre une phrase française il faut (au moins inconsciemment) abstraire sa structure, pour définir la sémantique d'un programme il est naturel de mettre en évidence sa structure syntaxique. Cette structure peut être présentée comme un arbre très proche de l'arbre syntaxique ou comme un

n-uplet ; ainsi, de l'affectation $x := 8$ on peut ne conserver, dans un certain contexte, que le couple $(x, 8)$ comme on le verra au paragraphe 6.

D'autre part, cette transformation de la représentation d'un programme permet de n'en conserver que les composants sémantiquement utiles, ainsi un certain nombre de symboles syntaxiques n'apparaîtront pas dans le programme transformé (par exemple le symbole *allera* en Algol 60 est redondant et la phrase *allera eti* peut fort bien être remplacée par *eti*).

Brièvement on ne conserve que la structure syntaxique des phrases en supprimant tout le « sucre syntaxique » nécessité par l'écriture linéaire et la facilité de manipulation et de lecture.

Il est également possible au cours de cette transformation de supprimer certaines constructions du langage en les représentant par d'autres (plus élémentaires ou dont le sens est plus simple à définir). Ainsi, par exemple, les cycles *pour* en Algol 60 pourront être représentés à l'aide de tests et de sauts dans l'optique d'une sémantique rendant facilement compte de ces notions ; de même une procédure fonction à n paramètres pourra être remplacée par une procédure sans résultat à n + 1 paramètres.

Enfin on peut expliciter dans la nouvelle représentation certains renseignements implicitement contenus dans la structure linéaire d'un programme, tels que la portée d'une variable dans le cas d'un langage à structure de bloc. Signalons qu'il est bien entendu possible de n'expliciter ces renseignements qu'au moment de la définition de la fonction sémantique au risque évidemment de la compliquer un peu. Ceci est souvent une question d'objectifs (implantation ou programmation) et d'outils de formalisation.

Ainsi de nombreuses méthodes de formalisation de la sémantique d'un langage évolué commencent par définir un nouveau langage appelé **langage pivot** (ou encore **langage noyau**) car il comporte des constructions plus simples que celles du langage initial, on le rencontre aussi sous le vocable **syntaxe abstraite** [LUC, 68].

Ensuite on définit une transformation (traduction) associant à chaque programme, un programme du langage pivot dont on détermine la sémantique ce que l'on peut schématiser par la figure 3.

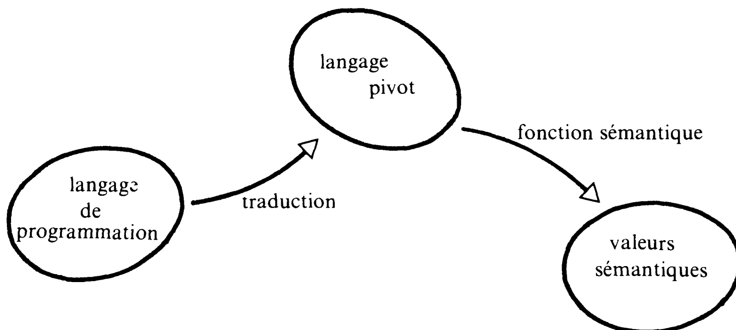


Figure 3.

A partir de cette notion de langage pivot on peut définir :

— la structure linéaire des programmes
— la sémantique du langage.

C'est la démarche proposée par McCARTHY [McC, 63] où on oppose **syntaxe concrète** (représentation linéaire) à **syntaxe abstraite** (langage pivot dans lequel les programmes sont des arbres), et c'est aussi la démarche suivie, implicitement, dans la définition d'ALGOL 68 où la sémantique s'appuie sur la structure syntaxique des différentes phrases du langage strict.

On retrouve une démarche similaire à celle proposée par certains linguistes, CHOMSKY notamment [CHO, 69] qui suggèrent, pour définir la sémantique d'une langue naturelle, de distinguer les notions de structure de surface et de structure profonde.

3 UN PETIT LANGAGE DE PROGRAMMATION

Dans la suite de ce chapitre nous présentons quatre méthodes différentes de la formalisation de la sémantique en les illustrant sur un petit langage de programmation NAIN. Présentons brièvement ce langage et tout d'abord sa forme syntaxique. Les programmes de NAIN sont définis par le système suivant :

$$\begin{aligned}
PROGRAMME &= BLOC\\
BLOC &= \underline{début}\ DECL\ ;\ INST\ \underline{fin}\\
DECL &= \underline{var}\ ID \cup \underline{cst}\ ID = EXP \cup \underline{proc}\ ID(ID : ID)\ INST\\
INST &= ID := EXP \cup BLOC \cup\\
&\quad \underline{tant\ que}\ EXP\ \underline{faire}\ INST\ \underline{fait} \cup\\
&\quad \underline{si}\ EXP\ \underline{alors}\ INST\ \underline{sinon}\ INST\ \underline{fsi} \cup\\
&\quad ID(EXP : ID) \cup\\
&\quad INST\ ;\ INST\ .
\end{aligned}$$

EXP désigne une expression quelconque, on peut se limiter aux expressions arithmétiques et logiques classiques. ID engendre les identificateurs (en nombre infini a priori). Nous laissons au lecteur le soin de terminer la définition de ce système. Signalons que la grammaire associée à ce système d'équations est syntaxiquement ambiguë, ce qui n'a aucune importance pour la définition de la sémantique.

Notons également que les procédures récursives sont acceptées mais qu'on ne considère pas de paramètres procédures.

Dans la suite nous supposons que les programmes NAIN considérés sont syntaxiquement corrects. De plus on a souvent besoin d'affirmer que chaque identificateur (de constante, de variable, de procédure) a été déclaré et que son utilisation est correcte relativement à son type (constant, variable, procédure). Ces contraintes peuvent être vérifiées statiquement c'est-à-dire indépendamment d'une élaboration quelconque du programme.

Remarquons que ce langage est un langage pivot si on admet que l'on

manipule des représentations linéaires schématiques (telles que <u>*début*</u> *D* ; *I*1 ; ... ; *In* <u>*fin*</u>) plutôt que des blocs.

4 SÉMANTIQUE INTERPRÉTATIVE OU OPÉRATIONNELLE

4.1 Notion de machine abstraite

Une sémantique interprétative est une méthode dans laquelle on associe à un couple (programme, donnée) des transitions d'un certain automate abstrait. On peut schématiser ceci par la figure 4.

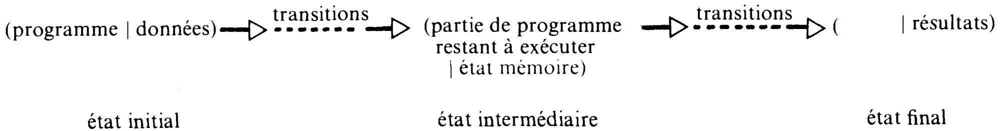


Figure 4.

Remarquons qu'il s'agit d'une définition implicite de la fonction sémantique qui est caractérisée par la donnée de la machine abstraite. Cette machine (ou automate) abstraite est essentiellement composée :

— d'un ensemble d'états. Un état contient en particulier une spécification de l'endroit où l'on est dans l'interprétation du programme ainsi que la valeur actuelle des identificateurs (état de mémoire) ;
— d'une fonction de transition d'un état à un autre.

Nous avons déjà défini un tel automate au chapitre 3 lorsque nous avons déroulé les calculs d'un schéma de programme. Les interprètes qui définissent ainsi la sémantique des schémas de programmes sont très simples car ils rendent facilement compte des notions impératives des programmes (sauts, conditionnelles, ...). Cependant si l'on veut exprimer toutes les constructions d'un langage de programmation réel l'automate se complique.

Dans la suite de ce paragraphe nous définissons une sémantique interprétative de NAIN. Commençons par formaliser la notion d'état de mémoire.

4.2 Etat de mémoire et modifications élémentaires

Comme nous l'avons déjà remarqué, toute définition de la sémantique passe par la spécification d'un formalisme permettant de préciser les propriétés des objets manipulés par un programme. Le cadre que nous utilisons ici dans le cas de NAIN doit permettre :

— d'exprimer les relations existant entre les objets d'un programme à un instant de son élaboration (état de mémoire),
— de rendre compte du passage d'un état de mémoire à un autre provoqué par l'exécution d'une instruction (notion de modification d'un état de mémoire).

Ce cadre constitue la structure d'information de NAIN que nous décrivons maintenant.

a) Formalisation de la notion d'état de mémoire

Après exécution du début de programme NAIN suivant

<u>*début*</u> <u>*proc*</u> $f(x : y)\, y := x + 1$;
  <u>*début*</u> <u>*var*</u> x ; $x := 1$;
    <u>*début*</u> <u>*cst*</u> $x = 8$.

On peut imaginer l'état de mémoire sous la forme schématisée par la figure 5.

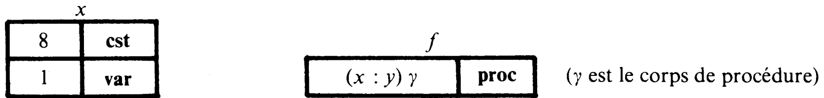


Figure 5 Image intuitive d'un état mémoire.

C'est-à-dire qu'à x est associée une pile dans laquelle on range les valeurs successivement désignées par cet identificateur ainsi que leur type. Il en est de même pour f dont la pile contient une seule valeur de type procédure constituée des paramètres d'entrée x et de sortie y ainsi que du corps γ. Pour arriver à une définition utilisable d'un tel état de mémoire (de manière à pouvoir prouver des propriétés) il faut abstraire de cette idée intuitive les notions caractéristiques. Pour cela nous utilisons un cadre ensembliste dans lequel les objets (identificateurs, types, piles, entiers, ...) sont des éléments de certains ensembles et sont reliés par certaines **fonctions d'accès ;** ainsi on peut schématiser l'état mémoire considéré précédemment par la figure 6.

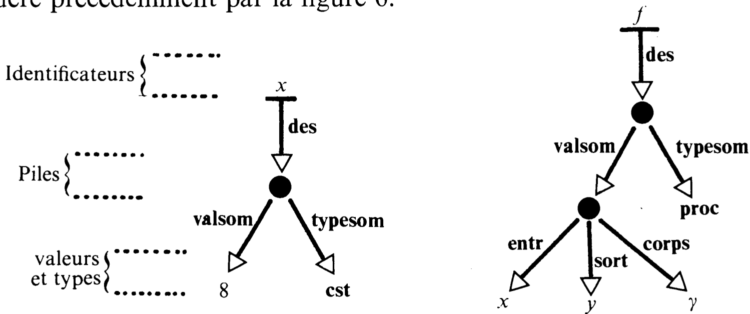


Figure 6 Image arborescente d'un état mémoire.

des (désigne), **valsom** (valeur du sommet), **typesom** (type du sommet), **entr** (paramètre d'entrée), **sort** (paramètre de sortie), **corps** (corps de la procédure) sont des fonctions d'accès permettant d'exprimer la structure logique de l'état. Ce sont des fonctions partielles définies dans certains des ensembles d'objets manipulés par le langage. Précisons tout de suite ces ensembles :

— **ID** ensemble des identificateurs
— **PI** ensemble des piles. Il contient en particulier la pile vide notée **nil.**
— **TY** = { **var, cst, proc** } ensemble des noms de type
— **PR** ensemble des textes de procédures

— **CR** = **INST** ensemble des corps de procédures, c'est-à-dire l'ensemble défini dans la syntaxe

— **VA** ensemble des valeurs (entiers, booléens...) ; pas plus que l'ensemble *EXP* des expressions de NAIN nous ne détaillons **VA.** Convenons qu'il contient un élément $\perp$ traduisant la non-initialisation d'une variable.

On peut alors expliciter les ensembles de départ et d'arrivée des fonctions d'accès précédemment citées :

$$\begin{array}{ll} \mathbf{des} & : \mathbf{ID} \to \mathbf{PI} \\ \mathbf{valsom} & : \mathbf{PI} \to \mathbf{VA} \cup \mathbf{PR} \\ \mathbf{typesom} & : \mathbf{PI} \to \mathbf{TY} \\ \mathbf{entr} & : \mathbf{PR} \to \mathbf{ID} \\ \mathbf{sort} & : \mathbf{PR} \to \mathbf{ID} \\ \mathbf{corps} & : \mathbf{PR} \to \mathbf{CR} \end{array}$$

et on exige que les trois dernières fonctions vérifient les propriétés :

(A 1) : $\mathbf{entr}((x : y)\, \gamma) = x$; $\mathbf{sort}((x : y)\, \gamma) = y$; $\mathbf{corps}((x : y)\, \gamma) = \gamma$.

Les définitions précédentes sont incomplètes si on omet de préciser ce que sont les piles : il est en effet nécessaire d'exprimer l'empilement d'une valeur et d'un type résultant de l'entrée dans un bloc et le dépilement en sortie de bloc. Pour cela on introduit deux nouvelles fonctions d'accès :

$$\mathbf{empil} : \mathbf{PI} \times (\mathbf{VA} \cup \mathbf{PR}) \times \mathbf{TY} \to \mathbf{PI}$$

qui formalise l'empilement d'une valeur et d'un type sur une certaine pile ; le résultat étant une nouvelle pile

$$\mathbf{depil} : \mathbf{PI} \to \mathbf{PI}$$

qui exprime la suppression du sommet d'une pile.

Ces deux fonctions doivent vérifier les propriétés suivantes :

$$\begin{array}{lll} (\text{A } 2) : & \mathbf{depil(empil(p, v, t)) = p} & \mathbf{valsom(empil(p, v, t)) = v} \\ & \mathbf{depil(nil) = nil} & \mathbf{typesom(empil(p, v, t)) = t} \,. \end{array}$$

Exercice 3

a) Compléter la représentation schématique de la figure 6 pour rendre compte de la pile antérieurement désignée par x.

b) Définir cet état mémoire par un ensemble d'égalités entre accès. □

Ainsi un état mémoire de NAIN est la donnée de la fonction **des,** que nous représentons souvent par son graphe **e** = (**ID, PI, des**). Si l'on voulait être tout à fait précis il faudrait détailler l'ensemble **VA** et ajouter un certain nombre de fonctions d'accès vérifiant certaines propriétés pour exprimer les opérations arithmétiques, logiques... Nous ne le ferons pas ici.

Dans la suite, utilisant la simplicité de NAIN nous facilitons le travail du lecteur en préférant aux représentations arborescentes (figure 6) des images intuitives (figure 5) dans lesquelles ne figurent pas explicitement les accès

mais qui permettent de les retrouver facilement. Ainsi la représentation de la figure 7 est préférée à celle de la figure 8.

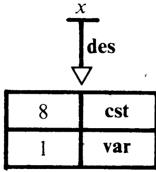


Figure 7.

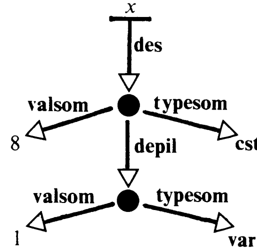
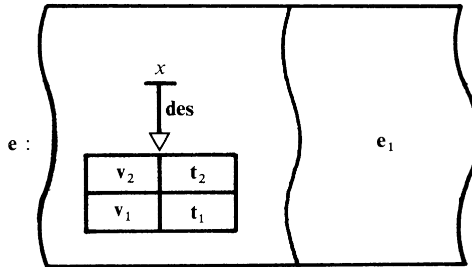


Figure 8.

b) Modification élémentaire d'un état de mémoire

Appelons $\mathscr{E}$ l'ensemble des états mémoires de NAIN. Une modification élémentaire de la structure d'information de NAIN est une application de $\mathscr{E}$ dans lui-même exprimant l'exécution des instructions élémentaires du langage ainsi que certaines « opérations de service » telles que le dépilement à la sortie d'un bloc. Définissons-les maintenant :

— **declcst**(x, **v**) exprime l'effet de la déclaration de constante $x \in$ **ID** pour la valeur **v** $\in$ **VA.** Intuitivement elle transforme un état de mémoire **e** $\in \mathscr{E}$ de la forme schématisée par la figure 9



(la partie $\mathbf{e}_1$ de **e** n'est reliée à x par aucun accès)

Figure 9 Etat de mémoire avant exécution de la déclaration de x.

en un état **e**$' \in \mathscr{E}$ de la forme de celui de la figure 10.

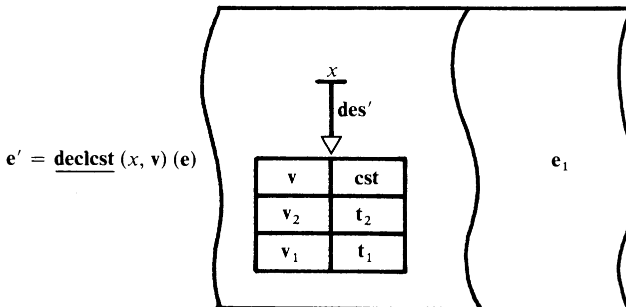


Figure 10 Etat de mémoire après déclaration de x.

Plus précisément **des**′ est définie par les relations :

$$\mathbf{des}'(x) = \mathbf{empil}(\mathbf{des}(x), \mathbf{v}, \mathbf{cst})$$

et $$\mathbf{des}'(y) = \mathbf{des}(y) \text{ pour tout } y \in \mathbf{ID}, y \neq x .$$

Nous convenons d'omettre par la suite la deuxième relation.

Nous allons définir maintenant les autres modifications :

— **declvar**(x) traduit la déclaration de la variable $x \in$ ID ; elle est définie par

$$\mathbf{des}'(x) = \mathbf{empil}(\mathbf{des}(x), \perp, \mathbf{var})$$

— **declproc**$(f, (x : y)\, \gamma)$ exprime la déclaration de la procédure $f \in \mathbf{ID}$ de texte $(x : y)\, \gamma \in \mathbf{Pr}$

$$\mathbf{des}'(f) = \mathbf{empil}(\mathbf{des}(f), (x : y)\, \gamma, \mathbf{proc})$$

— **lib**(x) formalise la suppression du sommet de la pile désignée par x

$$\mathbf{des}'(x) = \mathbf{depil}(\mathbf{des}(x))$$

— **affect**$(x, \mathbf{v})$ exprime l'affectation de la valeur $\mathbf{v} \in$ VA à la variable $x \in \mathbf{ID}$:

$$\mathbf{des}'(x) = \mathbf{empil}(\mathbf{depil}(\mathbf{des}(x)), \mathbf{v}, \mathbf{var}) .$$

Ceci revient à ne modifier que la valeur du sommet de la pile.

En plus de ces modifications élémentaires, qui sont en fait des schémas de modifications élémentaires puisqu'elles dépendent de paramètres, nous supposons l'existence d'une fonction qui permet de définir la valeur d'une expression pour un état de mémoire quelconque :

$$\mathbf{Eval} : EXP \times \mathscr{E} \to \mathbf{VA} .$$

Une telle fonction est définie simplement par récurrence sur la complexité de l'expression ; pas plus que *EXP* nous ne la précisons ici (voir exercice 4).

Exercice 4

Compléter la définition de la structure d'information de NAIN de manière à définir explicitement la fonction **Eval** précédente. On suppose que *EXP* est définie par :

$$EXP = EXPE \cup EXPB \text{ (expressions entières et expressions logiques)}$$

$$EXPE = (EXPE) + (EXPE) \cup (EXPE) - (EXPE) \cup (EXPE) \times (EXPE) \cup ID \cup ENT \text{ (}ENT\text{ représente les entiers)}$$

$$EXPB = \{ \underline{vrai}, \underline{faux} \} \cup \underline{non}\,(EXPB) \cup (EXPB)\,\underline{ou}\,(EXPB) \cup (EXPB)\,\underline{et}\,(EXPB) \cup (EXPE) \leqslant (EXPE) \cup (EXPE) < (EXPE) \cup (EXPE) = (EXPE) .$$

□

La structure d'information de NAIN étant décrite nous pouvons maintenant définir l'interprète abstrait.

4.3 Interprète abstrait de NAIN

a) Etats de l'interprète

Un **état de l'interprète** est un couple $\langle P \mid \mathbf{e} \rangle$ où $\mathbf{e}$ est un état de mémoire et P un morceau de programme. Plus précisément et de manière intuitive P représente la suite des instructions qu'il reste encore à exécuter pour terminer l'interprétation du programme P_0 de départ. Nous verrons que P contient un certain nombre de « marqueurs » nécessaires à l'interprétation (voir b4), b5), b6) ci-dessous) et qu'il n'est pas en général un facteur droit de P_0 car certaines instructions peuvent être dupliquées pour exprimer l'interprétation des itérations (voir b3) ci-dessous).

b) Transitions de l'interprète

La fonction de transition $\mathcal{T}$ de l'interprète abstrait associe à un état $\langle P \mid \mathbf{e} \rangle$, pour lequel elle est définie, un nouvel état $\langle P' \mid \mathbf{e}' \rangle$ où P' est le morceau de programme restant à interpréter après exécution de la première instruction α de P et $\mathbf{e}'$ est l'état obtenu à partir de $\mathbf{e}$ en exécutant α. Cette fonction est une fonction partielle ; le fait qu'elle ne soit pas définie pour certains états permet de rendre compte d'erreurs d'exécution (division par 0, utilisation dans un calcul d'une variable non initialisée, etc.).

$\mathcal{T}$ est définie par récurrence sur la complexité d'un programme en utilisant les modifications élémentaires du langage. Dans la suite nous présentons cette définition sur un état de la forme $\langle \alpha ; P \mid \mathbf{e} \rangle$.

b1) Interprétation d'une affectation

$\langle x := exp ; P \mid \mathbf{e} \rangle \xrightarrow{\mathcal{T}}$

$$\begin{array}{l} \underline{soit}\ \mathbf{v} = \mathbf{Eval}(exp, \mathbf{e}) \ ; \\ \underline{si}\ \mathbf{v} \neq \bot\ \underline{\text{et}}\ \mathbf{typesom(des(x))} = \mathbf{var} \\ \underline{alors}\ \langle P \mid \underline{\mathbf{affect}}(x, \mathbf{v})\ (\mathbf{e}) \rangle \ . \end{array}$$

Si la valeur de l'expression *exp* n'est pas définie, la pile associée à x est vide ou le type du sommet différent de **var,** $\mathcal{T}$ n'est pas définie. L'état $\langle x := exp ; P \mid \mathbf{e} \rangle$ est un état de non-satisfaction de l'automate (§ 4.4). Une telle situation peut se présenter lors de l'interprétation des autres types d'instruction.

Remarque : Le test $\mathbf{typesom(des(}x\mathbf{))} = \mathbf{var}$ est inutile dans le cas de NAIN puisqu'il est indépendant d'une exécution particulière du programme (on dit qu'on peut l'effectuer statiquement ou encore, pour être plus technique, qu'on peut le faire à la compilation). Son introduction ici ne pose aucune difficulté particulière et permet de comprendre ce que l'on ferait dans le cas de langages admettant des types « dynamiques ». □

b2) Interprétation d'une conditionnelle

$\langle\ \underline{si}\ exp\ \underline{alors}\ P1\ \underline{sinon}\ P2\ \underline{fsi} ; P \mid \mathbf{e} \rangle \xrightarrow{\mathcal{T}}$

$$\begin{array}{l} \underline{soit}\ \mathbf{v} = \mathbf{Eval}(exp, \mathbf{e}) ; \\ \quad \underline{si}\ \mathbf{v} = \mathbf{vrai}\ \underline{alors}\ (P1 ; P \mid \mathbf{e} \rangle \\ \quad \underline{si}\ \mathbf{v} = \mathbf{faux}\ \underline{alors}\ (P2 ; P \mid \mathbf{e} \rangle \ . \end{array}$$

*b*3) *Interprétation d'une itération*

$\langle$ *tant que exp faire P*1 *fait* ; *P* | **e** $\rangle \xrightarrow{\mathcal{I}}$
soit **v** = **Eval**(*exp*, **e**)
si **v** = **vrai** *alors* $\langle$ *P*1 ; *tant que exp faire P*1 *fait* ; *P* | **e** $\rangle$
si **v** = **faux** *alors* $\langle$ *P* | **e** $\rangle$.

*b*4) *Interprétation d'une déclaration de variable*

$\langle$ *début var x* ; *P*1 *fin P* | **e** $\rangle \xrightarrow{\mathcal{I}} \langle$ *P*1 ; *rest*(*x*) *fin* ; *P* | **declvar**(*x*) (**e**) $\rangle$.

On exprime d'une part la modification de l'état de mémoire par la déclaration de la variable *x* et d'autre part on place en fin du bloc contenant cette déclaration un « marqueur » *rest*(*x*) dont l'interprétation définira la restauration de l'ancienne valeur de *x* à la sortie de ce bloc (voir b7)).

*b*5) *Interprétation d'une déclaration de constante*

Elle est analogue à celle des déclarations de variables mais elle tient compte en plus de l'évaluation du deuxième membre :

$\langle$ *début cst x* = *exp* ; P_1 *fin P* | **e** $\rangle \xrightarrow{\mathcal{I}}$
soit v = **Eval**(*exp*, e) ;
si v $\neq \perp$ **alors** $\langle$ P_1 ; *rest*(*x*) *fin P* | **declcst**(*x*, **v**) (**e**) $\rangle$.

*b*6) *Interprétation d'une déclaration de procédure*

$\langle$ *début proc* $f(x : y)\, \gamma$; P_1 *fin P* | **e** $\rangle \xrightarrow{\mathcal{I}}$
$\langle$ P_1 ; *rest*(*f*) *fin P* | **declproc**(*f*, $(x : y)\, \gamma$) (**e**) $\rangle$.

*b*7) *Interprétation de la restauration d'une variable*

La dernière (pseudo) instruction à exécuter avant la fin d'un bloc concerne la restauration de l'ancien sommet de la pile désignée par l'identificateur défini dans ce bloc.

$\langle$ *rest*(*x*) *fin P* | **e** $\rangle \xrightarrow{\mathcal{I}} \langle$ *P* | **lib**(*x*) (**e**) $\rangle$.

Remarque

— Il y a une déclaration unique par bloc donc une et une seule pseudo-instruction de restauration.

— La pile désignée par *x* avant l'interprétation de *rest*(*x*) est nécessairement différente de **nil** puisque *rest*(*x*) n'est introduit dans un bloc que si *x* est déclaré en tête de ce bloc (voir *b*4), *b*5) et *b*6). □

*b*8) *Interprétation d'un appel de procédure*

$\langle$ *f*(*exp* : *res*) ; *P* | **e** $\rangle \xrightarrow{\mathcal{I}}$
soit *x* = **entr(valsom(des(**f**)))** ; *y* = **sort(valsom(des(**f**)))** ;
γ = **corps(valsom(des(**f**)))** ;

$\langle$ *début cst x* = *exp* ; *début var y* ; γ ; $\hat{r}$:= *y fin* ; *res* := $\hat{r}$ *fin P* | **e** $\rangle$.

On crée deux blocs emboîtés ; dans le premier on définit la constante x à partir du paramètre effectif *exp* ; dans le plus interne on déclare la variable y puis on introduit le corps γ de la procédure. Ensuite on affecte la valeur du paramètre y à une variable « standard » $\hat{r}$. Cette variable permet d'assurer l'indépendance entre le paramètre formel et le paramètre effectif de sortie : l'interprétation du bloc le plus interne se termine par celle de $\underline{\mathbf{lib}}(y)$ ce qui ferait perdre la valeur du résultat dans le cas où l'identificateur *res* serait identique à y et où on aurait $res := y$ à la place de $\hat{r} := y\ \underline{fin}\ ; res := \hat{r}$. $\hat{r}$ est un identificateur « standard » c'est-à-dire qu'il n'est pas déclaré dans un programme mais on le définit au niveau de l'état de mémoire initial (§ 4.4).

Remarquons que le mécanisme des piles permettant de retenir les objets successifs désignés par un identificateur il n'y a aucune difficulté à rendre compte des procédures récursives. L'interprétation de l'appel d'une telle procédure consiste à faire se succéder autant de déclarations, de corps de procédures et d'affectations qu'il y a d'appels emboîtés.

L'interprète abstrait étant défini on peut maintenant définir précisément la sémantique interprétative de NAIN.

4.4 Sémantique interprétative de NAIN

Un état de mémoire initial est un état dans lequel tous les identificateurs désignent la pile **nil** sauf :

— l'identificateur standard $\hat{r}$ qui désigne la pile | $\perp$ | **var** |,

— un certain nombre d'identificateurs désignant des constantes et rendant compte des données,

— un certain nombre d'identificateurs désignant des variables et rendant compte des résultats tels que $\widehat{impr}$.

Le calcul de l'interprète abstrait associé au programme P et à un état mémoire initial $\mathbf{e}_0$ est une suite d'états de l'interprète dont le premier est $\langle P \mid \mathbf{e}_0 \rangle$ et telle que l'on passe d'un état au suivant par application de $\mathscr{T}$. Le calcul **Cal** associé à P et $\mathbf{e}_0$ peut être de l'un des 3 types suivants :

i) **Cal** est **réussi** si son dernier état est de la forme $\langle \wedge \mid \mathbf{e} \rangle$ où $\wedge$ est le programme vide. Un tel état est un état de satisfaction pour l'interprète,

ii) **Cal** est **non réussi** si son dernier état n'est pas de la forme précédente et si $\mathscr{T}$ n'est pas défini sur lui (interprète bloqué),

iii) **Cal** est **infini** s'il n'existe pas de dernier élément (suite infinie d'états de mémoire) (l'interprète ne termine pas).

La fonction sémantique $\mathscr{S}_{\text{int}}$ interprétative définie par l'interprète précédent est caractérisée par : $\mathscr{S}_{\text{int}}[\![P]\!]\,(\mathbf{e}_0) = \mathbf{Cal}$.

Dans le cas où **Cal** est un calcul réussi, les résultats du programme sont les valeurs des variables résultats dans le dernier état mémoire.

Exemple 1 : interprétation d'un programme NAIN.

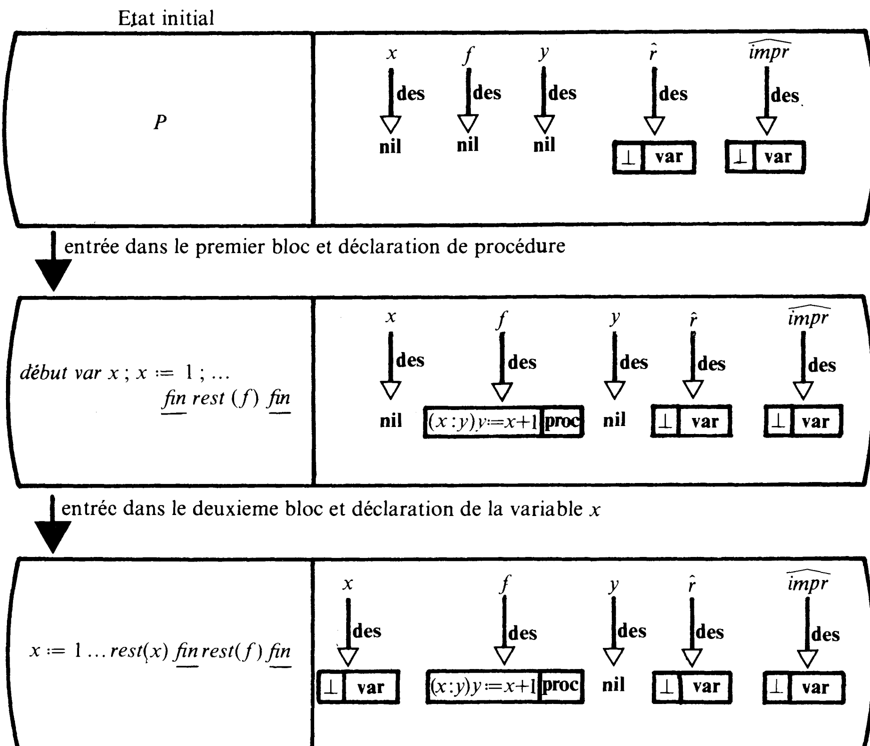
Considérons le programme P suivant :

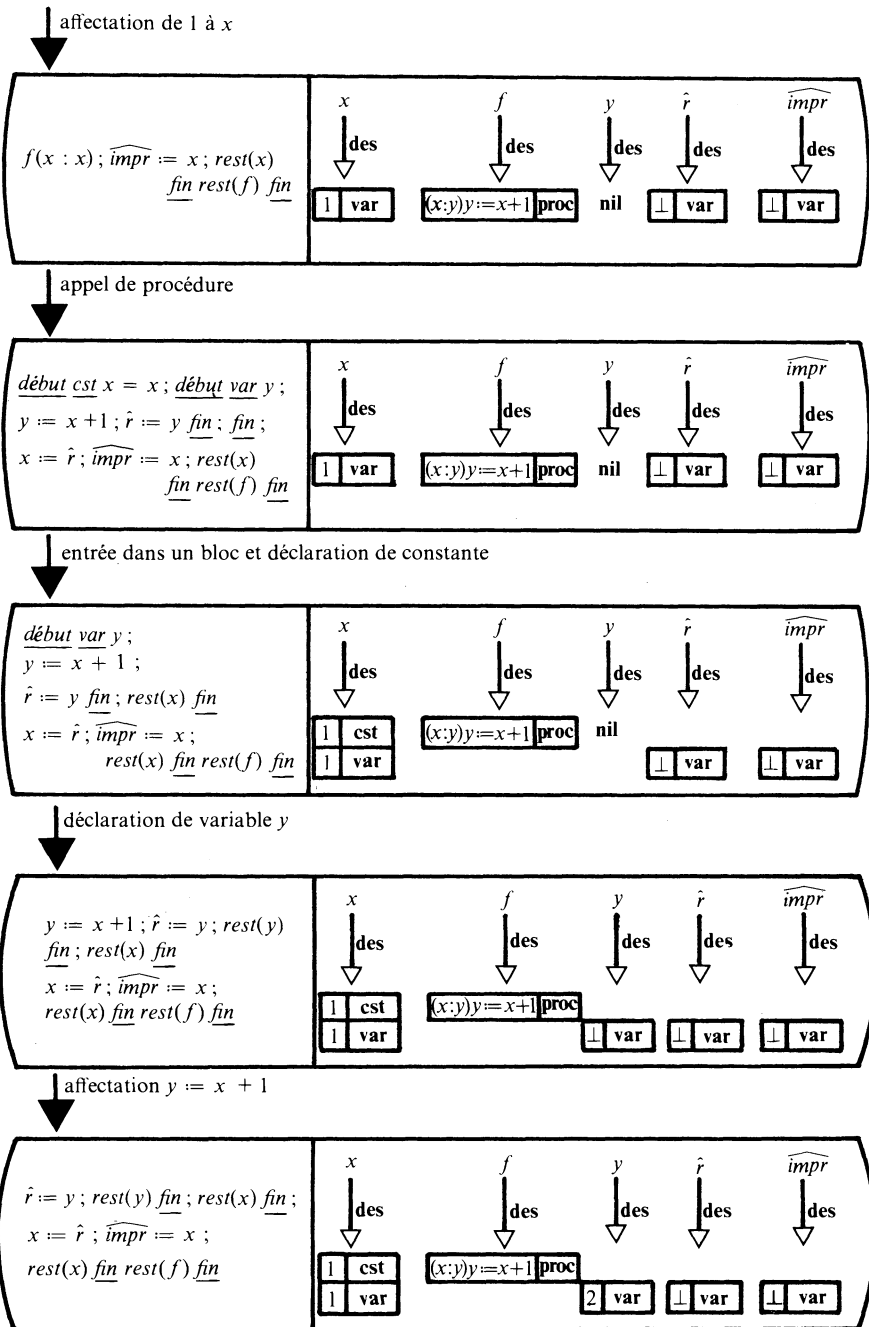
$$\underline{\textit{début}}\ \underline{\textit{proc}}\ f(x : y)\, y := x + 1\,;$$
$$\underline{\textit{début}}\ \underline{\textit{var}}\ x\,;\, x := 1\,;\, f(x : x)\,;\, \widehat{impr} := x\ \underline{\textit{fin}}\ \underline{\textit{fin}}$$

($\widehat{impr}$ est un identificateur de résultat).

Décrivons par la figure 11, la suite des états de l'interprète au cours de l'exécution de P en explicitant les états mémoire successifs. Pour simplifier les notations nous ne représentons pas le symbole « $\xrightarrow{\mathscr{I}}$ » entre chaque état mais nous mentionnons explicitement le type de la transition. Pour abréger la description il arrive que plusieurs transitions soient représentées simultanément.

Enfin il est très clair que dans la schématisation des états de mémoire nous ne mentionnons que les identificateurs concernant ce programme.




affectation de 1 à x
f(x : x) ; impr := x ; rest(x) fin rest(f) fin
x f y r impr
des des des des des
1 var
(x:y)y:=x+1 proc
nil
⊥ var
⊥ var
appel de procédure
début cst x = x ; début var y ;
y := x+1 ; r := y fin ; fin ;
x := r ; impr := x ; rest(x) fin rest(f) fin
entrée dans un bloc et déclaration de constante
début var y ;
y := x + 1 ;
r := y fin ; rest(x) fin
x := r ; impr := x ;
rest(x) fin rest(f) fin
1 cst
déclaration de variable y
y := x+1 ; r := y ; rest(y) fin ; rest(x) fin
x := r ; impr := x ;
rest(x) fin rest(f) fin
affectation y := x + 1
r := y ; rest(y) fin ; rest(x) fin ;
x := r ; impr := x ;
rest(x) fin rest(f) fin
2 var

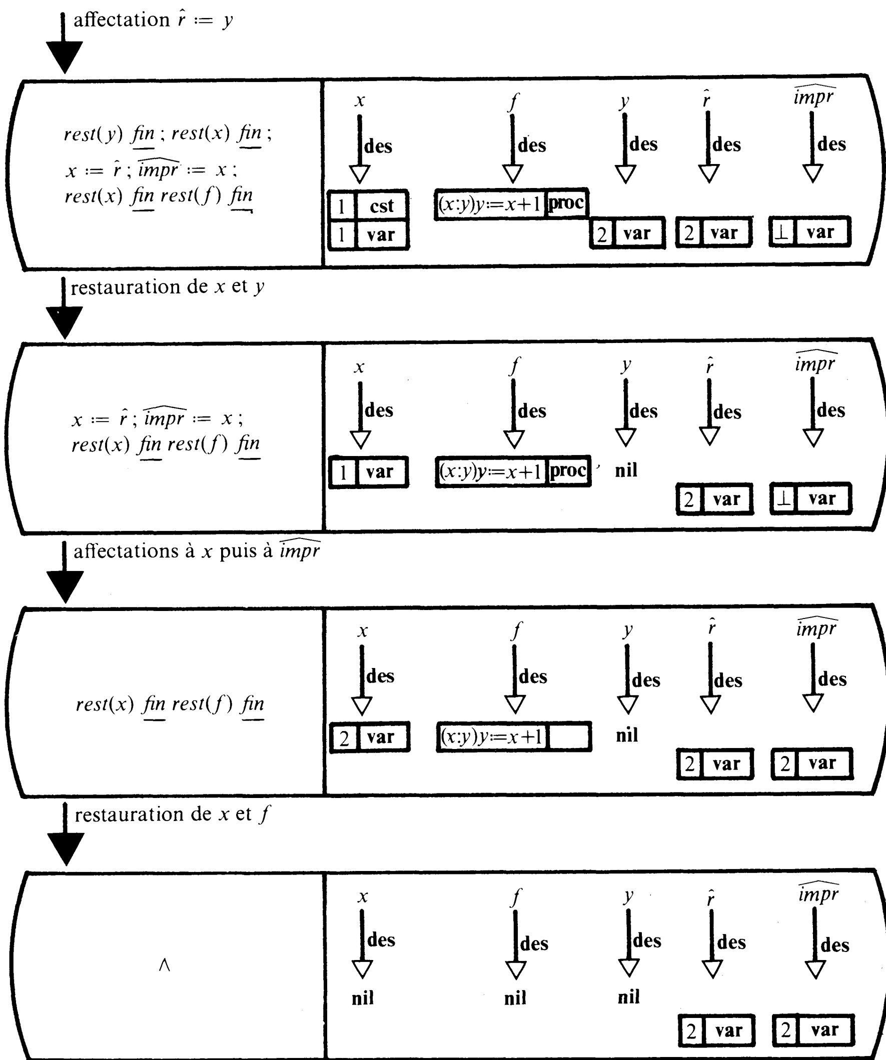


Figure 11 Exemple d'interprétation (le dernier état est un état de satisfaction).

* *Exercice* 5

Que devient l'interprétation de ce programme si la sémantique de l'appel de procédure est :

$$\langle\, f(exp : res) ; P \mid \mathbf{e} \,\rangle \xrightarrow{\mathcal{T}}$$
$$\underline{\mathbf{soit}}\ x = \mathbf{entr}(\mathbf{valsom}(\mathbf{des}(\mathrm{f}))) ; y = \mathbf{sort}(\mathbf{valsom}(\mathbf{des}(\mathrm{f}))) ;$$
$$\gamma = \mathbf{corps}(\mathbf{valsom}(\mathbf{des}(\mathrm{f})))$$
$$\langle\, \underline{\textit{début}}\ \underline{\textit{cst}}\ x = exp ; \underline{\textit{début}}\ \underline{\textit{var}}\ y ; \gamma ; res := y\ \underline{\textit{fin}}\ \underline{\textit{fin}} ; P \mid \mathbf{e} \,\rangle .$$

En déduire une condition syntaxique portant sur les appels de procédures pour que cette sémantique soit acceptable. □

Exercice 6

Ecrire en NAIN une procédure *fact* calculant n ! puis interpréter un programme appelant *fact* (2). □

Exercice 7

Préciser les adjonctions à l'interprète précédent permettant de rendre compte des instructions de saut. On pourra envisager successivement les deux cas suivants :

a) Les sauts ne se font qu'à l'intérieur d'un même bloc.
b) Les sauts se font à l'intérieur d'un même bloc ou vers un bloc plus externe.
Que se passe-t-il si l'on ne tient pas compte de la structure du bloc pour les sauts ? □

5 SÉMANTIQUE CALCULATOIRE

5.1 Introduction

L'approche précédente, très liée à la machine, présente plusieurs inconvénients :

— Elle semble difficile à comprendre pour l'utilisateur principal du langage qu'est le programmeur. La description pas à pas de l'exécution d'un programme est très « locale » et fournit des détails bien souvent inutiles au programmeur.
— Elle donne une définition très opérationnelle de la sémantique qui peut être utile pour imaginer ce que fait un programme existant mais d'un intérêt beaucoup moindre pour aider à concevoir un programme.
— Si elle est utile pour la construction d'interprète, elle est éloignée des problèmes du constructeur de compilateur qui lui, souhaite prévoir à l'avance la gestion de la mémoire et l'enchaînement des instructions.

La méthode que nous présentons ici [FIN, 76] utilise encore le concept de structure d'information pour formaliser la notion d'état de mémoire. Mais plutôt que de décrire ponctuellement et pas à pas l'action d'un programme, nous la définissons plus globalement en construisant l'**ensemble des calculs** qui lui sont associés. Cet ensemble est donc, ici, la valeur sémantique du programme considéré. Il en représente toutes les exécutions possibles.

Un calcul est défini dans ce cadre non plus comme une suite d'états mais comme une suite de modifications élémentaires de la structure d'information associée au langage.

Par exemple, au début de programme NAIN suivant :

$$\underline{\textit{début}}\ \underline{\textit{var}}\ x\ ;\ \underline{\textit{début}}\ \underline{\textit{cst}}\ y = x + 1\ ;\ x := x + 2$$

est associé le calcul :

$$\underline{\textbf{declvar}}(x).\underline{\textbf{declcst}}(y, \underline{\textbf{Eval}}(x + 1)).\underline{\textbf{affect}}(x, \underline{\textbf{Eval}}(x + 2))$$

(en reprenant les notations du paragraphe 4).

Dans la suite nous commençons par préciser la notion de calcul avant de définir la sémantique calculatoire de NAIN.

5.2 Notion de calcul

a) Calculs et ensembles de calculs

Intuitivement, un calcul est une suite d'exécution de phrases élémentaires. En utilisant le vocabulaire du paragraphe 4 :

Un **calcul** d'un langage algorithmique est une suite finie ou non d'éléments de l'ensemble $\mathcal{M}$ des modifications élémentaires de la structure d'information associée au langage.

Notons $\mathcal{M}^*$ le monoïde libre sur $\mathcal{M}$, $\mathcal{M}^{\omega}$ l'ensemble des calculs infinis et $\mathcal{M}^{\infty} = \mathcal{M}^* \cup \mathcal{M}^{\omega}$. La concaténation s'étend simplement à $\mathcal{M}^{\infty}$ qui devient alors un monoïde.

Remarque : Tout calcul fini $\mathbf{C} = \pi_1 . \pi_2 \dots \pi_n$ peut être interprété comme une modification $\pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1$ de la structure d'information en interprétant la concaténation comme la composition (dans l'ordre d'écriture) des modifications élémentaires, on assimilera par la suite ces deux notations. On peut ainsi obtenir une définition fonctionnelle d'un programme. Les calculs infinis ne sont pas interprétés.

Si la concaténation de $\mathcal{M}^{\infty}$ permet d'exprimer l'exécution des phrases séquentielles il faut pouvoir rendre compte de l'exécution des phrases conditionnelles ou itératives qui admettent plusieurs élaborations possibles selon la valeur des données. A de telles phrases on associe donc un **ensemble de calculs.**

□

Exemple 2

A l'instruction

$$\underline{si}\ x \geqslant 0\ \underline{alors}\ y := x\ \underline{sinon}\ y := -x\ \underline{fsi}$$

est associé l'ensemble de calculs

$$\{\ \underline{\mathbf{éga}}(\mathbf{Eval}(x \geqslant 0), \mathbf{vrai}) . \underline{\mathbf{affect}}(y, x)\ \} \cup \{\ \underline{\mathbf{diff}}(\mathbf{Eval}(x \geqslant 0), \mathbf{vrai}) . \underline{\mathbf{affect}}(y, -x)\ \}\ (*)$$

où $\underline{\mathbf{éga}}(\mathbf{u}, \mathbf{v})$ (resp. $\underline{\mathbf{diff}}(\mathbf{u}, \mathbf{v})$) est la modification élémentaire restriction de l'application identique à l'ensemble des états de mémoire pour lesquels **u** est égal à **v** (resp. **u** différent de **v**).

Plus précisément on définit ces deux modifications par :

$$\underline{\mathbf{éga}}(\mathbf{u}, \mathbf{v})(\mathbf{e}) = \begin{cases} \mathbf{e} & \text{si } \mathbf{u} = \mathbf{v} \text{ est vérifié dans l'état } \mathbf{e} \\ \mathbf{\Omega} & \text{sinon} \end{cases}$$

$$\underline{\mathbf{diff}}(\mathbf{u}, \mathbf{v})(\mathbf{e}) = \begin{cases} \mathbf{e} & \text{si } \mathbf{u} \neq \mathbf{v} \text{ est vérifié dans l'état } \mathbf{e} \\ \mathbf{\Omega} & \text{sinon} \end{cases}$$

(*) Dans la suite, pour simplifier les notations, nous omettons les accolades et confondons ainsi l'ensemble de calculs réduit à un élément avec cet élément.

où $\mathbf{\Omega}$ est un état mémoire particulier, l'état « indéfini », caractérisé par le fait que $\mathbf{m(\Omega)} = \mathbf{\Omega}$ pour toute modification élémentaire **m.** □

Plus généralement un ensemble de calculs est un élément de l'ensemble $\mathscr{P}(\mathscr{M}^{\infty})$ des parties de $\mathscr{M}^{\infty}$. Tous les ensembles de calculs sont ici construits à partir de modifications élémentaires par application de deux lois de composition internes dans $\mathscr{P}(\mathscr{M}^{\infty})$:

- La concaténation « . » qui est l'extension naturelle de celle de $\mathscr{M}^{\infty}$ définie par : $\mathbf{C}_1.\mathbf{C}_2 = \{\boldsymbol{\alpha}.\boldsymbol{\beta} \mid \boldsymbol{\alpha} \in \mathbf{C}_1, \boldsymbol{\beta} \in \mathbf{C}_2\}$ pour $\mathbf{C}_1, \mathbf{C}_2 \in \mathscr{P}(\mathscr{M}^{\infty})$.
- La réunion ensembliste « $\cup$ ».

b) Système de calculs associé à une phrase

L'ensemble de calculs associé à une phrase composée dépend de ceux qui sont associés aux phrases qui la composent. Il peut donc être défini par un système d'égalités. Par exemple à l'instruction suivante P de NAIN :

$$\underline{si}\ exp\ \underline{alors}\ P_1\ \underline{sinon}\ P_2\ \underline{fsi}$$

est associé le système $\mathscr{A}(P)$

$$\begin{cases} \tilde{\mathbf{P}} = \underline{\mathbf{éga}}(\mathbf{Eval}(exp), \mathbf{vrai}).\tilde{\mathbf{P}}_1 \cup \underline{\mathbf{éga}}(\mathbf{Eval}(exp), \mathbf{faux}).\tilde{\mathbf{P}}_2 \\ \mathscr{A}(P_1) \\ \mathscr{A}(P_2)\,. \end{cases}$$

Il est formé d'une part d'une équation liant l'inconnue $\tilde{\mathbf{P}}$ associée à P aux inconnues associées aux phrases composantes directes de P et d'autre part des systèmes associés à chacune des phrases P_1 et P_2.

De même, pour définir l'ensemble $\tilde{\mathbf{P}}$ de calculs associé à une itération P : $\underline{tant\ que}\ exp\ \underline{faire}\ E\ \underline{fait}$

on introduit l'équation :

$$\tilde{\mathbf{P}} = \underline{\mathbf{éga}}(\mathbf{Eval}(exp), \mathbf{vrai}).\tilde{\mathbf{E}}.\tilde{\mathbf{P}} \cup \underline{\mathbf{éga}}(\mathbf{Eval}(exp), \mathbf{faux})$$

qui traduit l'équivalence :

$$\tilde{\mathbf{P}} \equiv \underline{\mathbf{si}}\ exp\ \underline{\mathbf{alors}}\ \mathbf{E}\,;\tilde{\mathbf{P}}\ \underline{\mathbf{sinon}}\ \mathbf{vide}\ \underline{\mathbf{fsi}}\,.$$

Ainsi, pour une phrase P de la forme précédente, $\mathscr{A}(P)$ est un système à point fixe sur l'ensemble $\mathscr{P}(\mathscr{M}^{\infty})$ des calculs qui, muni de l'inclusion ensembliste, est un ensemble ordonné inductif (en fait il s'agit d'un treillis, voir le paragraphe 3 du chapitre 2).

Plus généralement, à toute phrase P et en particulier à tout programme de NAIN est associé un système à point fixe (appelé système de calculs) $\mathscr{A}(P)$ sur $\mathscr{P}(\mathscr{M}^{\infty})$ de la forme :

$$\mathscr{A}(P) : \begin{cases} \tilde{\mathbf{P}} = \mathbf{f}(\tilde{\mathbf{P}}_1, \tilde{\mathbf{P}}_2, \ldots, \tilde{\mathbf{P}}_{\mathbf{k}}) \\ \mathscr{A}(P_i) \quad i = 1, 2, \ldots, k \end{cases}$$

où les P_i sont des composantes de la phrase P.

La fonction **f** est la composée des fonctions « de base » concaténation et réunion qui sont continues dans $\mathscr{P}(\mathscr{M}^{\infty})$ muni de l'inclusion (exercice 8).

Le théorème du point fixe vu au chapitre 2 peut s'appliquer ici, et on peut affirmer l'existence d'une solution minimale. La composante relative à P dans cette solution minimale, est l'ensemble de calculs qui est la valeur sémantique de P.

Exercice 8

On considère les fonctions **u** et **c** de $(\mathscr{P}(\mathscr{M}^{\infty}))^2$ dans $\mathscr{P}(\mathscr{M}^{\infty})$ définis par :

$$\mathbf{u}(\mathbf{A}, \mathbf{B}) = \mathbf{A} \cup \mathbf{B} \text{ et } \mathbf{c}(\mathbf{A}, \mathbf{B}) = \mathbf{A}.\mathbf{B}\,.$$

Montrer que ces fonctions sont continues au sens du chapitre 2 de $(\mathscr{P}(\mathscr{M}^{\infty}))^2$ dans $\mathscr{P}(\mathscr{M}^{\infty})$. □

Précisons maintenant les systèmes de calculs associés aux différentes constructions de NAIN.

5.3 Système de calculs associé à une phrase de NAIN

1) *Affectation*

Soit P l'affectation $x := exp$.

$$\mathscr{A}(P) : \tilde{\mathbf{P}} = \underline{\mathbf{diff}}(\mathbf{Eval}(exp), \perp).\underline{\mathbf{éga}}(\mathbf{typesom}(\mathbf{des}(x)), \mathbf{var}).\underline{\mathbf{affect}}(x, \mathbf{Eval}(exp))\,.$$

2) *Conditionnelle*

Soit P la conditionnelle $\underline{si}$ *exp* $\underline{alors}$ P_1 $\underline{sinon}$ P_2 $\underline{fsi}$

$$\mathscr{A}(P) : \begin{cases} \tilde{\mathbf{P}} = \underline{\mathbf{éga}}(\mathbf{Eval}(exp), \mathbf{vrai}).\tilde{\mathbf{P}}_1 \cup \underline{\mathbf{éga}}(\mathbf{Eval}(exp), \mathbf{faux}).\tilde{\mathbf{P}}_2 \\ \mathscr{A}(P_1) \\ \mathscr{A}(P_2)\,. \end{cases}$$

Remarquons que si **Eval** $(exp) = \perp$ est vérifié dans un état de mémoire **e**, l'application à **e** du calcul associé à P conduit à $\mathbf{\Omega}$ puisque ni $\mathbf{Eval}(exp) = \mathbf{vrai}$ ni $\mathbf{Eval}(exp) = \mathbf{faux}$ ne sont vérifiées.

3) *Itération*

Soit P l'itération $\underline{tant}$ $\underline{que}$ *exp* $\underline{faire}$ P_1 $\underline{fait}$.

$$\mathscr{A}(P) : \begin{cases} \tilde{\mathbf{P}} = \underline{\mathbf{éga}}(\mathbf{Eval}(exp), \mathbf{vrai}).\tilde{\mathbf{P}}_1.\tilde{\mathbf{P}} \cup \underline{\mathbf{éga}}(\mathbf{Eval}(exp), \mathbf{faux}) \\ \mathscr{A}(P_1)\,. \end{cases}$$

Exercice 9

Expliciter le plus petit point fixe de l'équation associée à une itération. □

4) *Déclaration de variable*

Soit D la déclaration $\underline{var}$ x

$$\mathscr{A}(D) : \tilde{\mathbf{D}} = \underline{\mathbf{declvar}}(x)\,.$$

5) *Déclaration de constante*

Soit D la déclaration $\underline{cst}\ x = exp$

$$\mathscr{A}(D) : \tilde{\mathbf{D}} = \underline{\mathbf{diff}}(\mathbf{Eval}(exp), \perp).\underline{\mathbf{declcst}}(x, \mathbf{Eval}(exp)) .$$

6) *Déclaration de procédure*

Soit D la déclaration $\underline{proc}\ f(x : y)\ \gamma$

$$\mathscr{A}(D) : \begin{cases} \tilde{D} = \underline{\mathbf{declproc}}(f, (x : y)\ \gamma) \\ \mathscr{A}(\gamma) . \end{cases}$$

7) *Bloc*

Soit P le bloc $\underline{début}\ D ; P_1\ \underline{fin}$ et notons x l'identificateur défini par la déclaration D (variable, constante ou procédure)

$$\mathscr{A}(P) : \begin{cases} \tilde{\mathbf{P}} = \tilde{\mathbf{D}}.\tilde{\mathbf{P}}_1.\underline{\mathbf{lib}}(x) \\ \mathscr{A}(D) \\ \mathscr{A}(P_1) . \end{cases}$$

On traduit ainsi l'exécution du corps du bloc suivie de la restauration de l'ancien sommet de la pile désignée par x.

8) *Appel de procédure*

Soit P l'appel de procédure $f(exp : res)$. Le corps de la procédure désignée par f au moment de l'appel est une phrase à laquelle est associé un ensemble de calculs. Cet ensemble de calculs est défini lors de la déclaration de procédure (cf. 6) ci-dessus).

La sémantique de l'appel consiste alors essentiellement à introduire cet ensemble de calculs en associant les paramètres effectifs aux paramètres formels correspondants.

Plus précisément si l'on convient que

α représente l'identificateur **entr(valsom(des(**f**)))**

β représente l'identificateur **sort(valsom(des(**f**)))**

γ représente le corps de procédure **corps(valsom(des(**f**)))**

l'ensemble de calculs associé à P est défini par :

$$\mathscr{A}(P) : \begin{cases} \tilde{\mathbf{P}} = \underline{\mathbf{diff}}(\mathbf{Eval}(exp), \perp).\underline{\mathbf{declcst}}(\alpha, \mathbf{Eval}(exp)).\underline{\mathbf{declvar}}(\beta).\tilde{\gamma} \\ .\underline{\mathbf{affect}}(\hat{r}, \beta).\underline{\mathbf{lib}}(\beta).\underline{\mathbf{affect}}(res, \hat{r}).\underline{\mathbf{lib}}(\alpha) . \end{cases}$$

Le lecteur notera l'introduction de l'identificateur standard $\hat{r}$ permettant, comme dans le cas de la sémantique interprétative, d'éviter les « collisions » entre les paramètres effectifs et les paramètres formels.

Exercice 10

En supposant que les identificateurs de procédure sont tous différents dans un même programme, simplifier la définition précédente de $\mathscr{A}(P)$. □

9) *Séquence*

Dans le cas d'une définition interprétative (§ 4.3) on caractérisait l'exécution d'un morceau de programme en mettant en évidence la première instruction à exécuter dans ce morceau ; l'enchaînement séquentiel des diverses instructions en résultait immédiatement. Au contraire, la définition que nous présentons ici caractérise chaque type de phrase indépendamment de ce qui la suit, il est donc nécessaire de définir l'enchaînement séquentiel.

Soit P la phrase P_1 ; P_2 alors

$$\mathscr{A}(P) : \begin{cases} \tilde{\mathbf{P}} = \tilde{\mathbf{P}}_1 . \tilde{\mathbf{P}}_2 \\ \mathscr{A}(P_1) \\ \mathscr{A}(P_2) . \end{cases}$$

L'associativité de la concaténation étend cette définition au cas d'une séquence de longueur quelconque.

5.4 Sémantique calculatoire de NAIN

Soit P un programme NAIN. Le système de calculs qui lui est associé est de la forme :

$$\mathscr{A}(P) \begin{cases} \tilde{\mathbf{P}} = \mathbf{f}_0(\tilde{\mathbf{P}}_1, \tilde{\mathbf{P}}_2, \ldots, \tilde{\mathbf{P}}_\mathbf{k}) \\ \tilde{\mathbf{P}}_1 = \mathbf{f}_1(\tilde{\mathbf{P}}_1, \tilde{\mathbf{P}}_2, \ldots, \tilde{\mathbf{P}}_\mathbf{k}) \\ \vdots \\ \tilde{\mathbf{P}}_\mathbf{k} = \mathbf{f}_\mathbf{k}(\tilde{\mathbf{P}}_1, \tilde{\mathbf{P}}_2, \ldots, \tilde{\mathbf{P}}_\mathbf{k}) . \end{cases}$$

La sémantique calculatoire de P, notée $\mathscr{S}_{\text{cal}}[\![P]\!]$ est la composante relative à $\tilde{\mathbf{P}}$ de la solution minimale du système à point fixe $\mathscr{A}(P)$. On peut encore caractériser l'exécution de P pour un état mémoire initial e_0 (§ 4.4) en précisant que :

i) S'il existe un calcul fini $\mathbf{C} \in \mathscr{S}_{\text{cal}}[\![P]\!]$ tel que $\mathbf{C}(\mathbf{e}_0)$ soit différent de $\mathbf{\Omega}$ on dit que l'exécution du programme P se termine pour $\mathbf{e}_0$ dans l'état $\mathbf{C}(\mathbf{e}_0)$. Les résultats de cette exécution sont exprimés dans $\mathbf{C}(\mathbf{e}_0)$.

ii) Si pour tout calcul fini ou non $\mathbf{C} \in \mathscr{S}_{\text{cal}}[\![P]\!]$ on peut extraire un calcul fini $\mathbf{C}_1$ tel que $\mathbf{C}_1(\mathbf{e}_0) = \mathbf{\Omega}$ l'exécution du programme avorte pour $\mathbf{e}_0$.

iii) S'il existe un calcul infini $\mathbf{C} \in \mathscr{S}_{\text{cal}}[\![P]\!]$ tel que pour tout calcul fini $\mathbf{C}_1$ facteur gauche de $\mathbf{C}$ on ait $\mathbf{C}_1(\mathbf{e}_0) \neq \Omega$ l'exécution du programme boucle pour $\mathbf{e}_0$.

Notons que la définition de $\mathscr{S}_{\text{cal}}$ autorise non seulement la récursivité simple mais également la récursivité croisée (qui est interdite par la syntaxe de NAIN puisqu'un bloc ne peut contenir qu'une seule déclaration).

Exemple 3 : système de calculs associé à un programme NAIN.

Reprenons le programme P :

$$\underline{\textit{début}}\ \underline{\textit{proc}}\ f(x : y)\, y := x + 1 ;$$
$$\underline{\textit{début}}\ \underline{\textit{var}}\ x ; x := 1 ; f(x : x) ; \widehat{\textit{impr}} := x\ \underline{\textit{fin}}\ \underline{\textit{fin}}$$

et posons

$$P_1 : \underline{proc}\ f(x : y)\ y := x + 1$$
$$\gamma\ : y := x + 1$$
$$P_2 : \underline{début}\ \underline{var}\ x\,;\ x := 1\,;\ f(x : x)\,;\ \widehat{impr} := x\ \underline{fin}$$
$$P_3 : x := 1$$
$$P_4 : f(x : x)$$
$$P_5 : \widehat{impr} := x\,.$$

Alors

$$\mathscr{A}(P) : \begin{cases} \tilde{\mathbf{P}} = \tilde{\mathbf{P}}_1 . \tilde{\mathbf{P}}_2 . \underline{\mathbf{lib}}(f) \\ \tilde{\mathbf{P}}_1 = \underline{\mathbf{declproc}}(f,\ (x : y)\ y := x + 1) \\ \tilde{\gamma} = \underline{\mathbf{diff}}(\mathbf{Eval}(x + 1),\ \perp) . \underline{\mathbf{éga}}(\mathbf{typesom}(\mathbf{des}(y)),\ \mathbf{var}) . \\ \qquad \underline{\mathbf{affect}}(y,\ \mathbf{Eval}(x + 1)) \\ \tilde{\mathbf{P}}_2 = \underline{\mathbf{declvar}}(x) . \tilde{\mathbf{P}}_3 . \tilde{\mathbf{P}}_4 . \tilde{\mathbf{P}}_5 . \underline{\mathbf{lib}}(x) \\ \tilde{\mathbf{P}}_3 = \underline{\mathbf{diff}}(\mathbf{Eval}(1),\ \perp) . \underline{\mathbf{éga}}(\mathbf{typesom}(\mathbf{des}(x)),\ \mathbf{var}) . \\ \qquad \underline{\mathbf{affect}}(x,\ \mathbf{Eval}(1)) \\ \tilde{\mathbf{P}}_4 = \underline{\mathbf{diff}}(\mathbf{Eval}(x),\ \perp) . \underline{\mathbf{declcst}}(\alpha,\ \mathbf{Eval}(x)) . \mathbf{declvar}(\beta) . \tilde{\gamma} . \\ \qquad \underline{\mathbf{affect}}(\hat{r},\ \beta) . \underline{\mathbf{lib}}(\beta) . \underline{\mathbf{affect}}(x,\ \hat{r}) . \underline{\mathbf{lib}}(\alpha) \\ \tilde{\mathbf{P}}_5 = \underline{\mathbf{diff}}(\mathbf{Eval}(x),\ \perp) . \underline{\mathbf{éga}}(\mathbf{typesom}(\mathbf{des}(\widehat{impr})),\ \mathbf{var}) . \\ \qquad \underline{\mathbf{affect}}(\widehat{impr},\ \mathbf{Eval}(x)) \end{cases}$$

en utilisant les notations α, β (§ 5.3.8)) pour exprimer l'appel $f(x : x)$. La valeur sémantique du programme P est ainsi réduite au calcul suivant :

$$\mathscr{S}_{\text{cal}}[\![P]\!] = \underline{\mathbf{declproc}}(f,\ (x : y)\ \gamma) . \underline{\mathbf{declvar}}(x) . \tilde{\mathbf{P}}_3 . \tilde{\mathbf{P}}_4 . \tilde{\mathbf{P}}_5 . \underline{\mathbf{lib}}(x) . \underline{\mathbf{lib}}(f)$$

dans lequel $\tilde{\mathbf{P}}_4$ est le calcul :

$$\tilde{\mathbf{P}}_4 = \underline{\mathbf{diff}}(\mathbf{Eval}(x),\ \perp) . \underline{\mathbf{declcst}}(\alpha,\ \mathbf{Eval}(x)) . \underline{\mathbf{declvar}}(\beta) .$$
$$\underline{\mathbf{éga}}(\gamma,\ y := x + 1) . \underline{\mathbf{diff}}(\mathbf{Eval}(x + 1),\ \perp) . \underline{\mathbf{éga}}(\mathbf{typesom}(\mathbf{des}(y)),\ \mathbf{var}) .$$
$$\underline{\mathbf{affect}}(y,\ \mathbf{Eval}(x + 1)) . \underline{\mathbf{lib}}(\beta) . \underline{\mathbf{affect}}(x,\ \hat{r}) . \underline{\mathbf{lib}}(\alpha)\,.$$

On peut alors comparer la suite d'états mémoire définie par ce calcul à celle décrite par la sémantique interprétative. Intuitivement, on souhaite dire que les deux définitions de la sémantique de ce programme sont équivalentes en un certain sens. Plus généralement le lecteur est invité à comparer les définitions i) ii) et iii) ci-dessus aux définitions analogues du paragraphe 4.4. Une telle comparaison est esquissée au paragraphe 8. □

Exercice 11

a) Ecrire le système de calculs associé à l'appel de la procédure factorielle (on supposera qu'il n'y a qu'une seule procédure définie dans le programme où apparaît cette déclaration).

b) Quel est le calcul provoqué par l'appel de cette procédure pour n = 3 ? Même question pour n quelconque. □

Remarques

1) L'utilisation de piles pour rendre compte des itérations et des procédures récursives peut être évitée en introduisant une nouvelle fonction de base τ de $\mathscr{P}(\mathscr{M}^{\infty})$ dans lui-même qui a pour rôle de transformer tout identificateur local (au corps d'une itération ou d'une procédure récursive) en un nouvel identificateur. Cette technique permet un allégement de la structure d'information associée au langage.

2) Comme dans le cas de la méthode interprétative, la fonction **Eval** n'a pas été précisée. On pourrait la définir de la manière indiquée à l'exercice 4 ; **Eval** devient alors une fonction d'accès de **EXP** dans **VA.** Pour permettre d'exprimer plus spécifiquement les propriétés caractérisant les différents accès et pour définir précisément les modifications élémentaires il est agréable de se placer dans un cadre plus formel : les fonctions d'accès sont remplacées par des symboles fonctionnels, les compositions étant représentées par des schémas fonctionnels. Les diverses propriétés des accès, et en particulier les égalités entre accès, sont exprimées sous forme de théorèmes portant sur les schémas fonctionnels dans un certain système formel caractérisant la structure d'information du langage. Ceci permet en particulier d'introduire les phrases du langage pivot dans la structure d'information sous forme de schémas fonctionnels (rôle que jouent un peu les accès **entr, sort, corps** précédents). On peut alors rendre compte de langages pour lesquels une phrase peut à la fois décrire des traitements et fournir une valeur. □

6 SÉMANTIQUE FONCTIONNELLE OU DÉNOTATIONNELLE

Si l'on admet qu'un programme est une spécification de calculs, alors la méthode calculatoire est probablement la plus claire et la plus précise. Elle paraît certainement la mieux adaptée à la description d'un compilateur. Mais si l'on considère qu'un programme définit une fonction, on peut essayer de définir directement celle-ci ; il est probable qu'en se plaçant sur ce terrain, on peut s'éloigner de façon appréciable d'une hypothétique implantation, c'est ce que cherche à faire la méthode fonctionnelle qui va être exposée. Ainsi ce paragraphe a deux objectifs :

— D'une part, il tend à présenter une sémantique directe ou standard, c'est-à-dire une sémantique qui se place le plus loin possible d'une implantation du langage de programmation sur une machine fictive ou réelle. En effet, nous allons tenter d'associer directement au texte ou à une partie du texte d'un programme la fonction décrite par ce programme.

— D'autre part, il vise à initier le lecteur aux outils et aux concepts mathématiques de ce que divers auteurs appellent sémantique mathématique ou dénotationnelle ; elle est mathématique, parce que l'image du programme est une fonction mathématique définie sur des ensembles mathématiquement structurés, des treillis ; elle est dénotationnelle, parce que la signification qu'elle associe à chaque notation (un programme, une notation d'entier) est appelée sa dénotation (c'est une fonction, un entier).

Avant de donner la sémantique des instructions d'un programme, nous allons expliciter la sémantique des expressions et avant tout celle des entiers.

6.1 Sémantique des entiers

Habituellement, si l'on pense à un entier, on prend bien garde de distinguer d'une part son expression dans une numération donnée et un système de conventions donné qui est une suite de chiffres et de symboles et d'autre part le nombre qui lui est associé qui est une entité mathématique.

Soit **N** l'ensemble constitué des entiers notés **n** et d'un élément indéfini noté $\perp_{\mathbf{N}}$; **N** est ordonné au sens du chapitre 2 (voir figure 12); cette structure permet de tenir compte d'erreurs dans le calcul comme la division par zéro, de plus elle se révèle très utile lors de l'utilisation du point fixe qui définit la fonction calculée par une expression récursive.

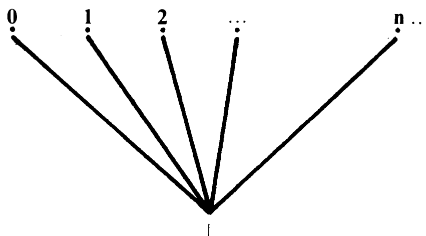


Figure 12 Le domaine des entiers.

La valeur sémantique $\mathcal{V}[\![n]\!]$ d'une notation n d'entier est donc un élément de **N**. On peut avoirs diverses notations pour le même entier, ainsi on pourrait écrire :

$$\begin{aligned}\mathcal{V}[\![106]\!] &= \mathcal{V}[\![1.06\,E + 2]\!] = \mathcal{V}[\![1101010\,B]\!] \\ &= \mathcal{V}[\![CVI]\!] = \mathcal{V}[\![cent\ six]\!] = \mathbf{106}\end{aligned}$$

suivant que l'on note cet entier en format décimal fixe ou flottant, en binaire, en caractères romains, en français (NAIN n'emploie que la première notation).

6.2 Sémantique des expressions

a) Introduction

Dans un langage de programmation, la sémantique d'une expression arithmétique fait correspondre un entier à un élément syntaxique notant une expression et décrit ici par *exp*. La « valeur » de l'expression dépend des valeurs associées aux identificateurs là où l'expression est évaluée. Cette dépendance peut se faire de deux manières, la première est directe, c'est le cas où l'identificateur est déclaré comme une constante, la deuxième est indirecte c'est le cas où l'identificateur est déclaré comme une variable : à l'identificateur, on associe d'abord une place en mémoire, puis à la place en mémoire on associe une valeur.

Par conséquent, nous avons besoin de définir deux domaines importants.

— L'environnement **e** est une fonction qui à un identificateur associe soit une valeur (cas d'une constante) soit une place en mémoire (cas d'une variable) soit une fonction (cas d'une procédure), ainsi le domaine des environnements est noté **Env,** celui des places **Pl,** celui des procédures **Pr,** celui des identificateurs **Id** ; nous écrivons :

$$\mathbf{Env} = \mathbf{Id} \to (\mathbf{Va} \oplus \mathbf{Pl} \oplus \mathbf{Pr})$$

$\oplus$ est la fonction qui fait la réunion disjointe de deux domaines ordonnés en confondant les éléments indéfinis.

Les domaines que nous notons **Va, Id** et **Pr** sont déduits respectivement des ensembles **VA, ID** et **PR** (§ 4.2a)) en leurs adjoignant un élément moins défini puis en les munissant d'une relation d'ordre.

— Le contenu mémoire **m** est une fonction qui à une place **p** fait correspondre la valeur contenue en **p**. Remarquons que la valeur **valsom**(**des**(x)) définie au paragraphe 4.2 coïncide avec **e**(x) si x est une constante et avec **m**(**e**(x)) si x est une variable.

Nous avons besoin par la suite de plusieurs fonctions définies à partir de domaines $\mathbf{D}$, $\mathbf{D}_1, \ldots, \mathbf{D}_i, \ldots$ quelconques. **B** est le domaine ordonné $\{$**vrai, faux,** $\perp_{\mathbf{B}}\}$.

— La fonction **si** ... **alors** ... **sinon** ... **fsi** de $\mathbf{B} \times \mathbf{D} \times \mathbf{D}$ vers **D** : **si b alors d sinon d′ fsi** prend la valeur **d** si **b** est **vrai,** la valeur **d′** si **b** est **faux** et la valeur $\perp_{\mathbf{B}}$ si **b** est $\perp_{\mathbf{B}}$.

On utilise plus généralement la fonction **cas** quand il y a plus de deux possibilités.

— La fonction $\varepsilon \mathrm{D}_i$, notée de manière post-fixée, de $\mathbf{D}_1 \oplus \cdots \oplus \mathbf{D}_n$ vers **B** ; à **d** elle associe **vrai** si **d** est dans $\mathbf{D}_i$, **faux** si **d** n'est pas dans $\mathbf{D}_i$ et $\perp_{\mathbf{B}}$ si **d** est indéfini.

Remarque : La sémantique se définit par de multiples compositions de fonctions ; l'écriture devient très lourde si on laisse les parenthèses dans ces compositions, aussi dans la suite nous ne maintiendrons les parenthèses que dans les cas où elles aident à la compréhension. □

b) *Définition*

Soit *Exp* le domaine des expressions, la sémantique d'une expression *exp* est la valeur d'une fonction $\mathscr{E}$ en *exp* ; $\mathscr{E}(exp)$ associe à chaque environnement une fonction des contenus-mémoire vers les valeurs, ainsi $\mathscr{E}$ est un élément du domaine :

$$Exp \to (\mathbf{Env} \to (\mathbf{Mem} \to \mathbf{Va})) .$$

Dans le cas d'un telle disposition des parenthèses nous écrirons plus simplement :

$$Exp \to \mathbf{Env} \to \mathbf{Mem} \to \mathbf{Va} .$$

Notons que nous aurions pu dire que $\mathscr{E}$ fait correspondre à l'expression *exp*,

à un environnement et à un contenu, une valeur ; autrement dit que $\mathscr{E}$ appartient au domaine :

$$(\mathbf{Exp} \otimes \mathbf{Env} \otimes \mathbf{Mem}) \to \mathbf{Va}$$

où $\otimes$ désigne le produit cartésien.

En fait ces domaines sont isomorphes et nous les confondons par la suite.

$\mathscr{E}$ est définie par cas :

— Cas où le contenu est $\perp_{\mathbf{Mem}}$

$$\mathscr{E}[\![exp]\!]\ \mathbf{e}\ \perp_{\mathbf{Mem}} = \perp_{\mathbf{N}}\ .$$

La sémantique de l'expression *exp* est alors indéfinie.

— Cas où l'expression est la notation d'un entier

$$\mathscr{E}[\![n]\!]\ \mathbf{em} = \mathscr{V}[\![n]\!]\ .$$

La sémantique est la valeur de cette notation d'entier, elle est indépendante de tout environnement et de tout contenu.

— Cas où l'expression est réduite à un identificateur :

$$\begin{aligned}\mathscr{E}[\![id]\!]\ \mathbf{em} = \ & \underline{\mathbf{cas}}\ \mathbf{e}[\![id]\!]\ \varepsilon\ \mathbf{Va}\ \underline{\mathbf{alors}}\ \mathbf{e}[\![id]\!]\ ;\\ & \quad \mathbf{e}[\![id]\!]\ \varepsilon\ \mathbf{Pl}\ \underline{\mathbf{alors}}\ \mathbf{me}[\![id]\!]\ ;\\ & \quad \mathbf{e}[\![id]\!]\ \varepsilon\ \mathbf{Pr}\ \underline{\mathbf{alors}}\ \perp_{\mathbf{Va}}\\ & \underline{\mathbf{fcas}}\end{aligned}$$

On a considéré trois cas suivant que $\mathbf{e}[\![id]\!]$ appartient à **Va,** à **Pl,** à **Pr,** les valeurs associées sont alors $\mathbf{e}[\![id]\!]$, $\mathbf{me}[\![id]\!]$ et $\perp_{\mathbf{Va}}$ respectivement ; la valeur $\perp_{\mathbf{Va}}$ dans le cas d'un identificateur de procédure correspond au fait qu'on ne peut pas attribuer un sens à un identificateur de procédure figurant dans une expression.

— Cas où l'expression est structurée (exemple de +)

$$\mathscr{E}[\![exp1 + exp2]\!]\ \mathbf{em} = \mathscr{E}[\![exp1]\!]\ \mathbf{em} + \mathscr{E}[\![exp2]\!]\ \mathbf{em}\ .$$

6.3 Une première sémantique des instructions

Définir la sémantique des instructions c'est définir une fonction $\mathscr{S}$ du domaine des instructions vers celui des fonctions des environnements vers les transformations d'états. $\mathscr{S}$ appartient donc au domaine

$$\mathbf{Inst} \to \mathbf{Env} \to \mathbf{Mem} \to \mathbf{Mem}$$

ou encore au domaine

$$(\mathbf{Inst} \otimes \mathbf{Env} \otimes \mathbf{Mem}) \to \mathbf{Mem}\ .$$

Par conséquent, la forme générale d'une définition sémantique d'instruction est $\mathscr{S}[\![i]\!]\ \mathbf{em} = \mathbf{m}'$.

a) Sémantique de l'affectation

Si l'affectation est de la forme $x := exp$, la sémantique de l'affectation dans l'environnement **e** est la transformation du contenu mémoire consistant à mettre à la place de $\mathbf{e}[\![x]\!]$ la valeur de *exp*

$$\mathcal{S}[\![x := exp]\!]\ \mathbf{em} = \underline{\mathbf{si}}\ \mathbf{e}[\![x]\!]\ \varepsilon\mathbf{Pl}\ \underline{\mathbf{alors}}\ \mathbf{m}[\mathbf{e}[\![x]\!]/\mathcal{E}[\![exp]\!]\ \mathbf{em}]\ .$$

$$\underline{\mathbf{sinon}}\ \bot_{\mathbf{Mem}}\ \underline{\mathbf{fsi}}\ .$$

Notons ici que $\mathbf{m}[x/\mathbf{y}]$ désigne la fonction qui à x associe $\mathbf{y}$ et à z différent de x associe $\mathbf{m}(z)$; on doit rapprocher cette substitution qui transforme la fonction **m** ou contenu mémoire, de la modification de la structure d'information $\underline{\mathbf{affect}}(x, \mathbf{v})$ définie au paragraphe 4.2b).

b) Sémantique d'une déclaration de variable

La sémantique d'un bloc commençant par une déclaration de x a pour effet d'appliquer le reste du bloc à l'environnement modifié en associant à x une place non allouée donnée par $\mathbf{a}(\mathbf{e})$

$$\mathcal{S}[\![\underline{\textit{début}}\ \underline{\textit{var}}\ x\ ;\ P\ \underline{\textit{fin}}]\!]\ \mathbf{em} = \mathcal{S}[\![P]\!]\ [\mathbf{e}[x/\mathbf{ae}]\ \mathbf{m}]\ .$$

Nous ne précisons pas ici comment trouver une place non allouée, une solution consiste à ordonner l'ensemble des places et à connaître à tout moment la première place non allouée.

Notons, pour le lecteur habitué à penser en termes d'exécution, qu'il n'y a pas « libération » de la place occupée à la fin du bloc, cela s'explique par la sémantique donnée pour la composition séquentielle.

c) Sémantique de la composition séquentielle

$$\mathcal{S}[\![P\ ;\ Q]\!]\ \mathbf{em} = \mathcal{S}[\![Q]\!]\ \mathbf{e}\mathcal{S}[\![P]\!]\ \mathbf{em}\ .$$

On applique $\mathcal{S}[\![Q]\!]$ au même environnement que $\mathcal{S}[\![P]\!]$, ainsi, on ne tient pas compte des déclarations faites dans les blocs intérieurs à P mais seulement de l'environnement tel qu'il est avant l'élaboration de P.

d) Sémantique de la conditionnelle

$$\mathcal{S}[\![\underline{\textit{si}}\ exp\ \underline{\textit{alors}}\ P\ \underline{\textit{sinon}}\ Q\ \underline{\textit{fsi}}]\!]\ \mathbf{em}$$
$$= \underline{\mathbf{si}}\ \mathcal{E}[\![exp]\!]\ \mathbf{em}\ \underline{\mathbf{alors}}\ \mathcal{S}[\![P]\!]\ \mathbf{em}\ \underline{\mathbf{sinon}}\ \mathcal{S}[\![Q]\!]\ \mathbf{em}\ \underline{\mathbf{fsi}}\ .$$

e) Sémantique de l'itération

Notant $\mu(\mathrm{F})$ le plus petit point fixe d'une fonction continue F (voir chapitre 2, § 2.2) nous poserons :

$$\mathcal{S}[\![\underline{\textit{tant que}}\ exp\ \underline{\textit{faire}}\ P\ \underline{\textit{fait}}]\!]\ \mathbf{em} =$$
$$\mu(\lambda\mathbf{f} : [\mathbf{Mem} \to \mathbf{Mem}]\ \{\ \underline{\mathbf{si}}\ \mathcal{E}[\![exp]\!]\ \mathbf{em}\ \underline{\mathbf{alors}}\ \mathcal{S}[\![P]\!]\ \mathbf{ef}(\mathbf{m})\ \underline{\mathbf{sinon}}\ \mathbf{m}\ \underline{\mathbf{fsi}}\ \})\ .$$

La sémantique d'une itération, appliquée à un environnement **e** est le plus petit point fixe d'une certaine fonctionnelle indiquée entre les parenthèses. Notons que cette définition est faite par induction sur la structure du pro-

gramme, c'est-à-dire qu'elle ne fait appel qu'à la sémantique d'éléments plus simples du texte, ici ces éléments sont *exp* et *P* ; ceci peut permettre de faire des raisonnements par induction sur le texte du programme qu'une définition circulaire du genre suivant n'aurait pas permis :

$$\mathcal{S}[\![\underline{tant\ que}\ exp\ \underline{faire}\ P\ \underline{fait}]\!]\ \mathbf{em} = \mathbf{si}\ \mathcal{E}[\![exp]\!]\ \mathbf{em}\ \underline{\mathbf{alors}}\ \mathcal{S}[\![i\,;\underline{tant\ que}\ exp\ \underline{faire}\ P\ \underline{fait}]\!]\ \mathbf{em}\ \underline{\mathbf{sinon}}\ \mathbf{em}\ \underline{\mathbf{fsi}}\,.$$

En effet cette dernière définition accroît la complexité du texte ; dans ce cas, une preuve doit s'appuyer sur la longueur (si elle existe !) des calculs, elle est alors plus complexe. On est en droit de se demander si une telle définition circulaire est une véritable définition, en effet n'aurait-on pas repoussé au niveau de la fonction $\mathcal{S}$ les problèmes posés par la récursivité ?

Exercice 12

Quelle est la sémantique de l'instruction vide ? □

f) Sémantique d'une déclaration de constante

Ce n'est guère plus difficile qu'une déclaration de variable ; on applique la sémantique $\mathcal{S}[\![P]\!]$ du corps du bloc à l'environnement modifié par $\mathcal{E}[\![exp]\!]$ en x ;

$$\mathcal{S}[\![\underline{début}\ \underline{cst}\ x = exp\,;P\ \underline{fin}]\!]\ \mathbf{em} = \mathcal{S}[\![P]\!]\ \mathbf{e}[x/\mathcal{E}[\![exp]\!]\ \mathbf{em}]\ \mathbf{m}\,.$$

g) Sémantique des procédures

Nous devons envisager ici la déclaration et l'appel.

— La déclaration de f a pour effet de calculer comme un plus petit point fixe la fonction associée à f et de placer cette fonction dans l'environnement, contrairement aux paragraphes 4 et 5 dans lesquels on stockait le texte de la procédure.

— L'appel $f(exp : y)$ a pour effet d'appliquer à $\mathcal{E}[\![exp]\!]$ la fonction $\mathbf{e}[\![f]\!]$ qui se trouve dans l'environnement **e** et de ranger le résultat à la place $\mathbf{e}[\![y]\!]$.

Examinons d'abord la signification d'un appel

$$\mathcal{S}[\![f(exp : y)]\!]\ \mathbf{em} = \underline{\mathbf{si}}\ \{\ \mathbf{e}[\![f]\!]\ \varepsilon\ \mathbf{Pr}\ \underline{\mathbf{et}}\ \mathbf{e}[\![y]\!]\ \varepsilon\ \mathbf{Pl}\ \}\ \underline{\mathbf{alors}}\ \mathbf{m}[\mathbf{e}[\![y]\!]/\mathbf{e}[\![f]\!]\ (\mathcal{E}[\![exp]\!]\ \mathbf{em})]\ \underline{\mathbf{sinon}}\ \bot_{\mathbf{Mem}}\ \underline{\mathbf{fsi}}\,.$$

La signification d'une déclaration est :

$$\mathcal{S}[\![\underline{début}\ \underline{proc}\ f(x : y)\ Q\,;P\ \underline{fin}]\!]\ \mathbf{em} = \mathcal{S}[\![P]\!]\ \mathbf{e'}\ \mathbf{m}$$

avec

$$\mathbf{e'} = \mathbf{e}[f/\mu(\lambda\mathbf{g} : \mathbf{Va} \to \mathbf{Va}\ \lambda\mathbf{v}\ \{(\mathcal{S}[\![Q]\!]\ \mathbf{e''}\ \bot_{\mathbf{Mem}})\ \mathbf{e''}[\![y]\!]\})]$$

dans lequel e″ signifie :

$$\mathbf{e''} = \mathbf{e}[f/\mathbf{g}]\ [x/\mathbf{v}]\ [y/\mathbf{ae}]\,.$$

Intuitivement, déclarer la procédure $f(x : y)$ consiste à :
i) expliciter le plus petit point fixe de l'équation

$$y = \lambda v \{ \mathscr{S}[\![Q]\!] \mathbf{e}'' \perp_{\mathbf{Mem}} \}$$

ou **e**″ est un nouvel environnement déduit de l'environnement **e** dans lequel la valeur du paramètre effectif v est associée à la constante x et la variable y est initialisée,
ii) créer un nouvel environnement **e**′ déduit de **e** dans lequel on associe à f ce plus petit point fixe.

La valeur finale trouvée dans l'emplacement associé à y sera transmise au paramètre effectif correspondant. On peut donc associer à $f(x : y)$ une fonction $\mathbf{g} : \mathbf{Va} \to \mathbf{Va}$ définie comme un plus petit point fixe.

Notons que ces définitions autorisent une procédure à être récursive, à utiliser une procédure ou une constante précédemment définie de nom différent de f et autorisent une déclaration de procédure à figurer en tête de bloc ; mais elles n'autorisent pas la récursivité croisée et les variables globales dans les procédures.

Exercice 13

Indiquer ce qui dans la définition régit ces différentes autorisations ou interdictions. □

6.4 Une autre approche de la sémantique des instructions : les continuations

Si l'on veut autoriser les branchements dans un programme, notamment les branchements au-delà du point où l'on se trouve et si l'on veut en rendre compte en sémantique fonctionnelle, il faut que la signification d'un composant de programme ne dépende pas seulement de ce qui « s'est passé » avant et qui apparaît dans l'environnement, elle doit aussi prévoir ce qui se « passera » après ce morceau de programme. C'est-à-dire qu'on doit envisager les **continuations** de morceau de programme et considérer que la sémantique d'un morceau de programme est une modification, définie à partir d'un environnement donné, des continuations possibles d'un programme. Partant des dernières instructions on arrive de modifications en modifications, à définir la sémantique du programme entier.

Maintenant il nous reste à donner un sens à ce qu'est une continuation d'un programme ; c'est une fonction **c** qui à un état mémoire **s** (l'état de la mémoire où l'on se trouve) associe un état mémoire $\mathbf{s}_f$ (l'état final correspondant à l'état **s** pour la continuation **c** considérée). Ainsi le domaine **Cont** des continuations est défini par :

$$\mathbf{Cont} = \mathbf{Mem} \to \mathbf{Mem}\,.$$

Nous n'allons examiner ici que deux constructions, le lecteur trouvera à la fin de ce sous-chapitre la définition complète de la nouvelle sémantique $\mathscr{C}$ ainsi définie. $\mathscr{C}$ appartient à

$$\mathbf{Inst} \to \mathbf{Env} \to \mathbf{Cont} \to \mathbf{Mem} \to \mathbf{Mem} = \mathbf{Inst} \to \mathbf{Env} \to \mathbf{Cont} \to \mathbf{Cont}\,.$$

a) *Sémantique de la composition séquentielle par les continuations*

$$\mathscr{C}[\![P\,;Q]\!]\,\mathbf{ecm} = \mathscr{C}[\![P]\!]\,\mathbf{e}(\mathscr{C}[\![Q]\!]\,\mathbf{ec})\,\mathbf{m}\,.$$

On remarque que la composition des fonctions ne se fait pas dans le même ordre que lors de la définition de $\mathscr{S}$ (§ 6.3c). En effet, on définit d'abord la continuation ($\mathscr{C}[\![Q]\!]$ **ec**) et on y applique $\mathscr{C}[\![P]\!]$ **e**.

b) Sémantique des branchements

Etendons la syntaxe de NAIN par une instruction de saut notée *aller a*. La syntaxe est modifiée ; DECL est remplacé par DECL′ (§ 3) :

$$DECL' = DECL \cup \underline{eti}\ ID : BLOC$$

et on ajoute le type d'instruction *aller a ID* aux instructions.

Intuitivement cela signifie qu'en tête d'un bloc B on déclare un bloc B' étiqueté (qui traite une erreur par exemple) auquel on peut se brancher par un *aller a* situé dans B ; une fois ce bloc B' exécuté on sort du bloc principal B. Le domaine des environnements est modifié ainsi :

$$\mathbf{Env}' = \mathbf{Id} \to [\mathbf{Pr} \oplus \mathbf{Va} \oplus \mathbf{Pl} \oplus \mathbf{Eti}]$$
$$\mathbf{Eti} = \mathbf{Cont}\,.$$

La sémantique de la déclaration d'étiquette est définie ainsi :

$$\mathscr{C}[\![\underline{début}\ \underline{eti}\ x : B' ; P\ \underline{fin}]\!]\ \mathbf{ec} = \mathscr{C}[\![P]\!]\ \mathbf{e}[x/\mathscr{C}[\![B']\!]\ \mathbf{ec}]\ \mathbf{c}\,.$$

On modifie l'environnement en x pour y placer une continuation.

La sémantique du branchement est définie ainsi :

$$\mathscr{C}[\![\underline{aller\ a}\ x]\!]\ \mathbf{ec} = \underline{\mathbf{si}}\ \mathbf{e}[\![x]\!]\ \varepsilon\ \mathbf{Eti}\ \underline{\mathbf{alors}}\ \mathbf{e}[\![x]\!]\ \underline{\mathbf{sinon}}\ \bot_{\mathbf{Cont}}\ \underline{\mathbf{fsi}}\,.$$

La valeur par $\mathscr{C}$ de l'instruction *aller a x* est la continuation qui se trouve en x.

Notons que $\mathscr{C}[\![\underline{aller\ a}\ x ; P]\!] = \mathscr{C}[\![\underline{aller\ a}\ x]\!]$.

6.5 Sémantique des programmes

Nous choisissons de définir la sémantique d'un programme comme une fonction qui à des données et une suite d'identificateurs notant les résultats associe les valeurs des résultats correspondant à ces identificateurs.

Posons : $\mathbf{Don} = [\mathbf{Id} \times \mathbf{Va}]^*$.

La fonction sémantique est la fonction

$$\mathscr{S}_{\text{fonct}} : P \to (\mathbf{Don} \otimes \mathbf{Id}^*) \to \mathbf{Va}^*\,.$$

inenv : $[\mathbf{Don} \otimes \mathbf{Id}^*] \to \mathbf{Env}$ est une fonction qui initialise l'environnement en allouant des places aux différents identificateurs de la donnée et ensuite en allouant des places aux identificateurs qui apparaissent dans les résultats mais pas dans les données.

inmem : $\mathbf{Don} \to \mathbf{Mem}$ est une fonction qui remplit les places allouées par **inenv** par les valeurs associées aux identificateurs dans la donnée.

res : $[\mathbf{Env} \otimes \mathbf{Mem} \otimes \mathbf{Id}^*] \to \mathbf{Val}^*$ est une fonction qui à un environnement, un contenu mémoire et une liste d'identificateurs fait correspondre la liste des résultats associés à ces identificateurs.

Nous ne donnons pas la définition formelle de **inenv, inmem** et **res.** On a alors

$$\mathscr{S}_{\text{fonct}}[\![P]\!]\ \mathbf{dy}^* = \mathbf{res}\ \{\ \mathbf{inenv}\ \mathbf{dy}^*\}\ \{\ \mathscr{S}[\![P]\!]\ (\mathbf{inenv}\ \mathbf{dy}^*)\ (\mathbf{inmem}\ \mathbf{d})\ \}\ \mathbf{y}^*$$

Id		identificateurs
Env	= **Id** → [**Va** ⊕ **Pl** ⊕ **Pr**]	environnement
Mem	= **Pl** → **Va**	contenu mémoire
Va	= **N** ⊕ **B**	valeurs de base
Pr	= **Va** → **Va**	procédures
Inst	= [**Id** ⊗ **Exp**]	affectation
	⊕ [**Inst** ⊗ **Inst**]	composition
	⊕ [**Exp** ⊗ **Inst** ⊗ **Inst**]	conditionnelle
	⊕ [**Exp** ⊗ **Inst**]	itération
	⊕ [**Id** ⊗ **Exp** ⊗ **Id**]	appel de procédure
	⊕ [**Id** ⊗ **Inst**]	déclaration de variable
	⊕ [**Id** ⊗ **Exp** ⊗ **Inst**]	déclaration de constante
	⊕ [**Id** ⊗ **Id** ⊗ **Id** ⊗ **Inst** ⊗ **Inst**]	déclaration de procédure

Figure 13 Domaines sémantiques et syntaxiques pour $\mathscr{S}$.

Inst′	= **Inst** ⊕ (**Id** ⊗ **Inst** ⊗ **Inst**)	déclaration d'étiquette
	⊕ **Id**	branchement
Cont	= **Mem** → **Mem**	continuations

Figure 14 Domaines sémantiques et syntaxiques pour $\mathscr{C}$.

$$\mathscr{S}\llbracket x := exp \rrbracket \, \mathbf{em} = \underline{\mathbf{si}}\ \mathbf{e}\llbracket x \rrbracket\ \varepsilon\ \mathbf{Pl}\ \underline{\mathbf{alors}}\ \mathbf{m}[\ \mathbf{e}\llbracket x \rrbracket / \langle\ \mathscr{E}\llbracket exp \rrbracket\ \mathbf{em}\ \rangle]\ \underline{\mathbf{sinon}}\ \bot_{\mathbf{Mem}}\ \underline{\mathbf{fsi}}$$

$$\mathscr{S}\llbracket P ; Q \rrbracket\ \mathbf{em} = \mathscr{S}\llbracket Q \rrbracket\ \mathbf{e}\mathscr{S}\llbracket P \rrbracket\ \mathbf{em}$$

$$\mathscr{S}\llbracket \underline{si}\ exp\ \underline{alors}\ P\ \underline{sinon}\ Q\ \underline{fsi} \rrbracket\ \mathbf{em} = \underline{\mathbf{si}}\ \mathscr{E}\llbracket exp \rrbracket\ \mathbf{em}\ \underline{\mathbf{alors}}\ \mathscr{S}\llbracket P \rrbracket\ \mathbf{em}\ \underline{\mathbf{sinon}}\ \mathscr{S}\llbracket Q \rrbracket\ \mathbf{em}\ \underline{\mathbf{fsi}}$$

$$\mathscr{S}\llbracket \underline{tant\ que}\ exp\ \underline{faire}\ P\ \underline{fait} \rrbracket\ \mathbf{em} = \mu(\lambda \mathbf{f} : [\mathbf{Mem} \to \mathbf{Mem}]\{\ \underline{\mathbf{si}}\ \mathscr{E}\llbracket exp \rrbracket\ \mathbf{em}\ \underline{\mathbf{alors}}\ \mathscr{S}\llbracket P \rrbracket\ \mathbf{efm}\ \underline{\mathbf{sinon}}\ \mathbf{m}\ \underline{\mathbf{fsi}}\ \})$$

$$\mathscr{S}\llbracket f(exp : y) \rrbracket\ \mathbf{em} = \underline{\mathbf{si}}\ \{\ \mathbf{e}\llbracket f \rrbracket\ \varepsilon\ \mathbf{Pr}\ \underline{\mathbf{et}}\ \mathbf{e}\llbracket y \rrbracket\ \varepsilon\ \mathbf{Pl}\ \}\ \underline{\mathbf{alors}}\ \mathbf{m}[\mathbf{e}\llbracket y \rrbracket / \mathbf{e}\llbracket f \rrbracket\ (\mathscr{E}\llbracket exp \rrbracket\ \mathbf{em})]\ \underline{\mathbf{sinon}}\ \bot_{\mathbf{Mem}}\ \underline{\mathbf{fsi}}$$

$$\mathscr{S}\llbracket \underline{début}\ \underline{var}\ x ; P\ \underline{fin} \rrbracket\ \mathbf{em} = \mathscr{S}\llbracket P \rrbracket\ \mathbf{e}[\mathbf{x}/\mathbf{ae}]\ \mathbf{m}$$

$$\mathscr{S}\llbracket \underline{début}\ \underline{cst}\ x = exp ; P\ \underline{fin} \rrbracket\ \mathbf{em} = \mathscr{S}\llbracket P \rrbracket\ \mathbf{e}[\mathbf{x}/\mathscr{E}\llbracket exp \rrbracket\ \mathbf{em}]\ \mathbf{m}$$

$$\mathscr{S}\llbracket \underline{début}\ \underline{proc}\ f(x : y)\ B ; P\ \underline{fin} \rrbracket\ \mathbf{em} = \mathscr{S}\llbracket P \rrbracket\ \mathbf{e}'\ \mathbf{m}$$

où

$$\mathbf{e}' = \mathbf{e}[f/\mu(\lambda \mathbf{g} : [\mathbf{Va} \to \mathbf{Va}].\lambda \mathbf{v}\ \{\ (\mathscr{S}\llbracket B \rrbracket\ \mathbf{e}''\ \bot_{\mathbf{Mem}})\ \mathbf{e}''\llbracket y \rrbracket\ \})]$$

où

$$\mathbf{e}'' = \mathbf{e}[f/\mathbf{g}]\ [x/\mathbf{v}]\ [y/\mathbf{ae}] .$$

Figure 15 Définition de la fonction sémantique $\mathscr{S}$.

$$\mathscr{C}\llbracket x : exp \rrbracket\ \mathbf{ecm} = \underline{\mathbf{si}}\ \mathbf{e}\llbracket x \rrbracket\ \varepsilon\ \mathrm{Pl}\ \underline{\mathbf{alors}}\ \mathbf{cm}[\mathbf{e}\llbracket x \rrbracket / \mathscr{E}\llbracket exp \rrbracket\ \mathbf{em}]\ \underline{\mathbf{sinon}}\ \bot_{\mathbf{Mem}}\ \underline{\mathbf{fsi}}$$

$$\mathscr{C}\llbracket P ; Q \rrbracket\ \mathbf{ecm} = \mathscr{C}\llbracket P \rrbracket\ \mathbf{e}(\mathscr{C}\llbracket Q \rrbracket\ \mathbf{ec})\ \mathbf{m}$$

$$\mathscr{C}\llbracket \underline{si}\ exp\ \underline{alors}\ P\ \underline{sinon}\ Q\ \underline{fsi} \rrbracket\ \mathbf{ecm} = \underline{\mathbf{si}}\ \mathscr{E}\llbracket exp \rrbracket\ \mathbf{em}\ \underline{\mathbf{alors}}\ \mathscr{C}\llbracket P \rrbracket\ \mathbf{ecm}\ \underline{\mathbf{sinon}}\ \mathscr{C}\llbracket Q \rrbracket\ \mathbf{ecm}\ \underline{\mathbf{fsi}}$$

$$\mathscr{C}\llbracket \underline{tant\ que}\ exp\ \underline{faire}\ P\ \underline{fait} \rrbracket\ \mathbf{e} = \mu(\lambda \mathbf{f} : [\mathbf{Cont} \to \mathbf{Cont}]\ \{\ \lambda \mathbf{c}.\lambda \mathbf{s}\ \underline{\mathbf{si}}\ \mathscr{E}\llbracket exp \rrbracket\ \mathbf{em}\ \underline{\mathbf{alors}}\ \mathscr{C}\llbracket P \rrbracket\ \mathbf{ef(c)}\ \mathbf{m}\ \underline{\mathbf{sinon}}\ \mathbf{cm}\ \underline{\mathbf{fsi}}\ \})$$

$$\mathscr{C}\llbracket f(exp : y) \rrbracket\ \mathbf{ecm} = \underline{\mathbf{si}}\ \{\ \mathbf{e}\llbracket f \rrbracket\ \varepsilon\ \mathbf{Pr} \wedge \mathbf{e}\llbracket y \rrbracket\ \varepsilon\ \mathbf{Pl}\ \}\ \underline{\mathbf{alors}}\ \mathbf{c}(\mathbf{m}[\mathbf{e}\llbracket y \rrbracket / \mathbf{e}]\llbracket f \rrbracket\ \mathscr{E}\llbracket exp \rrbracket\ \mathbf{em})\ \underline{\mathbf{sinon}}\ \bot_{\mathbf{Mem}}\ \underline{\mathbf{fsi}}$$

$$\mathscr{C}\llbracket \underline{début}\ \underline{cst}\ x = exp ; P\ \underline{fin} \rrbracket\ \mathbf{ecm} = \mathscr{C}\llbracket P \rrbracket\ \mathbf{e}[x/\mathscr{E}\llbracket exp \rrbracket\ \mathbf{em}]\ \mathbf{cm}$$

$$\mathscr{C}\llbracket \underline{début}\ \underline{var}\ x ; P\ \underline{fin} \rrbracket\ \mathbf{ecm} = \mathscr{C}\llbracket P \rrbracket\ \mathbf{e}[x/\mathbf{ae}]\ \mathbf{cm}[\mathbf{ae}/\bot_{\mathbf{Va}}]$$

$$\mathscr{C}\llbracket \underline{début}\ \underline{proc}\ f(x : y)\ B ; P \rrbracket\ \mathbf{ecm} = \mathscr{C}\llbracket P \rrbracket\ \mathbf{e}'\ \mathbf{cm}$$

où

$$\mathbf{e}' = \mathbf{e}[f/\mu(\lambda \mathbf{g}[\mathbf{Va} \to \mathbf{Va}]\ \lambda \mathbf{v} : \mathbf{Va}(\{\ \llbracket B \rrbracket\ \mathbf{e}''(\lambda \mathbf{m}.\mathbf{m})\ \bot_{\mathbf{Mem}}\ \}\ \mathbf{e}''\llbracket y \rrbracket))]$$

et où

$$\mathbf{e}'' = \mathbf{e}[f/\mathbf{g}]\ [x/\mathbf{v}]\ [y/\mathbf{ae}]$$

$$\mathscr{C}\llbracket \underline{début}\ \underline{eti}\ x : B' ; P\ \underline{fin} \rrbracket\ \mathbf{ec} = \mathscr{C}\llbracket P \rrbracket\ \mathbf{e}[x/\mathscr{C}\llbracket B' \rrbracket\ \mathbf{ec}]\ \mathbf{c}$$

$$\mathscr{C}\llbracket \underline{aller\ a}\ x \rrbracket\ \mathbf{ec} = \underline{\mathbf{si}}\ \mathbf{e}\llbracket x \rrbracket\ \varepsilon\ \mathbf{Eti}\ \underline{\mathbf{alors}}\ \mathbf{e}\llbracket x \rrbracket\ \underline{\mathbf{sinon}}\ \bot_{\mathbf{Mem}}\ \underline{\mathbf{fsi}}$$

Figure 16 Définition de la sémantique de $\mathscr{C}$.

6.6 Etude d'un exemple

Nous allons reprendre ici l'exemple déjà étudié aux paragraphes 4 et 5.
Nous supposerons que l'environnement **e** vérifie $\mathbf{e}[\![\widehat{impr}]\!] = \mathbf{p}_0$

$$\mathcal{S}[\![\underline{début}\ \underline{proc}\ f(x : y)\ y := x + 1\ ;\ \underline{début}\ \underline{var}\ x\ ;\ x := 1\ ;\ f(x : x)\ ,\ \widehat{impr} := x\ \underline{fin}\ \underline{fin}]\!]\ \mathbf{em}\ .$$

Comme la fonction $\lambda\mathbf{t}.\mathbf{t} + 1$ est la sémantique de la procédure f on peut écrire :

$$\mathcal{S}[\![\underline{début}\ \underline{var}\ x\ ;\ x := 1\ ;\ f(x : x)\ ;\ \widehat{impr} := x\ \underline{fin}]\!]\ \mathbf{e}[f/\lambda\mathbf{t}.\mathbf{t} + 1].\mathbf{m}$$

soit en posant $\mathbf{p}_1 = \mathbf{ae} = \mathbf{ae}[f/\lambda\mathbf{t}.\mathbf{t} + 1]$

$$= \mathcal{S}[\![x := 1\ ;\ f(x : x)\ ;\ \widehat{impr} := x]\!]\ \mathbf{e}[f/\lambda\mathbf{t}.\mathbf{t} + 1]\ [x/\mathbf{p}_1]\ \mathbf{m}$$
$$= \mathcal{S}[\![f(x : x)\ ;\ \widehat{impr} := x]\!]\ \mathbf{e}[f/\lambda\mathbf{t}.\mathbf{t} + 1]\ [x/\mathbf{p}_1]\ \ \mathbf{m}[\mathbf{e}[f/\lambda\mathbf{t}.\mathbf{t} + 1]\ [x/\mathbf{p}_1]\ [\![x]\!]/1]$$

ce qui se simplifie en

$$= \mathcal{S}[\![f(x : x)\ ;\ \widehat{impr} := x]\!]\ \mathbf{e}[f/\lambda\mathbf{t}.\mathbf{t} + 1]\ [x/\mathbf{p}_1]\ \mathbf{m}[\mathbf{p}_1/1]$$

que nous écrivons directement sous la forme

$$= \mathcal{S}[\![\widehat{impr} := x]\!]\ \mathbf{e}[f/\lambda\mathbf{t}.\mathbf{t} + 1]\ [x/\mathbf{p}_1]\ \mathbf{m}[\mathbf{p}_1/2]$$
$$= \mathcal{S}[\![\quad]\!]\ \mathbf{e}[f/\lambda\mathbf{t}.\mathbf{t} + 1]\ [x/\mathbf{p}_1]\ \mathbf{m}[\mathbf{p}_1/2]\ [\mathbf{p}_0/2]$$
$$= \mathbf{m}[\mathbf{p}_1/2]\ [\mathbf{p}_0/2]\ .$$

En utilisant les résultats du paragraphe 6.5, on montrerait que la sémantique du programme entier appliquée à la donnée $\mathbf{d} = \perp_{\mathbf{Don}}$ et à l'identificateur $\widehat{impr}$ est 2.

Revenons maintenant à la déclaration de procédure

$$\underline{proc}\ f(x : y)\ y := x + 1\ .$$

Avec les notations du paragraphe 6.3 g, on a à « calculer » :

$$\mathcal{S}[\![y := x + 1]\!]\ \mathbf{e}[f/\mathbf{g}]\ [x/\mathbf{v}]\ [y/\mathbf{p}_0]\ \perp_{\mathbf{Mem}} = \perp_{\mathbf{Mem}} [\mathbf{p}_0/\mathbf{v} + 1]$$

et ainsi sachant que $\mathbf{e}''[\![y]\!] = \mathbf{p}_0$

$$\lambda\mathbf{v}.\mathcal{S}[\![y := x + 1]\!]\ \mathbf{e}''\ \perp_{\mathbf{Mem}}\ \mathbf{e}''[\![y]\!] = \lambda\mathbf{v}.\mathbf{v} + 1$$

la fonctionnelle $\lambda\mathbf{g}.\lambda\mathbf{v}.\mathbf{v} + 1$ est constante et a pour point fixe

$$\lambda\mathbf{v}.\mathbf{v} + 1\ .$$

7 SÉMANTIQUE AXIOMATIQUE

Pour savoir ce que fait une instruction ou une construction d'un langage de programmation, il suffit de préciser comment cette instruction ou cette construction transforme un prédicat ; autrement dit cela revient à savoir comment intervient cette instruction ou cette construction dans l'élaboration de la preuve d'un programme.

La sémantique axiomatique consiste à donner pour chaque instruction ou chaque construction, un axiome ou une règle d'inférence. Ce sont les mêmes que ceux décrits au chapitre 4 paragraphes 3.2 et 6 pour la correction partielle, seul change l'objectif : dans un cas définir la sémantique, dans l'autre faire la preuve ; l'utilisation en est exactement la même dans un cas on dira qu'on a défini la signification d'un programme, par la manière dont il transforme les prédicats, dans l'autre on dira qu'on a prouvé que ce programme est partiellement correct par rapport à un prédicat de sortie et un prédicat d'entrée.

7.1 Axiomes et règles d'inférences relatives à NAIN

Nous allons simplement donner pour NAIN les axiomes et règles caractérisant les constructions de ce langage

a) *Règles de conséquence*

$$(\mathrm{IMP})\quad \frac{\mathbf{p}\ \{\ E\ \}\ \mathbf{q},\ \mathbf{q} \Rightarrow \mathbf{r}}{\mathbf{p}\ \{\ E\ \}\ \mathbf{r}} \qquad \frac{\mathbf{r} \Rightarrow \mathbf{p},\ \mathbf{p}\ \{\ E\ \}\ \mathbf{q}}{\mathbf{r}\ \{\ E\ \}\ \mathbf{q}}$$

b) *Affectation*

$$(\mathrm{AFF})\quad \mathbf{p}[x/exp]\ \{\ x := exp\ \}\ \mathbf{p}\ .$$

c) *Instruction vide*

$$(\mathrm{VID})\quad \mathbf{p}\ \{\quad \}\ \mathbf{p}\ .$$

d) *Conditionnelle*

$$(\mathrm{COND})\quad \frac{\mathbf{p}\ \underline{\mathbf{et}}\ t\ \{\ E\ \}\ \mathbf{q},\ \mathbf{p}\ \underline{\mathbf{et}}\ \underline{\mathbf{non}}\ t\ \{\ F\ \}\ \mathbf{q}}{\mathbf{p}\ \{\ \underline{si}\ t\ \underline{alors}\ E\ \underline{sinon}\ F\ \underline{fsi}\ \}\ \mathbf{q}}\ .$$

e) *Composition séquentielle*

$$(\mathrm{SEQ})\quad \frac{\mathbf{p}\ \{\ E\ \}\ \mathbf{q},\ \mathbf{q}\ \{\ F\ \}\ \mathbf{r}}{\mathbf{p}\ \{\ E\ ;F\ \}\ \mathbf{r}}\ .$$

f) *Itération*

$$(\mathrm{ITE})\quad \frac{\mathbf{p}\ \underline{\mathbf{et}}\ t\ \{\ E\ \}\ \mathbf{p}}{\mathbf{p}\ \{\ \underline{tant\ que}\ t\ \underline{faire}\ E\ \underline{fait}\ \}\ \mathbf{p}\ \underline{\mathbf{et}}\ \underline{\mathbf{non}}\ t}\ .$$

g) *Déclarations*

de constante

$$(\text{DEC}) \quad \frac{x = exp[x'/x] \mapsto \mathbf{p}[x'/x] \; \{ \; E \; \} \; \mathbf{q}[x'/x]}{\mathbf{p} \; \{ \; \underline{début} \; \underline{cst} \; x = exp \, ; E \; \underline{fin} \; \} \; \mathbf{q}}$$

de variable

$$\frac{\mathbf{p}[x'/x] \; \{ \; E \; \} \; \mathbf{q}[x'/x]}{\mathbf{p} \; \{ \; \underline{début} \; \underline{var} \; x \, ; E \; \underline{fin} \; \} \; \mathbf{q}}$$

x' n'a aucune occurrence libre ni dans **p** ni dans **q**, autrement dit si x n'a aucune occurrence libre dans **p** et **q**, on ignore simplement la déclaration pour le cas d'une variable. Le symbole $\boldsymbol{\alpha} \mapsto \boldsymbol{\beta}$ signifie que de $\boldsymbol{\alpha}$ on peut déduire $\boldsymbol{\beta}$.

h) *Procédures*

déclaration

$$\frac{\mathbf{r} \; \{ \; f(x : y) \; \} \; \mathbf{s} \mapsto \mathbf{r} \; \{ \; F \; \} \; \mathbf{s}, \; \mathbf{r} \; \{ \; f(x : y) \; \} \; \mathbf{s} \mapsto \mathbf{p} \; \{ \; E \; \} \; \mathbf{q}}{\mathbf{p} \; \{ \; \underline{début} \; \underline{proc} \; f(x : y) \; F \, ; E \; \underline{fin} \; \} \; \mathbf{q}}$$

appel

$$\frac{\mathbf{p} \; \{ \; f(x : y) \; \} \; \mathbf{q}}{\mathbf{r} \; \underline{\mathbf{et}} \; \mathbf{p}[exp/x, z/y] \; \{ \; f(exp : z) \; \} \; \mathbf{q}[exp/x, z/y] \; \underline{\mathbf{et}} \; \mathbf{r}}$$

où **r** et *exp* ne contiennent pas z.

7.2 Etude d'un exemple

Reprenons le petit programme que nous avons déjà étudié plusieurs fois (exemples 1 et 3).

Nous allons prouver que :

$$(1) \quad \mathbf{vrai} \; \{ \; \underline{début} \; \underline{proc} \; f(x : y) \; y : x + 1 \, ; \underline{début} \; \underline{var} \; x \, ; x := 1 \, ; f(x : \widehat{impr}) \; \underline{fin} \; \underline{fin} \; \} \; \widehat{impr} = 2 \, .$$

Remarquons que cet exemple diffère légèrement des exemples que nous avons traité aux paragraphes 4, 5 et 6 car nous avons changé

$$f(x : x) \, ; \widehat{impr} := x \text{ en } f(x : \widehat{impr}) \, .$$

En effet la sémantique axiomatique que nous proposons est incapable de rendre compte d'appels de procédures tels que f(x : x) (cf. spécification concernant l'appel en h) ci-dessus).

Lemme 1 : $\mathbf{vrai} \; \{ \; f(x : y) \; \} \; y = x + 1 \mapsto \mathbf{vrai} \; \{ \; y := x + 1 \; \} \; y = x + 1 \, .$

Immédiat d'après la règle de l'affectation et sans utiliser l'hypothèse (en effet la procédure n'est pas récursive).

Par la règle de déclaration de procédure, (1) découle alors du lemme 1 et de

$$(2) \; \mathbf{vrai} \; \{ \; f(x : y) \; \} \; y = x + 1 \mapsto \mathbf{vrai} \; \{ \; \underline{début} \; \underline{var} \; x \, ; x := 1 \, ; f(x : \widehat{impr}) \; \underline{fin} \; \} \; \widehat{impr} = 2 \, .$$

Or

$$\textbf{vrai}\ \{\ \underline{\textit{début}}\ \underline{\textit{var}}\ x\ ;\ x := 1\ ;\ f(x : \widehat{impr})\ \underline{\textit{fin}}\ \}\ \widehat{impr} = 2$$

découle de

(3) $$\textbf{vrai}\ \{\ x := 1\ ;\ f(x : \widehat{impr})\ \}\ \widehat{impr} = 2$$

d'après la règle de déclaration de variables et le fait que x n'apparaît pas dans les prédicats.

Par la règle de composition, (3) découle de

(4) $\textbf{vrai}\ \{\ x := 1\ \}\ x = 1$ (règle d'affectation et le fait que $1 = 1$ est équivalent à **vrai**)

(5) $$x = 1\ \{\ f(x : \widehat{impr})\ \}\ \widehat{impr} = 2$$

or par la première règle de conséquence, (5) découle de

$x = 1\ \{\ f(x : \widehat{impr})\ \}\ \widehat{impr} = x + 1\ \textbf{et}\ x = 1$ (règle d'appel)

$\widehat{impr} = x + 1\ \underline{\textbf{et}}\ x = 1 \Rightarrow \widehat{impr} = 2$ (théorème du calcul des prédicats).

8 COMPLÉMENTARITÉ DE DÉFINITIONS SÉMANTIQUES

8.1 Introduction

Dans les paragraphes précédents plusieurs définitions sémantiques du langage NAIN ont été présentées. On peut constater sur cet exemple que ces différentes sémantiques ne jouent pas le même rôle : alors que les sémantiques interprétatives et calculatoires semblent bien adaptées aux problèmes d'implantation, la sémantique fonctionnelle ou dénotationnelle et surtout la sémantique axiomatique peuvent être plus utilisables par le programmeur. De plus, les sémantiques calculatoires et fonctionnelles associent un sens global (ensemble de calculs ou fonctions) à un programme et peuvent être ainsi de bons outils pour la conception de nouveaux langages. La sémantique interprétative permet, quant à elle, de suivre, pas à pas, le déroulement d'un programme, elle est donc utile à la compréhension détaillée de celui-ci.

Les objectifs des différentes sémantiques sont donc nettement complémentaires. Encore faut-il que ces définitions donnent bien le « même sens » à chaque programme ou encore qu'elles soient cohérentes. Développons et illustrons cette idée en utilisant les définitions précédentes de la sémantique de NAIN.

8.2 Complémentarité des trois premières sémantiques de NAIN

Afin de préciser le concept de complémentarité de plusieurs sémantiques, commençons par introduire quelques notations :

— soit **cal** un calcul de l'interprète abstrait de NAIN (§ 4.4) de premier

état $\mathbf{e}_0$, définissons $\mathbf{Res}_{int}(\mathbf{cal})$ comme la valeur du résultat du calcul s'il est réussi. Plus précisément $\mathbf{Res}_{int}(\mathbf{cal})$ est le sommet de la pile désignée par l'identificateur « standard » $\widehat{impr}$:

$$\mathbf{Res}_{int}(\mathbf{cal}) = \begin{cases} \mathbf{valsom}_e\,(\mathbf{des}_e(\widehat{impr})) & \text{si } \mathbf{cal} \text{ est un calcul réussi de dernier état } \langle \wedge \mid \mathbf{e} \rangle, \text{ les fonctions d'accès se rapportent à } \mathbf{e} \\ \perp_{\mathbf{val}} & \text{élément indéfini de l'ensemble des valeurs (§ 6.2) si } \mathbf{cal} \text{ est non réussi ou infini.} \end{cases}$$

— Soit $\mathscr{C}$ un ensemble de calculs au sens de la sémantique calculatoire de NAIN et $\mathbf{e}_0$ un état de mémoire initial, $\mathbf{Res_{cal}}(\mathscr{C}, \mathbf{e}_0)$ est défini comme le résultat d'un calcul fini dont le dernier état n'est pas $\mathbf{\Omega}$, s'il en existe :

$$\mathbf{Res}_{cal}(\mathscr{C}, \mathbf{e}_0) = \begin{cases} \mathbf{valsom}_{\mathbf{c}(\mathbf{e}_0)}(\mathbf{des}_{\mathbf{c}(\mathbf{e}_0)}\,(\widehat{impr})) & \text{s'il existe un calcul fini } \mathbf{c} \in \mathscr{C} \text{ tel que } \mathbf{c}(\mathbf{e}_0) \neq \mathbf{\Omega} \\ \perp_{\mathbf{val}} & \text{s'il n'existe pas un tel calcul.} \end{cases}$$

De plus, pour simplifier les notations, on suppose que les programmes considérés n'admettent qu'une donnée $\mathbf{v}$ désignée par l'identificateur standard $\widehat{don}$ et qu'un résultat désigné par $\widehat{impr}$.

La complémentarité des trois premières sémantiques précédentes de NAIN peut alors s'exprimer sous la forme :

Théorème 1

Pour tout programme P de NAIN et toute donnée $\mathbf{d} = (\widehat{don}, \mathbf{v})$ de P on a :

$$\mathbf{Res}_{int}(\mathscr{S}_{int}[\![P]\!]\,(\mathbf{e}_0)) = \mathbf{Res}_{cal}(\mathscr{S}_{cal}[\![P]\!], \mathbf{e}_0) = \mathscr{S}_{fonct}[\![P]\!]\,\mathbf{d}\,\widehat{don}$$

où $\mathbf{e}_0$ est l'état de mémoire initial caractérisé par : $\mathbf{valsom}_{\mathbf{e}_0}(\mathbf{des}_{\mathbf{e}_0}(\widehat{don})) = \mathbf{v}$. □

Ce théorème signifie que, quelle que soit la sémantique considérée, un programme réalise la même fonction de transformation des données en résultats. Il se démontre par récurrence sur la structure des programmes.

Cette démonstration est longue puisqu'il est nécessaire de distinguer les différentes formes possibles des phrases de NAIN et elle dépasse le cadre de cet ouvrage. Contentons-nous de donner une idée intuitive de la démarche à suivre.

Il est nécessaire de considérer les différentes constructions du langage. Etudions par exemple, pour les deux premières sémantiques, la déclaration d'une variable :

$$\mathbf{Res}_{int}(\mathscr{S}_{int}[\![\underline{d\acute{e}but}\ var\ x\ ;\ P_1\ \underline{fin}]\!]\,(\mathbf{e}_0)) = \mathbf{Res}_{int}(\mathscr{S}_{int}[\![P_1\ ;\ rest(x)\ fin]\!]\,(\underline{\mathbf{declvar}}(x)\,(\mathbf{e}_0))$$

et

$$\mathbf{Res}_{cal}(\mathscr{S}_{cal}[\![P]\!], \mathbf{e}_0) = \mathbf{Res}_{cal}(\underline{\mathbf{declvar}}(\mathrm{x}).\mathscr{S}_{cal}[\![P_1]\!].\underline{\mathbf{lib}}(\mathrm{x}), \mathbf{e}_0)$$

En utilisant le résultat très simple suivant :

$$\mathbf{Res}_{cal}(\mathbf{m}.\mathscr{C}, \mathbf{e}) = \mathbf{Res}_{cal}(\mathscr{C}, \mathbf{m}\,(\mathbf{e}))$$

on se ramène à montrer que :

$$\mathbf{Res}_{\mathrm{int}}(\mathscr{S}_{\mathrm{int}}[\![P_1 \,;\, rest(x)\ fin]\!]\ (\mathrm{e})) =$$
$$\mathbf{Res}_{\mathrm{cal}}(\mathscr{S}_{\mathrm{cal}}[\![P_1]\!]\ \underline{\mathbf{lib}}(x),\ \mathbf{e})\ .$$

Ainsi en utilisant un raisonnement par induction et par cas, on réduit la preuve du théorème à celles de lemmes concernant les phrases élémentaires du langage (déclarations, affectations, appels).

8.3 Comparaison des sémantiques fonctionnelle et axiomatique

La sémantique axiomatique de NAIN est la plus « abstraite », c'est-à-dire la plus dégagée des considérations d'exécution et la plus utile pour le programmeur, il n'est pas étonnant qu'elle soit moins puissante que les autres au sens où elle ne permet pas de démontrer autant de propriétés d'un programme que les autres. On dit que le système formel défini par cette sémantique est incomplet. En particulier, elle ne définit pas la notion de terminaison d'un programme. Précisons cette notion d'incomplétude en comparant les sémantiques fonctionnelle et axiomatique. Soit **p** et **q**, deux prédicats et P une phrase de NAIN, on dit qu'une formule de la forme $\mathbf{p}\ \{\ P\ \}\ \mathbf{q}$ (§ 7) est **valide** si pour toute donnée $\mathbf{d} = (\widehat{don}, \mathbf{v})$ telle que $\mathbf{p}(\mathbf{v})$ soit vrai, le résultat

$$\mathscr{S}_{\mathrm{fonct}}[\![P]\!]\ \mathbf{d}\ \widehat{don}$$

vérifie le prédicat **q**. On peut alors énoncer :

Théorème 2

Pour tous prédicats **p**, **q** et tout programme P, si la formule $\mathbf{p}\ \{\ P\ \}\ \mathbf{q}$ est démontrable en utilisant les axiomes et règles d'inférence de la sémantique axiomatique alors $\mathbf{p}\ \{\ P\ \}\ \mathbf{q}$ est une formule valide. □

Ici encore la démonstration s'effectue par induction sur la structure des programmes. Notons bien cependant que la réciproque n'est pas vraie en général ; ceci résulte intuitivement du fait que le système axiomatique n'est pas assez puissant pour démontrer toutes les formules valides. Par exemple, on a constaté (§ 7) que la sémantique axiomatique donnée dans ce paragraphe est incapable de rendre compte d'appels de procédures de la forme $f(x : x)$.

8.4 En guise de conclusion sur les définitions sémantiques

Indiquons quelques voies de recherche qui tendent à rendre plus aisées les preuves de complémentarité et plus généralement de rendre plus manipulables les sémantiques formelles :

— Définir des sémantiques intermédiaires qui permettent de passer progressivement d'une sémantique à l'autre [MIL, 76].

— Définir un cadre mathématique dans lequel elles puissent toutes s'exprimer. Les différentes sémantiques représentent alors différentes facettes d'une même réalité mathématique [PLO, 75].

— Définir des sémantiques les plus abstraites possibles, au sens où on souhaite que ces définitions fassent le plus possible abstraction des choix d'implémentation. Par exemple, il est préférable de donner les propriétés algébriques de la fonction d'allocation que la définir explicitement ; ainsi celui qui implante aura une plus grande liberté et pourra effectuer des optimisations.

9 MÉTHODE DES ATTRIBUTS

9.1 Introduction

Considérons un langage de programmation P décrit par une grammaire algébrique non ambiguë. A chaque phrase du langage P est donc associé un arbre syntaxique, la méthode des attributs se propose d'étiqueter les nœuds de cet arbre par des renseignements d'ordre sémantique. Nous allons introduire la terminologie et les procédés de cette théorie au travers de l'exemple simple suivant.

Exemple 4

Considérons le langage P engendré par la grammaire

$$G = (\{ X, E, F \}, \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, + \}, ::=, X) \text{ telle que}$$

$$X ::= E$$

$$E ::= E + F$$

$$E ::= F$$

$$F ::= 0/1/2/ \dots /9 .$$

Une phrase de P est donc une suite finie de chiffres séparés par des +. Remarquons que l'axiome X n'intervient qu'en partie gauche des règles de G (§ 9.4).

Supposons que l'on veuille donner à P, la sémantique intuitive suivante : une phrase de P représente le nombre entier obtenu en faisant la somme des chiffres qui apparaissent dans cette phrase.

La description par la méthode des attributs de cette sémantique peut se faire de la manière suivante :

règles syntaxiques	règles sémantiques
$X ::= E$	$\mathbf{v}(X) = \mathbf{v}(E)$
$E_1 ::= E_2 + F$	$\mathbf{v}(E_1) = \mathbf{v}(E_2) + \mathbf{v}(F)$
$E ::= F$	$\mathbf{v}(E) = \mathbf{v}(F)$
$F ::= 0$	$\mathbf{v}(F) = 0$
$F ::= 1$	$\mathbf{v}(F) = \mathbf{1}$
⋮	⋮
$F ::= 9$	$\mathbf{v}(F) = \mathbf{9}$

Les règles syntaxiques sont pratiquement identiques aux règles de la grammaire G engendrant P. On a seulement modifié les règles contenant plusieurs

fois un même non-terminal, pour distinguer les diverses occurrences de cette lettre.

Les règles sémantiques se présentent comme des égalités associées à chaque règle syntaxique. Analysons l'une d'entre elles : $\mathbf{v}(E_1) = \mathbf{v}(E_2) + \mathbf{v}(F)$, elle signifie que pour calculer la valeur de l'attribut **v** en un point étiqueté E, d'un arbre syntaxique ayant des fils étiquetés E et F, il faut d'abord calculer la valeur de **v** en E, fils de E, puis la valeur en F et faire la somme de ces deux valeurs. Ces règles sémantiques permettent de calculer la valeur de l'attribut **v** en tout point d'un arbre syntaxique engendré par G.

Par exemple, l'évaluation sémantique de l'arbre de la figure 17 conduit à $\mathbf{v}(X) = \mathbf{14}$.

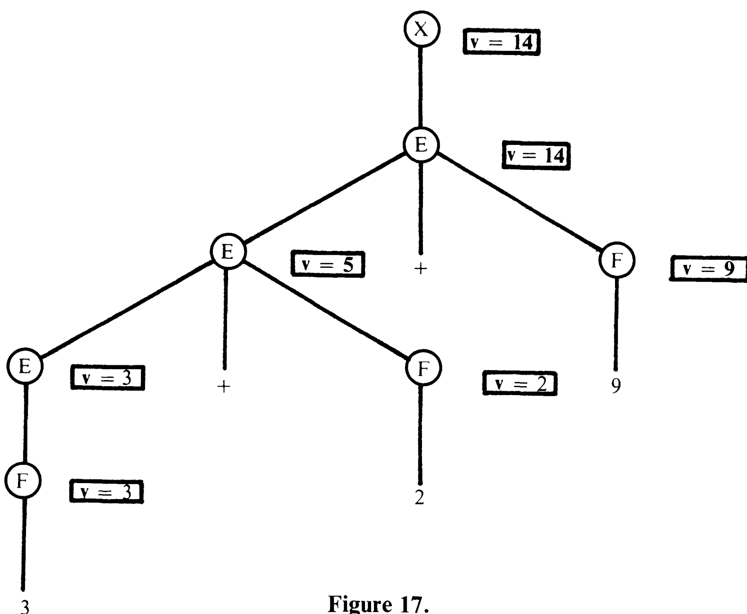


Figure 17.

L'évaluation sémantique, pour un tel arbre, est terminée quand on a calculé la valeur de l'attribut **v** en X. X joue donc un rôle de marqueur de fin de calcul et c'est une des raisons pour laquelle X n'intervient qu'en partie gauche des règles de G. □

Résumons les définitions que nous avons illustrées dans l'exemple précédent : pour décrire un langage P engendré par une grammaire $G = (N, T, ::=, X)$ par la méthode des attributs, on procède de la manière suivante :

— Pour chaque lettre Y du vocabulaire non-terminal de la grammaire G, on définit un ensemble d'attributs **At**(Y). A chaque attribut **a** est associé un ensemble de valeur $\mathbf{V_a}$ de l'attribut **a**.

— Pour chaque règle de la grammaire, on définit des relations liant les valeurs des attributs du premier membre et du deuxième membre de la règle.

Le problème est alors de calculer la valeur des attributs des différents nœuds d'un arbre syntaxique donné, de façon que les valeurs obtenues soient compatibles avec les relations que l'on a définies. Ce problème risque d'être insoluble si on n'impose pas de restrictions aux relations liant les valeurs des divers attributs. Dans le premier paragraphe, nous étudions un cas très particulier, où, de façon évidente, on sait calculer la valeur de tous les attributs. Dans les paragraphes suivants, on étudie des conditions moins restrictives, mais il faut alors donner des méthodes permettant de s'assurer que tous les calculs sont possibles. Cependant, remarquons que l'impossibilité de calculer la valeur d'un attribut en un point n'est pas toujours un phénomène regrettable ; en effet, on peut utiliser ce fait pour sélectionner, à l'intérieur du langage algébrique P, un sous-langage P′, qui est l'ensemble des phrases de P pour lesquels le calcul des attributs est toujours possible. Par exemple, on peut, par ce procédé, imposer que, dans un programme, tout identificateur utilisé soit déclaré. On étudie ce rôle de sélection des attributs dans le paragraphe 9.6.

On peut déjà remarquer, à partir de cette brève description, un certain nombre d'avantages de la méthode des attributs. Cette méthode est très localisée, en effet, c'est au niveau de chaque règle syntaxique que l'on définit des règles sémantiques. Une modification dans la sémantique d'une règle syntaxique aura donc des répercussions seulement sur les règles sémantiques associées. De plus, au niveau de chaque règle syntaxique, on peut en multipliant les attributs parcelliser le travail de définition de la sémantique. Dans le même ordre d'idée, on peut, par cette méthode, construire la définition sémantique par raffinements successifs, en commençant par tracer les grandes lignes de la sémantique avec peu d'attributs, puis en raffinant peu à peu cette définition, en augmentant le nombre des attributs et en utilisant le travail précédemment fait. Enfin, notons que la méthode des attributs est utilisable pour la construction de compilateurs de langages évolués réels [LOR, 74].

Cependant cette méthode présente également des inconvénients, en particulier elle est plus orientée vers l'implantation que vers la programmation.

9.2 Attributs synthétisés

Considérons une grammaire $G = (N, T, ::=, X)$ et, pour chaque lettre Y de N, un ensemble **At**(Y) d'attributs. On associe à chaque attribut **a** un ensemble $\mathbf{V}_{\mathbf{a}}$, appelé ensemble des valeurs de **a**. On dit que les attributs de cette grammaire sont **synthétisés** si, pour calculer la valeur des attributs du premier membre d'une règle, il suffit de connaître les valeurs des attributs des lettres du deuxième membre de la règle. Autrement dit, dans un arbre syntaxique, la valeur des attributs d'un nœud dépend exclusivement de la valeur des attributs des fils de ce nœud. Il est donc évident que l'on pourra calculer de proche en proche la valeur des attributs des différents nœuds d'un arbre syntaxique en procédant des feuilles de l'arbre vers la racine. Traitons tout de suite un exemple de ce type proposé par KNUTH [KNU, 68b].

Exemple 5

Considérons la grammaire :

$G = (\{ N, L, B \}, \{ 0, 1, \}, ::=, N)$ telle que

$N ::= L.L$

$N ::= L$

$L ::= LB$

$L ::= B$

$B ::= 0$

$B ::= 1$

Cette grammaire engendre le langage

$$P = \{ 0, 1 \} \{ 0, 1 \}^{*}.\{ 0, 1 \} \{ 0, 1 \}^{*} \cup \{ 0, 1 \} \{ 0, 1 \}^{*} .$$

On veut attribuer à ce langage une sémantique signifiant, intuitivement, « un mot α de P représente un nombre rationnel écrit en système binaire ». Nous allons décrire cette sémantique en utilisant les attributs suivants :

— attribut **v** dont la valeur est, intuitivement, le nombre rationnel représenté par le mot dérivant de B, de N ou de L.

— attribut **l** dont la valeur est, intuitivement, la longueur du mot dérivant de L.

On a donc $\mathbf{At}(B) = \mathbf{At}(N) = \{ \mathbf{v} \}$ et $\mathbf{At}(L) = \{ \mathbf{l}, \mathbf{v} \}$.

D'autre part, les attributs **v** et **l** prendront leurs valeurs dans $\mathbf{V}_{\mathbf{v}} = \mathbb{Q}$ et $\mathbf{V}_{\mathbf{l}} = \mathbb{N}$ respectivement.

Donnons maintenant les règles sémantiques qui permettent de calculer les valeurs des attributs aux différents points d'un arbre syntaxique :

règles syntaxiques	règles sémantiques
$B ::= 0$	$\mathbf{v}(B) = 0$
$B ::= 1$	$\mathbf{v}(B) = 1$
$L ::= B$	$\mathbf{v}(L) = \mathbf{v}(B), \mathbf{l}(L) = \mathbf{1}$
$L_1 ::= L_2 B$	$\mathbf{v}(L_1) = 2\,\mathbf{v}(L_2) + \mathbf{v}(B), \mathbf{l}(L_1) = \mathbf{l}(L_2) + 1$
$N ::= L$	$\mathbf{v}(N) = \mathbf{v}(L)$
$N ::= L_1.L_2$	$\mathbf{v}(N) = \mathbf{v}(L_1) + \mathbf{v}(L_2)/2 ** \mathbf{l}(L_2)$

Comme dans l'exemple 4, on a indicé les lettres pour distinguer les différentes occurrences d'une même lettre.

Considérons, par exemple, l'arbre syntaxique associé à 1 011.01 (figure 18) et étiquetons l'arbre par la valeur des attributs des différents nœuds.

(Les flèches précisent l'ordre dans lequel se font les calculs.)

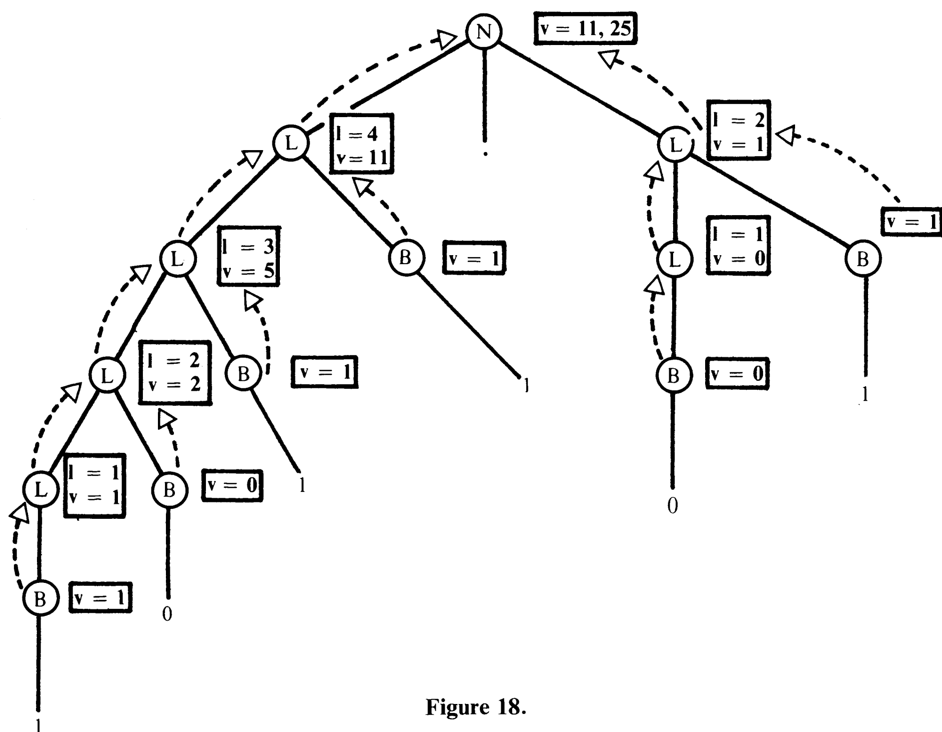


Figure 18.

Avant de poursuivre, donnons une formalisation du concept de sémantique définie par attributs synthétisés. Cette formalisation permet d'introduire, dans un cas simple, les définitions et les notations que nous verrons dans le cas général au paragraphe suivant.

On veut définir par attribut synthétisé une sémantique d'un langage L décrit par une grammaire algébrique $G = (N, T, ::= ; X)$. Pour cela, on procède de la façon suivante :

— Pour chaque Y de N, on note $\mathbf{V}_Y$ l'ensemble $\prod_{a \in \mathbf{At}(Y)} \mathbf{V}_a$ produit des ensembles de valeurs des attributs de Y.

Remarquons que l'on n'associe pas d'attributs aux éléments terminaux de G. On pose donc $\mathbf{At}(a) = \varnothing$ pour chaque terminal a de G.

— Pour chaque règle r de la forme $Y ::= Y_1\, Y_2 \ldots Y_n$, on se donne une fonction $\mathbf{F}(r)$ de $\mathbf{V}_{Y_1} \times \mathbf{V}_{Y_2} \times \cdots \times \mathbf{V}_{Y_n}$ dans $\mathbf{V}_Y$. Cette fonction permet de calculer les valeurs des attributs de Y, connaissant celles des attributs de Y_1, $Y_2, \ldots Y_n$.

On utilise alors ces données de la manière suivante : S est un arbre syntaxique engendré par G, y est un nœud de cet arbre étiqueté par le non-terminal $\tilde{y}$. Par définition d'un arbre syntaxique, le nœud y possède des successeurs y_1, $y_2, \ldots, y_n$ étiquetés par $\tilde{y}_1, \ldots, \tilde{y}_n$ et il existe dans G une règle r de la forme

$\tilde{y} ::= \tilde{y}_1 \tilde{y}_2 \dots \tilde{y}_n$. Considérons la famille $\boldsymbol{a}(y) = (\mathbf{a}(y))_{\mathbf{a} \in \mathbf{At}(\tilde{y})}$. On considère $\boldsymbol{a}(y)$ comme une inconnue. On écrit alors toutes les équations de la forme

$$\boldsymbol{a}(y) = \mathbf{F}(r)\,(\boldsymbol{a}(y_1), \dots, \boldsymbol{a}(y_n)),$$

en prenant pour y tous les nœuds de S qui sont étiquetés par des non-terminaux. Il est alors possible d'ordonner ces équations de façon que toutes les inconnues $\boldsymbol{a}(y)$ puissent être calculées. Il suffit, en effet, de commencer par les équations qui correspondent à des règles du type $Y ::= \alpha$ avec $\alpha \in T^*$ pour lesquelles le second membre ne contient pas d'inconnues, puis de continuer en remontant jusqu'à la racine. Ce résultat signifie simplement que l'on peut calculer de manière unique les valeurs des attributs synthétisés en tout point d'un arbre syntaxique.

Exercice 14

On considère la grammaire

$$G = (\{ X, E \}, \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, / \}, ::=, X)$$

telle que

$$X ::= E/E$$
$$E ::= CE \mid C$$
$$C ::= 0 \mid 1 \mid 2 \dots \mid 9 .$$

Expliciter à l'aide d'attributs synthétisés la sémantique de L(G) décrite intuitivement par « si α/β est un mot de L(G), il représente le nombre rationnel p/q tel que p (respectivement q) s'écrive α (respectivement β) en écriture décimale ». □

Exercice 15

On considère la grammaire $G = (\{ P, M, ; \}, \{ a, b, c, \}, ::=, P)$ telle que

$$P ::= M ; P \mid M$$
$$M ::= \wedge \mid aM \mid bM \mid cM .$$

Expliciter à l'aide d'attributs synthétisés la sémantique de L(G) décrite intuitivement par « si $\alpha_0 ; \alpha_1 ; \alpha_2 ; \dots ; \alpha_n$ est un mot de L(G), il représente le polynôme formel $\alpha_0 + \alpha_1 X + \cdots \alpha_n X^n$. Montrer que cette méthode de définition de la sémantique amène à calculer les polynômes suivant le schéma de Horner ». □

Exercice 16

On considère la grammaire $G = (\{ E, T, ; \}, \{ 0, 1, \dots, 9 \}, ::=, E)$ telle que

$$E ::= T ; E \mid T$$
$$T ::= CT \mid C$$
$$C ::= 0 \mid 1 \mid \dots \mid 9 .$$

Expliciter, à l'aide d'attributs synthétisés, la sémantique de L(G) décrite intuitivement par « si α_0 ; α_1 ; ... ; α_n est un mot de L(G) il représente le nombre

$$\frac{x_1}{0\,!} + x_2 \frac{x_0}{1\,!} \cdots + x_n \frac{x_0^{n-1}}{(n-1)\,!}$$

$x_0, x_1, \dots x_n$ ayant comme écriture décimale $\alpha_0, \alpha_1, \dots \alpha_n$ ». □

9.3 Attributs hérités : un exemple

Reprenons l'exemple de la représentation binaire des nombres (exemple 5) et analysons la définition des attributs que l'on a donnée au paragraphe précédent.

Pour l'arbre syntaxique considéré sur la figure 18, on a trouvé $\mathbf{v}(N) = \mathbf{11{,}25}$ (en décimale) c'est-à-dire :

$$\mathbf{v}(N) = \mathbf{2}^3 + \mathbf{2}^1 + \mathbf{2}^0 + \mathbf{2}^{-2}\,.$$

On s'attendait à trouver ce résultat qui est satisfaisant pour l'esprit. Mais en certains points de l'arbre syntaxique les valeurs des attributs, calculées par cette méthode, ne sont pas très naturelles. En effet, on pouvait s'attendre à trouver pour $\mathbf{v}(B)$ respectivement $\mathbf{2}^3$, $\mathbf{0}$, $\mathbf{2}^1$, $\mathbf{2}^0$, $\mathbf{0}$, $\mathbf{2}^{-2}$ quand on se déplace de gauche à droite dans les feuilles de l'arbre ; alors que l'on trouve uniformément **1**.

Cette définition de $\mathbf{v}(B)$ est impossible à réaliser avec des attributs synthétisés, en effet, ce qui est en dessous de B et qui, seul, peut être pris en compte par des attributs synthétisés ne nous donne aucun renseignement sur la place de 1 dans le mot considéré. Pour connaître cette place, il faut compter le nombre d'utilisations de règle L ::= LB pour atteindre l'occurrence du B considéré. On est donc amené à considérer des attributs dont la valeur est définie en allant du haut vers le bas. De tels attributs sont dits attributs **hérités.**

Dans le cas particulier de cet exemple, on est amené à définir la sémantique de ce langage en posant :

1° $\mathbf{At}(N) = \{\,\mathbf{v}\,\}$,
$\mathbf{At}(L) = \{\,\mathbf{v}, \mathbf{s}, \mathbf{l}\,\}$,
$\mathbf{At}(B) = \{\,\mathbf{v}, \mathbf{s}\,\}$,

l a la même signification que précédemment ($\mathbf{V_l} = \mathbb{N}$),
s donne le « poids » du bit B au bit le plus à droite dérivant de L($\mathbf{V_s} = \mathbb{N}$),
v donne la valeur de la suite dérivant de L ou de B dans le contexte où elle est placée ($\mathbf{V_v} = \mathbb{Q}$).

2° B ::= 0 $\mathbf{v}(B) = \mathbf{0}$
B ::= 1 $\mathbf{v}(B) = \mathbf{2}^{\mathbf{s}(B)}$
L ::= B $\mathbf{v}(L) = \mathbf{v}(B), \mathbf{s}(B) = \mathbf{s}(L), \mathbf{l}(L) = \mathbf{1}$
L_1 ::= L_2 B $\mathbf{v}(L_1) = \mathbf{v}(L_2) + \mathbf{v}(B), \mathbf{s}(B) = \mathbf{s}(L_1)$
$\mathbf{s}(L_2) = \mathbf{s}(L_1) + \mathbf{1}, \mathbf{l}(L_1) = \mathbf{l}(L_2) + \mathbf{1}$
N ::= L $\mathbf{v}(N) = \mathbf{v}(L), \mathbf{s}(L) = \mathbf{0}$
N ::= $L_1 . L_2$ $\mathbf{v}(N) = \mathbf{v}(L_1) + \mathbf{v}(L_2), \mathbf{s}(L_1) = \mathbf{0}, \mathbf{s}(L_2) = -\,\mathbf{l}(L_2)$.

Reprenons sur la figure 19 le calcul des valeurs des attributs de l'exemple du paragraphe précédent.

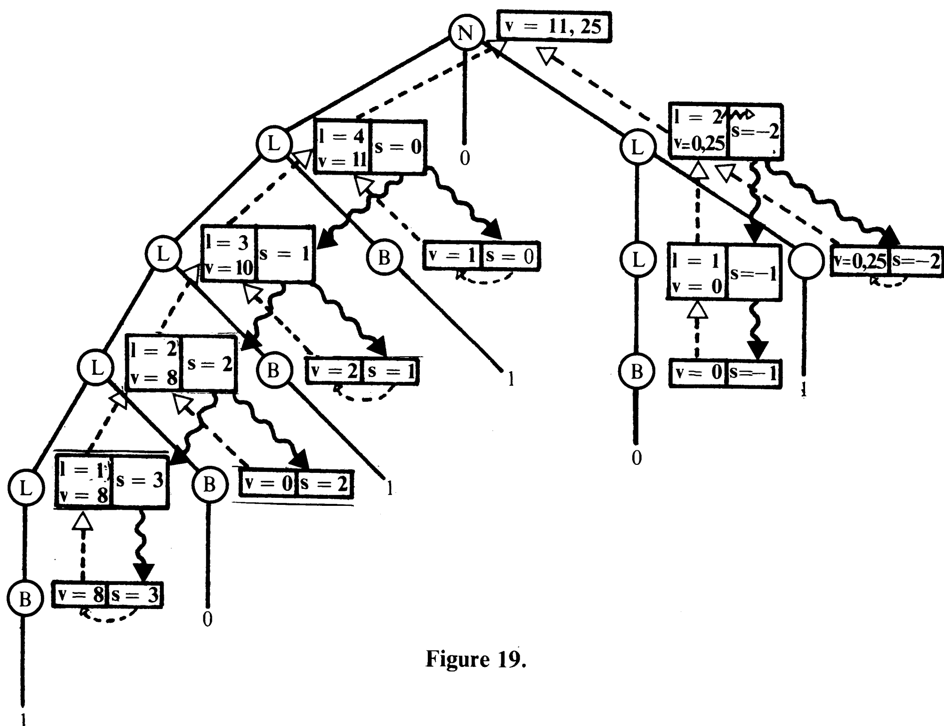


Figure 19.

Contrairement à ce qui se passe dans le cas des attributs synthétisés, on peut, en utilisant des attributs hérités, donner aux points de l'arbre syntaxique des valeurs d'attributs qui dépendent du contexte général dans lequel est placé ce point. En revanche, il n'est nullement évident que l'on puisse effectivement calculer la valeur des attributs en tout point d'un arbre syntaxique. Il est possible qu'il y ait des circuits dans les définitions sémantiques, qui empêchent une élaboration complète des valeurs des attributs. Ce problème est traité au paragraphe 9.5.

Exercice 17

Reprendre les exercices 14, 15, 16 quand on a à sa disposition les attributs hérités. □

Exercice 18

On considère la grammaire suivante

$$G = (\{ E \}, \{(\,,\,), +, -, x, /, \uparrow, \square \}, ::=,),$$

telle que

$$X ::= E\,,$$
$$E ::= (E) \mid E + E \mid -E \mid E - E \mid E \times E \mid E/E \mid E \uparrow E \mid \square\,.$$

La sémantique de L(G) est « si α est un mot de L(G), α représente la fonction f_α de $\mathbb{R}^n$ dans $\mathbb{R}$ telle que n est le nombre d'occurrences de $\square$ dans α et $f_\alpha(x_1, \dots x_n)$ est obtenu en remplaçant les occurrences de $\square$ par $x_1, x_2, \dots x_n$ dans cet ordre ».

1) Exprimer la sémantique de L(G) par la méthode des attributs ;

2) Construire un attribut **var** de E, tel que **var**(E) $\in \{-\mathbf{1}, +\mathbf{1}\}$ et que var(E) = **1** si et seulement si une augmentation de l'expression qui est en dessous de E fait toujours augmenter l'expression totale. $\square$

9.4 Définition formelle de la méthode des attributs

Considérons un langage P défini par une grammaire algébrique G = (N, T, ::=, *X*). Les seules conditions imposées à cette grammaire sont que cette grammaire soit réduite supérieurement et inférieurement et que l'axiome X apparaisse uniquement en partie gauche des règles de G. Rappelons qu'une grammaire est dite réduite inférieurement et supérieurement si tout non-terminal peut se dériver en un mot du langage engendré et si tout non-terminal apparaît dans un mot dérivant de l'axiome. Pour définir une sémantique de P par la méthode des attributs on utilise les outils suivants :

— Pour chaque symbole Y de N, on se donne un ensemble éventuellement vide de symbole **At**(Y) réunion de deux sous-ensembles disjoints $\mathbf{At_s}(Y)$ et $\mathbf{At_h}(Y)$. Les éléments de $\mathbf{At_s}(Y)$ sont appelés attributs synthétisés de Y et ceux de $\mathbf{At_h}(Y)$ attributs hérités de Y. On impose de plus que $\mathbf{At_h}(X) = \varnothing$ et par convention si *a* est un symbole terminal de G on pose $\mathbf{At}(a) = \varnothing$.

— A chaque attribut **a** est associé un ensemble $\mathbf{V_a}$ appelé ensemble des valeurs de l'attribut **a**. Pour chaque Y de N $\cup$ T on note $\mathbf{V_Y}$ l'ensemble $\prod_{\mathbf{a} \in \mathbf{At}(Y)} \mathbf{V_a}$.

— Pour chaque règle r de G de la forme $Y_0 ::= Y_1\, Y_2 \dots Y_n$, on se donne des applications qui permettent de calculer les attributs synthétisés de Y_0, en fonction des attributs de $Y_1, Y_2, \dots, Y_n$ et de calculer les attributs hérités de chaque Y_i, en fonction des attributs des Y_j. Pour chaque règle r, on dispose donc des applications suivantes :

$$F_0[r] : \mathbf{V}_{Y_1} \times \cdots \times \mathbf{V}_{Y_n} \longrightarrow \prod_{\mathbf{a} \in \mathbf{At_s}(Y_0)} \mathbf{V_a}$$

$$1 \leqslant i \leqslant n \; F_i[r] : \mathbf{V}_{Y_0} \times \mathbf{V}_{Y_1} \times \cdots \times \mathbf{V}_{Y_n} \longrightarrow \prod_{\mathbf{a} \in \mathbf{At_h}(Y_i)} \mathbf{V_a} \,.$$

On emploiera d'autre part les notations suivantes : si **V** est un ensemble qui est un produit d'ensembles $\mathbf{V_a}$, par abus d'écriture on notera $\mathbf{pr_a}$ la fonction de **V** dans $\mathbf{V_a}$ qui pour chaque famille de **V** donne sa composante sur $\mathbf{V_a}$. On notera d'autre part : $F_i[r, \mathbf{a}]$ la fonction $\mathbf{pr_a} \circ F_i(r)$ $(0 \leqslant i \leqslant n)$. En général les fonctions $F_i[r, \mathbf{a}]$ ne dépendent pas de tous leurs arguments ; nous verrons, au paragraphe suivant, comment décrire formellement les arguments efficaces de ces fonctions et nous supposons ici que l'on sait dire, intuitivement, quels arguments de $F_i[r, \mathbf{a}]$ sont efficaces.

On utilise alors ces outils de la façon suivante :

Soit S un arbre syntaxique engendré par la grammaire G. Pour chaque nœud y_0 de S étiqueté par $\tilde{y}_0$, on crée des noms de variables $\mathbf{a}(y_0)$ associés à chaque attribut **a** de $\mathbf{At}(\tilde{y}_0)$. On note $\mathfrak{a}(y_0)$ la famille des $\mathbf{a}(y_0)$ quand **a** parcourt $\mathbf{At}(\tilde{y}_0)$:

$$\mathfrak{a}(y_0) = (\mathbf{a}(y_0))_{\mathbf{a} \in \mathbf{At}(y_0)} \,.$$

Supposons que $\tilde{y}_0$ soit un non-terminal, y_0 n'est pas alors une feuille de S et admet donc des fils qui sont dans l'ordre $y_1, y_2, \dots, y_n$. Par définition d'un arbre syntaxique, il existe

une règle r de G de la forme $\tilde{y}_0 ::= \tilde{y}_1 \tilde{y}_2 \ldots \tilde{y}_n$ où $\tilde{y}_i$ désigne l'étiquette de y_i. Pour chaque y_0 étiqueté par un non-terminal $\tilde{y}_0$, on pose alors les équations suivantes :

$$\mathbf{a}(y_0) = F_0[r, \mathbf{a}] (a(y_1), \ldots, a(y_n)) \qquad \text{pour tout } \mathbf{a} \in \mathbf{At_s}(\tilde{y}_0) ,$$
$$\mathbf{a}(y_i) = F_i[r, \mathbf{a}] (a(y_0), a(y_1), \ldots, a(y_n)) \qquad \text{pour tout } \mathbf{a} \in \mathbf{At_h}(\tilde{y}_0) .$$

On obtient alors un système $\mathscr{S}(S)$, où chaque inconnue $\mathbf{a}(y)$ intervient une fois et une seule en partie gauche d'une équation.

Exemple 6

Reprenons l'exemple du paragraphe 9.3 et considérons la règle de grammaire $L_1 ::= L_2\ B$ à laquelle sont associées les règles sémantiques :

$$\mathbf{v}(L_1) = \mathbf{v}(L_2) + \mathbf{v}(B)\,,\quad \mathbf{l}(L_1) = \mathbf{l}(L_2) + 1\,,\quad \mathbf{s}(L_2) = \mathbf{s}(L_1) + 1\,,\quad \mathbf{s}(B) = \mathbf{s}(L_1)\,;$$

v et **l** sont des attributs synthétisés alors que **s** est hérité. Les fonctions qui sont associées à cette règle et qui permettent de redécouvrir les règles sémantiques ci-dessus sont :

$$F_0[L_1 ::= L_2\ B] : (\mathbf{V_v} \times \mathbf{V_s} \times \mathbf{V_l}) \times (\mathbf{V_v} \times \mathbf{V_s}) \rightarrow \mathbf{V_v} \times \mathbf{V_l}$$

telle que $F_0[L_1 ::= L_2\ B]\,((\alpha, \beta, \gamma), (\delta, \varepsilon)) = (\alpha + \delta, \gamma + 1)$

$$F_1[L_1 ::= L_2\ B] : (\mathbf{V_v} \times \mathbf{V_s} \times \mathbf{V_l}) \times (\mathbf{V_v} \times \mathbf{V_s} \times \mathbf{V_l}) \times (\mathbf{V_v} \times \mathbf{V_s}) \rightarrow \mathbf{V_s}$$

telle que $F_1[L_1 ::= L_2\ B]\,((\alpha, \beta, \gamma), (\alpha', \beta', \gamma'), (\alpha'', \beta'')) = \beta + 1$

$$F_2[L_1 ::= L_2\ B] : (\mathbf{V_v} \times \mathbf{V_s} \times \mathbf{V_l}) \times (\mathbf{V_v} \times \mathbf{V_s} \times \mathbf{V_l}) \times (\mathbf{V_v} \times \mathbf{V_s}) \rightarrow \mathbf{V_s}$$

telle que $F_2[L_1 ::= L_2\ B]\,((\alpha, \beta, \gamma), (\alpha', \beta', \gamma'), (\alpha'', \beta'')) = \beta$.

Si dans un arbre syntaxique S, on a une occurrence de cette règle syntaxique $L_1 ::= L_2\ B$, on aura dans le système $\mathscr{S}(S)$ les équations suivantes, en supposant que les nœuds étiquetés par L_1, L_2 et B ont pour noms y_1, y_2 et y_3 (cf. figure 20)

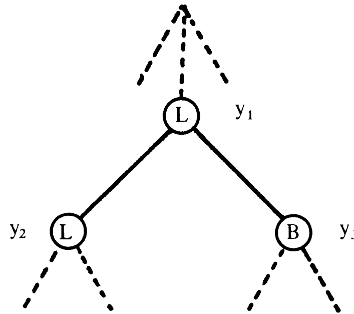


Figure 20.

$$\mathbf{v}(y_1) = F_0[L_1 ::= L_2\ B, \mathbf{v}]\,((\mathbf{v}(y_2), \mathbf{s}(y_2), \mathbf{l}(y_2)), (\mathbf{v}(y_3), \mathbf{s}(y_3)))$$
$$\mathbf{l}(y_1) = F_0[L_1 ::= L_2\ B, \mathbf{l}]\,((\mathbf{v}(y_2), \mathbf{s}(y_2), \mathbf{l}(y_2)), (\mathbf{v}(y_3), \mathbf{s}(y_3)))$$
$$\mathbf{s}(y_2) = F_1[L_1 ::= L_2\ B, \mathbf{s}]\,((\mathbf{v}(y_1), \mathbf{s}(y_1), \mathbf{l}(y_1)), (\mathbf{v}(y_2), \mathbf{s}(y_2), \mathbf{l}(y_2)), (\mathbf{v}(y_3), \mathbf{s}(y_3)))$$
$$\mathbf{s}(y_3) = F_2[L_1 ::= L_2\ B, \mathbf{s}]\,(\mathbf{v}(y_1), \mathbf{s}(y_1), \mathbf{l}(y_1)), (\mathbf{v}(y_2), \mathbf{s}(y_2), \mathbf{l}(y_2)), (\mathbf{v}(y_3), \mathbf{s}(y_3)) .$$

Ce système admet une solution si, et seulement si on peut ordonner ses équations de façon que si l'équation $\mathbf{a}(y_i) = F_i[r, \mathbf{a}]$ (...) a le numéro j, alors chacun des arguments efficaces de $F_i[r, \mathbf{a}]$ correspond à une variable $\mathbf{a}'(z)$ qui est définie dans une équation de rang inférieur strictement à j. Pour chaque système de ce type il est facile de savoir s'il admet ou non une solution. En effet, ce problème se ramène immédiatement à tester l'existence d'un circuit dans un graphe. En revanche, il est moins simple de savoir si tous les arbres syntaxiques S engendrés par G vont donner naissance à des systèmes $\mathscr{S}(S)$ ayant toujours une solution. Nous allons présenter au paragraphe suivant des algorithmes permettant de répondre à cette question.

9.5 Algorithmes testant les circularités dans les définitions des attributs

a) Formalisation de la notion de variable efficace

Nous reprenons dans ce paragraphe les notations introduites précédemment. Le premier concept à formaliser est celui d'argument efficace d'une fonction du type $F_i[r, \mathbf{a}]$. On réalise ceci en associant à chaque règle r de G de la forme $Y_0 ::= Y_1 Y_2 \dots Y_n$ un ensemble E(r) défini par

$$E(r) = \bigcup_{i=0}^{n} (\mathbf{At}(Y_i) \times \{ i \})$$

et une relation Γ(r) sur E(r) telle que :

$$(\mathbf{a}, i)\ \Gamma(r)\ (\mathbf{b}, j) \Rightarrow (j = 0\ \underline{\text{et}}\ \mathbf{b} \in \mathbf{At_s}(Y_0))\ \underline{\text{ou}}\ (j > 0\ \underline{\text{et}}\ \mathbf{b} \in \mathbf{At_h}(Y_j))\,.$$

Intuitivement, on doit construire Γ(r) de telle sorte que $(\mathbf{a}, i)\ \Gamma(r)\ (\mathbf{b}, j)$ signifie que la valeur de l'attribut **a** de Y_i est utilisée pour calculer la valeur de l'attribut **b** de Y_j.

Exemple 7

Considérons encore une fois la règle $L_1 ::= L_2\ B$ à laquelle on a associé les règles sémantiques

$$\mathbf{v}(L_1) = \mathbf{v}(L_2) + \mathbf{v}(B),\ \mathbf{l}(L_1) = \mathbf{l}(L_2) + 1,\ \mathbf{s}(L_2) = \mathbf{s}(L_1) + 1,\ \mathbf{s}(B) = \mathbf{s}(L_1)\,.$$

On a alors :

$$E(r) = \{ (\mathbf{v}, 0), (\mathbf{v}, 1), (\mathbf{v}, 2), (\mathbf{l}, 0), (\mathbf{l}, 1)\ (\mathbf{s}, 0), (\mathbf{s}, 1), (\mathbf{s}, 2) \}$$

et Γ(r) défini par

$$(\mathbf{v}, 1)\ \Gamma(r)\ (\mathbf{v}, 0) \quad (\mathbf{v}, 2)\ \Gamma(r)\ (\mathbf{v}, 0)$$

$$(\mathbf{l}, 1)\ \Gamma(r)\ (\mathbf{l}, 0)$$

$$(\mathbf{s}, 0)\ \Gamma(r)\ (\mathbf{s}, 2) \quad (\mathbf{s}, 0)\ \Gamma(r)\ (\mathbf{s}, 1)\,.$$

On préférera dans la pratique une représentation de Γ(r) sous une forme graphique du type de la figure 21.

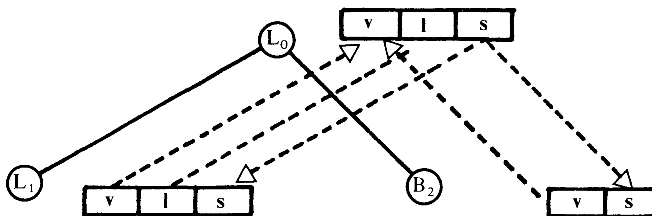


Figure 21.

b) Prolongement des relations Γ(r) *aux arbres syntaxiques généralisés*

Afin de pouvoir expliciter des algorithmes testant la non-circularité dans les définitions des attributs, nous sommes conduits à introduire la notion d'arbre syntaxique généralisé.

Un arbre syntaxique généralisé S engendré par une grammaire algébrique $G = (N, T, ::=, X)$ est un arbre répondant aux mêmes conditions qu'un arbre syntaxique sauf que sa racine est une lettre quelconque du vocabulaire de G. A un tel arbre S, on peut associer un système d'équation $\mathcal{S}(S)$ de manière analogue à ce qui a été fait dans le cas d'un arbre syntaxique au paragraphe 9.4. Notons cependant qu'en général $\mathcal{S}(S)$ n'a pas de solution, car les attributs hérités de la racine de S ne sont pas définis.

Considérons un arbre syntaxique généralisé S, engendré par la grammaire G, et le système $\mathcal{S}(S)$ associé. Nous allons utiliser les relations Γ(r) introduites précédemment pour définir une relation Γ(S) sur l'ensemble $\mathfrak{a}(S)$ des inconnues du système. Notons y_0 un point de S, $y_1, y_2, \ldots, y_n$ ses fils et r la règle :

$$\tilde{y}_0 ::= \tilde{y}_1\ \tilde{y}_2 \cdots \tilde{y}_n\,.$$

On pose $\mathbf{a}(y_i)\ \Gamma(S)\ \mathbf{b}(y_j)$ si, et seulement si on a $(\mathbf{a}, i)\ \Gamma(r)\ (\mathbf{b}, j)$. Intuitivement $\mathbf{a}(y_i)\ \Gamma(S)\ \mathbf{b}(y_j)$ signifie que, dans le système associé à S, l'équation de membre gauche $\mathbf{b}(y_j)$ a, dans son second membre, $\mathbf{a}(y_j)$ comme argument efficace.

c) Condition de non-circularité

Rappelons qu'étant donnée une relation Γ, on appelle fermeture transitive de Γ et on note Γ^+, la plus petite relation transitive contenant Γ. On montre que $\Gamma^+ = \bigcup_{n=1}^{\infty} \Gamma^n$, Γ^n étant le produit de la relation Γ, n fois par elle-même. Nous sommes maintenant capables d'énoncer correctement le problème que l'on cherche à résoudre. Il s'agit de montrer que pour tout arbre syntaxique S engendré par G la relation Γ(S) est sans circuit ; c'est-à-dire qu'il faut vérifier la formule suivante où BL(G) désigne l'ensemble des arbres syntaxiques engendrés par G :

$$(1) \qquad (\forall S \in BL(G)) \quad (\forall \mathbf{a}(y) \in \mathfrak{a}(S))\,(\underline{\text{non}}\ \mathbf{a}(y)\ \Gamma(S)^+\ \mathbf{a}(y))\,.$$

d) Transformation de la condition (1)

Sous la forme ci-dessus la condition de non-circularité (1) ne se prête pas à un traitement algorithmique. Nous allons ici en donner une forme équivalente conduisant, grâce à des formules de récurrence, à des algorithmes de résolution.

Appelons BLG(G) l'ensemble des arbres syntaxiques généralisés engendrés par G. Pour un arbre S de BLG(G), appelons y le nœud racine de S et $\mathfrak{a}'(S)$ les variables de $\mathfrak{a}(S)$ de la forme $\mathbf{a}(y)$. Montrons alors que la formule (1) est équivalente à la formule (2) suivante :

$$(2) \qquad (\forall S \in BLG(G)) \quad (\forall \mathbf{a}(y) \in \mathfrak{a}'(S))\,(\text{non}\ \mathbf{a}(y)\ \Gamma(S)^+\ \mathbf{a}(y))$$

(la condition de non-circularité (2) teste seulement les circularités à la racine des arbres, mais la vérification doit se faire sur tous les éléments de BLG(G) et non plus seulement sur BL(G)). La démonstration de l'équivalence des formes (1) et (2) résulte immédiatement des deux remarques suivantes :

— une circularité dans la définition des attributs pour un arbre syntaxique S implique une circularité dans la définition des attributs pour le sous-arbre syntaxique généralisé S′ dont la racine est le nœud le plus « haut » dans S concerné par cette circularité.

— Si un arbre syntaxique généralisé S présente une circularité dans la définition des

attributs, on peut le compléter en un arbre syntaxique qui présente évidemment cette même circularité (le fait que la grammaire G soit réduite intervient ici).

e) Formule de récurrence à la base des algorithmes de non-circularité

Dans la suite nous supposons que les nœuds des arbres syntaxiques généralisés sont toujours choisis de la manière suivante :

0 pour la racine, 1, 2, ... pour les successeurs de la racine.

Un tel arbre S est représenté par la figure 22.

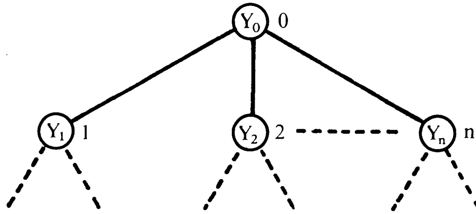


Figure 22.

Si $S_1, S_2, \ldots, S_n$ désignent les sous-arbres de S de racines respectives $Y_1, \ldots, Y_n$ on écrit alors $S = Y_0 \times (S_1 + S_2 + \cdots + S_n)$.

Soit S un arbre syntaxique généralisé, soit r la règle de grammaire correspondant à la racine de S. On considère la relation $\Delta(S)$ qui est la restriction de $\Gamma(S)^+$ aux variables $\mathbf{a}(y)$ telles que $y \in [0 : n]$. D'après le paragraphe précédent, pour que la définition des attributs soit sans circularité il faut et il suffit que la relation $\Delta(S)$ soit sans circularité pour tout S (on a vu qu'il suffisait même de prendre $y = 0$). Les nouvelles relations présentent l'intérêt de vérifier des relations de récurrence qui vont permettre de les calculer. En effet, si S peut se mettre sous la forme $S = Y_0 \times (S_1 + S_2 + \cdots + S_n)$, on obtient :

$$(3) \qquad \Delta(S) = [\Phi_1(\Delta(S_1)) \ \underline{\text{ou}} \ldots \underline{\text{ou}} \ \Phi_{n(r)} \ (\Delta(S_n)) \ \underline{\text{ou}} \ \Gamma(r)]^+$$

où Φ_k est un opérateur sur les relations défini par :

$$[\mathbf{a}(k) \ \Phi_k(R) \ \mathbf{a}'(k) \Leftrightarrow \mathbf{a}(0) \ R \ \mathbf{a}'(0)]$$

(cet opérateur transforme la numérotation pour tenir compte de la numérotation canonique des nœuds des arbres) et $\Gamma(r)$ est la relation associée à la règle r étendue aux variables par l'abus d'écriture $\mathbf{a}(i) \ \Gamma(r) \ \mathbf{a}'(i') \Leftrightarrow (\mathbf{a}, i) \ \Gamma(r) \ (\mathbf{a}', i')$.

Les relations (3) forment un système infini liant les relations $\Delta(S)$. Le caractère infini de ce système ne permet pas une résolution pratique. Pour pallier cette difficulté, on va regrouper les relations $\Delta(S)$ dans des ensembles disjoints en posant :

$$\delta(Y) = \{ \Delta(S) \mid \text{la racine de S est étiquetée par Y} \} .$$

Remarquons que les ensembles $\delta(Y)$ sont finis : En effet les relations $\Delta(S)$ sont définies sur l'ensemble fini $\{ \mathbf{a}(i) \mid \mathbf{a} \text{ est un attribut et } 0 \leqslant i \leqslant m \}$ où m est la longueur maximale d'une règle.

Si Y est un non-terminal, $\delta(Y)$ vérifie l'équation à point fixe suivante :

$$\delta(Y) = \{ [\Phi_1(R_1) \ \underline{\text{ou}} \ \Phi_2(R_2) \ \underline{\text{ou}} \ldots \underline{\text{ou}} \ \Phi_n(R_n) \ \underline{\text{ou}} \ \Gamma(r)]^+ \mid \text{r est une règle du type } Y ::= Y_1 \, Y_2 \ldots Y_n \text{ et } R_i \in \delta(Y_i) \text{ pour } i \in [1 : n] \} .$$

Si Y est un terminal, $\delta(Y)$ contient uniquement la relation vide notée Ω. Ces caractérisations de $\delta(Y)$ sont à la base des algorithmes de non-circularité. Il suffit en effet de calculer ces ensembles et de vérifier la condition :

$$(\forall X \in N)\ (\forall R \in \delta(X))\ (\forall \mathbf{a})\ (\forall i)\ (\text{non } \mathbf{a}(i)\ R\ \mathbf{a}(i))\,.$$

Ceci pourra se faire de manière effective puisque l'ensemble N, les ensembles $\delta(X)$ et les ensembles sur lesquels sont définies les relations de $\delta(X)$ sont finis.

f) Un algorithme testant la non-circularité

Nous allons calculer les ensembles $\delta(Y)$ par approximation successive en considérant des arbres syntaxiques de hauteur croissante (la hauteur d'un arbre est par définition la longueur d'un plus long chemin allant de la racine à une feuille). On note h(S) la hauteur d'un arbre S.

On pose par définition

$\delta_k(Y) = \{\ \Delta(S) \mid S$ est un arbre syntaxique généralisé de racine étiqueté par Y et de hauteur au plus $k + 1\ \}$.

En utilisant les relations caractérisant les $\delta(Y)$ on obtient :

$$\begin{aligned}
&Y \in T && \Rightarrow \forall k \in \mathbb{N}\ \ \delta_k(Y) = \{\ \Omega\ \}\\
&Y \in N && \Rightarrow \delta_0(Y) = \varnothing\\
&Y \in N\ \underline{\text{et}}\ k > 0 && \Rightarrow \delta_k(Y) = \{\ [\Phi_1(R_1)\ \underline{\text{ou}} \dots \underline{\text{ou}}\ \Phi_n(R_n)\ \underline{\text{ou}}\ \Gamma(r)]^+ \mid\\
&&& \qquad r \text{ est du type } Y ::= Y_1 \dots Y_n \text{ et } R_i \in \delta_{k-1}(Y_i)\\
&&& \qquad \text{pour } i \in [1 : n]\ \}\,.
\end{aligned}$$

De façon évidente, on a $\delta(Y) = \bigcup_{k \geq 0} \delta_k(Y)$ mais il n'est pas nécessaire de calculer $\delta_k(Y)$ pour tous les k ; en effet, les ensembles $\delta_k(Y)$ vérifient les deux propriétés :

$$\begin{cases} (\forall Y)\ \delta_k(Y) = \delta_{k+1}(Y) \Rightarrow (\forall Y)\ \delta_{k+1}(Y) = \delta_{k+2}(Y)\\ \qquad \delta_k(Y) \subseteq \delta_{k+1}(Y)\,. \end{cases}$$

Donc on aura $\delta(Y) = \delta_{k_0}(Y)$ où k_0 est le premier entier tel que $\delta_{k_0}(Y) = \delta_{k_0+1}(Y)$. De plus cet entier k_0 existe car tous les ensembles sur lesquels on travaille sont finis.

g) Exemples d'application de l'algorithme précédent

L'écriture à la main de cet algorithme est fastidieuse. Nous nous sommes donc limités à un exemple simple (qui est cependant plus complexe que ceux que l'on cite habituellement dans la littérature).

Exemple 8

Considérons la grammaire $G = (\{\ X, A, B\ \}, \{\ a, b\ \}, ::=, X)$ telle que

$$\begin{aligned} X &::= BA\\ B &::= BA \mid b\\ A &::= a\,. \end{aligned}$$

Considérons par ailleurs la définition par attributs suivante :

$$\begin{aligned}
&\mathbf{At_s}(X) = \{\ \mathbf{a}, \mathbf{b}, \mathbf{c}\ \}\,, && \mathbf{V_a} = \mathbf{V_b} = \mathbf{V_c} = \mathbb{N}\\
&\mathbf{At_s}(B) = \{\ \mathbf{a}, \mathbf{d}\ \}\,, && \mathbf{V_d} = \mathbb{N}\\
&\mathbf{At_h}(B) = \{\ \mathbf{c}\ \}
\end{aligned}$$

$$\mathbf{At}_{\mathbf{s}}(\mathrm{A}) = \varnothing$$
$$\mathbf{At}_{\mathbf{h}}(\mathrm{A}) = \{\ \mathbf{c}\ \}\ .$$

Les règles sémantiques associées aux règles syntaxiques sont décrites dans la figure 23.

Nom de la règle	Règle syntaxique	Règle sémantique
r	X ::= BA	$\mathbf{a}(X) = \mathbf{c}(A) + 1$ $\mathbf{b}(X) = \mathbf{a}(B) + 2$ $\mathbf{c}(X) = \mathbf{0}$ $\mathbf{c}(B) = \mathbf{2} \times \mathbf{a}(X) + \mathbf{1}$ $\mathbf{c}(A) = \mathbf{b}(X) + \mathbf{1}$
r′	B_1 ::= B_2 A	$\mathbf{a}(B_1) = \mathbf{c}(A) + \mathbf{1}$ $\mathbf{d}(B_1) = \mathbf{d}(B_2) + \mathbf{c}(B_2)$ $\mathbf{c}(B_2) = \mathbf{c}(B_1) + \mathbf{1}$ $\mathbf{c}(A) = 2 \times \mathbf{d}(B_1) + 3$
r″ r‴	B ::= *b* A ::= *a*	$\mathbf{a}(B) = \mathbf{0}$ $\mathbf{d}(B) = \mathbf{0}$

Figure 23.

Cette définition par attributs est-elle toujours sans circularité ? Commençons par construire les relations $\Gamma(r)$, $\Gamma(r')$, $\Gamma(r'')$, $\Gamma(r''')$ (figure 24).

Relation	Définition formelle	Représentation graphique de P(r)
Γ(r)	$(\mathbf{c}, 2)\ \Gamma(r)\ (\mathbf{a}, 0)$ $(\mathbf{a}, 1)\ \Gamma(r)\ (\mathbf{b}, 0)$ $(\mathbf{a}, 0)\ \Gamma(r)\ (\mathbf{c}, 1)$ $(\mathbf{b}, 0)\ \Gamma(r)\ (\mathbf{c}, 2)$	X_0 [a b c] B_1 [c a d] A_2 [c]
Γ(r′)	$(\mathbf{c}, 2)\ \Gamma(r')\ (\mathbf{a}, 0)$ $(\mathbf{c}, 0)\ \Gamma(r')\ (\mathbf{c}, 1)$ $(\mathbf{d}, 1)\ \Gamma(r')\ (\mathbf{d}, 0)$ $(\mathbf{d}, 0)\ \Gamma(r')\ (\mathbf{c}, 2)$ $(\mathbf{c}, 1)\ \Gamma(r')\ (\mathbf{d}, 0)$	B_0 [c a d] B_1 [c a d] A_2 [c]
Γ(r″)	relation vide Ω	rien
Γ(r‴)	relation vide Ω	rien

Figure 24.

Construisons maintenant les ensembles $\delta_k(Y)$ pour $Y = X, A, B, a, b$ (figure 25).

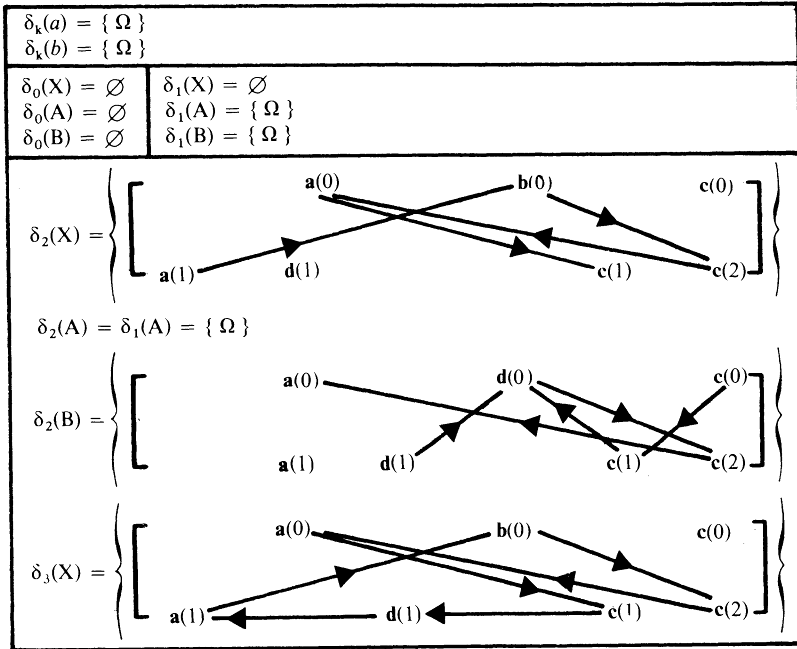


Figure 25.

On rencontre une circularité dans la définition de $\delta_3(X)$; il est alors inutile de poursuivre l'algorithme. On pourra d'ailleurs vérifier en essayant de calculer les attributs des différents nœuds de l'arbre syntaxique de la figure 26.

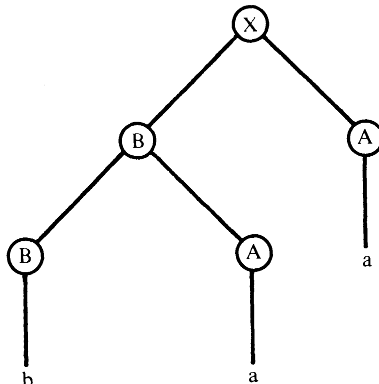


Figure 26. □

Exemple 9

Appliquons l'algorithme ci-dessus à l'exemple du paragraphe 9.3

$$G = (\{ N, L, B \}, \{ 0, 1, . \}, ::=, N).$$

Les définitions des règles sémantiques et de Γ sont résumées par la figure 27.

Nom de la règle	Règle syntaxique	Règle sémantique	Γ	Représentation de $\Gamma(r)$
r_1	$B ::= 0$	$v(B) = 0$	relation vide	rien
r_2	$B ::= 1$	$v(B) = 2^{s(B)}$	$(s, 0)\ \Gamma\ (r_2)\ (v, 0)$	B_0 v s
r_3	$L ::= B$	$v(L) = v(B)$ $s(B) = s(L$ $l(L) = 1$	$(v, 1)\ \Gamma\ (r_3)\ (v, 0)$ $(s, 0)\ \Gamma\ (r_3)\ (s, 1)$	L_0 v l s B_1 v s
r_4	$L_1 ::= L_2\ B$	$v(L_1) = v(L_2) + v(B)$ $s(B) = s(L_1)$ $s(L_2) = s(L_1) + 1$ $l(L_1) = l(L_2) + 1$	$(v, 1)\ \Gamma\ (r_4)\ (v, 0)$ $(v, 2)\ \Gamma\ (r_4)\ (v, 0)$ $(s, 0)\ \Gamma\ (r_4)\ (s, 2)$ $(s, 0)\ \Gamma\ (r_4)\ (s, 1)$ $(l, 1)\ \Gamma\ (r_4)\ (l, 0)$	L_0 v l s L_1 v l s B_2 v s
r_5	$N ::= L$	$v(N) = v(L)$ $s(L) = 0$	$(y, 1)\ \Gamma\ (r_5)\ (v, 0)$	N_0 v L_1 v l s
r_6	$N ::= L_1 . L_2$	$v(N) = v(L_1) + v(L_2)$ $s(L_1) = 0$ $s(L_2) = -l(L_2)$	$(v, 1)\ \Gamma\ (r_6)\ (v, 0)$ $(v, 3)\ \Gamma\ (r_6)\ (v, 0)$ $(l, 3)\ \Gamma\ (r_6)\ (s, 3)$	N_0 v L_1 v l s L_2 v l s

Figure 27.

Commençons le calcul des ensembles $\delta_k(Y)$ pour $Y = 0, 1, ., N, B, L$ (figure 28).

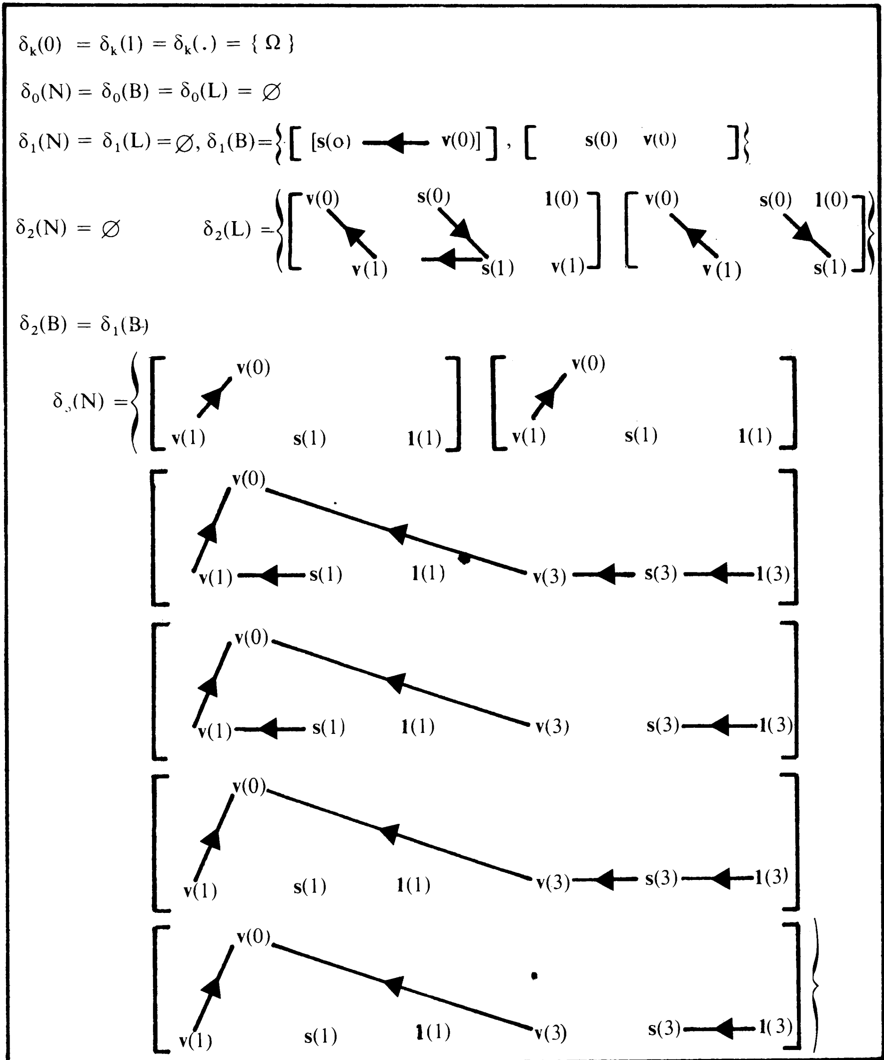


Figure 28.

Exercice 19

Finir le calcul des ensembles $\delta_k(\ )$ précédents. Qu'en déduit-on ? Ce résultat est-il surprenant ? □

9.6 Rôle de sélection des attributs

Comme nous l'avons annoncé dans l'introduction, on peut utiliser la méthode des attributs pour définir un sous-langage du langage initialement

retenu. On convient, en général, d'éliminer les phrases pour les arbres syntaxiques desquels le calcul des attributs n'est pas possible. Traitons deux exemples, l'un théorique, l'autre plus pratique pour illustrer cet aspect de la méthode des attributs :

Exemple 10

Considérons la grammaire $G = (N, T, ::=, X)$ définie par

$$N = \{ X, A, B, C \}, \quad T = \{ a, b, c \}$$

et les règles $X ::= ABC$, $A ::= Aa \mid a$, $B ::= Bb \mid b$, $C ::= Cc \mid c$; la grammaire G engendre $L = a^* \, a \, b^* \, b \, c^* \, c$; nous allons sélectionner à l'aide de la méthode des attributs le sous-langage

$$L' = \{ a^n b^n c^n \mid n \geqslant 1 \} \text{ de } L .$$

On prend $\mathbf{At}(X) = \{ \mathbf{v} \}$

$\mathbf{At}(A) = \mathbf{At}(B) = \mathbf{At}(C) = \{ \mathbf{n} \}$

avec $\mathbf{V_n} = \mathbf{V_v} = \mathbb{N}$.

Les règles syntaxiques et sémantiques sont :

règles syntaxiques	règles sémantiques
$A ::= a$	$\mathbf{n}(A) = \mathbf{1}$
$A_1 ::= A_2 \, a$	$\mathbf{n}(A_1) = \mathbf{n}(A_2) + \mathbf{1}$
$B ::= b$	$\mathbf{n}(B) = \mathbf{1}$
$B_1 ::= B_2 \, b$	$\mathbf{n}(B_1) = \mathbf{n}(B_2) + \mathbf{1}$
$C ::= c$	$\mathbf{n}(C) = \mathbf{1}$
$C_1 ::= C_2 \, c$	$\mathbf{n}(C_1) = \mathbf{n}(C_2) + \mathbf{1}$
$X ::= ABC$	$\mathbf{v}(X) = \underline{\mathbf{si}}\ \mathbf{n}(A) = \mathbf{n}(B) = \mathbf{n}(C)\ \underline{\mathbf{alors}}\ \mathbf{1}\ \underline{\mathbf{sinon}}\ \mathbf{v}(X)$.

On a procédé de la manière suivante : l'attribut **n** a collecté dans l'arbre syntaxique un certain nombre de renseignements. Au point X on vérifie à l'aide de l'attribut **v** que les conditions requises sont vérifiées. Si ces conditions ne sont pas vérifiées le calcul de l'attribut **v** en X boucle et la phrase est rejetée.

□

Exercice 20

En reprenant la même grammaire G que dans l'exemple précédent, sélectionner par la méthode des attributs le sous-langage suivant de L

$$L_1 = \{ a^n b^n c^p ; n \geqslant 1, p \geqslant 1 \} \cup \{ a^p b^n c^n ; n \geqslant 1, p \geqslant 1 \} .$$

□

Remarquons que cette méthode de sélection est extrêmement efficace et qu'en théorie elle permet de sélectionner un sous-langage quelconque d'un langage algébrique (voir exercice 21).

Dans la pratique, on utilise cette méthode pour rendre compte de contraintes

contextuelles souvent présentes dans les langages de programmation et qui ne peuvent être maîtrisées par les grammaires algébriques.

Exercice 21

Soit L un langage algébrique engendré par G = (N, T, ::=, X) et L′ un sous-langage de L défini par une propriété caractéristique P(α) de ses mots. Montrer que L′ peut être défini à partir de L par la méthode des attributs. □

Exemple 17

Considérons le petit langage de programmation P engendré par la grammaire G décrite par la figure 29 :

Règles syntaxiques	Signification intuitive des symboles utilisés
⟨ P ⟩ ::= *début* ⟨ PD ⟩ ; ⟨ PI ⟩ *fin*	⟨ P ⟩ = programme
⟨ PD ⟩ ::= ⟨ SD ⟩	⟨ PD ⟩ = partie déclaration
	⟨ PI ⟩ = partie instruction
⟨ SD ⟩ ::= ⟨ D ⟩ | ⟨ D ⟩ ; ⟨ SD ⟩	⟨ SD ⟩ = suite de déclaration
⟨ D ⟩ ::= *entier* ⟨ SID ⟩ | *réel* SID ⟩	⟨ D ⟩ = déclaration
⟨ SID ⟩ ::= ⟨ ID ⟩ | ⟨ ID ⟩ ; ⟨ ID ⟩	⟨ SID ⟩ = suite d'identificateurs
⟨ ID ⟩ ::= *a* | *a* ⟨ ID ⟩	⟨ ID ⟩ = identificateur
⟨ PI ⟩ ::= ⟨ SI ⟩	⟨ SI ⟩ = suite d'instruction
⟨ SI ⟩ ::= ⟨ I ⟩ | ⟨ I ⟩ ; ⟨ SI ⟩	⟨ I ⟩ = instruction
⟨ I ⟩ ::= ⟨ MG ⟩ ::= ⟨ MD ⟩	⟨ MG ⟩ = membre gauche
⟨ MG ⟩ ::= ⟨ ID ⟩	⟨ MD ⟩ = membre droit
⟨ AD ⟩ ::= ⟨ E ⟩	⟨ E ⟩ = expression
⟨ E ⟩ ::= ⟨ ES ⟩ | ⟨ E ⟩ ⟨ 0 ⟩ ⟨ ES ⟩	⟨ ES ⟩ = expression simple
⟨ 0 ⟩ ::= + | −	⟨ 0 ⟩ = opérateur
⟨ ES ⟩ ::= ⟨ EN ⟩ | ⟨ ID ⟩ |(⟨ E ⟩) ⟨ CR ⟩	⟨ EN ⟩ = entier
	⟨ CR ⟩ = constante réelle
⟨ EN ⟩ ::= ⟨ EP ⟩ | 0	⟨ EP ⟩ = entier positif
⟨ EP ⟩ ::= ⟨ EP ⟩ ⟨ C ⟩ | ⟨ NZ ⟩	
⟨ C ⟩ ::= 0 | ⟨ NZ ⟩	⟨ C ⟩ = chiffre
⟨ NZ ⟩ ::= 1 | 2 | ... | 9	⟨ NZ ⟩ = non zéro
⟨ CR ⟩ ::= ⟨ EN ⟩.⟨ EN ⟩ | ⟨ EN ⟩ .	

Figure 29.

Nous allons sélectionner par la méthode des attributs le sous-langage P′ de P qui correspond aux contraintes contextuelles suivantes :

1) tout identificateur utilisé dans la partie instruction doit être déclaré,
2) tout identificateur utilisé en partie droite d'une instruction doit avoir été

utilisé en partie gauche dans une instruction précédente (c'est-à-dire toute variable doit être initialisée),

3) aucun identificateur ne peut être déclaré plus d'une fois.

On va utiliser quatre attributs **d** (déclaré), **i1**, **i2** (initialisé) et **v** (vérification). $\mathscr{P}(a^*)$ est le domaine associé aux trois premiers attributs, { 1 } est associé au dernier. Reprenons maintenant la grammaire règle par règle en indiquant les règles sémantiques associées (figure 30).

Règles syntaxiques	Règles sémantiques	Commentaires
$\langle ID \rangle ::= a$ $\langle ID_1 \rangle ::= a \langle ID_2 \rangle$ $\langle SID \rangle ::= \langle ID \rangle$ $\langle SID_1 \rangle ::= \langle ID \rangle \langle SID_2 \rangle$	$\mathbf{d}(\langle ID \rangle) = \{a\}$ $\mathbf{d}(\langle ID_1 \rangle) = \{a\}\, \mathbf{d}(\langle ID_2 \rangle)$ $\mathbf{d}(\langle SID \rangle) = \mathbf{d}(\langle ID \rangle)$; $\mathbf{v}(\langle SID \rangle) = 1$ $\mathbf{d}(\langle SID_1 \rangle) = \mathbf{d}(\langle ID \rangle) \cup \mathbf{d}(\langle SID_2 \rangle)$ $\mathbf{v}(\langle SID_1 \rangle) =$ si $\mathbf{d}(\langle ID \rangle) \subseteq \mathbf{d}(SID_2)$ alors $\mathbf{v}(\langle SID_1 \rangle)$ sinon $\mathbf{v}(\langle SID_2 \rangle)$	On traite ici la partie déclaration, l'attribut **d** est synthétisé dans cette partie et enregistre les identificateurs qui sont déclarés. L'attribut **v** sert à tester les doubles déclarations qui l'amènent à boucler.
$\langle D \rangle ::=$ <u>entier</u> $\langle SID \rangle$	$\mathbf{d}(\langle D \rangle) = \mathbf{d}(\langle SID \rangle)$ $\mathbf{v}(\langle D \rangle) = \mathbf{v}(\langle SID \rangle)$	Remarquons que dans la pratique on ne fera pas boucler l'évaluation de **v**, mais on remplacera cela par un message d'erreur.
$\langle SD \rangle ::= \langle D \rangle$	$\mathbf{d}(\langle SD \rangle) = \mathbf{d}(\langle D \rangle)$ $\mathbf{v}(\langle SD \rangle) = \mathbf{v}(\langle D \rangle)$	
$\langle SD_1 \rangle ::= \langle D \rangle ; \langle SD_2 \rangle$	$\mathbf{d}(\langle SD_1 \rangle) = \mathbf{d}(\langle D \rangle) \cup \mathbf{d}(\langle SD_2 \rangle)$ $\mathbf{v}(\langle SD_1 \rangle) =$ si $\mathbf{d}(\langle D \rangle) \cap \mathbf{d}(\langle SD_2 \rangle) \neq \varnothing$ alors $\mathbf{v}(\langle SD_1 \rangle)$ sinon $\mathbf{v}(\langle SD_2 \rangle) \wedge \mathbf{v}(\langle D \rangle)$	(On a employé dans l'évaluation de **v** une fonction standard notée $\wedge$ qui vaut 1 si et seulement si ses 2 arguments valent 1.)
$\langle PD \rangle ::= \langle SD \rangle$	$\mathbf{d}(\langle PD \rangle) = \mathbf{d}(\langle SD \rangle)$ $\mathbf{v}(\langle PD \rangle) = \mathbf{v}(\langle SD \rangle)$	
$\langle P \rangle ::=$ <u>début</u> $\langle PD \rangle$; $\langle PI \rangle$ <u>fin</u>	$\mathbf{d}(\langle P \rangle) = \mathbf{d}(\langle PD \rangle)$ $\mathbf{v}(\langle P \rangle) = \mathbf{v}(\langle PD \rangle) \wedge \mathbf{v}(\langle PI \rangle)$ $\mathbf{i}_1(\langle P \rangle) = \varnothing$; $\mathbf{i}_1(\langle PI \rangle) = \varnothing$ $\mathbf{d}(\langle PI \rangle) = \mathbf{d}(\langle P \rangle)$	Ici on transmet, par l'intermédiaire de l'attribut **d**, de l'information provenant de la partie déclaration vers la partie instruction. En particulier dans la partie instruction l'attribut **d** sera hérité.
$\langle PI \rangle ::= \langle SI \rangle$	$\mathbf{d}(\langle SI \rangle) = \mathbf{d}(\langle PI \rangle)$ $\mathbf{i}_1(\langle BI \rangle) = \varnothing$ $\mathbf{v}(\langle PI \rangle) = \mathbf{v}(\langle SI \rangle)$	
$\langle SI \rangle ::= \langle I \rangle$	$\mathbf{d}(\langle I \rangle) = \mathbf{d}(\langle SI \rangle)$ $\mathbf{i}_1(\langle I \rangle) = \mathbf{i}_1(\langle SI \rangle)$ $\mathbf{i}_2(\langle SI \rangle) = \mathbf{i}_2(\langle I \rangle)$ $\mathbf{v}(\langle SI \rangle) = \mathbf{v}(\langle I \rangle)$	L'attribut $\mathbf{i}_1$ donne les variables qui sont déjà initialisées à cet endroit du programme. $\mathbf{i}_1$ est un attribut hérité $\mathbf{i}_2$ donne la variable qui est initialisée par l'instruction I. $\mathbf{i}_2$ sert à construire $\mathbf{i}_1$ et c'est un attribut synthétisé.
$\langle SI_1 \rangle ::= \langle I \rangle ; \langle SI_2 \rangle$	$\mathbf{d}(\langle I \rangle) = \mathbf{d}(\langle SI_1 \rangle)$ $\mathbf{d}(\langle SI_2 \rangle) = \mathbf{d}(\langle SI_1 \rangle)$ $\mathbf{i}_1(\langle I \rangle) = \mathbf{i}_1(\langle SI_1 \rangle)$ $\mathbf{i}_1(\langle SI_2 \rangle) = \mathbf{i}_1(\langle SI_1 \rangle) \cup \mathbf{i}_2(\langle I \rangle)$ $\mathbf{v}(\langle SI_1 \rangle) = \mathbf{v}(\langle I \rangle) \wedge \mathbf{v}(\langle SI_2 \rangle)$	
$\langle I \rangle ::= \langle MG \rangle \langle MD \rangle$	$\mathbf{i}_2(\langle I \rangle) = \mathbf{i}_2(\langle MG \rangle)$ $\mathbf{i}_1(\langle MG \rangle) = \mathbf{i}_1(\langle I \rangle)$ $\mathbf{d}(\langle MD \rangle) = \mathbf{d}(\langle I \rangle)$ $\mathbf{d}(\langle MG \rangle) = \mathbf{d}(\langle I \rangle)$ $\mathbf{v}(\langle I \rangle) = \mathbf{v}(\langle MG \rangle) \wedge \mathbf{v}(\langle MD \rangle)$	

Règles syntaxiques	Règles sémantiques	Commentaires
⟨ MG ⟩ ::= ⟨ Id ⟩	**v**(⟨ MG ⟩) = si **d**(⟨ ID ⟩) ⊂ **d**(⟨ MG ⟩) alors 1 sinon **v**(⟨ MG ⟩) $\mathbf{i}_2$(⟨ MG ⟩) = **d**(⟨ ID ⟩)	
⟨ MD ⟩ ::= ⟨ E ⟩	$\mathbf{i}_1$(⟨ E ⟩) = $\mathbf{i}_1$(⟨ MD ⟩) **d**(⟨ E ⟩) = **d**(⟨ MD ⟩) **v**(⟨ MD ⟩) = **v**(E)	
⟨ E ⟩ ::= ⟨ ES ⟩	$\mathbf{i}_1$(⟨ ES ⟩) = $\mathbf{i}_1$(⟨ ES ⟩) **d**(⟨ E ⟩) = **d**(⟨ ES ⟩) **v**(⟨ E ⟩) = **v**(⟨ ES ⟩)	
⟨ E_1 ⟩ ::= ⟨ E_2 ⟩ ⟨ 0 ⟩ ⟨ ES ⟩	$\mathbf{i}_1$(⟨ E_2 ⟩) = $\mathbf{i}_1$(⟨ E_1 ⟩) $\mathbf{i}_1$(⟨ ES ⟩) = $\mathbf{i}_1$(⟨ E_1 ⟩) **d**(⟨ E_2 ⟩) = **d**(⟨ E_1 ⟩) **d**(⟨ ES ⟩) = **d**(⟨ E_1 ⟩) **v**(⟨ E_1 ⟩) = **v**(⟨ E_2 ⟩) ∧ **v**(⟨ ES ⟩)	
⟨ ES ⟩ ::= ⟨ EN ⟩	**v**(⟨ ES ⟩) = **1**	
⟨ ES ⟩ ::= ⟨ CR ⟩	**v**(⟨ ES ⟩) = **1**	
⟨ ES ⟩ ::= ⟨ ID ⟩	**v**(⟨ ES ⟩) = si **d**(⟨ ID ⟩) ⊆ **d**(⟨ ES ⟩) ∩ $\mathbf{i}_1$(⟨ ES ⟩) alors **1** sinon **v**(⟨ ES ⟩)	
⟨ ES ⟩ ::= ⟨ E ⟩	**v**(⟨ ES ⟩) = **v**(⟨ E ⟩) **d**(⟨ E ⟩) = **d**(⟨ ES ⟩) $\mathbf{i}_1$(⟨ E ⟩) = $\mathbf{i}_1$(⟨ ES ⟩)	
⟨ EN ⟩ ::= ⟨ EP ⟩ | 0	rien	
⟨ EP ⟩ ::= ⟨ EP ⟩ ⟨ C ⟩ | ⟨ NZ ⟩	rien	
⟨ NZ ⟩ ::= 1 | 2 | ... | 9	rien	
⟨ CR ⟩ ::= ⟨ EN ⟩.⟨ EN ⟩	rien	
⟨ CR ⟩ ::= ⟨ EN ⟩.	rien	

Figure 30.

On se persuade facilement que l'évaluation de l'attribut **v** n'est possible, pour un programme engendré par G, que si ce programme vérifie les conditions imposées ci-dessus. □

On remarque, sur cet exemple proche de la réalité, que la méthode des attributs est facile à appliquer et est même assez naturelle ; cependant, si on veut automatiser l'évaluation des attributs sur des exemples réels, on risque d'obtenir des temps de calcul prohibitifs. Donner des algorithmes performants et une méthodologie de l'évaluation des attributs ont donc fait l'objet de travaux, dont un bon exemple est le projet Delta de LOHRO [LOH, 74].

Exercice 22

Pour le petit langage de programmation défini ci-dessus, sélectionner par la méthode des attributs les sous-langages qui vérifient les conditions suivantes :

1) Tout identificateur est déclaré et initialisé avant d'être utilisé. Les instructions ne contiennent que des objets d'un même type (entier ou réel).

2) Tout identificateur est déclaré et initialisé avant d'être utilisé. Un identificateur ne peut figurer qu'une fois en partie gauche d'une instruction (programmation en affectation unique) et tout identificateur déclaré est utilisé. □

Exercice 23

On considère toujours le petit langage de programmation défini ci-dessus. On considère seulement les programmes vérifiant les conditions 1), 2) et 3) énoncées ci-dessus et ne comportant pas d'expression mixte. On veut associer à chaque programme P une fonction $f_p : a^* \to \mathbb{R} \cup \{ w, d \}$ telle que :

$f_p(a^n) = w \Leftrightarrow a^n$ n'est pas déclaré dans P ,

$f_p(a^n) = d \Leftrightarrow a^n$ est déclaré mais pas utilisé dans P ,

$f_p(a^n) = b \Leftrightarrow a^n$ est déclaré et utilisé dans P et la valeur de a^n à la fin de l'exécution de P est b .

Comment définir cette fonction avec la méthode des attributs ?

Reprendre cet exercice en supposant que l'on autorise les expressions mixtes et que les conversions se font automatiquement. □

10 COMMENTAIRES BIBLIOGRAPHIQUES

Paragraphe 1 *à* 4

Signalons tout d'abord quelques ouvrages ou articles qui permettent de suivre l'évolution générale des travaux sur la sémantique des langages de programmation :

McCARTHY [McC, 63] présente un certain nombre d'idées importantes telles que la notion de vecteur d'état, la notion de syntaxe abstraite et la notion de sémantique opérationnelle.

STEEL [STE, 66] est un « instantané » des diverses approches proposées en 1964. Le lecteur y trouvera les germes des points de vue opérationnel (McCARTHY) et fonctionnel (STRACHEY) qui se sont développés considérablement dans la suite. L'auteur présente aussi certaines approches ayant connu un succès plus modeste, comme celles qui sont fondées sur les algorithmes de Markov (Van WIJNGAARDEN) ou sur le λ-calcul (LANDIN).

de BAKKER [BAK, 69] consiste en un panorama très détaillé de l'état de l'art en 1968. On peut remarquer l'apparition d'un nouveau point de vue basé sur les idées que FLOYD a développé à propos de la justification des programmes : l'approche axiomatique. On y trouve également quelques timides essais sur l'utilisation de définitions récursives.

Paragraphe 5

BLICKLE [BLI, 73] et FINANCE [FIN, 76] présentent deux formalisations basées sur la notion de calcul ; la première définit un calcul comme une suite d'états de mémoire alors que la deuxième le caractérise comme une suite de modifications élémentaires d'une information.

Paragraphe 6

Le but avoué des créateurs de la sémantique dénotationnelle, Christopher STRACHEY et Dana SCOTT, était de définir **la** sémantique d'un programme. Se référant à une fonction mathématique, image d'un programme, la plus abstraite et la plus indépendante d'une machine possible, on doit pouvoir démontrer (facilement !) des équivalences (en fait des égalités) de programmes ou de sémantiques. Citons quelques-unes des études faites à ce sujet :

GORDON [GOR, 75] montre l'équivalence pour le langage LISP entre une sémantique opérationnelle (la fonction EVAL) et la sémantique dénotationnelle. L'étude de PLOTKIN [PLO, 75] est plus fouillée et s'applique à LCF et par conséquent à tous les langages plus ou moins apparentés au λ-calcul.

MILNE [MIL, 74] (cf. aussi [MIL, 76]) définit différentes sémantiques dénotationnelles d'un langage de programmation (évoquant certains aspects d'ALGOL 68 : le parallélisme et les modes récursifs), il montre leur équivalence.

MILNER [MILR, 76] étudie sur un exemple simple de langage, quatre sémantiques, deux opérationnelles et deux dénotationnelles (avec ou sans continuations) ; il démontre leur équivalence et aborde le problème de la complexité de telles démonstrations, il propose de les mécaniser en utilisant LCF.

WADSWORTH et MORIS (impublié) ont introduit le concept de « continuation » décrit dans l'article [STR, 74] ; REYNOLDS [REY, 74] montre l'équivalence des sémantiques avec et sans « continuation » sur le λ-calcul.

D'autres auteurs essaient de définir la sémantique dénotationnelle d'un langage réaliste, citons par exemple APT et de BAKKER [APT, 76], DONAHUE [DON, 76], TENNENT [TEN, 76] pour PASCAL, le groupe de VIENNE BEKIC et *al.* [BEK, 74] pour PL1, TENNENT [TEN, 74] pour SNOBOL 4.

Le lecteur souhaitant une initiation plus complète en sémantique dénotationnelle trouvera dans l'article de TENNENT [TEN, 76] l'exposé des concepts qui motivent l'introduction des outils principaux et, pour revenir aux sources, il doit lire l'article de SCOTT et STRACHEY [SCO, 71].

Paragraphe 7

On trouvera dans HOARE et WIRTH [HOA, 73] la définition axiomatique de PASCAL. L'article de GORELICK [GORE, 75] présente un système axiomatique complet définissant la sémantique d'un petit langage de programmation.

Etudes comparatives

HOARE, LAUER [HOA, 74] est l'un des premiers articles présentant le concept de complémentarité. A cette occasion on trouvera une étude comparative des principales méthodes de formalisation de la sémantique, un peu analogue à celle que nous avons présentée ici.

DONAHUE [DON, 75] concerne la définition d'une importante partie de PASCAL présentée à la fois d'un point de vue dénotationnel et d'un point de vue axiomatique.

MILNER [MILR, 76] étudie la complémentarité des sémantiques opérationnelles et dénotationnelles.

Autres études

Dans des domaines plus particuliers citons trois méthodes.

La méthode de VIENNE : on trouve une présentation précise de cette méthode interprétative dans [LUC, 68] ; ce rapport est assez complet mais sa lecture est rendue un peu fastidieuse par l'introduction d'une terminologie spécifique introduite par les auteurs ; le lecteur trouve une présentation plus simple dans [NEU, 71] qui indique les idées à la base de cette formalisation.

La méthode des attributs : dans [KNU, 68] on trouve des notions de base sur la méthode, on y introduit la notion d'attributs hérités. Un algorithme de non-circularité est présenté ; cet algorithme est corrigé dans [KNU, 71]. Le projet DELTA [LOR, 74] [BLA, 73] se propose de décrire et de construire des compilateurs en utilisant la méthode des attributs ; des algorithmes de non-circularité sont présentés dans [LOR, 75]. Dans [AMI, 76] on propose une méthode de preuve utilisant des attributs logiques.

La sémantique algébrique développée par NIVAT et son équipe [COU, 78] s'attache essentiellement à rendre compte des schémas récursifs en introduisant un cadre algébrique bien adapté : le Magma libre.

Enfin signalons au lecteur intéressé par les nouvelles voies de recherche les travaux d'ASHCROFT, WADGE [ASH, 76] qui présentent la définition formelle de LUCID qui est un langage déclaratif bien adapté aux preuves de correction.

> L'évidence est toujours l'ennemie de la correction. C'est pourquoi on invente un symbolisme nouveau et difficile dans lequel rien ne semble évident. Nous définissons ensuite des règles d'emploi de ces symboles et tout devient mécanique.
>
> B. RUSSEL, *Les Mathématiques et les métaphysiciens.*

11 SOLUTIONS D'EXERCICES

Exercice 1

Pour décrire l'exécution, indiquons les identificateurs locaux en fonction de la profondeur de l'appel :

— L'appel $f(n)$ provoque celui de $f(m_1)$ avec $l_1 = n$ et m : goto n

— L'appel $f(m_1)$ conduit à exécuter l'instruction étiquetée m_2 : goto l_2 avec $l_2 = m_1$.

Ainsi l'appel de $f(n)$ provoque successivement le branchement en m_1 puis en n ce qui termine l'exécution.

Exercice 2

La réponse est *faux*, puisque les valeurs logiques PL/1 sont confondues avec les entiers (*vrai* avec 1 et *faux* avec 0) et l'évaluation de cette expression s'effectue de gauche à droite. Remarquons qu'ici la définition du langage n'est pas ambigüe, mais étant trop éloignée du sens intuitif elle est la cause d'erreurs de programmation.

Exercice 3

a) Voir la figure 31

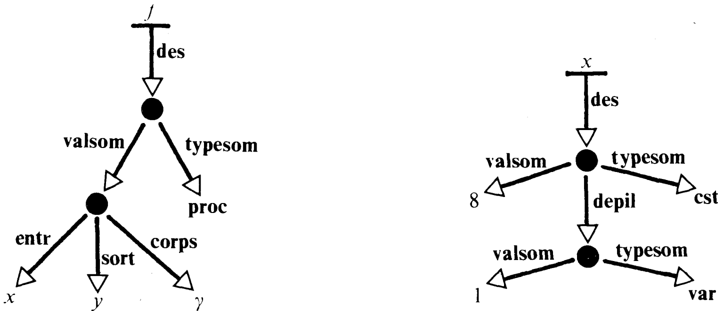


Figure 31.

b) Pour x :

valsom des x = 8 **valsom depil des** x = 1
typesom des x = **cst** **typesom depil des** x = **var**

Exercice 4

VA = **VE** ∪ **VB**
plus : **VE** × **VE** → **VE** **infe** : **VE** × **VE** → **VB**
moins : **VE** × **VE** → **VE** **infs** : **VE** × **VE** → **VB**
mult : **VE** × **VE** → **VE** **egal** : **VE** × **VE** → **VB**

Ainsi

Eval($e_1 + e_2$) = **plus(Eval**(e_1), **Eval**(e_2))
Eval(x) = **valsom(des** x)
etc...

Exercice 5

L'appel de procédure devient (il n'y a plus de $\hat{r}$) (figure 32)

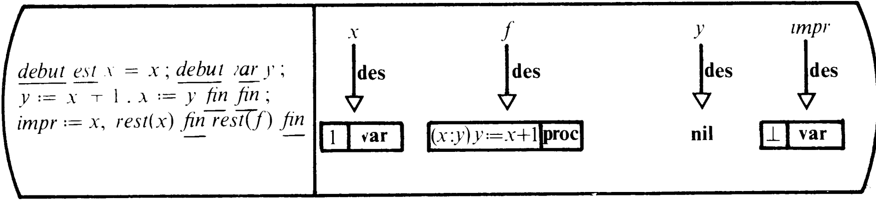


Figure 32.

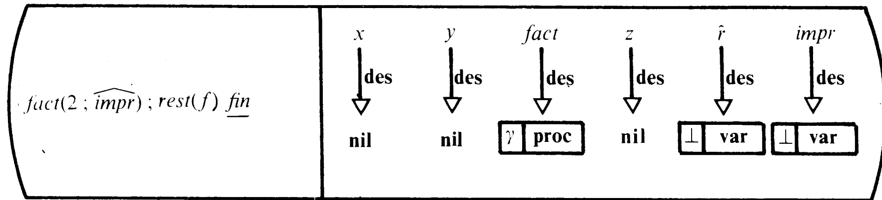
Ce qui est incorrect puisqu'on ne peut affecter de valeur à la constante x. Si maintenant on suppose que le paramètre d'entrée x est une variable, cette sémantique n'a pas d'intérêt puisqu'à la sortie du bloc représentant l'appel la valeur de x est perdue. L'introduction de $\hat{r}$ permet d'éviter de telles « collisions » entre paramètres effectifs et paramètres formels. Une autre manière d'éviter ce type de problème est d'interdire que le paramètre effectif résultat soit égal à l'un des paramètres formels.

Exercice 6

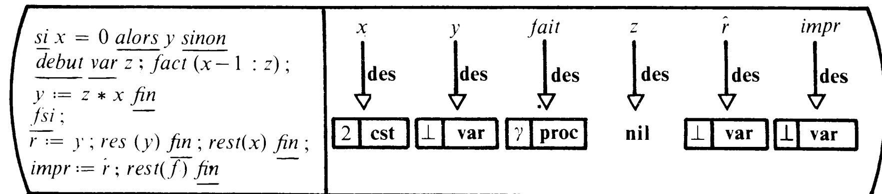
Programme :

<u>début</u> <u>proc</u> *fact*(x : y) <u>si</u> $x = 0$ <u>alors</u> $y := 1$ <u>sinon</u> <u>début</u> <u>var</u> z ; *fact*($x-1$, z) ;
$y := z * x$
<u>*fin*</u>
<u>*fsi*</u> ;
fact(2 : $\widehat{impr}$)
<u>*fin*</u>

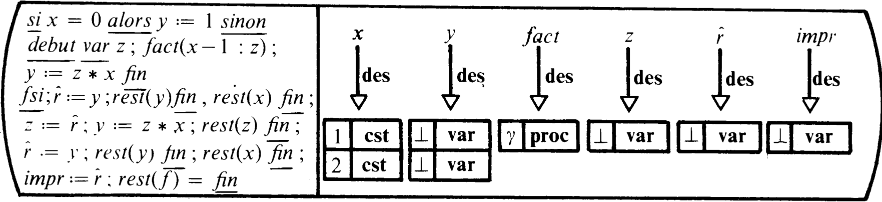
En notant γ le corps de la procédure, l'exécution est décrite par la figure 33



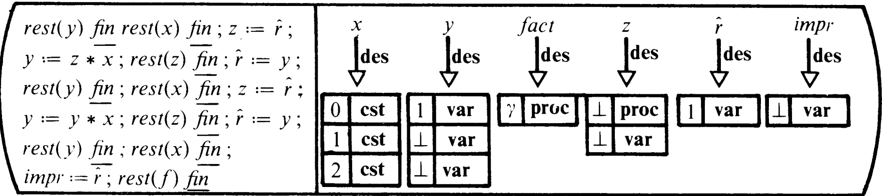
après un premier appel :



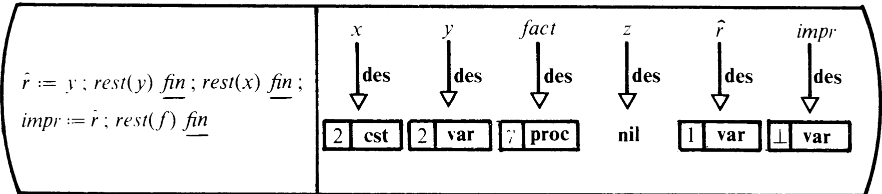
après déclaration de z et deuxième appel de fact



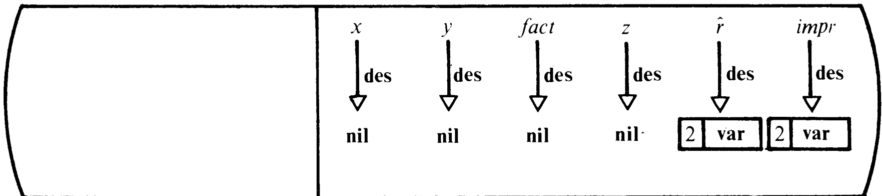
après une nouvelle déclaration de z, un troisième appel de *fact* et l'affectation de 1 à y suivie de celle de y à $\hat{r}$:



finalement, après exécution des différentes affectations et restaurations



et donc :



Exercice 7

a) Il est nécessaire de conserver le texte complet du programme dans l'état mémoire, de manière à rendre compte des sauts en arrière. Si le saut s'effectue dans un même bloc :

$$\langle\ \underline{allera}\ l\,;\ Q \mid \mathbf{e}\ \rangle \rightarrow \langle\ P_2 \mid \mathbf{e}\ \rangle$$

si P est de la forme P_1 ; l : P_2 où l et <u>allera</u> l sont dans le même bloc alors l'état mémoire **e** ne change pas. On suppose ici que toutes les étiquettes sont différentes dans un programme ce qui évite de les empiler.

b) Si le saut peut s'effectuer dans un bloc extérieur, une restauration des identificateurs déclarés entre ce bloc extérieur et le bloc d'où l'on part est nécessaire. On peut associer, au niveau du langage pivot, cet ensemble d'identificateurs à chaque étiquette... Alors :

$$\langle\, \underline{allera}\ l ;\, Q \mid \mathbf{e} \,\rangle \to \langle\, P_2 \mid \underline{\mathbf{lib}}(x_1) \circ \underline{\mathbf{lib}}(x_2) \circ \ldots \circ \underline{\mathbf{lib}}(x_n)\,(\mathbf{e}) \,\rangle$$

si P est de la forme P_1 ; l : P_2 et $x_1, \ldots, x_n$ sont les identificateurs associés à l.

Exercice 8

Soient $(\mathrm{A_n})$, $(\mathrm{B_n})$ deux suites croissantes de $\mathscr{P}(\mathscr{M}^\infty)$ convergeant respectivement vers A et B :

— $\lim\limits_{n} \mathrm{A_n} \cup \mathrm{B_n} = \mathrm{A} \cup \mathrm{B}$ car

$$\mathrm{card}(\mathrm{A} \cup \mathrm{B} - \mathrm{A_n} \cup \mathrm{B_n}) \leqslant \mathrm{card}(\mathrm{A}) - \mathrm{card}(\mathrm{A_n}) + \mathrm{card}(\mathrm{B}) - \mathrm{card}(\mathrm{B_n})$$

— $\lim\limits_{n} \mathrm{A_n\, B_n} = \mathrm{AB}$. Il suffit de poser $\mathrm{C_n} = \mathrm{A} - \mathrm{A_n}$ et $\mathrm{D_n} = \mathrm{B} - \mathrm{B_n}$, de la convergence vers $\varnothing$ de $\mathrm{C_n}$ et $\mathrm{D_n}$ on déduit que :

$$\mathrm{AB} = \mathrm{A_n\, B_n} \cup \mathrm{C_n\, B_n} \cup \mathrm{A_n\, D_n} \cup \mathrm{C_n\, D_n} \text{ est limite de } \mathrm{A_n\, B_n}\,.$$

Exercice 9

$$\tilde{\mathbf{P}}_0 = (\underline{\mathbf{ega}}(\mathbf{Eval}(exp),\ \mathbf{vrai}).\tilde{\mathbf{P}}_1)^*.\underline{\mathbf{ega}}(\mathbf{Eval}(exp),\ \mathbf{faux})\,.$$

Exercice 10

On connaît le corps γ de la procédure désignée par f. Il est alors inutile de l'introduire dans l'état de mémoire (concept statique). Alors avec x et y paramètres formels :

$$\mathscr{A}(P) \begin{cases} \tilde{\mathbf{P}} = \underline{\mathbf{diff}}(\mathbf{Eval}(exp),\ \bot).\mathbf{declcst}(x,\ \mathbf{Eval}(exp)).\mathbf{declvar}(y).\tilde{\gamma} \\ \qquad \underline{\mathbf{affect}}(\hat{r},\ y).\underline{\mathbf{lib}}(y).\underline{\mathbf{affect}}(res,\ \hat{r}).\underline{\mathbf{lib}}(x) \end{cases}$$

Exercice 11

a) D : <u>proc</u> $fact(x : y)$ <u>si</u> $x = 0$ <u>alors</u> $y := 1$ <u>sinon</u> <u>début</u> <u>var</u> z ; $fact(x-1 : z)$;
$y := z * x$
<u>fin</u>
<u>fsi</u>

$$\mathscr{A}(P) : \begin{cases} \tilde{\mathbf{P}} = \underline{\mathbf{diff}}(\mathbf{Eval}(a),\ \bot).\underline{\mathbf{declcst}}(x,\ \mathbf{Eval}(a)).\mathbf{declvar}(y). \\ \quad (\underline{\mathbf{ega}}(x,\ 0).\underline{\mathbf{affect}}(y,\ 1) \cup \underline{\mathbf{diff}}(x,\ 0).\underline{\mathbf{declvar}}(z).\widetilde{fact(x-1\ :\ z)}. \\ \quad \underline{\mathbf{affect}}(y,\ z * x).\underline{\mathbf{lib}}(z)).\underline{\mathbf{affect}}(\hat{r},\ y).\mathbf{lib}(y).\mathbf{lib}(x).\underline{\mathbf{affect}}(b,\ \hat{r})\,. \end{cases}$$

b) Déduisons de l'équation précédente le seul calcul provoqué par l'appel $fact(z : \widehat{impr})$.

$$\begin{aligned}\widetilde{fact(z : \widehat{impr})} = {} & \underline{\mathbf{diff}}(\mathbf{Eval}(2), \bot).\underline{\mathbf{declcst}}(x, 2).\underline{\mathbf{declvar}}(y).\underline{\mathbf{diff}}(x, 0).\underline{\mathbf{declvar}}(z).\\ & \underline{\mathbf{diff}}(\mathbf{Eval}(x-1), \bot).\underline{\mathbf{declcst}}(x, \mathbf{Eval}(x-1)).\underline{\mathbf{declvar}}(y).\\ & \underline{\mathbf{diff}}(x, 0).\underline{\mathbf{declvar}}(z).\underline{\mathbf{diff}}(\mathbf{Eval}(x-1), \bot).\underline{\mathbf{declcst}}(x, \mathbf{Eval}(x-1)).\\ & \underline{\mathbf{declvar}}(y).\underline{\mathbf{diff}}(x, 0).\underline{\mathbf{affect}}(y, 1).\underline{\mathbf{affect}}(\hat{r}, y).\underline{\mathbf{lib}}(y).\\ & \underline{\mathbf{lib}}(x).\underline{\mathbf{affect}}(z, \hat{r}).\underline{\mathbf{affect}}(y, z * x).\underline{\mathbf{lib}}(z).\underline{\mathbf{affect}}(\hat{r}, y).\\ & \underline{\mathbf{lib}}(y).\underline{\mathbf{lib}}(x).\underline{\mathbf{affect}}(z, \hat{r}).\underline{\mathbf{affect}}(y, z * x).\underline{\mathbf{lib}}(z).\\ & \underline{\mathbf{affect}}(\hat{r}, y).\underline{\mathbf{lib}}(y).\underline{\mathbf{lib}}(x).\underline{\mathbf{affect}}(\widehat{impr}, \hat{r})\ .\end{aligned}$$

Plus généralement, le seul calcul « significatif » d'un appel $fact(n : \widehat{impr})$ est :

$$\begin{aligned}\widetilde{fact(n : \widehat{impr})} = {} & \underline{\mathbf{diff}}(\mathbf{Eval}(n), \bot).\underline{\mathbf{declcst}}(x, \mathbf{Eval}(n)).\underline{\mathbf{declvar}}(y).\\ & [\underline{\mathbf{diff}}(x, 0).\underline{\mathbf{declvar}}(z).\underline{\mathbf{diff}}(\mathbf{Eval}(x-1), \bot).\\ & \mathbf{declcst}(x, \mathbf{Eval}(x-1)).\underline{\mathbf{declvar}}(y)]^{n}.\underline{\mathbf{ega}}(x, 0).\underline{\mathbf{affect}}(y, 1).\\ & \underline{\mathbf{affect}}(\hat{r}, y).\underline{\mathbf{lib}}(y).\underline{\mathbf{lib}}(x).\underline{\mathbf{affect}}(z, \hat{r}).\\ & [\underline{\mathbf{affect}}(y, z * x).\underline{\mathbf{lib}}(z).\underline{\mathbf{affect}}(\hat{r}, y).\underline{\mathbf{lib}}(y).\underline{\mathbf{lib}}(x).\mathbf{affect}(z, \hat{r})]^{n-1}.\\ & \underline{\mathbf{affect}}(y, z * x).\underline{\mathbf{lib}}(z).\underline{\mathbf{affect}}(\hat{r}, y).\underline{\mathbf{lib}}(y).\underline{\mathbf{lib}}(x).\underline{\mathbf{affect}}(\widehat{impr}, \hat{r})\ .\end{aligned}$$

Exercice 12

$$\mathscr{S}[\![\quad]\!]\,\mathbf{em} = \mathbf{m}\ .$$

Exercice 13

La sémantique proposée autorise les procédures récursives. On associe en effet à f une fonction définie comme plus petit point fixe d'une équation.

Une procédure peut également appeler une procédure déclarée précédemment. Par exemple si P est le programme :

$\underline{\textit{début}}$ $\underline{\textit{proc}}$ f(x : y) B ; P_1 $\underline{\textit{fin}}$

et si P_1 est le programme :

$\underline{\textit{début}}$ $\underline{\textit{proc}}$ g(x_1, y_1) B_2 ; P_2 $\underline{\textit{fin}}$

alors :

$$\mathscr{S}[\![P]\!]\,\mathrm{em} = \mathscr{S}[\![P_1]\!]\,\mathbf{e}_1\,\mathbf{m}$$

et l'environnement $\mathbf{e}_1$ contient la définition de la fonction associée à f.

Bien entendu ceci ne permet pas la récursivité croisée. L'usage des variables globales est également interdit puisque dans la définition de f le corps de procédure B est exécuté sur l'état de mémoire totalement indéfini.

Bibliographie

[AMI, 76] AMIRCHAHY, M. ; NEEL, D. ; PAIR, C. (1976), Preuves de descriptions de traitements de textes par attributs, Rapport LABORIA, num. 163, IRIA, Rocquencourt.

[APT, 76] APT, K. R. ; de BAKKER, J. W. (1976), Exercices in denotational semantics, Mathematical Foundations of Computer Science (A. Mazurkiewicz, ed.), Lecture Notes in Computer Science, vol. 45, Springer, Berlin.

[ARS, 74] ARSAC, J. (1974), Langages sans étiquettes et transformations de programmes, Automata Languages and Programming (J. Loekx, ed.), Lecture Notes in Computer Science, 14, Springer, Berlin.

[ARS, 76] ARSAC, J. (1976), Méthodes et Outils en programmation, Congrès de l'AFCET, pp. 123-146, Gif-sur-Yvette.

[ARS, 77] ARSAC, J. (1977), La construction de programmes structurés, Dunod, Paris.

[ASH, 71] ASHCROFT, E. A. ; MANNA, Z. (1971), The Translation of « GOTO » Programs to « WHILE » Programs, Proc. IFIP Congress, North-Holland Publ., Amsterdam, pp. 250-255.

[ASH, 76] ASHCROFT, E. A. ; WADGE, W. W. (1976), Lucid : a formal system for writing and proving programs, SIAM J. Comput., 5, 3, pp. 336-354.

[BAK, 67] de BAKKER, J. W. (1967), Formal definition of programming languages, Mathematical Center Tracts 16, Mathematisch Centrum, Amsterdam.

[BAK, 69] de BAKKER, J. W. (1969), Semantics of programming languages, Advances information systems science (J. T. Tou, ed.), Plenum Press, 2, pp. 173-227.

[BAK, 74] de BAKKER, J. W. (1974), Fixed points in programming theory, Foundations of Computer Science (J. W. de Bakker, ed.), pp. 1-49, Mathematical Center Tracts 63, Mathematisch Centrum, Amsterdam.

[BAK, 76] de BAKKER, J. W. (1976), Correctness proofs for assignement statements, Report IW 55/76, Mathematisch Centrum, Amsterdam.

[BAK, 75] de BAKKER, J. W. ; MERTENS, L. G. L. T. (1975). On the completeness of inductive assertion methods, J. Comput. System Sci. 11, pp. 323-357.

[BAK, 72] de BAKKER, J. W. ; de ROEVER, W. P. (1972), A calculus for recursive program schemes, Automata, Languages and Programming (M. Nivat, ed.), pp. 167-196, North-Holland, Amsterdam.

[BAS, 75] BASU, S. K. ; YEH, R. T. (1975), Strong verification on software engineering, IEEE Trans. Software Engrg., SE 1, num. 3.

[BEK, 74] BEKIC, H.; BJORNER, O.; HENHAPL, W.; JONES, C. B.; LUCAS, P. (1974), A formal definition of a PL/1 subset, Technical report TR 25.139, IBM Laboratory Vienna.

[BER, 75] BERRY, G. (1975), Bottom-up computation of recursive programs, Rapport LABORIA, num. 133, IRIA, Rocquencourt.

[BIR, 67] BIRKHOFF, G. (1967), Lattice Theory, AMS, Providence.

[BLA, 73a] BLAIZOT, L. (1973), Delta : système de description de langages et traducteur par attributs, Rapport LABORIA, num. 20, IRIA, Rocquencourt.

[BLA, 73b] BLAIZOT, F.; BOULIER, P. (1973), Projet Delta. Description d'un langage algorithmique à structure de blocs, Rapport LABORIA, num. 35, IRIA, Rocquencourt.

[BLI, 73] BLICKLE, A. (1973), An algebraic approach to mathematical theory of programs, C.C.P.A.S. Reports 119, Warsaw.

[BOH, 66] BOHM, C.; JACOPINI, G. (1966), Flow diagrams turing machines and languages with only two formation rules, Comm. ACM. 9, 5, pp. 366-371.

[BOY, 75] BOYER, R. S.; ELSPAS, B.; LEVITT, K. N. (1975), SELECT a formal system for testing and debuging programs by symbolic execution, International Conference on reliable softivare, Sigplan Notes avril 75, Los-Angeles.

[BUR, 69] BURSTALL, R. M. (1969), Proving properties of programs by structural induction, Comput. J. 12, pp. 41-48.

[BRI, 73] BRINCH HANSEN, P. (1973), Concurrent programming Concepts, Computing Surveys 5, 4, pp. 221-245.

[CAD, 73] CADIOU, J. M. (1973), Recursive definitions of partial functions and their computations, Ph. D. Thesis, Comput. Sc. Department, Stanford University.

[CHA, 73] CHANG, C.; LEE, R. C. (1973), Symbolic Logic and mechanical theorem proving, Academic Press, New-York.

[CHO, 69] CHOMSKY, N. (1969), Structures syntaxiques, Seuil, Paris.

[CHU, 41] CHURCH, A. (1941), The Calculi of Lambda-Conversion, Ann. of Math. Studies 6, Princeton University Press, Princeton.

[CLI, 72] CLINT, M.; HOARE, C.A.R. (1972), Program proving : jumps and functions, Acta Informat. 1, 3, pp. 214-224.
(Voir aussi : ASHCROFT, E. A.; CLINT, M.; HOARE, C.A.R. (1976), Remark on Program proving : jumps and functions by M. Clint and C. A. R. Hoare, Acta Informat. 6, pp. 317-318.)

[CLI, 73] CLINT, M. (1973), Program proving : coroutines, Acta Informat. 2, 1, pp. 50-63.

[COL, 75] COLMERAUER, A. (1975), Les grammaires de métamorphose, Université de Marseille Luminy.

[CON, 71] CONWAY, J. H. (1971), Regular Algebras and Finite Machine, Chapman and Hall, London.

[COU, 74] COURCELLE, B. (1974), Sur la traduction des schémas récursifs monadiques en schémas itératifs, Rapport LABORIA, num. 69, IRIA, Rocquencourt.

[COU, 76a] COURCELLE, B.; NIVAT, M. (1976), Algebraic families of interpre-

tations, Proceedings of 17th Symposium on Foundations of Computer Science, Houston, — ou Rapport LABORIA, num. 189, IRIA, Rocquencourt.

[COU, 76b] COURCELLE, B.; VUILLEMIN, J. (1976), Completeness results for the equivalence of recursive schemes, J. Comput. System Sci. 12, pp. 179-197.

[COU, 78] COURCELLE, B.; COUSINEAU, G.; NIVAT, M. (1978), Theory of programs schemes, à paraître.

[CUR, 58] CURRY, H. B.; FEYS, R. (1958), Combinatory logic, North-Holland, Amsterdam.

[DAH, 72] DAHL, O. J.; DIJKSTRA, E. W.; HOARE, C. A. R. (1972), Structured Programming, Academic Press, New York.

[DES, 37] DESCARTES, R. (1637), Discours de la méthode, Classiques Larousse, Paris.

[DIJ, 68] DIJKSTRA, E. W. (1968), Go to statement considered as Harmful, Comm. ACM. 11, 3, pp. 147-148.

[DIJ, 75] DIJKSTRA, E. W. (1975), Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM. 18, 8, pp. 453-457.

[DIJ, 76] DIJKSTRA, E. W. (1976), A discipline of programming, Prentice Hall series in Automatic Computation, New Jersey.

[DON, 76] DONAHUE, J. E. (1976), Complementary definitions of programming language semantics, Lecture Notes in Computer Science, 48, Springer, Berlin.

[ELC, 69] ELCOCK, E. W.; FORSTER, J. M. (1969), ABSYS 1 : an incremental compiler for assertions : an introduction, Machine Intelligence 4 (B. Melzer and D. Michie, ed.), pp. 423-429.

[ELC, 71] ELCOCK, E. W.; FORSTER, J. M. (1971), ABSET : a programming language based on sets : motivations and examples, Machine Intelligence 6 (B. Meltzer and D. Michie, ed.), pp. 467-492.

[ENG, 73] ENGELER, E. (1973), Introduction to the theory of computation, Academic Press, New-York.

[ENG, 74] ENGELFRIET, J. (1974), Simple program schemes and formal languages, Lecture Notes in Computer Sciences, 20, Springer, Berlin.

[FIN, 76] FINANCE, J. P. (1976), Une méthode de formalisation de la sémantique d'un langage de programmation, Rev. Française Automat. Informat. Recherche Opérationnelle, Sér. Rouge, 10, 8, pp. 5-32, et 10, 12, pp. 5-21.

[FLO, 67] FLOYD, R. W. (1967), Assigning meanings to programs, Proc. Symposium in applied Mathematics, Mathematical Aspects of Computer Science (J. T. Schwartz, ed.), 19, pp. 19-32, American Mathematical Society.

[FRA, 75] FRANK, C.; ROCHFELD, A.; TABOURIER, Y. (1975), La programmation structurée en informatique, Edition d'organisation, Paris.

[GAR, 73] GARLAND, S. J.; LUCKHAM, D. C. (1973), Programs schemes, recursion schemes and formal languages, J. Comput. System Sci. 7, pp. 119-166.

[GER, 77] GERBIER, A. (1977), Mes premières constructions de programmes, Lecture Notes in Computer Science, 55, Springer, Berlin.

[GOR, 75] GORDON, M. (1975), Operational reasoning and denotational semantics, Construction, Amélioration et Vérification de Programmes, Colloque IRIA, pp. 83-98, Arc et Senans, IRIA, Rocquencourt.

[GORE, 75] GORELICK, G. A. (1975), A complete axiomatic systems for proving assertion about recursive and non-recursive programs, Department of Computer Science Tech. Rep. n° 75, University of Toronto.

[GRE, 75] GREIBACH, S. (1975), Theory of programs structures : schemes, semantics, verification, Lecture Notes in Computer Science, 36, Springer, Berlin.

[HIT, 72] HITCHCOCK, P. ; PARK, D. (1972), Induction rules and proofs of termination, Proc. IRIA Symposium on Automata, Formal Languages and Programming (M. Nivat, ed.), North-Holland, Amsterdam.

[HEW, 72] HEWITT, C. (1972), Description and theoritical analysis (using schemata) of PLANNER : a language for proving theorems and manipulating models in a robot, MIT, Art. Intell. Lab., Cambridge, Mass.

[HOA, 69] HOARE, C. A. R. (1969), An axiomatic basis of computer programming, Comm. ACM. 12, 10, pp. 576-580, 583.

[HOA, 70] HOARE, C. A. R. (1970), Procedures and parameters : an axiomatic approach, Symposium on Semantics of Algorithmic languages (E. Engeler, ed.), Lectures Notes in Mathematics, vol. 188, pp. 102-116, Springer, Berlin.

[HOA, 71] HOARE, C. A. R. (1971), Proof of a program : FIND, Comm. ACM. 14, 1, pp. 39-45.

[HOA, 73] HOARE, C. A. R. ; WIRTH, N. (1973), An axiomatic definition of the Programming language Pascal, Acta Informat. 2, pp. 335-355.

[HOA, 74] HOARE, C. A. R. ; LAUER, P. E. (1974), Consistent and complementary formal theories of the semantics of programming languages, Acta Informat. 3, pp. 135-153.

[HOP, 69] HOPCROFT, J. E. ; ULLMAN, J. D. (1969), Formal languages and their relation to automata, Addison-Wesley, Reading.

[HOW, 77] HOWDEN, W. E. (1977), Symbolic testing and the DISSECT symbolic evaluation system, IEEE transactions on software engineering, 3, 4.

[IAN, 60] IANOV, Y. I. (1960), The Logical Schemes of Algorithms, *in* Problems of Cybernetics 1, pp. 82-140, Pergamon Press, New-York.

[IGA, 73] IGARASHI, S. ; LONDON, R. L. ; LUCKHAM, D. C. (1973), Automatic program verification I : logical basis and its implementation, Computer Science Report 365, Standford University, Stanford.

[KAS, 73] KASAMI, T. ; PETERSON, W. ; TOKURA, N. (1973), On the capabilities of while, repeat, and exit statements, Comm. ACM. 16, 8, pp. 503-512.

[KAT, 75] KATZ, S. ; MANNA, Z. (1975), A closer look at termination, Acta Informat. 5, pp. 333-352.

[KEN, 75] KENNEDY, K. ; SCHWARTZ, J. (1975), An Introduction to the Set Theoritic Language SETL, Computers and Mathematics with Applications, 1, 97.

[KIN, 76] KING, J. C. (1976), Symbolic Execution and Program Testing, Comm. ACM. 19, 7, pp. 385-394.

[KLE, 71] KLEENE, S. C. (1971), Logique mathématique, Coll. U, Armand Colin, Paris.

[KNU, 68a] KNUTH, D. E. (1968), The art of programming, Addison Wesley, New York.

[KNU, 68b] KNUTH, D. E. (1968), Semantics of context free languages, Math. Systems Theory 2, pp. 127-145.

[KNU, 71] KNUTH, D. E. (1971), Semantics of context free languages, Math. System Theory 5, pp. 95-96.

[KNU, 74] KNUTH, D. E. (1974), Structured programming with go to statements, Comput. Surveys 6, 4, pp. 261-301.

[KOT, 76] KOTT, L. (1976), Approche par le magma libre d'un langage de programmation type ALGOL : sémantique et schémas de programme, Thèse de 3e cycle, Université de Paris VII.

[KOS, 74] KOSARAJU RAO, S. (1974), Analysis of structured programs, J. Comput. System Sci. 9, pp. 232-255.

[KRI, 69] KRIVINE, J. L. (1969), Théorie axiomatique des ensembles, Collection Sup., Presses universitaires de France, Paris.

[LAN, 65] LANDIN, P. J. (1965), A correspondence between Algol 60 and Church's lambda notation, Comm. ACM. 8, 2, pp. 89-101 et 8, 3, pp. 158-165.

[LAU, 68] LAUER, P. (1968), Formal definition of Algol 60, Technical report TR 25.88, IBM Laboratory, Vienna.

[LEA, 74] LEAVENWORTH, B.; SAMMET, J. (1974), An overview of non procedural languages, Proc. Symp. Very High Level Lang., SIGPLAN Notices 9, 4, pp. 1-12.

[LED, 75] LEDGARD, H.; MARCOTTY, M. (1975), A genealogy of control structures, Comm. ACM. 18, 11, pp. 629-639.

[LER, 75] LEROY, H. (1975), La fiabilité des programmes, Travaux de l'institut d'informatique de Namur, num. 3, Presses universitaires de Namur, Namur.

[LOR, 74] LORHO, B. (1974), De la définition à la traduction des langages de programmation : méthode des attributs sémantiques, Thèse d'Etat, Université Paul Sabatier, Toulouse.

[LOR, 75] LORHO, B.; PAIR, C. (1975), Algorithms for checking consistency of attribute grammars, Construction, Amélioration et Vérification de Programmes, Colloque IRIA, pp. 29-54, Arc et Senans, IRIA, Rocquencourt.

[LUC, 68] LAUER, P.; LUCAS, P.; STIGLEITNER, H. (1968), Method and notation for the formal definition of programming languages, Technical Report TR 25.087, IBM Laboratory, Vienna.

[LUC, 70] LUCKHAM, D. C.; PARK, D. M. R.; PATERSON, M. S. (1970), On formalized computer programs, J. Comput. System Sci. 4, pp. 220-249.

[LUC, 75] LUCKHAM, D. C.; SUZUKI, N. (1975), Automatic program verification IV : proof of termination within a weak logic of programs, Computer Science Report 522, Stanford University, Stanford.

[McC, 63] McCARTHY, J. (1963), A basis for a mathematical theory of computation, Computer Programming and Formal Systems (P. Braffort and D. Hirsberg, ed.), North-Holland, Amsterdam.

[McC, 70] McLANE, S.; BIRKHOFF, G. (1970), Algèbre, Gauthier-Villars, Paris.

[MAN, 72] MANNA, Z.; VUILLEMIN, J. (1972), Fixpoint approach to the theory of computation, Comm. ACM. 15, 7, pp. 528-536.

[MAN, 73] MANNA, Z.; NESS, S.; VUILLEMIN, J. (1973), Inductive methods for proving properties of programs, Comm. ACM. 16, 8, pp. 491-502.

[MAN, 74a] MANNA, Z. (1974), Mathematical theory of computation, McGraw-Hill, New-York.

[MAN, 74b] MANNA, Z.; PNUELI, A. (1974), Axiomatic approach to total correctness, Acta Informat. 3, pp. 243-263.

[MAN, 71] MANNA, Z.; WALDINGER, J. R. (1971), Towards automatic program synthesis, Symposium on semantics of algorithmic language (E. ENGELER, ed), pp. 270-310, Lecture notes in mathematics, 188, Springer Verlag, Berlin.

[MAR, 54] MARKOV, A. A. (1954), Theory of algorithms, USSR Academy of Science, Moscou (traduit en anglais par Israeli, Program for Scientific Translations, vol. 42, Jerusalem, 1962).

[MAR, 75] MARQUE-PUCHEU, M. (1975), Application de la méthode des attributs à la définition des compilateurs. Construction, Amélioration et Vérification de Programmes, Colloque IRIA, pp. 262-283, Arc et Senans, IRIA, Rocquencourt.

[MIL, 74] MILNE, R. E. (1974), The formal semantics of computer languages and their implementations, Technical Monograph PRG-13, Programming Research Group, University of Oxford, Oxford.

[MIL, 76] MILNE, R. E.; STRACHEY, C. (1976), A theory of programming language semantics, part a et b, Chapman et Hall, London.

[MILR, 76] MILNER, R. (1976), Program semantics and mechanised proof, Foundations of Computer Science II (K. R. Apt et J. W. de Bakker, ed.), Mathematical Centre Tracts 82, pp. 3-44, Mathematisch Centrum, Amsterdam.

[NAU, 63] NAUR, P. (éditeur) (1963), Revised report on the algorithmic language Algol 60, Comm. ACM. 6, 1, pp. 1-17.

[NEU, 71] NEUHOLD, E. J. (1971), The formal description of programming languages, IBM Systems J. 2, pp. 86-113.

[NEW, 75] NEWTON, G. E. (1975), A partially annotated bibliography of top-down and go-to-less programming, Construction, Amélioration et Vérification de Programmes, Colloque IRIA, pp. 435-473, Arc et Senans, IRIA, Rocquencourt.

[NIV, 75] NIVAT, M. (1975), On the interpretation of polyadic recursive program schemes, Symp. Mathematica, vol. 15, Academic Press, New-York.

[PAI, 74] PAIR, C. (1974), Formalization of the notions of data, information and information structure, Data Base Management Systems, (Klimbie-Koffeman, ed.), pp. 149-167, North-Holland.

[PAI, 76] PAIR, C. (1976), Du problème à sa solution, Journées logique et programmation IRIA, Le Bischenberg, publication du centre de recherche en informatique de Nancy.

[PAI, 77] PAIR, C. (1977), La construction des programmes, publication du centre de recherche en informatique de Nancy, 77-R-019.

[PAR, 70] PARK, D. (1970), Fixpoint induction and proofs of program semantics, Machine Intelligence, vol. 5, pp. 59-70 (B. Meltzer et D. Michie, ed.), Edinburgh University Press, Edinburgh.

[PAR, 76] PARK, D. (1976), Finiteness is mu-ineffable, Theorical Computer Science 3, pp. 173-181.

[PLO, 75a] PLOTKIN, G. D. (1975), LCF considered as a programming language, Construction, Amélioration et Vérification de Programmes, Colloque IRIA, pp. 243-261, Arc et Senans, IRIA, Rocquencourt.

[PLO, 75b] PLOTKIN, G. D. (1975), Call by Name, Call by Value and the λ-calculus, Theorical Computer Science 1, pp. 125-159.

[REM, 74] REMY, J. L. (1974), Structures d'information, formalisation des notions d'accès et de modification d'une donnée, Thèse de Spécialité, Université de Nancy I.

[REY, 74] REYNOLDS, J. C. (1974), On the relation between direct and continuation semantics, Automata Languages and Programming (J. Loekx, ed.), Lecture Notes in Computer Science, 14, pp. 141-156, Springer, Berlin.

[ROE, 74] de ROEVER, W. P. (1974), Operational, mathematical and axiomatized semantics for recursive procedures and data structures, Report ID 1/74, Mathematisch Centrum, Amsterdam.

[ROE, 75] de ROEVER, W. P. (1975), First order reduction of call by name to call by value, Construction, Amélioration et Vérification de Programmes, Colloque IRIA, pp. 413-434, Arc et Senans, IRIA, Rocquencourt.

[ROU, 75] ROUSSEL, P. (1975), Prolog : manuel d'utilisation, Université de Marseille Luminy.

[SCH, 75] SCHWARTZ, J. (1975), On programming : an interim report on the SETL project, Courant Institute of mathematical Sciences, New York University.

[SCO, 77] SCOTT, D. (1977), Logic and Programming languages, Comm. ACM. 20, 9, pp. 634-641.

[SCO, 71] SCOTT, D. ; STRACHEY, C. (1971), Towards a mathematical semantics for computer languages, Proceedings of the symposium on Computers and Automata (J. Fox, ed.), pp. 19-46, Polytechnic Institute of Brookly Press, New York et Technical Monograph PRG-6 (1972), Programming Research Group, University of Oxford, Oxford.

[SHO, 67] SHOENFIELD, J. R. (1967), Mathematical Logic, Addison-Wesley, Reading.

[SIN, 76] SINTZOFF, M. (1976), La conception et l'écriture du logiciel, Congrès AFCET, Gif-sur-Yvette.

[SIG, 75] Proceedings of the International Conference on Reliable Software 21-23 août 75, Sigplan notice, 10, 6, Los-Angeles.

[STE, 66] STEEL (ed.) (1966), Formal languages description languages, Proc. IFIP, North-Holland, Amsterdam.

[STR, 74] STRACHEY, C. ; WADSWORTH, C. P. (1974), Continuations : a mathematical semantics for handling full jumps, Technical Monograph PRG-11, Programming Research Group, University of Oxford, Oxford.

[TAK, 74] TAKUMI KASAI (1974), Translatability of flowcharts into while programs, J. Comput. System Sci. 9, pp. 177-195.

[TEN, 74] TENNENT, R. D. (1974), Mathematical semantics of Snobol 4, Dept. of Computing and Information Science, Queen's University Kingston, Ontario.

[TEN, 76] TENNENT, R. D. (1976), The denotational semantics of programming languages, Comm. ACM. 19, 8, pp. 437-453.

[TEN, 77] TENNENT, R. D. (1977), Language Design Methods based on Semantic Principles, Acta Informat., 8, pp. 97-112.

[TUR, 36] TURING, A. M. (1936), On computable Numbers with an application to the Entcheidungsproblem, Proc. London Math. Soc. 42, pp. 230-265.

[VUI, 73] VUILLEMIN, J. (1973), Proof Techniques for Recursive Programs, Thèse, Standford, ou note de travail, IRIA, Rocquencourt.

[VUI, 74] VUILLEMIN, J. (1974), Correct and optimal implementations of Recursion in a Simple Programming Language, J. Comput. System Sci. vol. 9, pp. 332-354.

[WAR, 75] WARNIER, J. D. (1975), L'organisation des données d'un système ; les procédures de traitement et leurs données, Editions d'organisation, Paris.

[WEG, 70a] WEGNER, P. (1970), The Vienna definition language, Center for Computer and Information Sciences, TR 70-21-2, Brown University.

[WEG, 70b] WEGNER, P. (1970), Programming language semantics, Courant Computer Science Symposium 2, Formal Semantics of Programming Languages (Rustin, ed.), Prentice Hall Inc. Englewood Cliffs, New-Jersey.

[WIJ, 75] Van WIJNGAARDEN, A. et al. (1975), Revised report on the algorithmic language Algol 68, Acta Informat. 5, 1-3.

Notations

Notation	Pages
$\overline{\overline{\text{def}}}$	11.
t[i : j]	13.
g(T)	13.
d(T)	13.
λ	14.
$\mathscr{F}$(E, F)	15.
$\sqsubseteq$	15.
Ω	15, 103, 116, 264.
Simpl (e)	15, 103.
Dev (e)	15.
fact	16.
μ, μ_i	18, 122.
$\perp$	18, 253.
inf	19.
sup	19.
f_{91}	23.
*	24, 117, 118.
<u>tête</u>	30.
<u>queue</u>	30.
ar	48.
$\mathscr{X}$	48, 50, 94.
$\mathscr{F}$	48, 73, 94.
$\mathscr{P}$	48, 73, 94.
sch ($\mathscr{F}$, $\mathscr{X}$)	48.
I_0	49, 52.
I	49, 52.
:=	50.
STOP	50.
miroir	52.
I[x] (d)	54.
<u>**Tr**</u> (π)	59.
$\overline{\mathscr{P}}$	59.
$\equiv$	62.
$\simeq$	62.
$\equiv_F$	62.
$\mathscr{D}$	74.
res	78.
cal	78.
$\underset{FN}{\leqslant}$	81.
$\underset{SE}{\leqslant}$	81.
$\underset{CA}{\leqslant}$	81.
$\mathscr{G}$	81.
BJ_n	85.
RE_n	89.
REC_n	91.
DRE_n	91.
$DREC_n$	91.
I^{ε}	93.
$\mathscr{F}'$	94.
$\Leftarrow$	95.
$I[\![\tau]\!]$	97.
$t[\underline{\varphi}, \varphi]$	101.
Φ_V	102.
Φ_M	102.
Φ_P	102.
;	114.
p_A	147.
C(x ; α)	150.
$p_A \{ \alpha \} p_B$	150, 177.
$\prec$	167.
f_A	169.
(AFF)	178, 280.
(CMD)	178, 280.
(IMP)	178, 280.
(ITE)	178, 280.
(SEQ)	178, 280.
$\circ$	208.
$\rightarrow$	208.
(APP)	216.
$\mathscr{M}^{\infty}$	263.
$\mathscr{A}$(P)	264.
$\mathscr{S}_{int}$	258.
$\mathscr{S}_{cal}$	267.
$\mathscr{S}_{fonct}$	276.
$\mathscr{C}$	275, 278.
(DEC)	281.
At	286.
$\mathbf{V_a}$	286.
$\boldsymbol{a}$(y)	290.
F_i[r]	293.

Index des sujets

A

Ackerman (fonction d'), 112, 220.
Algèbres relationnelles, 119.
Algorithme d'Euclide, 156.
Appel de procédure, 118, 216, 257, 266, 274, 281.
Appel par nom, *101*, 102.
Appel par substitution pleine, *101*, 102.
Appel par valeur, *101*, 102, 215.
Application continue, 19.
Approximations successives, 15.
Arbre binaire, 13.
Arbre syntaxique, 248, 286.
Arité, 48.
Arrêt des programmes, 68, 71.
Assertion, 146, 148.
Attribut, 285, 286.
Attribut hérité, 291.
Attribut synthétisé, 285.
Automate abstrait, 251.
Axiomatique (sémantique), 280.
Axiome d'affectation, 178, *280*.
Axiome du point fixe, 123.

B

Bien-fondé (ordre —, ensemble —), *31*, 167.
Borne supérieure, inférieure, *19*.
BJ-schémas, 85.

C

Calcul, *54*, 74, 100, 121, 263.
Calculable, *68*, 248.
Calculabilité, 248.
Calcul d'un interprète, 258.
Calcul relationnel récursif, 114.
Chaîne, *19*.
Chemin compatible, 150.
Chemin direct, 151.
Complétude de la méthode de Floyd, 209, 212.
Compteur (méthode des —), 169, 171, 190.
Condition de cheminement, 150.
Condition de comptabilité, 150.
Contenu mémoire, 271, 277.
Continu, 19.
Continuation, 275.
Contrôle (structure de —), 73, 241.
Converge, 15.
Conversion fonctionnelle, sémantique par calcul, stricte, 81.
Corontines, 187.
Corps (de procédure), 216.
Correct (localement), 151.
Correct (partiellement), 148.
Correct (totalement), 148.
Correction locale, 151.
Correction partielle, 148, *151*, 179, 206, *218*.
Correction totale, 148, 189.
Croissante (fonction), 19, 22.
Cycle, 151, 152.

D

Dénotationnelle (sémantique —), 248, *269*.
Description instantanée, 53.
Décidable, 67.
Déclaration, 250, 257, 266, 273, 280.
Déclaration récursive, 94.
Définition algorithmique, 10.
Division euclidienne, 146.
Domaine d'interprétation, 52.
Données, 7.

Donnée structurée, 8, 10.
Drapeau hollandais, 201.
𝒟-schéma, 74.
Dynamique, 200.

E

Elargissement d'un prédicat, 154.
Enoncé, 2, 7, 9, 200.
Enoncé récursif, 11.
Entrée (point d'—), 91, 152.
Entrée (paramètre d'—), 215.
Entier, 270.
Environnement, 271.
Equation récursive, 12, 18.
Equivalence de langages, 27.
Etat d'un automate, 251.
Etat mémoire, 53, 251.
Etat d'un programme, 53.
Etiquetage, 150.
Evaluation symbolique, 57, 222.
Exit, 86.

F

Factorielle, 12, 216.
Fonction d'accès, 8, 252.
Fonction continue, 19.
Fonction d'évaluation, 190.
Fonction sémantique, 242, 247.
Fonction 91, 23, 24, 31, 219.
Fonctionnelle, *15*, 97.
Formule, 10.

H

Hanoï (tours de —), 14.

I

Implantation d'un langage, 243, 282, 287.
Inconnue, 7.
Indécidable, 67.
Indéterminisme, 56, 113.
Inductif, 19.
Infini (calcul —) (voir terminaison), 258.
Initialisation, *52*.
Insertion dichotomique, 201.
Interprétatif, 247, 251.
Interprétation, 49, 52, 95.
Interprétation libre, de Herbrand, 56.
Interprète, 256.
Invariant, 147, *153*, 206, 207.
Isologie, isologue, 47, 62.
Itération, itératif, 72, 257, 265.

L

Lambda-notation, 14.
Langage d'énoncé, 9, 10.
Langage pivot, 248.
Langage noyau, 249.
Langage de programmation, 243, 245, 250.
Langage NAIN, 250.
LCF, 23, 33.

M

Machine abstraite, 251.
Majorant, minorant, 18.
Maximum, minimum, 18.
Mémoire (état —), 251.
Mémoire (contenu —), 271.
Méthode ascendante, descendante, 154, 182.
Méthode de Floyd, *148*, 209.
Modification d'un état de mémoire, 254, 263, 273.
Moins défini que, 15.
Monadique, 69, 103.
μ-calcul, 122.

N

Nœud, 13.
Nom (appel par —), 101, *102*, 245.
NAIN (langage —), 250.

O

Ordre bien fondé, *31*, 167.
Organigramme, 51, 75, 146.
Opérateur, 15.

P

Paramètre effectif, 258.
Paramètre d'entrée, 215.
Paramètre d'une procédure, 215, 244.
Paramètre résultat, 216, 244.
Parcours d'un arbre linaire, 13.
Parcours d'une forêt, 28.
Pas de calcul, 121.
pgcd, 7, 10, 11, 156, 215.
Pile associée à un identificateur, 252.
Place, 271.
Plus petite solution, 2, 15.
Point fixe, *15*, *18*, 94, 103, 123, 264, 273.
Point de coupure, 150.
Point d'entrée, de sortie, 75, 152.
Prédicat, 8, 50, 114, 146, 280.
Preuve de programme, 146.
Problème, 1, *8*.
Problème des n reines, 199, 221.
Procédure, 93, 119, 215, 250.
Programme avec branchement, 50.
Programme récursif, 95, 215.
Programme itératif, 176.
Propriété admissible, 26.
Propriété de Park, 123.

R

Racine, 13.
Raffinements successifs, 192, 199.
Recherche associative, 200.
Recherche dichotomique, 174.
Règle de calcul, 100.
Règle de Park, 23, *24*.
Règle de preuve des appels, 215.
Règle du report, 112.
Règle de Scott, 23, *25*, *26*, 123, 219.
Règle de troncation, 25.
Règle d'induction structurelle, 23, *31*
Règle d'inférence, 177, 216, 280.
Règle fortement sûre, 109.
Règle sûre, 103.
Relation, 8, 113.
Relation universelle, 116.
RE-schéma, 89.
Résoudre un problème, 10.
Résultat du calcul d'un programme, 54.
Réussi (calcul —), 54, 122, 258.
Retour de procédure, 215.
Retour (prédicat de —), 216.

S

Schéma avec branchement, 50.
Schéma de programme, 3, *50*.
Schéma de Horner, 290.
Schéma d'Ianov, 58.
Schéma fonctionnel, 48, 56.
Schéma interprété, 52.
Schéma itératif, 72.
Schéma récursif, 95.
Schéma répétitif, 88.
BJ-schéma, 85.
$\mathscr{D}$-schéma, 74.
RE-schéma, 89.
Sémantique, 4, *242*.
Sémantique axiomatique, *280*, 284.
Sémantique calculatoire, 267.
Sémantique dénotationnelle, 122, 248, *269*.
Sémantique des entiers, 270.
Sémantique des expressions, 270.
Sémantique fonctionnelle, *269*, 284.
Sémantique interprétative, 251.
Sémantique opérationnelle, 122, *251*.
Sémantique par attributs, 285.
Semi-décidable, 67.
Séquence d'exécution, 54.
Simplification d'un terme, 100.
Sous-tableau, 13, 217.
Stationnaire (chaîne —), 19, 20.
Statique, 200.

Stricte (application —), 108.
Structure de contrôle, 73.
Structure de donnée, 8.
Structure d'information, 247.
Substitution (règle de la — pleine), 101, 102.
Symboles fonctionnels, 10, 48.
Symboles relationnels, 10.
Syntaxe abstraite, 249.
Syntaxe concrète, 250.
Syntaxe d'un langage, 250.
Synthèse de programme, 224.

T

Terme, 94.
Terme relationnel, 122.
Terminaison, 167, 220.
Termine pour p, 148.
Théorème du point fixe, 18, 21.
Trace d'un schéma de Ianov, 58.
Trace d'un calcul relationnel, 121.
Transformation d'énoncé, 194, 200.
Transition d'un automate, 251.
Transition d'un interprète, 256.
Treillis distributifs (axiomes des —), 118.
Tri par interclassement, 13, 217.
Tri topologique, 192.
Type abstrait, 1.

V

Valeur (appel par —), 101, 102, 215.
Valeur calculée le long d'un chemin, 150, 160.
Valeur sémantique, 242.
Vecteur d'état, 246.
Vérification, 145.

Imprimé en France. — JOUVE, 18, rue Saint-Denis, 75001 PARIS
Dépôt légal de la 1re édition : 4e trimestre 1978
Dépôt légal : Février 1984
N° d'imprimeur : 12220
Achevé d'imprimer : Février 1984