

Quand on proclama que la Bibliothèque comprenait tous les livres, la première réaction fut un bonheur extravagant...

A l'espoir éperdu succéda, comme il est naturel, une dépression excessive.

La Bibliothèque de Babel 1941
Jorge Luis BORGES.

OUTILS LOGIQUES POUR L'INTELLIGENCE ARTIFICIELLE

CHEZ LE MEME EDITEUR

Du même auteur : DESSINS GEOMETRIQUES ET ARTISTIQUES AVEC
VOTRE MICRO-ORDINATEUR

NOUVEAUX DESSINS GEOMETRIQUES ET ARTISTIQUES
AVEC VOTRE MICRO-ORDINATEUR

M.G. Monteil et PROGRAMMES D'INTELLIGENCE ARTIFICIELLE EN BASIC
R. Schomberg

J.N. Chatain et SYSTEMES EXPERTS : METHODES ET OUTILS
A. Dussauchoy

J. Pitrat TEXTES, ORDINATEURS ET COMPREHENSION

M. James INTRODUCTION A L'INTELLIGENCE ARTIFICIELLE
SUR MICRO-ORDINATEUR

A. Barr et LE MANUEL DE L'INTELLIGENCE ARTIFICIELLE
E.A. Feigenbaum Tome 1

J.L. Laurière INTELLIGENCE ARTIFICIELLE
Résolution de problèmes par l'Homme et la machine

J.P. Balpe INITIATION A LA GENERATION DE TEXTES
EN LANGUE NATURELLE
Exemples de programmes en Basic

Ch. Ernst INTRODUCTION AUX SYSTEMES EXPERTS DE GESTION

Y. Shirai et INTELLIGENCE ARTIFICIELLE
J.I. Tsujii Concepts, techniques et applications

OUTILS LOGIQUES POUR L'INTELLIGENCE ARTIFICIELLE

Jean-Paul DELAHAYE

*Agrégé de Mathématiques
Docteur d'Etat en Mathématiques
Laboratoire d'Informatique Fondamentale de Lille
(Laboratoire Associé au CNRS n° 369)
Equipe d'Intelligence Artificielle du Professeur COMYN*

Préface de
Maurice NIVAT

DEUXIÈME ÉDITION
Revue et corrigée



61, boulevard Saint-Germain – 75005 PARIS
1987

Si vous désirez être tenu au courant de nos publications, il vous suffit d'adresser votre carte de visite au :

Service « Presse », Editions EYROLLES,
61, Boulevard Saint-Germain
75240 PARIS CEDEX 05,

en précisant les domaines qui vous intéressent.
Vous recevrez régulièrement un avis de parution des nouveautés en vente chez votre libraire habituel.

« La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1^{er} 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. ».

AVANT-PROPOS

Les méthodes et techniques de l'Intelligence Artificielle font intervenir de nombreux concepts mathématiques et logiques, et il est parfois difficile à celui qui doit les utiliser de trouver des exposés qui soient suffisamment complets (sans l'être trop) lui permettant de s'initier rapidement et correctement.

Le but de cet ouvrage est justement de présenter aux non spécialistes, certains des concepts mathématiques et logiques importants en informatique et dont le rôle est essentiel en Intelligence Artificielle.

Les thèmes principaux abordés sont : la calculabilité et la récursivité ; les systèmes formels ; le calcul propositionnel ; le calcul des prédictats du premier ordre ; la manipulation des formules logiques ; les méthodes de démonstration automatique (unification, résolution, stratégies) ; le langage Prolog.

Chaque chapitre est accompagné d'exercices et de références bibliographiques.

Issu d'un cours de maîtrise, ce livre, par sa forme et le souci didactique qui l'anime, s'adresse aussi bien aux étudiants qu'aux informaticiens qui se sentent concernés par l'Intelligence Artificielle.

PREFACE

L'Intelligence Artificielle est à la mode. Les médias nous font miroiter de très nombreux miracles qu'elle devrait bientôt opérer. Dans le secret de quelques laboratoires, d'ailleurs, de petits miracles s'opèrent : j'entends par là que, sur des sujets bien délimités, on construit des systèmes qui imitent fort bien le raisonnement humain que ce soit celui du joueur d'échecs, du réparateur d'ordinateur en panne ou du médecin formulant son diagnostic. Imiter le raisonnement humain n'est peut-être pas la bonne formule : ce qui est sûr, c'est que ces systèmes obtiennent des résultats qui sont meilleurs que ceux de beaucoup de praticiens chevronnés. Vous serez inexorablement battu par la machine jouant aux échecs si vous n'êtes pas vous-même un grand maître et ce sont aux diagnostics des meilleurs médecins que l'on compare ceux que formule le système MYCIN, par exemple (avec de très troublantes conclusions quant au comportement des médecins et des machines, et les dosages respectifs de rigueur et d'intuition chez les uns et les autres). Ne voir que ces systèmes dits experts comme résultats des très nombreux efforts de recherche poursuivis dans le monde en Intelligence Artificielle depuis longtemps serait vraiment très restrictif : tous les problèmes de reconnaissance, que ce soit du langage écrit pour extraire d'un texte un contenu sémantique, du langage parlé pour le transcrire, de formes visuelles pour analyser une image et faire prendre des décisions à un robot, à un militaire ou à un médecin, appartiennent bien à l'Intelligence Artificielle. Le sens, ou la signification que perçoivent les yeux et les oreilles ne sont pas immédiatement traductibles en formules et, de toute façon, doivent être entièrement explicités pour que la machine puisse les découvrir. Si l'objet qui présente une surface plane portée par quatre pieds a de fortes chances d'être une table, beaucoup de tables n'ont pas quatre pieds et, d'ailleurs, qu'est-ce qu'un pied ?

Il n'en existe pas moins de très remarquables et efficaces systèmes de reconnaissance, surtout de la parole et de formes graphiques, qui sont effectivement utilisés dans l'industrie. La formation du sens dans un texte écrit est plus mystérieuse encore et la langue n'a pas livré tous ses secrets. Non plus que les longues chaînes d'ADN dans lesquelles s'inscrivent nos codes génétiques : découvrir la façon dont nos caractères physiques héréditaires sont traduits et représentés, comme on décrypte les messages codés de l'ennemi, n'est plus de l'Intelligence Artificielle, c'est de l'intelligence tout court. Il s'agit bien de comprendre quelque chose qu'aujourd'hui nous ne comprenons pas, peut-être le plus fascinant mystère de l'humanité : pour ce faire, on utilise des techniques mises au point pour l'Intelligence Artificielle et ce n'est pas le moindre mérite de cette dernière.

De fait, si l'Intelligence Artificielle a vingt-cinq ans, on peut situer sa naissance dans les laboratoires du MIT et de Stanford, au début des années soixante, les problèmes de la connaissance et de "l'art de penser" qui consiste à utiliser les connaissances que l'on a pour tirer des conclusions, émettre des jugements et des opinions, ou simplement choisir un comportement sont eux très anciennement posés : c'est bien pour résoudre ces problèmes qu'Aristote a inventé le syllogisme et fondé ce que l'on appelle la logique. Affinée par des siècles de philosophie, de scolastique et de rhétorique, la logique est restée une discipline très littéraire jusqu'à la fin du XIX^e siècle, où C.S. PIERCE et G. FREGE, indépendamment, ont commencé à calculer dans les ensembles de formules abstraites représentant des propositions auxquelles ont pouvait attribuer une valeur de vérité (vrai ou faux).

On ne retracera pas ici l'histoire du formidable développement de la logique mathématique depuis lors : en moins de cent ans elle est devenue un épais corps de doctrine qui a marqué toute la pensée contemporaine. Le théorème d'incomplétude de GÖDEL a été un des grands événements de l'esprit comme la relativité générale d'EINSTEIN ou les relations d'incertitude d'HEISENBERG. Il a été prolongé par d'autres résultats dont nous en mentionnons un ici, parce qu'il est pertinent à notre propos et que nul ne peut y songer sans arrêt quand il essaie de résoudre un problème d'Intelligence Artificielle : il existe des théorèmes effectivement démontrables dans l'arithmétique du premier ordre dont on ne verra jamais la preuve pour l'excellente raison que le nombre de caractères qu'il faudrait pour l'écrire est supérieur à celui des atomes qui composent notre planète. Cet énoncé de M. FISCHER et M. RABIN qui a étonné, amusé et troublé les participants du Congrès de l'IFIP de 1978, dit éloquemment ce que tous les logiciens apprennent vite à leurs dépens, à savoir que les chaînes de déduction qui constituent les preuves formelles dans n'importe quel système logique ont une forte tendance à être longues.

Ainsi l'Intelligence Artificielle est née du jour où l'on a disposé d'ordinateurs assez puissants pour leur faire effectuer ces longues preuves que l'on ne pouvait faire à la main. Cela a peu modifié les mathématiques dont il est vrai que c'est moins la preuve elle-même qui constitue la difficulté de la démonstra-

tion d'un résultat mais bien l'intuition que ce résultat est vrai et démontrable : les énoncés mathématiques démontrés par des ordinateurs restent peu nombreux et appartiennent à des domaines très spéciaux (géométrie élémentaire, structures algébriques pauvres, etc.).

Par contre, très vite ont été développées des techniques de programmation permettant de calculer sur des masses de faits représentés par des formules traduites en arbres ou mieux en listes. Le langage LISP inventé à Stanford en 1958 est toujours bien vivant et constitue l'un des outils privilégiés de tous les chercheurs en Intelligence Artificielle. Evidemment, il a été beaucoup modifié au cours des temps : il serait fallacieux de croire que les gains de rapidité dans l'exécution de programmes LISP soient seulement dus au progrès des ordinateurs et à leur gain en puissance. Il y a eu aussi de remarquables progrès réalisés dans la compréhension des phénomènes de calcul qui se sont traduits par des améliorations considérables du langage LISP lui-même et des techniques pour l'interpréter et le compiler.

Puis, il y a eu PROLOG, inventé à Marseille par A. COLMERAUER, en 1971 ; resté longtemps assez confidentiel mais qui apparaît aujourd'hui comme un autre outil privilégié de l'I.A. Il est en fait fort différent de LISP. Dans sa version simplifiée (ou pure), c'est vraiment le calcul des prédictats du premier ordre dans lequel on écrit non pas un programme impératif fait d'une suite d'instructions données à la machine mais un ensemble d'axiomes et la spécification du résultat : l'obtention de ce résultat passe par la preuve de cette spécification, l'interprète ou le compilateur PROLOG fonctionnant comme un démonstrateur de théorèmes.

Bien entendu, dans le langage PROLOG que l'on utilise, il y a la possibilité d'écrire bien d'autres choses qui sont en fait des indications pour guider et raccourcir ces preuves et augmenter d'autant la rapidité du calcul. Il y a bien d'autres langages tels SIMULA ou ce que l'on appelle aujourd'hui les langages orientés objets, qui sont autant d'outils fort intéressants.

Ce que nous voulons dire ici, c'est que tout cela n'est pas un jeu : bien au contraire les progrès de ces langages, comme aussi des langages plus simples d'interrogation de bases de données, relationnelles ou non, sont le résultat d'un intense effort de recherche mené dans le monde depuis vingt-cinq ans et d'innombrables expériences. Leur efficacité actuelle remarquable (même si elle est encore perfectible, n'est rendue possible que par des progrès considérables faits dans la connaissance des mécanismes de la dénotation des fonctions (la notion de fonction ayant elle-même été beaucoup élargie) et de la démonstration par application de règles et d'heuristiques ou stratégies pour cheminer au plus court dans les arbres de preuves.

On a trop longtemps dit et trop de gens croient encore que la programmation est chose facile. Elle ne l'est ni en théorie ni en pratique : l'ouvrage que nous présentons aujourd'hui M. DELAHAYE devrait suffire à en convaincre les nombreux lecteurs que nous lui souhaitons. Nous savons gré pour nous à

M. DELAHAYE de combler ce qui nous apparaissait comme une lacune, c'est-à-dire un ouvrage simple qui, partant de la logique mathématique, présente les mécanismes de la programmation PROLOG en expliquant non seulement à quoi ils servent mais pourquoi ils ont été introduits et à quelle nécessité ils répondent. Cet ouvrage est bien plus qu'un manuel d'utilisation de PROLOG, encore qu'il en soit un excellent, c'est ce que les américains appellent un "rationale" (peut-être en français doit-on dire une justification) de PROLOG.

Issu d'un cours professé à Lille, cet ouvrage est à vocation pédagogique : M. DELAHAYE a su, pensons-nous, s'adresser au programmeur moyen en ne supposant pas grand chose de connu et en illustrant abondamment tous les concepts introduits : son livre peut être utilisé par tous. Une nouvelle génération de programmeurs doit être formée qui, en plus de savoir comment faire, devront également savoir pourquoi il faut faire ainsi. Et sans doute, le manuel de M. DELAHAYE en est un exemple que l'on aimerait voir suivi pour constituer la bibliothèque du programmeur professionnel du futur.

Maurice NIVAT

*Professeur à l'Université de PARIS-VII
Président des Conseils Scientifiques du
programme mobilisateur de la filière
électronique et de l'ADI (Agence pour
le Développement de l'Informatique)*

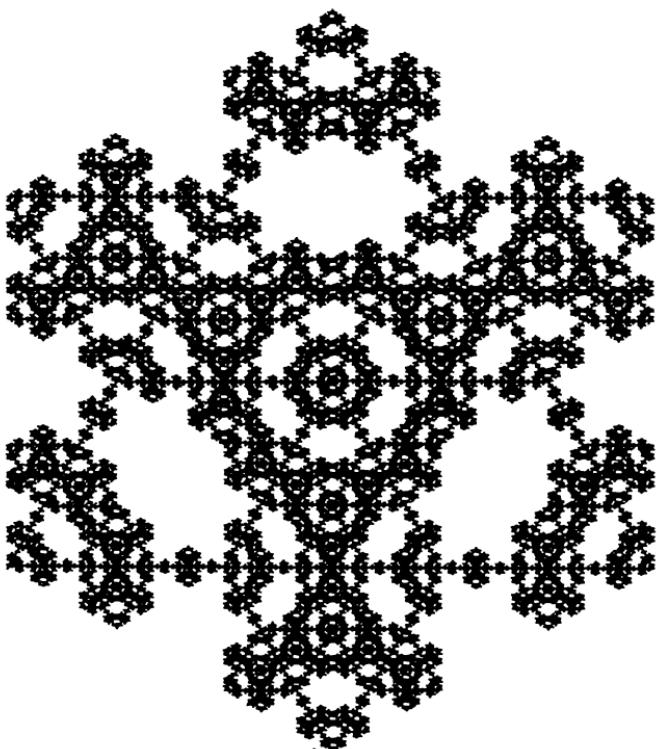
TABLE DES MATIÈRES

Introduction	1
Chapitre 1. — Présentation générale de l'Intelligence Artificielle	5
1. Introduction	6
2. L'Intelligence Artificielle	6
3. Les Systèmes Experts	13
Exercices	18
Bibliographie	21
Chapitre 2. — Récursivité, Décidabilité	29
1. Introduction	30
2. Ensembles finis, infinis dénombrables et infinis non dénombrables ..	30
3. Programmes à i entrées entières et j sorties entières	34
4. Fonctions récursives ; thèse de Church	38
5. Ensembles récursifs et récursivement énumérables	41
6. Prédicats décidables et semi-décidables	45
Exercices	47
Bibliographie	51
Chapitre 3. — Systèmes formels	53
1. Introduction	54
2. Définition	54
3. Résultats généraux	60
4. Etude d'un exemple	63
Exercices	64
Bibliographie	68

Chapitre 4. — Le calcul propositionnel	71
1. Introduction	72
2. Définition du calcul propositionnel P_0 et exemples de théorèmes	72
3. La notion d'interprétation pour les formules de P_1	76
4. Résultats	81
Exercices	87
Bibliographie	91
Chapitre 5. — Le calcul des prédictats du premier ordre	93
1. Introduction	94
2. Termes, Atomes, Formules, Variables libres et liées	94
3. Le système formel Pr du calcul des prédictats du premier ordre	99
4. Interprétations	101
5. Résultats	107
6. La notion de Théorie Axiomatique	108
7. Quelques résultats de décidabilité et d'indécidabilité	113
Exercices	114
Bibliographie	119
Chapitre 6. — Préparation des Formules ; Modèles de Herbrand	123
1. Introduction	124
2. Mise sous forme prénexe	124
3. Théorème de Skolem	127
4. Théorème de Herbrand	130
5. Un algorithme de démonstration automatique	133
Exercices	138
Bibliographie	142
Chapitre 7. — Unification, Résolutions	143
1. Introduction	144
2. Résolution sans variable	144
3. Unification	150
4. Résolution avec variables	155
5. Stratégies d'utilisation de la résolution	164
Exercices	185
Bibliographie	190

Chapitre 8. — Le Langage Prolog	193
1. Introduction	194
2. Un exemple	194
3. La résolution Prolog	202
4. Contrôle, Arithmétique, Traitement de listes	213
Exercices	240
Bibliographie	243
Index	245

INTRODUCTION



Les dessins qui illustrent la première page de chaque chapitre sont des objets fractals de dimension de Hausdorff proche de 2. Ils sont tous construits en partant d'un segment, d'un triangle équilatéral ou d'un carré dont on a remplacé récursivement chaque segment par un motif simple. Ce motif est facile à retrouver avec un peu d'attention. Les dessins ont été réalisés à partir du programme FRACTALE GÉNÉRALE du livre « Nouveaux Dessins Géométriques et Artistiques avec votre micro-ordinateur » de l'auteur, paru aux Editions Eyrolles en 1985.

Le livre qui suit correspond à un cours d'une quinzaine de semaines, enseigné en 1984-1985 dans la Maîtrise d'Informatique de l'Université de Lille.

Le polycopié distribué aux étudiants a servi de base pour la rédaction du texte principal, les exercices, quant à eux, ont été mis au point à partir des séances de travaux dirigés qui accompagnaient le cours.

Le niveau de l'ouvrage est celui d'un deuxième cycle pour étudiants en informatique. En conséquence le bagage mathématique requis est modeste ; nous avons d'ailleurs essayé de le limiter le plus possible et nous pensons que tout bachelier scientifique est préparé à lire ce livre.

Il y a certes quelques démonstrations mais, le plus souvent, elles peuvent être passées, l'essentiel étant les résultats eux-mêmes et les idées qui leur donnent un sens. On remarquera que parfois nous avons préféré donner des indications sur les principes des raisonnements plutôt que les formalisations complètes et détaillées des preuves. Ceux qui le regrettent pourront consulter les ouvrages spécialisés que nous indiquons dans la bibliographie de chaque chapitre.

Le but de ce livre est de présenter certains concepts mathématiques et logiques importants en informatique et qui jouent un rôle essentiel en Intelligence Artificielle.

Les concepts que nous considérons s'articulent autour :

- **de la théorie de la calculabilité** (chapitre 2) introduite en se basant sur une certaine familiarité supposée des lecteurs avec la programmation, plutôt que, comme c'est généralement l'usage, à partir des machines de Turing ;
- **de la notion de système formel** (chapitre 3) dont la compréhension, comme l'a magnifiquement démontré D. Hofstadter, est un point central pour tout travail sur la pensée et donc pour l'Intelligence Artificielle.
- **de la logique des propositions et des prédicats** (chapitres 4 et 5) qui constitue la seule théorie mathématiquement développée et formalisée du savoir, du raisonnement et de la vérité et qui donc est un outil de base pour la représentation et la manipulation (informatique ou non) des connaissances ;
- **des méthodes de démonstration automatique** (chapitres 6 et 7) qui, là encore, constituent des instruments fondamentaux pour tout travail concernant le raisonnement et qui, de plus, permettent de donner une assise théorique aux systèmes experts et à certains langages de l'Intelligence Artificielle (comme Prolog).

A cette partie centrale théorique, nous avons ajouté deux autres chapitres plus « pratiques ». L'un situé en tête donne un aperçu rapide des buts, domaines et difficultés de l'Intelligence Artificielle ainsi que quelques notions concernant les systèmes experts. L'autre, placé à la fin (chapitre 8) fournit une présentation du langage Prolog qui illustre et complète les concepts théoriques des autres chapitres. Insistons sur l'idée qu'aucune véritable compréhension de Prolog ne peut être obtenue sans une bonne assimilation des notions théoriques qui précèdent. Il est certes possible de manipuler Prolog et d'écrire des pro-

grammes Prolog sans connaître le contenu complet de ce cours, d'ailleurs les manuels d'initiation à Prolog évitent le plus souvent le recours à toute théorie (ce qui a pour effet de rendre ce langage parfois un peu mystérieux et presque magique). Il n'empêche que le fondement de ce langage est là, dans ces mathématiques et que quiconque veut comprendre ce qu'il fait et pourquoi, lorsqu'il programme en Prolog, ne peut se dispenser d'un certain effort abstrait.

A la fin de chaque chapitre, on trouvera des exercices et une bibliographie commentée. Les bibliographies ne prétendent pas être complètes, elles sont là pour guider le choix de ceux qui souhaitent approfondir certains aspects du cours. La plupart des références données sont celles d'articles et de livres généraux, qui eux, contiennent des bibliographies spécialisées et parfois complètes.

La conception de ce cours reflète le besoin ressenti d'aborder les problèmes posés en Intelligence Artificielle de façon rigoureuse et de considérer comme fondamentales dans un programme de maîtrise les propriétés des systèmes formels et des notions telles que la résolution. Le plaidoyer de Maurice Nivat pour l'enseignement de la logique dans le rapport qu'il remit au Ministère de l'Education Nationale et au Ministère de l'Industrie et de la Recherche 1983, m'a conduit à penser qu'il était le mieux placé pour donner une appréciation sur le contenu et l'orientation de cet ouvrage ; je tiens à le remercier très vivement pour ses encouragements et le très grand honneur qu'il m'a fait en acceptant de préfacer ce livre.

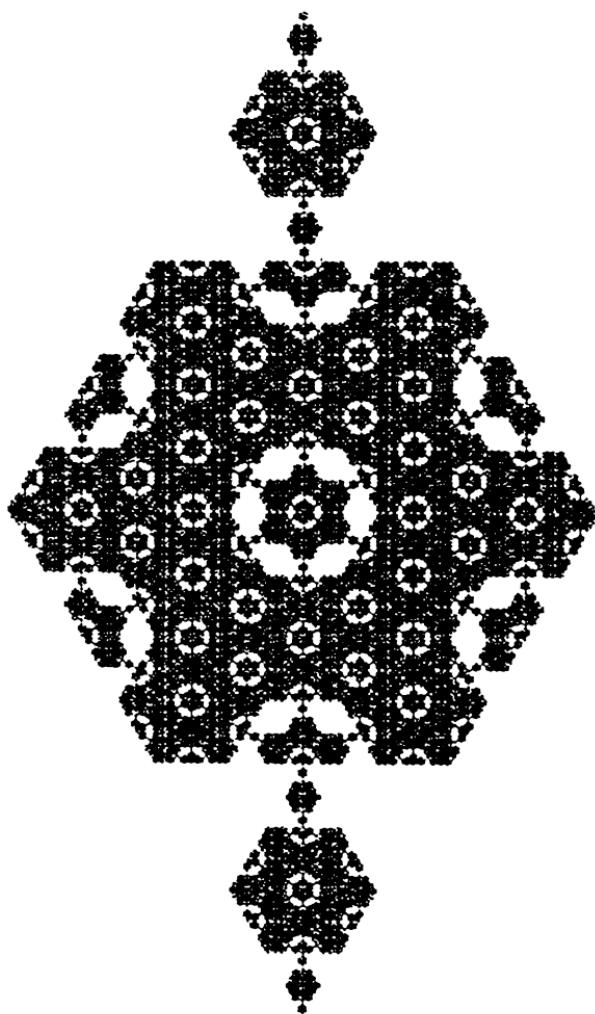
Un cours analogue avait été enseigné durant les années 1982-1983 et 1983-1984 par le Professeur Comyn. Il me faut reconnaître ici la dette que j'ai vis-à-vis de ces premières versions. J'en ai en effet repris l'ossature principale et la philosophie générale. Outre le fait qu'il m'ait offert de le remplacer dans l'enseignement de son cours, G. Comyn m'a bien souvent guidé dans mon travail, m'encourageant et facilitant les recherches documentaires. C'est lui qui, par ses critiques, remarques et conseils a rendu possible l'élaboration de la première version de mon polycopié et son perfectionnement. Qu'il trouve ici l'expression de tous mes remerciements dévoués et sincères.

Je me dois aussi de citer Eliane et Alain Cousquer, le Professeur Dauchet, mes collègues du département informatique de l'IUT de Villeneuve-d'Ascq et de l'UER d'IEEA de l'Université des Sciences et Techniques de Lille, et les étudiants de maîtrise de l'année 1984-1985, car tous, par leurs remarques et leur soutien critique, ont contribué à la mise au point de ce livre. Mme Jeanine Descarpentries a dactylographié la première version du manuscrit et a, par sa diligente et compétente attention, grandement facilité ma tâche, qu'elle trouve exprimée toute ma reconnaissance.

Enfin, les Editions Eyrolles, par la qualité de leur travail, ont permis la composition (rendue difficile par mon goût immodéré des doubles indices) et l'impression rapide et réussie de cet ouvrage, je tiens à les en remercier vivement.

CHAPITRE 1

PRÉSENTATION GÉNÉRALE DE L'INTELLIGENCE ARTIFICIELLE



1. — INTRODUCTION

Ce court chapitre a plus pour but d'éveiller la curiosité du lecteur que de réellement lui fournir une présentation complète de l'Intelligence Artificielle.

Les nombreuses références bibliographiques (le plus souvent possible en français) que nous donnons à la fin du chapitre doivent lui permettre de poursuivre son initiation dans les domaines spécialisés de cette vaste discipline, qu'il choisira selon ses désirs.

Exception faite pour les systèmes experts dont nous expliquons un peu le fonctionnement général, nous avons préféré ne pas entrer dans le détail des techniques et exposer les difficultés, souvent insoupçonnées au départ, qu'on rencontre lorsqu'on cherche à rendre « intelligentes » des machines. Ceci de manière à provoquer la réflexion, car après plus de vingt années de recherche et de progrès, il est clair que, contrairement à ce que certains aimeraient croire, il reste plus de problèmes à résoudre qu'on en a vraiment résolus.

2. — L'INTELLIGENCE ARTIFICIELLE

(a) Le test de Turing

En 1950, alors qu'on mettait au point les premiers calculateurs numériques électroniques, le mathématicien anglais Alan TURING publiait un article important concernant ce que, plus tard, on allait appeler l'INTELLIGENCE ARTIFICIELLE. Il considérait la question « les machines peuvent-elles penser ? » et, plutôt que d'essayer de donner les définitions nécessaires à une réflexion sur cette question elle-même, TURING proposait de remplacer la question par une autre, étroitement liée et exprimée en termes presque totalement dépourvus d'ambiguïté :

« Ce nouveau type de problème peut être présenté sous forme d'un jeu, que nous appellerons « jeu de l'imitation ». Il nécessite trois joueurs, un homme (A), une femme (B), et un troisième joueur qui posera les questions (C), et qui peut appartenir à l'un ou l'autre sexe. Ce dernier restera dans une pièce séparée de celle où se trouve le couple. Le but du jeu, pour ce troisième joueur, est de déterminer, chez les deux autres joueurs, qui est l'homme et qui est la femme. Il les connaît sous le nom de X et Y, et doit dire, au terme du jeu, soit « X est A et Y est B », soit « X est B et Y est A ». Pour ce faire, il a droit à des questions du type :

« X peut-il me dire de quelle longueur sont ses cheveux ? »

Supposons que X est en fait A, c'est à X de répondre. La règle du jeu, pour A, c'est de tenter d'induire C en erreur. Par conséquent, il pourra répondre, par

exemple : « J'ai les cheveux coupés au carré, les plus longues mèches atteignent environ vingt-deux centimètres ».

Pour éviter, naturellement, que les voix n'aident C, les réponses devront être écrites, si possible à la machine. Le mieux serait un téléimprimeur entre les deux pièces ; ou encore, autre solution, les questions et les réponses peuvent être transmises par un intermédiaire. Le but du jeu pour la joueuse B est d'aider le joueur C. La meilleure tactique, en ce qui la concerne, est peut-être encore de ne donner que des réponses justes. Elle peut ajouter des déclarations telles que « C'est moi la femme, ne l'écoutez pas », mais, comme le joueur A peut tout aussi bien émettre pareil discours, cela n'avancera pas à grand chose.

Et nous posons maintenant cette question : « Que se passera-t-il si l'on fait tenir par une machine le rôle de B dans ce jeu ? ». Le joueur C, en pareil cas, échouera-t-il aussi souvent que lorsque le jeu est joué avec un homme et une femme pour partenaires ? Ces questions prennent la place de notre interrogation première, « Les machines sont-elles capables de penser ? »⁽¹⁾.

Cette nouvelle façon de poser le problème de l'intelligence des machines est appelée « Test de TURING ».

La réponse proposée par TURING dans son article est « oui, je crois qu'il sera possible, dans une cinquantaine d'années, de programmer des ordinateurs pour les faire jouer si bien au jeu de l'imitation qu'un interrogateur n'aura pas plus de 70 pour cent de chance de procéder à l'identification exacte après cinq minutes d'interrogation ».

A l'heure d'aujourd'hui, le test de TURING est loin d'avoir été satisfait. A vrai dire, peu de gens pensent qu'en l'an 2000 il sera satisfait comme le croyait TURING.

(b) Les domaines de l'Intelligence Artificielle

L'idée de base de TURING fournit un moyen assez simple de définir l'Intelligence Artificielle et on peut dire que l'Intelligence Artificielle, c'est l'ensemble des disciplines informatiques tendant à faire que des ordinateurs imitent des comportements intelligents humains.

(1) Cette traduction est extraite du livre de H. Dreyfus (voir références bibliographiques [DRE 84]). Une traduction complète de l'article de A. Turing se trouve dans [AND 84].

Cette définition peut, bien sûr, être discutée (voir par exemple [BOB 85]) et plutôt que de chercher à énoncer des principes généraux qui permettraient de faire l'unanimité sur ce qui est de l'Intelligence Artificielle et sur ce qui n'en est pas, il est sans doute préférable de faire la liste de ce qui en constitue les différents domaines :

- la traduction automatique, et l'aide à la traduction [NAG 83] [WIN 84]
- les jeux (échecs, bridge, dames, etc.) [BER 80] [DEW 84] [FIN 80] [LAU 83] [LEV 85]
- la démonstration automatique de théorèmes (voir les chapitres 6 et 7)
- la reconnaissance des formes et la vision artificielle [GUI 74] [SIM 84] [WIN 84]
- la compréhension de la parole [LENI 81]
- la compréhension et la manipulation des langues naturelles [BON 84] [PIT 78] [WIN 83] [PIT 85]
- la modélisation des diverses sortes de raisonnements (déductions incertaines, analogie, etc.) [CHO 83] [KAI 83] [DUB 85]
- la réalisation d'apprentissages automatiques [LEN 85] [WIN 84]
- les systèmes experts [COR 84] [DAV 82] [FIE 84] [GON 83] [GON 84] [PIN 81] [SEY 84] [SHO 76] [FAR 85]

(c) Difficultés de l'Intelligence Artificielle

Un certain nombre de choses relativement faciles pour un être humain sont aujourd'hui encore impossibles à obtenir vraiment d'une machine. Citons par exemple :

- la conduite d'une automobile,
- la lecture d'un texte manuscrit,
- la compréhension du langage parlé courant,
- la traduction d'un texte écrit, d'une langue dans une autre.

Parallèlement, un certain nombre de choses difficiles ou même impossibles pour un être humain sont couramment réalisées par les machines dont nous disposons :

- l'exécution fiable de grands calculs numériques,
- le jeu de haut niveau aux échecs, au bridge, aux dames, au poker, au réversi,
- la manipulation et le traitement de grandes quantités de données symboliques,
- l'intégration exacte ou approchée de fonctions et, plus généralement, la manipulation formelle d'expressions mathématiques.

Ceux qui se posent la question de la possibilité de l'intelligence des machines (ou simplement de la possibilité de faire un jour au test de TURING) se divisent en deux camps aussi convaincus l'un que l'autre d'avoir raison : ceux qui y croient et ceux qui n'y croient pas.

Le camp de ceux qui n'y croient pas est très bien défendu par le philosophe américain Hubert DREYFUS. Celui-ci, s'appuyant sur les prédictions tapageuses du début de l'Intelligence Artificielle, sur l'optimisme excessif et la naïveté de certains chercheurs, sur les échecs momentanés de plusieurs grands projets (comme celui de la traduction automatique), nie la possibilité d'une véritable Intelligence Artificielle.

Le camp de ceux qui croient à la possibilité d'une Intelligence Artificielle est, bien sûr, défendu par beaucoup de chercheurs de ce domaine. Il est remarquable que, dès 1950, TURING, dans son article, répondait par avance à la plupart des objections qui ont depuis lors été avancées contre l'idée de systèmes informatiques pouvant faire un jour au test qu'il proposait.

Les neuf objections qu'il passe en revue sont les suivantes :

- l'objection théologique : penser est une fonction de l'« âme immortelle de l'homme »,
- l'objection de l'autruche : ce serait trop terrible !
- l'objection mathématique : le théorème d'incomplétude de GÖDEL « démontre » que l'homme sera toujours plus que la machine,
- l'objection issue de la conscience : l'intelligence véritable nécessite une conscience de soi, ce qui est impossible à un mécanisme,
- l'objection provenant de diverses incapacités : il ne sera jamais possible de fabriquer une machine qui fasse X (X variant au fur et à mesure des succès de l'Intelligence Artificielle),
- l'objection de Lady LOVELACE : aucune création véritable ne peut provenir d'une machine,
- l'objection que le système nerveux est continu et que les machines électroniques sont discrètes,
- l'objection qu'il n'y a pas de règles de conduite et que la véritable intelligence n'est donc pas formalisable,
- l'objection de la perception extra-sensorielle impossible à une machine alors que (peut-être) elle l'est à l'homme.

Pour chacune des neuf objections, il propose une réponse (que vous pouvez chercher à imaginer) et qui, bien sûr, satisfait les partisans et est discutée par les contradicteurs.

Même ceux qui sont convaincus de la possibilité de faire un jour au test de TURING doivent prendre la mesure des difficultés et du long chemin qu'il y a à parcourir, avant d'obtenir des machines simulant l'intelligence.

Les problèmes les plus difficiles (et on ne peut pas y échapper) sont ceux de la « connaissance commune » et de l'« intelligence générale », nécessaire, par exemple, dans la traduction automatique (depuis longtemps, on sait que des considérations purement syntaxiques sont insuffisantes pour réaliser de bonnes traductions et qu'une certaine compréhension est indispensable).

Le texte suivant tiré d'un rapport du MIT de GOLDSTEIN et PAPERT (voir [DRE 84], p. 392-3) donne une idée des difficultés qu'on peut rencontrer dans la compréhension de récits simples :

« Soit le fragment de récit suivant :

Aujourd'hui, c'était l'anniversaire de Jack. Penny et Janet sont allés dans un magasin pour acheter des cadeaux. Janet s'est décidée pour un cerf-volant. « Surtout pas » a dit Penny. « Jack a déjà un cerf-volant ». Il t'obligerait à *le* rapporter.

L'objectif est de bâtir une théorie susceptible d'expliquer comment le lecteur comprend que le pronom « *le* » se rapporte au nouveau cerf-volant, et non pas à celui que Jack possède déjà. Les critères purement syntaxiques - désigner par antécédent, par exemple, le dernier substantif nommé - ne s'appliquent manifestement pas ici, et le résultat en serait un contresens dans l'interprétation de la dernière phrase : on en déduirait en effet que Jack risque d'obliger Janet à rapporter *le cerf-volant qu'il possède déjà...* Il est clair que l'on ne peut pas décider que « *le* » désigne le cerf-volant neuf si l'on ne dispose pas d'indications sur les pratiques commerciales de notre société. On pourrait après tout imaginer un monde tout différent, dans lequel on ne rapporterait pas chez le marchand les objets neufs, mais bien les objets usagés...

Certains lecteurs s'étonneront peut-être de remarquer qu'il n'est fait mention nulle part dans le texte des points suivants :

- a) les cadeaux achetés par Penny et Janet sont destinés à Jack ;
- b) la possession d'un objet implique nécessairement que l'on ne veuille pas d'un autre exemplaire de ce type d'objet ».

Toujours dans le but de vous mettre en garde contre l'idée que l'intelligence doit être simple à programmer, je vous propose de réfléchir aux petits problèmes suivants appelés « tests de BONGARD » (exemples tirés du livre de D. HOFSTADTER [HOF 80]).

Dans chaque problème, les différentes boîtes sont numérotées :

I A	I B	II A	II B
I C	I D	II C	II D
I E	I F	II E	II F

et la question posée est « de quelle manière les boîtes de la classe I diffèrent-elles des boîtes de la classe II ? ».

Imaginons qu'on essaie de mettre au point un système qui puisse aussi bien que nous réussir à résoudre ces problèmes.

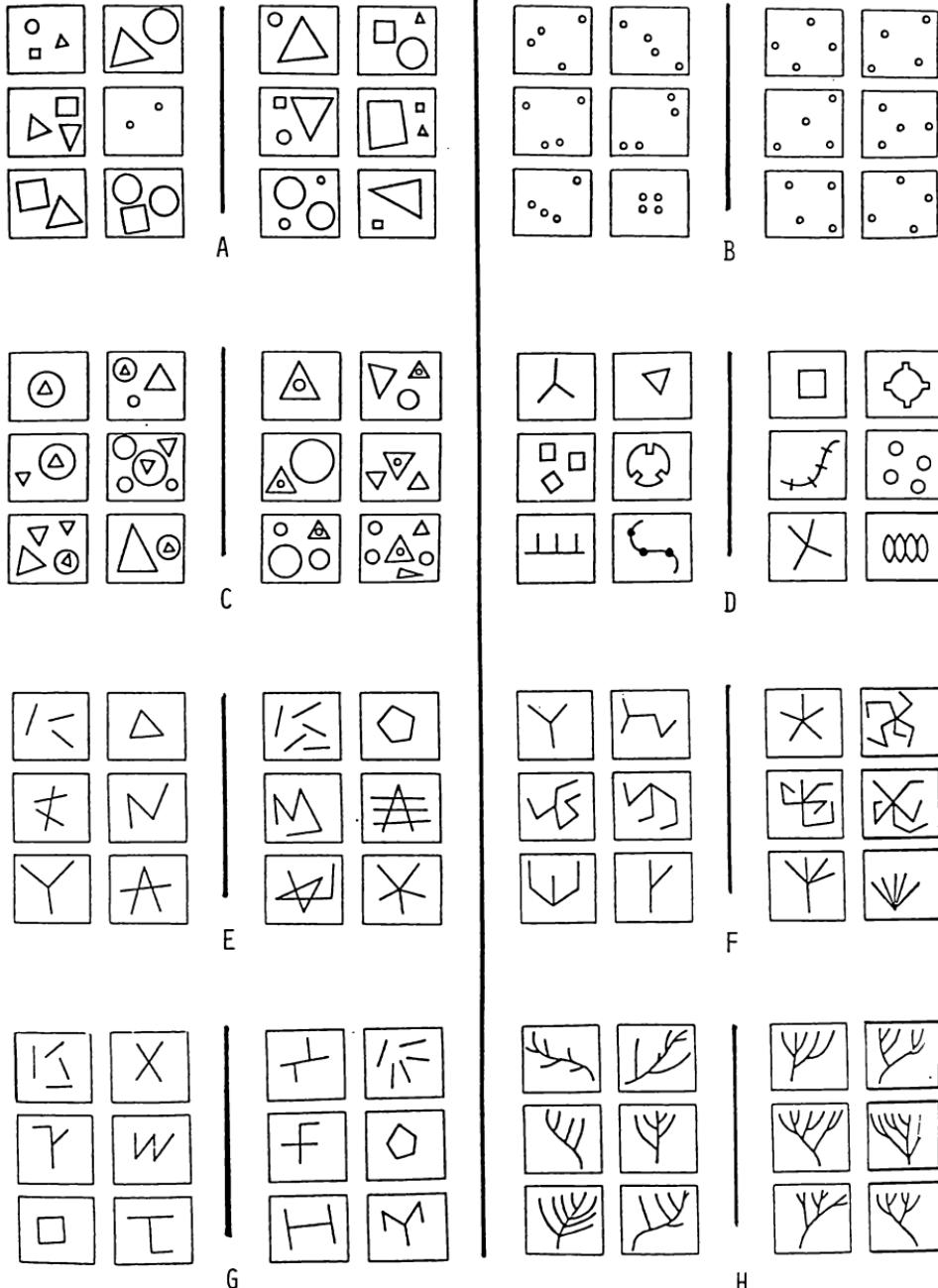


Fig. 1

Ce système devrait être capable :

- de reconnaître des formes simples (triangles, ronds, traits, croix, etc.),
- de pouvoir les classer, quand c'est nécessaire, en petites formes et grosses formes en faisant abstraction du reste (problème A),
- de découvrir que les formes sont groupées ou éparses (problème B),
- de savoir ce que sont l'intérieur et l'extérieur d'une forme géométrique (problème C),
- de compter des indentations, des creux, des traits (problème D),
- de connaître plusieurs définitions différentes de la notion de traits continus (problèmes E, F, G),
- d'avoir la notion de ce qu'est un arbre, etc., etc.

• 3. — *LES SYSTÈMES EXPERTS*

L'impossibilité actuelle qu'il y a à programmer des systèmes informatiques pour qu'ils puissent mimer un comportement intelligent général (comme nous en avons tous un !) a conduit les chercheurs en Intelligence Artificielle à créer des programmes ne visant qu'un domaine limité et facile à mettre sous forme symbolique. Ces programmes qui concentrent en eux les connaissances d'une discipline restreinte de telle façon qu'elles soient utilisables, sont appelés des systèmes experts (ils sont destinés à remplacer des experts humains).

Les premiers et plus célèbres sont :

- DENDRAL, 1977, B. G. BUCHANAN
Chimie moléculaire, STANFORD.
- MYCIN, 1977, E. H. SHORTLIFFE
Diagnostic des infections bactériennes et indications thérapeutiques,
STANFORD.
- PROSPECTOR, 1978, P. E. HART, R. O. DUDA
Aide à la recherche en exploitation minière, STANFORD.

Il existe aujourd'hui des centaines de systèmes experts utilisés et en développement [FAR 85].

Principe de fonctionnement d'un système expert

Un ensemble de connaissances est fixé.

Exemple : (emprunté à [COR 84])

Base de connaissance :

R1	si B et D et E	alors F
R2	si G et D	alors A
R3	si C et F	alors A
R4	si B	alors X
R5	si D	alors E
R6	si X et A	alors H
R7	si C	alors D
R8	si X et C	alors A
R9	si X et B	alors D

Ces connaissances correspondent à des règles de raisonnement qui ont été tirées d'un certain domaine de connaissances : diagnostic médical, analyse de circuit, exploration minière, etc.

Exemple :

si (est mammifère) et (est carnivore) et (a des rayures noires) et (est de couleur fauve) alors (est tigre).

La mise au point de ces règles est une partie très délicate et très longue dans la constitution des systèmes experts. D'une part parce que les experts humains qu'on interroge pour élaborer ces bases de connaissances, en général, n'ont pas formalisé sous forme de règles (comme celles indiquées) tout ce qu'ils savent de leur domaine. D'autre part, parce que certaines règles sont utilisées inconsciemment et qu'il est donc nécessaire de réfléchir en profondeur à tout ce qui constitue le savoir de l'expert.

En médecine, il est souvent nécessaire d'énoncer les règles sous des formes non catégoriques, par exemple :

si le site de culture est le sang
et si l'organisme est gram négatif
et si l'organisme est de forme bâtonnet
et si le patient est un hôte à risque
alors

il est vraisemblable à 0,6 que l'organisme est le pseudomonas aeroginosa (extrait de MYCIN).

En plus de l'ensemble de connaissances, on donne :

Une base de faits

B, C

(ce qui correspond par exemple à des résultats d'analyses pour un patient donné)

Ensuite on pose une question :

But

H

c'est-à-dire, on se demande si H résulte des faits par application des règles données dans la base de connaissance.

Il y a aussi des formes plus générales de questions faisant intervenir des variables (voir le chapitre 8).

C'est là qu'intervient ce qu'on appelle le *moteur d'inférences* du système expert, c'est-à-dire le mécanisme (algorithme) qui va tenter de résoudre le problème posé.

Schématiquement, on peut considérer qu'il y a deux types de fonctionnement pour un moteur d'inférences, le chaînage avant et le chaînage arrière.

Chaînage avant

On part de la base de faits et on déclenche des règles dont les prémisses sont entièrement contenues dans cette base de faits. On obtient ainsi une nouvelle base de faits et on poursuit jusqu'à ce que :

ou bien, on tombe sur H
ou bien, plus aucune règle ne puisse s'appliquer.

Lorsque plusieurs règles peuvent s'appliquer, le moteur d'inférence choisit l'une d'elles, éventuellement en appliquant des règles de choix qu'on nomme *métarègles*. Lorsqu'on a affaire à de vastes ensembles de règles, la croissance des faits déduits est souvent exponentielle et donc, la définition et la mise en place de ces métarègles est fondamentale.

Exemple (suite de la page 12) :

base de faits	
	B C
R4 donne X :	B,C, X
R8 donne A :	A,B,C,X
R6 donne H :	A,B,C,X,H succès

(la règle de choix que nous avons employée pour déterminer quelle règle utiliser quand il y en a plusieurs possibles, est : prendre celle de plus petit numéro).

Chaînage arrière

On regarde les règles qui ont le but fixé dans leurs conséquences. On considère chacune de ces règles, si l'une d'elles a toutes ses prémisses dans la base de faits, on a un succès, sinon on considère les prémisses comme de nouveaux buts et on recommence.

On est ainsi amené à explorer un arbre qui, dans les cas simples, est fini (comme pour notre exemple). Différentes procédures permettent d'explorer un arbre⁽¹⁾. Certaines règles (ou la manière d'écrire certaines règles en PROLOG) déterminent le type d'exploration qui est entrepris par le moteur d'inférences.

(1) Voir chapitre 7.

Exemple :

But H

Seule la règle R6 possède H comme conséquence, donc :

Nouveaux buts X,A

La règle R4 possède X comme conséquence, donc :

Nouveaux buts A,B

B est un fait, donc :

Nouveau but A

3 règles possèdent A comme conséquence, ce qui détermine 3 branches dans l'arbre d'exploration.

Branche 1 application de R2

Nouveaux buts G,D

G n'est jamais conséquence donc *échec*

Branche 2 application de R3

Nouveaux buts C, F

C est un fait, donc :

Nouveau but F

R1 donne

Nouveaux buts B,D,E

B est un fait, donc :

Nouveaux buts D,E

R7 donne

Nouveaux buts C,E

C est un fait donc :

Nouveau but E

R5 donne

Nouveau but D

R7 donne

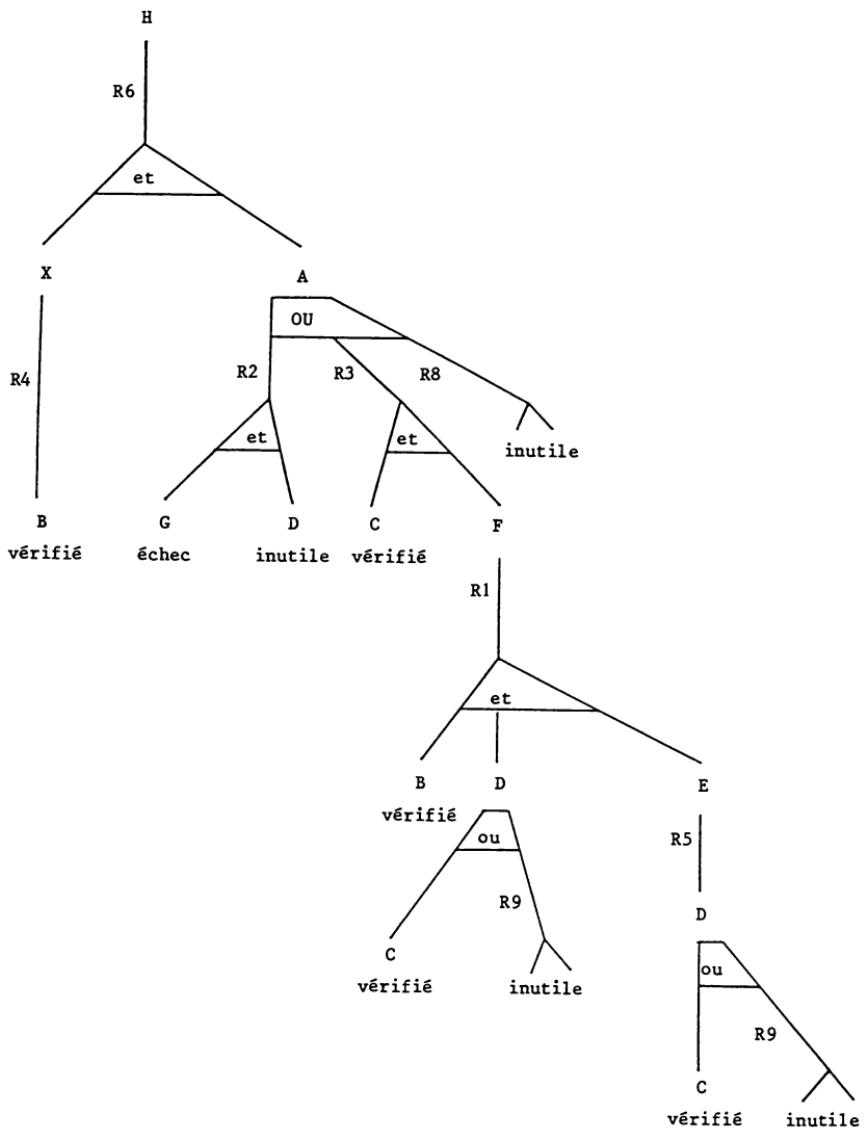
Nouveau but C

C est un fait

donc succès

Il est inutile d'essayer la branche 3.

On peut schématiser les choses comme suit :



Signalons que lorsqu'on utilise un procédé de chaînage arrière, on dit qu'on cherche à *effacer* les buts.

Les méthodes de chaînage avant et chaînage arrière ont chacune leurs intérêts propres, selon la structure des problèmes l'une peut être plus intéressante que l'autre.

Beaucoup de règles avec les mêmes prémisses pénalisent le chaînage avant et inversement beaucoup de règles avec les mêmes conclusions pénalisent le chaînage arrière.

Le principe du chaînage avant repose sur ce qu'on appelle en logique le *modus ponens*

si $A \rightarrow B$ et A , alors B .

Le principe du chaînage arrière repose sur ce qu'on appelle en logique le *modus tollens*

si $A \rightarrow B$ et non B , alors non A

(on considère que la négation du but est ajoutée aux faits et on recherche par applications successives du modus tollens la négation d'un fait, ce qui donnera une contradiction et donc la « démonstration » du but).

Indiquons encore qu'en général un système expert est conçu de manière :

- a) à pouvoir être mis à jour en permanence (modification de la base de connaissances)
- b) à pouvoir indiquer en clair quel raisonnement il a utilisé (explication des résultats fournis).

EXERCICES

Exercice sur les objections de TURING

Proposer une réponse à chacune des neuf objections envisagées par Turing.
Comparer avec celle qu'il a lui-même donnée (voir [AND 84]).

Exercice sur les différents chaînages

A partir de la base de faits : A, K et des règles :

R1	A, B, C	→ D
R2	I, H	→ B
R3	H, F	→ B
R4	A	→ I
R5	E, F	→ D
R6	A	→ F
R7	K, L	→ E
R8	A	→ L

- réaliser du chaînage avant jusqu'à ce qu'aucune règle ne puisse plus donner de faits supplémentaires ;
- construire l'arbre associé au chaînage arrière pour la question : D ? (lorsque plusieurs règles sont en compétition on les prendra par numéros croissants).

Exercice sur les moteurs d'inférences

Dans un langage de programmation de votre choix écrire et tester un programme pouvant réaliser la gestion de règles et de faits (du type envisagé en cours) et capable d'effectuer du chaînage avant ou arrière, à la demande (on trouvera des solutions dans [FAR 85]).

Exercice sur l'utilisation de coefficients de vraisemblance

On considère une base de faits finie :

F_1, F_2, \dots, F_n .

A chaque fait F_i est associé un coefficient de vraisemblance $C(F_i)$ compris entre 0 et 1 et que l'on peut interpréter comme une sorte de probabilité pour que F_i soit vrai a priori.

On considère un ensemble fini de règles entre ces faits R_1, R_2, \dots, R_m .

Chaque règle R_i étant du type :

$F_{i_1}, F_{i_2}, \dots, F_{i_p} \rightarrow F_j$ avec $i_1, i_2, \dots, i_p, j \in \{1, 2, \dots, n\}$.

A chaque règle R_i est associé un coefficient $C(R_i)$ compris entre 0 et 1 et qui mesure la force de la règle ($C(R_i) = 1$ pour une règle absolument sûre).

On considère l'algorithme suivant :

- Initialisation
- **BOOL** : = VRAI
- tant que **BOOL** = VRAI faire
 - **BOOL** = FAUX
 - pour **i** : = 1 à **m** faire
 - évaluer le coefficient de la queue de la règle R_i à l'aide de la formule :
 - $C'(F_j) := C(F_{i_1}) \times C(F_{i_2}) \times \dots \times C(F_{i_p}) \times C(R_i)$
 - si $C'(F_j) > C(F_j)$ alors faire
 - $C(F_j) := C'(F_j)$
 - **BOOL** : = VRAI
 - fin - de - si
 - fin - de - pour
 - fin - de - tant - que
- Impression des résultats
- fin.

(La formule :

$$C'(F_j) := C(F_{i_1}) \times C(F_{i_2}) \times \dots \times C(F_{i_p}) \times C(R_i)$$

est utilisée pour évaluer la vraisemblance de F_j par la règle i . Si cela donne une meilleure évaluation que celle donnée a priori on prend celle-là. Ce calcul est réalisé et utilisé tant qu'un cycle d'utilisation de toutes les règles n'est pas resté sans effet).

Question 1

Montrer que si on ne choisit que des coefficients de vraisemblance égaux à 0 ou 1, l'algorithme décrit est équivalent à un algorithme de chaînage avant.

Question 2

Donner un exemple de situation où un cycle d'utilisation des règles n'est pas suffisant pour obtenir une évaluation définitive des coefficients de vraisemblance (c'est-à-dire où la séquence d'instructions placées dans le « tant que » est exécutée au moins 3 fois).

Question 3

Démontrer que l'algorithme s'arrête toujours.

Donner une majoration du nombre de cycles d'utilisation des règles en fonction des données.

Question 4

Ecrire un programme correspondant à cet algorithme dans un langage de votre choix.

Question 5

Reprendre les questions 1-4 avec la formule :

$$C'(F_j) := \min \{ C(F_{i_1}), C(F_{i_2}), \dots, C(F_{i_p}), C(R_i) \}$$

puis avec la formule :

$$C'(F_j) := \min \{ C(F_{i_1}), C(F_{i_2}), \dots, C(F_{i_p}) \} \times C(R_i)$$

BIBLIOGRAPHIE

- [AND 84] ANDERSON A.-R. et GUIEZE G.
Pensée et Machines.
Collection Milieux, Champ Vallon, Paris, 1984.
(Traduction de : Minds and Machine, Prentice Hall, 1964).
{Recueil d'articles sur la possibilité d'une pensée mécanique. Contient l'article de Turing de 1950}
- [AUB 84] AUBERT et SCHOMBERG
Pratiquer l'Intelligence Artificielle.
Editions Eyrolles, Paris, 1984.
{Court livre sur quelques aspects de l'Intelligence Artificielle}
- [BAR 85] BARR A. et FEIGENBAUM E. A.
Le livre de l'Intelligence Artificielle
Editions Eyrolles, Paris, 1985.
(Traduction de : The Handbook of Artificial Intelligence, Kaufman, Los Altos, California, 1981).
{Traité général et très complet, abordant tous les thèmes de l'Intelligence Artificielle}
- [BER 80] BERLINER H.
L'ordinateur champion du monde de Backgammon.
Bibliothèque pour la Science : les progrès des mathématiques. Belin, Paris, 1980.
{Comment le programme BKG 9.8 a réussi à battre le champion du monde en titre de Backgammon}
- [BOB 85] BOBROW G. et HAYES J. (editors)
Artificial Intelligence - Where Are We ?
Artificial Intelligence, 25, 1985, pp. 375-415.
{Résultats d'une enquête auprès de personnalités réputées de l'Intelligence Artificielle : S. Amarel, J. Feldman, R. Shank, M. Boden, C. Longuet-Higgins, P. Mc Corduck, N. Nilsson, A. Sloman, J.-S. Brown, J. Mc Carthy, D. Michie, A. Newell, T. Winograd, H. Dreyfus, B. Mteltzer, H. Berliner}
- [BOD 77] BODEN M.
Artificial Intelligence and Natural Man
Basic Books, 1977.
{Etudes des rapports entre la psychologie humaine et les méthodes et programmes de l'Intelligence Artificielle}
- [BON 81] BONNET A.
Applications de l'Intelligence Artificielle : les systèmes experts.
R.A.I.R.O. Informatique, 15, n° 4, pp. 325-341, 1981.
{Introduction générale aux systèmes experts}

- [BON 84] BONNET A.
L'Intelligence Artificielle : Promesses et Réalités.
InterEditions, Paris, 1984.
{Introduction concise et bien faite à quelques aspects de l'Intelligence Artificielle ; centrée principalement sur les problèmes de représentation des connaissances, les langues naturelles et les systèmes experts}
- [CHO 83] CHOURAQUI E.
Computational Model of Reasonning.
Colloque Franco-Britannique « Machine et Esprit », Londres, 15-17 avril 1983.
{Présentation de différents modes de raisonnement classiques et non classiques utilisables en Intelligence Artificielle}
- [COR 84] CORDIER M.-O.
Les systèmes experts.
La Recherche, janvier 1984, pp. 60-70.
{Introduction générale, constituant un bon point de départ pour aborder les systèmes experts}
- [COT 84] COT N., ADAM A., COULON D., DESLEES J.-P., KAISER D., LAURENT J.-P.
Réflexion sur l'enseignement de l'Intelligence Artificielle en France.
Rapport à l'attention de M. Malgrange, chargé de mission à l'Education Nationale, juillet 1984.
- [DAV 82] DAVIES R. et LENAT D.
Knowledge-based Systems in Artificial Intelligence.
Mc Graw-Hill Book Company, New York, 1982.
{Sur les systèmes experts}
- [DEK 84] DEKKER A. et ROBIN M.
Super calculateur ou Intelligence Artificielle.
Sciences et Techniques n° 3, avril 1984, pp. 40-51.
{Etude des développements technologiques, liés à l'Intelligence Artificielle, en France, au Japon et aux Etats-Unis}
- [DEW 84] DEWDNEY A.-K.
Les ordinateurs jouent aux dames.
Pour la Science, novembre 1984, pp. 164-174
{Exposé des problèmes et méthodes liés à la programmation du jeu de dames}
- [DRE 84] DREYFUS H. L.
Intelligence Artificielle : Mythes et Limites.
Flammarion, Paris, 1984.
(Traduction de : What computers can't do : the limit of Artificial Intelligence, Harper and Row Publishers, New York 1979).
{Livre polémique à propos de la possibilité d'une véritable Intelligence Artificielle. Très intéressant quoique peu constructif}
- [DUB 85] DUBOIS D. ET PRADE H.
Théorie des possibilités. Applications à la représentation des connaissances en informatique.

- Masson, Paris, 1985.
{Ensembles flous, raisonnements approchés, environnements imprécis, informations incomplètes et incertaines. Avec des exemples et des programmes}
- [FAR 85] FARRENY H.
Les Systèmes Experts : principes et exemples.
Editions CEPADUES, Toulouse 1985.
{Le plus complet des livres français sur les systèmes experts}
- [FEI 84] FEIGENBAUM E. et Mc CORDUCK P.
La cinquième génération. Le pari de l'Intelligence Artificielle à l'aube du 21^e siècle.
InterEditions, Paris, 1984.
{Livre non technique traitant des divers grands projets de développement en Intelligence Artificielle, dans le monde et particulièrement au Japon}
- [FIE 84] FIESCHI M.
L'Intelligence Artificielle en Médecine. Des Systèmes Experts.
Collection Méthode + Programmes, Masson, Paris, 1984.
{Présentation, études, exemples de systèmes experts médicaux}
- [FIN 80] FINDLER N.
L'Ordinateur joue au poker.
Bibliothèque pour la Science : Les Progrès des Mathématiques, Belin, Paris, 1980, pp. 83-89.
{Les problèmes de prises de décision, de psychologie à propos de la programmation d'un jeu de hasard}
- [GON 83] GONDTRAN M.
Introduction aux systèmes experts.
Bulletin de la Direction des Etudes et Recherches de l'EDF, série C2, 1983, pp. 5-42.
{Article introductif}
- [GON 84] GONDTRAN M.
Introduction aux systèmes experts.
Editions Eyrolles, Paris, 1984.
{Ouvrage bref de présentation des systèmes experts. Bien conçu pour une rapide prise de contact avec le sujet}
- [GUI 74] GUIHOT G. et JOUANNAUD J.-P.
Intelligence Artificielle et Reconnaissance de formes.
La Recherche, mars, 1974, pp. 214-220.
{Quelques aspects de l'Intelligence Artificielle}
- [HOF 80] HOFSTADTER D. R.
Gödel, Escher, Bach : an Eternal Golden Braid.
Vintage Books Editions, New York, 1980. InterEditions, Paris, 1985.
{Livre de réflexion en tout point passionnant, sur la logique et l'informatique, dont les derniers chapitres, consacrés à l'Intelligence Artificielle, apportent des points de vue et des idées d'un grand intérêt}

- [KAI 83] KAISER D.
Examen des diverses méthodes utilisées en représentation des connaissances.
Laboratoire de Recherche en Informatique, Université de Paris Sud, 91405 Orsay, 1983.
{Document de base sur le problème de la représentation des connaissances}
- [KRU 84] KRUTCH J.
Expériences d'Intelligence Artificielle en Basic.
Editions Eyrolles, Paris, 1984.
{Quelques idées de l'Intelligence Artificielle mises en programme pour un micro-ordinateur. Forcément limité}
- [LAU 83] LAURENT J.P. et MORLAYER B.
L'Intelligence Artificielle : la démarche, les domaines.
Gazette des Mathématiciens (Société Mathématique de France) n° 22, septembre 1983, pp. 116-152.
{Bonne introduction générale}
- [LAU 82] LAURIERE J.-L.
Représentation et Utilisation des Connaissances.
Techniques et Sciences Informatiques - vol. 1 n° 1 et 2, 1982.
{Ces deux articles constituent une excellente introduction aux systèmes experts}
- [LAU 83] LAURIERE J.-L.
Intelligence Artificielle et Jeux d'Echecs
La Recherche, janvier 1983, pp. 30-39.
{Discussion sur les différentes techniques de programmation du jeu d'échecs}
- [LAU 83] LAURIERE J.-L.
Snark, Symbolic Normalized Acquisition and Representation of Knowledge.
Rapport de l'Institut de Programmation n° 430, novembre 1983.
{Documents autour du langage pour systèmes experts SNARK}
- [LAU 85] LAURIERE J.-L.
Intelligence Artificielle - Résolution de Problèmes par l'homme et la machine.
Editions Eyrolles, Paris, 1985.
- [LEN 85] LENAT D.
Les logiciels et l'Intelligence Artificielle.
Pour la Science, novembre 1984, pp. 132-143.
{Article d'introduction générale}
- [LENI 81] LENINSON S. et LIBERMANN M.
La reconnaissance de la parole par ordinateur.
Pour la Science n° 44, juin 1981, pp. 88-101.
{Les problèmes, des solutions pour les résoudre, l'état de l'art dans le domaine de la reconnaissance de la parole}

- [LEV 85] LEVY D.
Les logiciels de jeux de 1995.
 Science et Vie Micro n° 16, avril 1985.
 {Bilan et perspectives des programmes de jeux pour le Backgammon, le Reversi et les Echecs}
- [MCC 79] MC CORDUCK P.
Machines Who Think.
 W.-A. Freeman Company, 1979.
 {Histoire de l'Intelligence Artificielle}
- [MOT 84] MOTO - OKA T.
Les ordinateurs de la cinquième génération.
 La Recherche n° 154, avril 1984, pp. 516-525.
 {Le projet japonais de développement de la « cinquième génération » présenté par un Japonais}
- [NAG 83] NAGAO M.
La traduction automatique.
 La Recherche, décembre 1983, pp. 1530-1541.
 {Historique, méthodes et perspectives en traduction automatique}
- [NIL 80] NILSSON N.
Principles of Artificial Intelligence.
 Tioga Publishing Company, Palo Alto, 1980.
 {Livre de base en Intelligence Artificielle ; synthétique et général}
- [PAV 84] PAVELLE R., ROTHSTEIN M. et FITCH J.
L'Algèbre Informatique.
 Bibliothèque Pour la Science : L'Intelligence de l'Informatique. Belin, Paris, 1984.
 {Article sur le calcul formel}
- [PIN 81] PINSON S.
Représentation des connaissances dans les systèmes experts.
 R.A.I.R.O. Informatique. Vol. 15, n° 4, 1981, pp. 343-367.
 {Article de base sur le sujet}
- [PIT 78] PITRAT J.
La programmation informatique du langage.
 La Recherche, octobre 1978, pp. 876-881.
 {La représentation et la manipulation des connaissances portées par les langues naturelles}
- [PIT 85] PITRAT J.
Textes, ordinateur et compréhension.
 Editions Eyrolles, Paris, 1985.
- [RIC 83] RICH E.
Artificial Intelligence.
 Mac Graw-Hill Inc., New York, 1983.
 {Comme le livre de Nilsson, il s'agit d'un ouvrage de base en Intelligence Artificielle}

- [SEY 84] SEYDEN E.
Systèmes experts : simuler l'intelligence.
Sciences et Techniques n° 4, mai 1984, pp. 46-63.
{Article d'introduction comportant en particulier une liste assez complète des systèmes experts réalisés ou en cours de réalisation avec quelques informations sur chacun d'eux}
- [SHO 76] SHORTLIFFE E. H.
Mycin : computer based medical consultation. American Elsevier, 1976.
{A propos de Mycin}.
- [SIM 85] SIMONS G. L.
Les ordinateurs de demain : la cinquième génération.
Masson, Paris, 1985. (Traduction de : Toward Fifth-Generation Computers, National Computing Centre Ltd, 1983).
{Le grand projet de développement de l'Intelligence Artificielle des Japonais, les réactions occidentales à ce projet}
- [SIM 84] SIMON J.-C.
La reconnaissance des formes par algorithmes.
Masson, 1984.
- [VAJ 84] VAJOU M.
Alan Turing ou la mécanisation de l'intelligence.
Sciences et Avenir, n° Spécial Informatique, 1984.
{A propos de A. Turing}
- [WAL 82] WALTZ D.
L'Intelligence Artificielle.
Pour la Science, décembre 1982, pp. 34-49.
{Article d'introduction générale}
- [WAT 78] WATERMAN D. A. et HAYES-ROTH F.
Pattern directed inference systems.
Academic Press, New York, 1978.
- [WIN 83] WINograd T.
Language as a Cognitive Process.
Addison Wesley, Publishing Company, 1983.
- [WIN 84] WINograd T.
Les logiciels de traitement des langues naturelles.
Pour la Science, novembre 1984, pp. 91-103.
- [WINS 84] WINSTON P. H.
Artificial Intelligence (2^d Edition).
Addison Wesley Publishing Company, Reading 1984.
{Ouvrage général sur l'Intelligence Artificielle constituant une bonne introduction}
- [ZZ1 85] ZZ1
La Recherche.
Numéro Spécial Intelligence Artificielle, octobre 1985.
- [ZZ2 85] ZZ2
Dossier Spécial Intelligence Artificielle.
Revue Enjeux, septembre 1985.

Périodiques liés à l'Intelligence Artificielle :

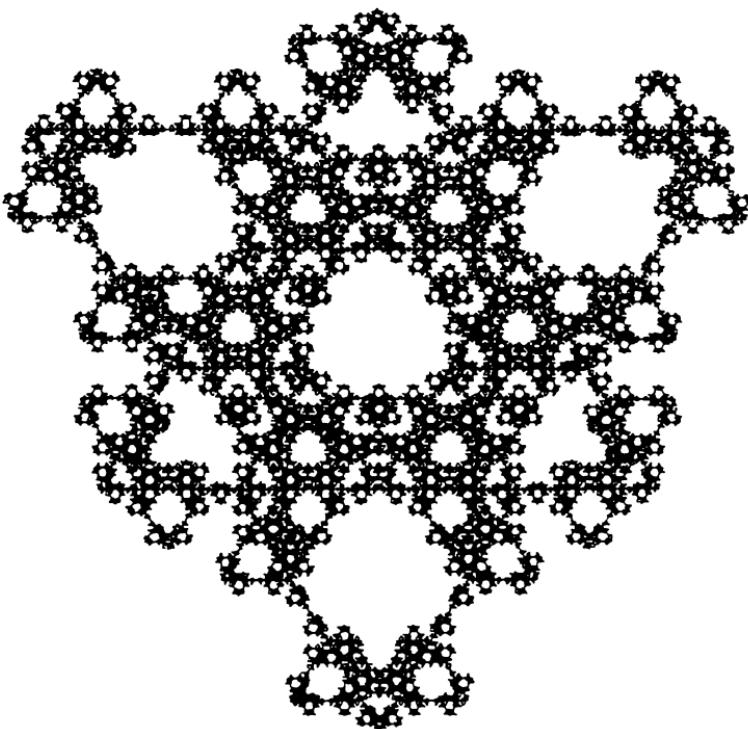
- Artificial Intelligence (North Holland).
- Cognitive Science.
- The Journal of Logic Programming (North Holland).
- Logic Programming Newsletter (Departamento de Informatica, Universidade Nova de Lisboa).
- ECCAI Newsletter (European Coordinating Committee for Artificial Intelligence) (Austrian Society for Artificial Intelligence P.O. Box 177 Vienna, Austria).
- La Lettre de l'Intelligence Artificielle (43, rue de la Victoire, Paris).
- New Generation Computing (Springer Verlag).
- FGCS (Future Generation Computing Systems) (North Holland).
- Expert Systems (Learned Information Ltd Oxford).

Comptes rendus de congrès périodiques ou séries de livres :

- IJCAI (International Joint Conference on Artificial Intelligence).
- AAAI (American Association for Artificial Intelligence).
- Machine Intelligence (University of Edinburgh).
- International Logic Programming Conference.

CHAPITRE 2

RÉCURSIVITÉ, DÉCIDABILITÉ



1. — INTRODUCTION

La notion de décidabilité d'une question et les notions liées (semi-décidabilité, récursivité) sont fondamentales pour l'informatique en général. Ceci car elles précisent et même caractérisent les choses qui sont à sa portée :

- les problèmes qu'un programme est susceptible de résoudre, ce sont les problèmes décidables, c'est-à-dire correspondant à un prédicat décidable, ce dernier terme ayant un sens mathématique rigoureux et indépendant de tout langage de programmation ;
- les fonctions qu'un programme peut calculer, ce sont les fonctions récursives, ce terme lui aussi pouvant être défini mathématiquement et donc sans aucune ambiguïté.

Nous avons choisi de présenter ces notions car en plus de leur intérêt général, elles sont indispensables pour définir, formuler et expliquer certaines idées de la logique mathématique qui nous concernent ici. La théorie des systèmes formels que nous verrons au chapitre suivant s'appuie sur ces notions ; de même certains résultats essentiels à propos du calcul des prédicats et des méthodes de démonstration automatique ne peuvent s'exprimer que dans le langage de la décidabilité.

Un point de vue totalement rigoureux et formalisé (comme celui des machines de Turing) aurait nécessité trop de place et trop d'efforts d'abstraction, c'est pourquoi nous avons choisi une présentation qui, fondée sur une certaine familiarité du lecteur avec la programmation, permet d'arriver rapidement aux idées et résultats centraux.

Les lecteurs désirant approfondir les questions de ce chapitre trouveront dans les références et commentaires bibliographiques les informations nécessaires.

Le paragraphe 2 contient un rappel sur les ensembles finis, infinis dénombrables et infinis non dénombrables. Ce rappel sera sans doute inutile à certains lecteurs.

2. — ENSEMBLES FINIS, INFINIS DÉNOMBRABLES, INFINIS NON DÉNOMBRABLES

On appelle *bijection* (ou application bijective) entre l'ensemble A et l'ensemble B toute application $f : A \rightarrow B$ telle que :

[$\forall x, y \in A (x \neq y \Rightarrow f(x) \neq f(y))$] (injectivité)
et

[$\forall y \in B \exists x \in A (f(x) = y)$] (surjectivité)

(chaque élément de B est l'image d'un élément unique de A)

Par exemple les relations :

$$f(0) = a, f(1) = b, f(2) = c, \dots, f(25) = z$$

définissent une bijection entre $\{0, \dots, 25\}$ et $\{a, b, c, \dots, z\}$

Lorsqu'il existe une bijection entre A et B, on dit que A et B sont équivalents (on dit aussi que A et B ont même cardinal).

Le fait que deux ensembles soient équivalents signifie intuitivement qu'ils contiennent la « même quantité d'éléments ».

Les ensembles équivalents à un ensemble de la forme $\{0, 1, 2, \dots, n-1\}$, $n \in \mathbb{N}$, sont dits finis⁽¹⁾. Un ensemble qui n'est pas fini est dit infini.

Les ensembles équivalents à \mathbb{N} sont dits infinis dénombrables. Les ensembles infinis non équivalents à \mathbb{N} sont dits infinis non dénombrables.

Lorsque A est infini dénombrable, toute application bijective entre \mathbb{N} et A est appelée numérotation de A.

L'ensemble des nombres entiers pairs est infini dénombrable car l'application $a : n \rightarrow 2n$ est une numérotation de cet ensemble. Lorsqu'il s'agira de numérotation, au lieu d'utiliser une notation fonctionnelle du type $a : n \rightarrow 2n$ nous utiliserons le plus souvent une notation du type $a_0 = 0, a_1 = 2, \dots, a_n = 2n, \dots$

L'ensemble des nombres premiers est infini dénombrable. Une numérotation de cet ensemble peut être définie par :

$$p_0 = 2$$

$p_n = \text{« le plus petit entier non multiple de } p_0, p_1, \dots, p_{n-1} \text{ »}.$

Une telle définition est appelée définition par récurrence.

Proposition 1 — Un sous-ensemble infini d'un ensemble infini dénombrable est infini dénombrable.

Démonstration :

Soit A = { $a_0, a_1, \dots, a_n, \dots$ } un ensemble infini dénombrable, donné avec une numérotation.

Soit B infini, $B \subset A$.

(1) Nous ne définirons pas les notions d'ensembles et d'entiers que nous considérons comme primitives. La notation \mathbb{N} désigne l'ensemble de tous les entiers 0, 1, 2, etc.

On définit une numérotation de B en posant :

$$b_o = a_{i_o} \text{ avec } i_o = \min\{i \in \mathbb{N} \mid a_i \in B\}$$

$$b_j = a_{i_j} \text{ avec } i_j = \min\{i \in \mathbb{N} \mid i > i_{j-1}, a_i \in B\}.(1)$$

(même procédé que pour numérotter l'ensemble des nombres premiers).

□

Proposition 2 — Soit A un ensemble infini non dénombrable, soit B un sous-ensemble infini dénombrable de A. Il existe des éléments de A qui ne sont pas dans B.

Démonstration :

C'est évident car si ce n'était pas le cas, toute numérotation de B serait une numérotation de A et donc A serait dénombrable contrairement à sa définition.

□

L'existence d'ensembles infinis non dénombrables peut choquer l'intuition car elle signifie qu'il y a des infinis essentiellement différents, et même, d'après la proposition 2, qu'il y a des infinis « plus infinis » que d'autres. C'est pourquoi nous allons donner plusieurs démonstrations de la proposition 3.

Proposition 3 — Il existe des ensembles infinis non dénombrables.

Démonstration 1 :

L'ensemble \mathbb{R}^+ de tous les nombres réels positifs est non dénombrable.

En effet supposons le contraire et soit $r_0, r_1, \dots, r_n, \dots$ une numérotation de \mathbb{R}^+ .

Chaque nombre réel positif r peut se développer en base 10, c'est-à-dire s'écrire sous la forme $r = r^0, r^1 r^2 \dots r^p \dots$ avec $r^0 \in \mathbb{N}$, $r^i \in \{0, 1, 2, \dots, 9\}$.

Cette écriture signifiant :

$$r = r^0 + r^1 10^{-1} + r^2 10^{-2} + \dots + r^p 10^{-p} + \dots$$

Si on impose à la suite r^i de ne pas se terminer par une infinité de 9 le développement de r est unique et à chaque développement $r^0, r^1 r^2 \dots r^p \dots$ correspond un nombre réel unique.

(1) Pour tout ensemble d'entiers A, on désigne par $\min A$ l'élément minimum de A, c'est-à-dire l'élément $a \in A$ tel que : $\forall b \in A : a \leq b$.

Ecrivons chaque nombre réel de notre numérotation sous forme développée :

$$r_0 = r_0^0, r_0^1 r_0^2 \dots r_0^p \dots$$

$$r_1 = r_1^0, r_1^1 r_1^2 \dots r_1^p \dots$$

...

$$r_n = r_n^0, r_n^1 r_n^2 \dots r_n^p \dots$$

...

La suite s^i définie par :

$s^i = 0$ si r_i^i est impair,

$s^i = 1$ si r_i^i est pair,

fournit le développement en base 10 d'un nombre réel $s = s^0, s^1 s^2 \dots s^n \dots$ qui ne se termine pas par une infinité de 9 (car il n'en comporte aucun !) et tel que :

$$s \neq r_0 \text{ car } s^0 \neq r_0^0$$

$$s \neq r_1 \text{ car } s^1 \neq r_1^1$$

etc.

Le nombre réel positif s n'est donc pas parmi $r_0, r_1, \dots, r_n, \dots$ et donc contrairement à sa définition $r_0, r_1, \dots, r_n, \dots$ n'est pas une numérotation de \mathbb{R}^+ .

□

Démonstration 2 :

L'ensemble $P(E)$ des parties d'un ensemble E n'est jamais équivalent à E (et donc $P(\mathbb{N})$ n'est pas dénombrable).

En effet supposons donnée $f : E \rightarrow P(E)$, bijective.

Soit $A = \{x \in E \mid x \notin f(x)\}$. Soit a tel que $f(a) = A$.

Si $a \notin f(a)$, alors $a \in A$, donc $a \in f(a)$. C'est impossible.

Si $a \in f(a)$, alors $a \notin A$, donc $a \notin f(a)$. C'est aussi impossible.

□

Démonstration 3 :

Si E a plus de deux éléments, l'ensemble des applications de E dans E (qu'on notera $A(E, E)$) n'est pas équivalent à E (et donc $A(\mathbb{N}, \mathbb{N})$ n'est pas dénombrable).

En effet supposons donnée $f : x \mapsto f_x ; E \rightarrow A(E, E)$, bijective.

Soient e_1 et e_2 dans E , $e_1 \neq e_2$.

Considérons $A = \{x \mid f_x(x) = e_1\}$

Soit g définie par :

$$\left. \begin{array}{l} g(x) = e_2 \text{ si } x \in A \\ g(x) = e_1 \text{ si } x \notin A \end{array} \right\} (*)$$

Soit e_3 tel que $f_{e_3} = g$.

Si $e_3 \in A$ alors $f_{e_3}(e_3) = e_1$, mais $f_{e_3} = g$,
donc $f_{e_3}(e_3) = g(e_3) = e_2$ d'après (*).

• Si $e_3 \notin A$ alors $f_{e_3}(e_3) \neq e_1$, mais $f_{e_3} = g$,
donc $f_{e_3}(e_3) = g(e_3) = e_1$ d'après (*).

□

Remarque :

La démonstration 2 montre en particulier que :

P(IN) (l'ensemble des parties de IN) est infini non dénombrable, c'est-à-dire « a plus d'éléments que IN »

P(P(IN)) (l'ensemble des parties de P(IN)) ne peut pas être mis en bijection avec P(IN), comme il a au moins autant d'éléments, on peut dire que P(P(IN)) « est plus infini que P(IN) ».

etc.

Il y a donc une infinité de sortes d'ensembles infinis.

3. — PROGRAMMES A I ENTRÉES ENTIÈRES ET J SORTIES ENTIÈRES

On suppose qu'on dispose d'un langage comme le BASIC, le PASCAL, le FORTRAN, etc., qui fonctionne sur une machine pour laquelle il n'y a pas de limitation de la mémoire. On suppose en plus que ce langage permet d'utiliser des entiers aussi grands qu'on veut, et que l'ordinateur qu'on utilise est totalement fiable.

- On appelle *programme à une entrée entière* tout programme dont la première instruction exécutable est une instruction du type :

entrer un entier n ,

n'ayant aucun autre ordre d'entrée, ne comportant aucun retour possible à cette instruction (pas de « aller en » sur cette instruction).

- On appelle *programme à une sortie entière* tout programme se terminant par :

imprimer k
arrêt.

et n'ayant aucun autre ordre d'impression et aucun autre ordre arrêt.

On définit de la même façon la notion de programme à *plusieurs entrées entières, à plusieurs sorties entières*.

On désignera par $P_{e_i s_j}$ l'ensemble des programmes à i entrées entières et j sorties entières.

Proposition 4 — Quels que soient i et j dans IN, l'ensemble $P_{e_i s_j}$ est dénombrable.

Démonstration :

Notons $\Sigma = \{a_1, a_2, \dots, a_n\}$ l'ensemble des symboles du langage utilisé.

L'ensemble des programmes corrects (sans erreur de syntaxe) est un sous-ensemble de l'ensemble de toutes les suites finies⁽¹⁾ d'éléments de Σ .

Raisonnement 1

L'ensemble de toutes les suites finies d'éléments de Σ (noté Σ^*) est dénombrable (voir exercices). $P_{e_i s_j}$ est un sous-ensemble infini de Σ^* donc $P_{e_i s_j}$ est dénombrable (proposition 1).

Raisonnement 2 (plus direct)

Montrer que $P_{e_i s_j}$ est dénombrable revient à trouver une numérotation $P_0, P_1, \dots, P_n, \dots$ des programmes à i entrées entières et j sorties entières (à chaque programme est attribué un numéro unique et des programmes différents ont des numéros différents). Voici un procédé « concret » pour réaliser une telle numérotation :

- On considère les éléments de Σ^* de longueur 1, il y en a un nombre fini. Parmi eux, il y en a peut-être qui sont dans $P_{e_i s_j}$, on les numérote P_0, P_1, \dots, P_{n_1} (en les classant, par exemple par ordre alphabétique).
 - ...
 - On considère les éléments de Σ^* de longueur ℓ , il y en a un nombre fini. Parmi eux, il y en a peut-être qui sont dans $P_{e_i s_j}$, on les numérote $P_{n_{\ell-1}+1}, \dots, P_{n_\ell}$ (en les classant par exemple par ordre alphabétique).
- Etc...

□

(1) A la place de suite finie d'éléments d'un ensemble A nous direns parfois « mot de A » ou « mot fini de A ». Nous utiliserons indifféremment la notation (x_1, x_2, \dots, x_m) ou $x_1 x_2 \dots x_m$ pour désigner une suite finie (un mot) de longueur m .

Fixons-nous, une fois pour toutes, une numérotation de $P_{e_1 s_1}$ (nous supposerons que c'est celle donnée dans le raisonnement 2 de la proposition 4 mais d'autres numérotations sont possibles) :

$$P_{e_1 s_1} = \{P_0, P_1, \dots, P_n, \dots\}$$

Si, pour la donnée $k \in \text{IN}$, le programme P_j imprime (au bout d'un temps fini, bien sûr), l'entier r (et ensuite arrive à « arrêt »), on écrira :

$$P_j(k) = r.$$

Si, pour la donnée $k \in \text{IN}$, le programme P_j ne s'arrête jamais, on écrira :

$$P_j(k) = \perp.$$

On utilisera des notations analogues pour les programmes de $P_{e_i s_j}$, $i \in \text{IN}$, $j \in \text{IN}$.

Proposition 5 — (existence d'un programme universel)

Il existe un programme $Q \in P_{e_2 s_1}$ tel que :

$$\begin{aligned} Q(j, k) &= r \quad \text{si } P_j(k) = r, \\ Q(j, k) &= \perp \quad \text{si } P_j(k) = \perp. \end{aligned}$$

Démonstration (non formelle)

Le langage que nous utilisons est suffisamment puissant pour qu'on puisse écrire un programme qui :

- (a) à partir de j reconstitue le programme P_j (étant donné que la numérotation choisie est celle du raisonnement 2 de la proposition 4, il n'y a pas de problème pour cette reconstitution).
- (b) exécute P_j avec la donnée k .

De manière un peu plus précise la structure du programme Q est la suivante :

- entrer l'entier j
- entrer l'entier k
- [reconstitution de P_j]
- [calcul de $P_j(k)$]
- imprimer $P_j(k)$
- arrêt.

Lorsque $P_j(k) = \perp$ l'exécution de Q avec les données j et k bouclera dans la partie [calcul de $P_j(k)$], on aura donc bien :

$$Q(j, k) = \perp.$$

Sinon au bout d'un temps fini le programme Q imprimera l'entier $P_j(k)$ et s'arrêtera.



Remarque :

Il est essentiel que le langage utilisé soit d'une certaine puissance. Par exemple, il faut pouvoir réaliser des additions, des multiplications, des boucles pour, des tests. Nous ne précisons pas plus mais les langages comme le BASIC, le PASCAL, le FORTRAN ont cette puissance. Le programme Q est appelé programme universel car à lui tout seul, il « mime » tous les programmes de $P_{e_1s_1}$.

Proposition 6 — Il n'existe aucun programme $R \in P_{e_2s_1}$ tel que :

$$R(j, k) = 0 \quad \text{si } P_j(k) = \perp$$

$$R(j, k) = 1 \quad \text{sinon (c'est-à-dire s'il existe } r \in \text{IN tel que } P_j(k) = r).$$

Avant de démontrer ce résultat, faisons quelques commentaires.

La définition du programme R semble assez proche de la définition du programme Q dont nous avons vu l'existence à la proposition 5, son but serait de détecter quand le programme n° j s'arrête pour la donnée k, et quand il ne s'arrête pas. Une telle détection des programmes qui bouclent est donc impossible. Quelle que soit la puissance du langage utilisé, et même avec une machine idéalisée comme celle que nous considérons (rappelons que nous avons supposé que nous n'avions aucune limitation de mémoire et que nous pouvions utiliser des entiers aussi grands que nécessaire), certaines choses *parfaitement précises et relativement simples à formuler* ne peuvent pas être programmées. Ce résultat, formulé de manière un peu différente, a été établi par A.-M. TURING, en 1936 [TUR 36].

Démonstration :

Raisonnons par l'absurde en supposant que R existe. A partir de R on peut construire un programme S $\in P_{e_1s_1}$ tel que :

$$S(j) = 0 \quad \text{si } P_j(j) = \perp,$$

$$S(j) = 1 \quad \text{sinon.}$$

La construction du programme S à partir du programme R peut se faire par exemple de la façon suivante :

- entrer l'entier j
- [exécution de R avec les données (j, j) et affectation du résultat $R(j, j)$ à la variable ℓ]
- imprimer ℓ
- arrêt

A partir de S on peut construire un programme T $\in P_{e_1s_1}$ tel que :

$$T(j) = 0 \quad \text{si } P_j(j) = \perp$$

$$T(j) = \perp \quad \text{sinon}$$

La construction du programme T à partir du programme S peut se faire par exemple de la façon suivante :

- entrer l'entier j
- [exécution de S avec la donnée j, et affectation du résultat $S(j)$ à la variable r]
- (*) • si $r = 1$ alors aller en (*)
- imprimer r
- arrêt

Le programme T est un programme de $P_{e_1s_1}$, il existe donc un entier h tel que $T = P_h$.

Regardons ce qui se passe pour $P_h(h)$.

Si $P_h(h) = \perp$ par définition de T, on a $T(h) = 0$, mais puisque $T = P_h$, ceci donne $P_h(h) \neq \perp$, c'est impossible.

Si $P_h(h) \neq \perp$ par définition de T, on a $T(h) = \perp$, mais puisque $T = P_h$, ceci donne $P_h(h) = \perp$, c'est impossible.

R ne peut donc pas exister.

□

4. — FONCTIONS RÉCURSIVES, THÈSE DE CHURCH

Soit f une fonction partielle de IN dans IN⁽¹⁾.

On dit que F est *récursive* (on dit aussi *calculable*) s'il existe $P \in P_{e_1s_1}$ tel que :

$$P(n) = \perp \quad \text{si } n \notin \text{def } f$$

$$P(n) = f(n) \quad \text{si } n \in \text{def } f$$

def f désigne l'ensemble de définition f.

Autrement dit, une fonction récursive est une fonction qu'on peut programmer.

(1) L'usage veut en théorie de la récursivité qu'à la place de *fonction définie partout* (qu'on appelle *application* dans l'enseignement secondaire français) on dise *fonction totale*, et qu'à la place de *fonction non forcément définie partout* (qu'on appelle simplement *fonction* dans l'enseignement secondaire français) on dise *fonction partielle*.

Proposition 7 — Il existe des fonctions de IN dans IN qui ne sont pas récursives.

Démonstration :

D'après la proposition 3 l'ensemble $A(\text{IN}, \text{IN})$ des applications (= fonctions totales) de IN dans IN n'est pas dénombrable. L'ensemble $F(\text{IN}, \text{IN})$ des fonctions (= fonctions partielles) de IN dans IN n'est donc pas dénombrable (si $F(\text{IN}, \text{IN})$ était dénombrable $A(\text{IN}, \text{IN})$ qui en est un sous-ensemble infini, serait dénombrable d'après la proposition 1). Soit Φ l'application qui à chaque fonction récursive f associe le programme de plus petit numéro possible dans $P_{e_1 s_1}$ qui calcule f . Cette application Φ est une bijection entre l'ensemble des fonctions récursives et un sous-ensemble infini S de $P_{e_1 s_1}$. Puisque $P_{e_1 s_1}$ est dénombrable (proposition 4), S est aussi dénombrable (proposition 1) et donc l'ensemble des fonctions récursives aussi.

L'ensemble des fonctions de IN dans IN n'est pas dénombrable, l'ensemble des fonctions récursives de IN dans IN est dénombrable, donc d'après la proposition 2, il existe des fonctions de IN dans IN qui ne sont pas récursives. □

Remarque :

Un ensemble non dénombrable contient en fait beaucoup plus d'éléments qu'un ensemble dénombrable (car par exemple, une infinité dénombrable d'ensembles dénombrables disjoints, ne suffit pas à faire un ensemble non dénombrable). La démonstration indiquée montre donc que : rares sont les fonctions de IN dans IN qui sont récursives. Cependant toutes les fonctions courantes ($n \rightarrow 2n$; $n \rightarrow n!$; $n \rightarrow \langle\langle n\text{-ième nombre premier}\rangle\rangle$, etc) sont récursives car il est immédiat de construire pour chacune d'elles un programme qui la calcule.

La démonstration de la proposition 7 semble conduire à un paradoxe : elle nous dit que la plupart des fonctions de IN dans IN sont non récursives alors que nous n'arrivons pas, à première vue, à imaginer une seule fonction de IN dans IN non récursive.

La démonstration de la proposition 8 en définissant explicitement une fonction totale non récursive (à partir de laquelle on pourrait en définir une infinité d'autres) va lever ce paradoxe apparent.

Proposition 8 — Il existe des fonctions totales de IN dans IN qui ne sont pas récursives.

Démonstration :

Soit la fonction $f : \text{IN} \rightarrow \text{IN}$ définie par :

$$f(n) = 1 \quad \text{si} \quad P_n(k) \text{ est défini pour tout } k \in \text{IN}$$

$$f(n) = 0 \quad \text{sinon.}$$

Supposons que f soit récursive. Soit P le programme qui calcule f . Avec P , on construit facilement un programme Q tel que :

$$Q(n) = P_n(n) + 1 \text{ si } f(n) = 1$$

$$Q(n) = 1 \quad \quad \quad \text{si } f(n) = 0$$

$Q(n)$ est toujours défini et $Q \in P_{e_{1s_1}}$ donc $Q = P_k$ pour un certain k . Par définition, on a :

$$P_k(k) = Q(k) = P_k(k) + 1, \text{ ce qui est impossible.}$$

□

Remarque :

Les fonctions récursives (qui sont les fonctions qu'on peut calculer avec des programmes) ne sont donc pas toutes les fonctions de IN dans IN, et même, certaines fonctions simples et parfaitement définies sur IN tout entier, ne sont pas récursives. Ceci signifie que le pouvoir des langages de programmation est limité. Ce qui est extraordinaire, c'est qu'en fait il est limité exactement de la même façon, quel que soit le langage⁽¹⁾. Dès qu'un langage contient suffisamment de primitives de base, il peut calculer toutes les fonctions récursives, mais quoique ce soit qu'on ajoute en plus, on n'arrive pas à calculer plus que les fonctions récursives (en fait on n'arrive pas à dépasser le pouvoir du mécanisme élémentaire de ce qu'on appelle les machines de TURING, voir bibliographie). Toutes les tentatives faites jusqu'à présent pour étendre les possibilités de calcul des langages se sont révélées inutiles et à chaque fois que l'étude a été menée à fond, on a pu établir que les fonctions calculées par les langages proposés étaient exactement les fonctions récursives.

Tous les mathématiciens, logiciens et informaticiens qui se sont intéressés à ces questions sont d'ailleurs persuadés que :

toutes les fonctions de IN dans IN calculables par le moyen d'algorithmes sont des fonctions récursives.

Cette affirmation appelée *Thèse de CHURCH* n'est pas démontrable (alors que sa réciproque est quasi évidente) car l'expression « calculable par le moyen d'algorithmes » n'a pas de sens formellement défini. Cependant, puisque les extensions envisagées jusqu'à présent de la notion d'algorithme n'ont jamais réussi à calculer autre chose que des fonctions récursives, on est en droit de considérer la thèse de Church comme « expérimentalement établie » : la notion mathématique de fonction récursive et la notion « physico-philosophique » de fonctions calculables par le moyen d'algorithmes coïncident exactement.

(1) Il s'agit bien sûr, ici, uniquement du pouvoir théorique des langages. Chaque langage a ses spécificités propres qui rendent plus facile la programmation de tel ou tel type de problème.

5. — ENSEMBLES RÉCURSIFS ET RÉCURSIVEMENT ÉNUMÉRABLES

Soit $A \subset \text{IN}$.

On dit que A est *récursif* ssi :

(α) la fonction $f : \text{IN} \rightarrow \text{IN}$ définie par :

$$f(n) = 1 \text{ si } n \in A$$

$$f(n) = 0 \text{ si } n \notin A$$

est récursive.

ou bien :

(α') Il existe un programme à une entrée qui, pour toute donnée $n \in \text{IN}$ imprime (au bout d'un temps fini !) :

OUI si $n \in A$

NON si $n \notin A$

On dit que A est *récursivement énumérable* (on écrit r.e.) ssi :

(β) Il existe une fonction récursive telle que :

$$A = \text{def } f$$

ou bien :

(β') Il existe un programme à zéro entrée qui imprime successivement tous les éléments de A .

Le fait que (α) \Leftrightarrow (α') et (β) \Leftrightarrow (β') résulte de façon immédiate de la définition des fonctions récursives en termes de programmes⁽¹⁾.

Proposition 9 — Si A est récursif, alors A est récursivement énumérable.

Démonstration :

Soit P un programme à une entrée qui, pour toute donnée $n \in \text{IN}$ imprime OUI si $n \in A$ et imprime NON si $n \notin A$.

On modifie ce programme de telle façon qu'à la place d'imprimer OUI, il imprime 1 (ou un entier quelconque) et qu'à la place d'imprimer NON, il boucle. La fonction récursive f associée à ce nouveau programme vérifie $A = \text{def } f$. Donc A est r.e.

D'une manière plus explicite voici comment à partir de P obtenir le programme Q dont la fonction récursive f vérifie $A = \text{def } f$.

- entrer un entier n
- [exécution de P avec la donnée n et affectation du résultat à la variable f]

(1) Pour (β) \Leftrightarrow (β') on utilise un principe analogue à celui de la démonstration de la proposition 11 (b).

- (*) • si $\ell = \text{“NON”}$ alors aller en (*)
- imprimer 1
- arrêt.

□

Proposition 10 — Si A est r.e. et que IN - A (le complémentaire de A dans IN) est aussi r.e., alors A est récursif.

Démonstration :

Soit P_A le programme qui imprime successivement tous les éléments de A, et P_{IN-A} celui qui imprime tous les éléments de IN-A.

Notons $P_A[m]$ le m-ième entier imprimé par P_A et $P_{IN-A}[m]$ le m-ième entier imprimé par P_{IN-A} .

Considérons alors le programme suivant :

- entrer un entier n
- BO : = VRAI
- m : = 0
- tant que BO = VRAI faire
 - si $P_A[m] = n$ alors faire
 - s : = 1
 - BO : = FAUX
 - fin - de - si
 - si $P_{IN-A}[m] = n$ alors faire
 - s : = 0
 - BO : = FAUX
 - fin -de - si
 - m : = m + 1
 - fin - de - tant - que
- imprimer s
- arrêt.

Ce programme (que, bien sûr, il faut traduire dans le langage qu'on s'est fixé au départ) pour la donnée de n, explore simultanément la liste des entiers imprimés par P_A et la liste des entiers imprimés par P_{IN-A} ; dès qu'il a trouvé n (ce qui se produit au bout d'un temps fini car $A \cup (IN-A) = IN$) il imprime 1 si $n \in A$ et 0 si $n \notin A$. La fonction récursive associée à ce programme montre que A est récursif (définition (α)).

□

Proposition 11

- (a) il existe des ensembles récursifs,
- (b) il existe des ensembles récursivement énumérables et non récursifs,
- (c) il existe des ensembles non récursivement énumérables.

Démonstration :

- (a) IN est bien sûr récursif, de même que par exemple l'ensemble des entiers pairs et l'ensemble des nombres premiers.
- (b) Soit $A = \{m \mid P_m(m) \neq \perp\}$
A est récursivement énumérable. En effet, considérons le programme suivant :
 - $n := 1$
 - pour $m := 0$ à n faire
 - si (le programme P_m avec la donnée m fournit un résultat en moins de n unités de temps) et (m non déjà imprimé) alors imprimer m
 - fin - de - pour
 - $n := n + 1$
 - aller en (*)

Pour tout $m \in A$, il existe un entier n tel que « le programme P_m avec la donnée m fournit un résultat en moins de n unités de temps », donc tout $m \in A$ sera imprimé par notre programme. De plus, seuls des $m \in A$ peuvent être imprimés, donc le programme indiqué montre que A est récursivement énumérable.

Si A était récursif, alors la fonction f définie par :

$$\begin{aligned} f(n) &= 1 \text{ si } P_n(n) \neq \perp \\ f(n) &= 0 \text{ si } P_n(n) = \perp \end{aligned}$$

serait récursive et on pourrait construire un programme tel que :

$$\begin{aligned} Q(n) &= P_n(n) + 1 & \text{si } f(n) = 1 \\ Q(n) &= 1 & \text{si } f(n) = 0 \end{aligned}$$

ce qui est impossible (voir proposition 8), l'ensemble A est donc récursivement énumérable sans être récursif.

- (c) IN-A n'est ni récursif ni récursivement énumérable.

Si IN-A était r.e., alors, d'après la proposition 10, A serait récursif, ce qui n'est pas le cas (d'après (b)).

Si IN-A était récursif, alors IN-A serait r.e., ce qui (nous venons de le montrer) est faux.

□

Soit E un ensemble fini. Nous supposerons que les éléments de E sont numérotés :

$$E = \{e_1, e_2, \dots, e_n\}$$

Nous supposerons aussi que les éléments de E sont des symboles du langage de programmation que nous utilisons. Si ce n'est pas le cas, il suffit d'identifier e_1 avec 1, e_2 avec 2, ..., e_n avec n .

Nous noterons E^* l'ensemble de toutes les suites finies d'éléments de E .

On considère alors la numérotation suivante des éléments de E^* :

$$\alpha_0 = () \quad (\text{la suite vide})$$

$$\alpha_1 = (e_1), \alpha_2 = (e_2), \dots, \alpha_n = (e_n)$$

$$\alpha_{n+1} = (e_1, e_1), \alpha_{n+2} = (e_1, e_2), \dots, \alpha_{n+n^2} = (e_n, e_n)$$

...

$$\alpha_{n+n^2+\dots+n^{p-1}+1} = \underbrace{(e_1, e_1, \dots, e_1)}_{p \text{ fois}}, \dots, \alpha_{n+n^2+\dots+n^p} = \underbrace{(e_n, e_n, \dots, e_n)}_{p \text{ fois}}$$

...

Les définitions que nous allons donner et les résultats que nous allons obtenir sont indépendants de la numérotation choisie pour E et de la numérotation choisie pour E^* à condition toutefois que la numérotation choisie pour E^* soit « programmable » (c'est-à-dire qu'il existe un programme P qui, pour la donnée de m , imprime α_m) ce qui est le cas pour celle définie juste au-dessus.

Soit $A \subset E^*$.

On dit que A est récursifssi :

(γ) l'ensemble $\{n \mid \alpha_n \in A\}$ est récursif.

ou bien

(γ') il existe un programme qui, pour toute donnée $\alpha \in E^*$ imprime (au bout d'un temps fini !) :

OUI si $\alpha \in A$,

NON si $\alpha \notin A$.

On dit que A est récursivement énumérablessi :

(δ) l'ensemble $\{n \mid \alpha_n \in A\}$ est r.e.

ou bien

(δ') il existe un programme à zéro entrée qui imprime successivement tous les éléments de A .

Le fait que (γ) \Leftrightarrow (γ') et (δ) \Leftrightarrow (δ') résulte de la définition des fonctions récursives et de ce que la numérotation de A^* est « programmable ».

De même que nous avons défini les notions d'ensembles récursifs et d'ensembles récursivement énumérables pour des parties de E^* (lorsque E est fini), on peut définir les notions d'ensembles récursifs et d'ensembles récursivement énumérables pour des parties de IN^* (l'ensemble des suites finies d'entiers) pour des parties de E^k (E ensemble fini, k entier) et pour des parties de IN^k (k entier), et plus généralement pour les parties de F^k (k entier) et pour les parties de F^* si ces notions sont définies pour F .

A chaque fois, les propositions 9, 10 et 11 restent vraies.

6. — PRÉDICATS DÉCIDABLES ET SEMI-DÉCIDABLES

Considérons une formule contenant des variables comme, par exemple :

$$n \geq m$$

$$\exists p : p^2 = q$$

la suite α commence par 0101

On appelle cela un prédictat et on utilise, par exemple, les notations :

$$P(n,m)$$

$$R(q)$$

$$S(\alpha)$$

Pour chaque attribution de valeurs à $n \in IN$ et $m \in IN$, $P(n,m)$ est ou bien vrai ou bien faux (par exemple $P(3,2)$ est vrai, $P(2,3)$ est faux).

De même, pour chaque attribution de valeurs à $q \in IN$, $R(q)$ est ou bien vrai ou bien faux (par exemple, dans IN , $R(4)$ est vrai, $R(5)$ est faux). De même pour $S(\alpha)$.

Soit un prédictat général $T(x_1, x_2, \dots, x_n)$ ayant n variables x_1, x_2, \dots, x_n variant dans F , où F est un ensemble pour lequel les notions de parties récursives et de parties récursivement énumérables ont été définies (par exemple $F = IN$, $F = E^*$ avec E fini, $F = IN^*$, $F = IN^k$).

On dit que T est *décidable* (ou récursif)ssi :

(Θ) $\{(x_1, x_2, \dots, x_n) \mid T(x_1, x_2, \dots, x_n) \text{ est vrai}\}$ est récursif.

ou bien :

(Θ') il existe un programme P qui, pour toute donnée x_1, x_2, \dots, x_n dans F imprime (au bout d'un temps fini) :

- OUI si $T(x_1, x_2, \dots, x_n)$ est vrai,
- NON si $T(x_1, x_2, \dots, x_n)$ est faux.

On dit que T est *semi-décidable* ssi :

- (μ) $\{(x_1, x_2, \dots, x_n) \mid T(x_1, x_2, \dots, x_n) \text{ est vrai}\}$
est r.e..

ou bien

- (μ') il existe un programme P qui, pour toute donnée x_1, x_2, \dots, x_n dans F :
- imprime OUI (au bout d'un temps fini) si $T(x_1, x_2, \dots, x_n)$ est vrai,
 - continue de tourner indéfiniment si $T(x_1, x_2, \dots, x_n)$ est faux,

ou bien

- (μ'') il existe un programme à zéro entrée qui imprime tous les n -uplets (x_1, \dots, x_n) pour lequel $T(x_1, x_2, \dots, x_n)$ est vrai.

Un prédicat décidable correspond à une famille infinie de questions pour laquelle il existe un algorithme qui, quelle que soit la question, répond en un temps fini (non fixé à l'avance) et de manière correcte à la question.

Par exemple :

« n est un diviseur de m » : $T(n, m)$

est décidable.

Un prédicat semi-décidable correspond à une famille infinie de questions, pour laquelle il existe un algorithme qui, pour chaque question dont la réponse est positive, l'indique en un temps fini et qui, pour les autres, n'indique jamais rien.

Le prédicat « n est le numéro d'un programme à une entrée qui, pour la donnée n , s'arrête en un temps fini » : $T(n)$, est semi-décidable mais n'est pas décidable.

L'algorithme (qu'on peut mettre sous forme de programme) qui montre la semi-décidabilité est le suivant :

- A partir de n , reconstituer le programme à une entrée entière qui a pour numéro n .
- Faire tourner ce programme avec la donnée n .
- Si le programme s'arrête, imprimer OUI.

Ce prédicat n'est pas décidable car sinon, l'ensemble $\{m \mid P_m(m) \neq \perp\}$ serait récursif, ce qui est faux (proposition 11).

En raisonnant comme au § 5, on montre que :

- Si le prédicat $T(x_1, \dots, x_n)$ est décidable, alors il est semi-décidable.
- Si le prédicat $T(x_1, \dots, x_n)$ et sa négation non $T(x_1, \dots, x_n)$ sont semi-décidables, alors $T(x_1, \dots, x_n)$ est décidable.
- Il existe des prédicats qui ne sont pas semi-décidables (par exemple « n est le numéro d'un programme qui ne s'arrête pas »).

Les exemples donnés ici, de prédicts non décidables et non semi-décidables peuvent sembler artificiels. Sachez qu'il y a d'autres cas d'indécidabilité en logique, en théorie des langages, en arithmétique et en informatique qui présentent un intérêt évident.

Nous verrons au chapitre 5 que le prédictat :

« la formule F du calcul des prédictats du 1^{er} ordre est un théorème »

est un prédictat semi-décidable et non décidable. Ce résultat a en particulier pour conséquence qu'il ne peut exister aucun moteur d'inférences complètement satisfaisant pour des langages basés sur le calcul des prédictats du premier ordre comme PROLOG.

L'Intelligence Artificielle est confrontée dès le départ à ces résultats incontournables d'indécidabilité. En avoir conscience nous semble fondamental et peut éviter de rechercher des méthodes dont il est établi par avance qu'elles ne peuvent pas exister.

EXERCICES

Exercices sur les ensembles infinis dénombrables

- (a) Montrer que $\text{IN} \times \text{IN}$ est dénombrable.
- (b) Montrer que l'ensemble des suites finies d'éléments d'un ensemble fini $E = \{e_1, e_2, \dots, e_n\}$ est dénombrable.
- (c) Montrer que l'ensemble des suites finies d'éléments d'un ensemble infini dénombrable $\{e_1, e_2, \dots, e_n, \dots\}$ est dénombrable.
- (d) Montrer que l'ensemble des suites infinies d'éléments d'un ensemble E ayant deux ou plus de deux éléments est un ensemble infini non dénombrable.

Exercices sur les ensembles finis et infinis

Indiquer, pour chacun des ensembles suivants, s'il est fini, infini dénombrable ou infini non dénombrable.

- (a) Z , l'ensemble des entiers négatifs et positifs.
- (b) Q , l'ensemble des nombres rationnels.
- (c) $\{(x, y) \mid x \in Z, y \in Z, x^2 + y^2 \leq 100\}$.
- (d) $\{(x, y) \mid x \in Q, y \in Q, x^2 + y^2 \leq 100\}$.
- (e) $\{(x, y) \mid x \in IR, y \in IR, x^2 + y^2 \leq 100\}$.
- (f) $\{\alpha \in \text{IN}^* \mid \sum \alpha_i = 5\}$.

IN^* désigne ici l'ensemble des mots finis (suites finies) d'éléments de IN , si $\alpha \in \text{IN}^*$ on note $\alpha = \alpha_1 \alpha_2 \dots \alpha_n$

- (g) {fonctions croissantes de IN dans IN }.

Exercices sur les programmes à i entrées et j sorties

(a) Existe-t-il un programme $P \in P_{e_2 s_1}$ tel que :

$$\begin{aligned} P(k, \emptyset) &= P_k(P_k(\emptyset)) && \text{si } P_k(\emptyset) \neq \perp \\ &&& \text{et } P_k(P_k(\emptyset)) \neq \perp \\ &= \perp && \underline{\text{sinon.}} \end{aligned}$$

(b) Existe-t-il un programme $P \in P_{e_3 s_2}$ tel que :

$$\begin{aligned} P(k, m, n) &= P_k(m) + P_k(n) + 1 && \text{si } P_k(m) \neq \perp \\ &&& \text{et } P_k(n) \neq \perp, \\ &= P_k(m) && \text{si } P_k(m) \neq \perp \\ &&& \text{et } P_k(n) = \perp, \\ &= P_k(n) && \text{si } P_k(n) \neq \perp \\ &&& \text{et } P_k(m) = \perp \\ &= 0 && \text{si } P_k(m) = P_k(n) = \perp ? \end{aligned}$$

Exercice sur la détection des boucles dans les programmes à zéro entrée et une sortie

Comme dans le cours, on numérote les programmes à zéro entrée et une sortie.

$$P_{e_0 s_1} = \{Q_0, Q_1, Q_2, \dots\}$$

Montrer qu'il n'existe pas de programme $U \in P_{e_1 s_1}$ tel que :

$U(i) = 0$ si Q_i ne s'arrête jamais.

$U(i) = 1$ si Q_i s'arrête (en un temps fini).

Exercices sur les fonctions récursives

(a) Montrer que toute fonction totale f de IN dans IN, décroissante, est récursive.

(b) Soit g une fonction totale de IN dans IN non injective. Construire une fonction totale h de IN dans IN telle que :

h soit non récursive,

$g \circ h$ soit récursive.

Exercices sur les fonctions non récursives

Définir une fonction totale de IN dans IN qui soit :

(a) croissante et non récursive,

(b) non récursive et telle que $f \circ f$ soit récursive.

Exercices sur les fonctions récursives et non récursives

Indiquer en justifiant brièvement votre réponse quelles sont, parmi les fonctions suivantes, celles qui sont récursives et celles qui ne le sont pas :

- (a) $f(n)$ = nombre de programmes de moins de n symboles.
- (b) $f(n) = 0$ s'il y a une infinité de programmes dans $P_{e_1s_1}$ tel que $P(0) = n = 1$ sinon.
- (c) $f(n) = n$ -ième chiffre dans le développement décimal de π .
- (d) $f(\ell) = 1$ pour tout $\ell \in \text{IN}$ si il existe
 $n \in \text{IN} - \{0\}$, $m \in \text{IN} - \{0\}$, $p \in \text{IN} - \{0\}$
 $q \in \text{IN} - \{0, 1, 2\}$ tels que $n^q + m^q = p^q$.
 $f(\ell) = 0$ pour tout $\ell \in \text{IN}$, sinon.
- (e) $f(n) = 1$ si $P_n(k) \neq 0$ pour tout $k \in \text{IN}$,
= 0 sinon.
- (f) $f(n) = \perp$ si $P_n(k) \neq 0$ pour tout $k \in \text{IN}$,
= 1 sinon.

Exercices sur les ensembles récursifs et récursivement énumérables

Indiquer, en justifiant votre réponse, quels sont, parmi les ensembles suivants, ceux qui sont :

- récursifs,
- récursivement énumérables et non récursifs,
- non récursivement énumérables.

- (a) $\{2^n \mid n \in \text{IN}\}$
- (b) $\{p \mid p \text{ est le produit de deux nombres premiers}\}$
- (c) $\{\alpha \in \text{IN}^* \mid \alpha \text{ contient } 0\}$
- (d) $\{\alpha \in \text{IN}^* \mid \alpha \text{ ne contient pas } 0\}$
- (e) $\{P \in P_{e_1s_1} \mid P(0) = 0\}$
- (f) $\{P \in P_{e_1s_1} \mid P(0) \neq 0\}$
- (g) $\{(P, Q) \mid P \in P_{e_1s_1}, Q \in P_{e_1s_1} : \forall n : P(n) = Q(n)\}$

Exercice sur les ensembles récursifs et récursivement énumérables.

Soit A un ensemble récursivement énumérable.

Montrer que s'il existe un programme qui énumère les éléments de A par ordre croissant alors A est récursif.

Exercice sur les ensembles récursifs et récursivement énumérables

Etablir de façon détaillée que les propriétés (α) et (α') sont équivalentes. De même pour (β) et (β') ; (γ) et (γ') ; (δ) et (δ') ; (Θ) et (Θ') ; (μ) , (μ') et (μ'') .

Exercices sur les prédictats décidables et semi-décidables

Indiquer en justifiant votre réponse quels sont, parmi les prédictats suivants, ceux qui sont :

- décidables,
- semi-décidables et non décidables,
- non semi-décidables.

- (a) $m = n + p$
- (b) $\exists n \ m = n + p$
- (c) n est la somme de 2 nombres premiers
- (d) n est le numéro d'un programme de $P_{e_1s_1}$ qui contient un ordre « si... alors... »
- (e) n est le numéro d'un programme de $P_{e_1s_1}$ qui exécute un ordre « si... alors... » pour la donnée n .
- (f) n est le numéro d'un programme de $P_{e_1s_1}$ qui n'exécute aucun ordre « si... alors... » pour la donnée n .

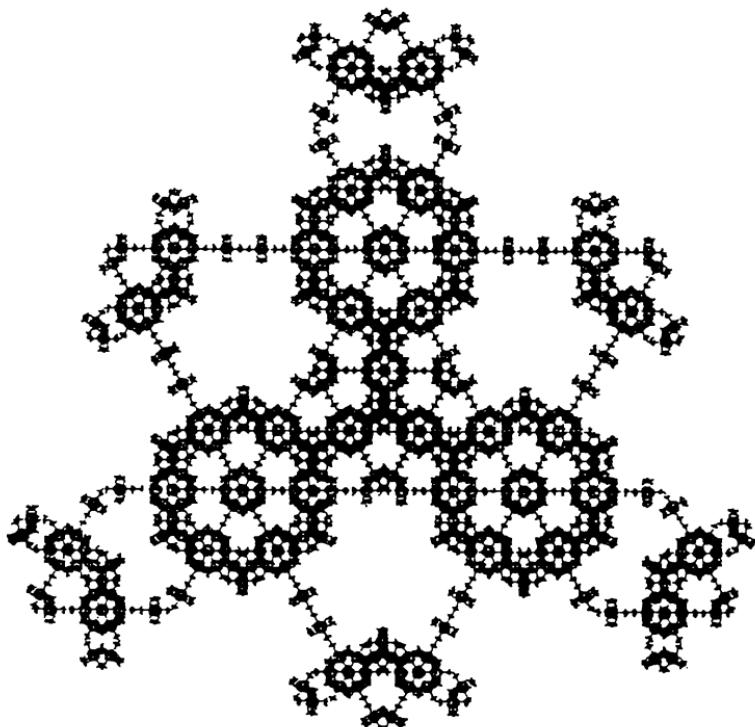
BIBLIOGRAPHIE

- [AZR 73] AZRA J.-P. et JAULIN B.
Récursivité.
Collection « Programmation », Gauthier-Villars, Paris, 1973.
{Les notions de récursivité sont introduites à l'aide de programmes dans un langage spécifique ne comportant que 5 types d'instructions. Assez complet, mais relativement difficile}.
- [CAR 75] CARREZ C.
Récursivité et Décidabilité.
Polycopié de l'Université des Sciences et Techniques de Lille, 1975.
{Présentation en termes de simulateur digital}.
- [DAV 58] DAVIS M.
Computability and Unsolvability.
Mac Graw Hill, New York, 1958.
{De même que le livre de ROGERS cité plus bas il s'agit d'un ouvrage de référence}
- [HAN 84] HANDLER D.
La machine universelle.
Science et Avenir, numéro spécial Informatique, 1984.
{Article de vulgarisation présentant la notion de machine de Turing}
- [HOF 80] HOFSTADTER D. R.
Gödel, Escher, Bach : an eternal golden braid.
Vintage Books Editions, New York, 1980. InterEditions, Paris, 1985.
{Le chapitre XIII constitue une présentation concise, très bien expliquée, des concepts de récursivité introduits par le biais d'un langage de programmation structuré. De nombreuses variantes de la thèse de Church sont énoncées et discutées. Astucieux et profond}.
- [HOP 84] HOPCROFT J.
Les machines de Turing.
Pour la Science, juillet 1984, pp. 46-57.
{Par un spécialiste, une présentation précise de la notion de machine de Turing et de quelques résultats de base}
- [KFO 82] KFOURY A. J., MOLL R. N. et ARBIB M. A.
A Programming Approach to Computability.
Springer-Verlag, New York, 1982.
{En partant d'une version restreinte du langage PASCAL les auteurs introduisent les notions de calculabilité et de récursivité, et les étudient.
A conseiller pour qui veut approfondir le sujet du chapitre sans toutefois tomber dans les traités généraux destinés aux chercheurs}

- [KLE 71] KLEENE S. C.
Logique mathématique.
Librairie Armand Colin, Paris, 1971.
(Traduction de : Mathematical logic, John Wiley and Sons, New York, 1967).
{Le chapitre 5 de ce livre classique que nous citerons encore plusieurs fois, est un exposé très bien commenté des résultats fondamentaux concernant la décidabilité, abordés par la méthode des machines de Turing.}
- [ROG 67] ROGERS A.
Theory of Recursive Functions and Effective Computability.
Mc Graw Hill Book Company, New York, 1967.
{Il s'agit du traité le plus complet sur les questions de récursivité}
- [SAL 73] SALOMAA A.
Formal Languages.
Academic Press, New York, 1973.
{Dans le cadre de la théorie des langages, on trouvera au chapitre 1 la présentation des machines de Turing et d'autres mécanismes abstraits, puis, dans les autres chapitres divers résultats d'indécidabilité « concrets » c'est-à-dire à propos de problèmes posés indépendamment des numérotations de programme}
- [TUR 36] TURING A. M.
On computable numbers, with an application to the Entscheidungsproblem.
Proceeding of the London Mathematical Society Vol. 42, 1936, pp. 230-265 et Vol. 43, 1937, pp. 544-546.
{Article original de Turing dans lequel se trouve la première définition de ce qu'on appelle aujourd'hui les machines de Turing, ainsi que divers résultats les concernant}
- [YAS 71] YASUHARA A.
Recursive Function Theory and Logic.
Academic Press, New York, 1971.
{Introduction rigoureuse aux concepts récursivistes et à leurs applications en logique. Très bon manuel que nous citerons dans plusieurs autres chapitres.}

CHAPITRE 3

SYSTÈMES FORMELS



I. — INTRODUCTION

La théorie des systèmes formels constitue un cadre général dans lequel on peut exprimer et étudier, de façon entièrement rigoureuse et mathématique, les notions d'axiomatique et de mécanisme déductif. En particulier, les concepts de démonstration et de théorème deviennent des concepts mathématiques objets, au sujet desquels on peut établir des résultats (qualifiés de métamathématiques) au même titre qu'on établit des résultats au sujet des nombres entiers ou des matrices inversibles. Ces résultats indiquent des propriétés générales applicables à tous les mécanismes et systèmes de déduction, et concernent à ce titre les moteurs d'inférences des systèmes experts. Il n'est peut-être pas exagéré de dire que cette théorie est le noyau abstrait de la théorie des systèmes experts.

En plus de ces aspects directement liés aux problèmes de déduction logique et mathématique la notion de système formel joue d'autres rôles importants.

En épistémologie par exemple on admet que toute science au fur et à mesure de ses progrès tend à se formaliser de plus en plus, l'idéal étant atteint (en mathématique bien sûr mais aussi dans certains domaines de la physique) lorsque la théorie s'exprime entièrement dans un système formel.

En informatique toutes les manipulations réalisées (y compris les manipulations numériques) sont des manipulations symboliques formelles, et donc, les problèmes des possibilités combinatoires et du pouvoir de représentation des systèmes formels y sont essentiels. C'est encore plus vrai en Intelligence Artificielle où on a à modéliser (c'est-à-dire à formaliser) des connaissances et des comportements. Il s'agit de problèmes difficiles, à la fois profonds et pratiques, c'est pourquoi les outils et les résultats des mathématiciens qui travaillent sur ces questions depuis plusieurs dizaines d'années doivent être pris en compte et utilisés quand c'est possible.

La courte présentation proposée ici est très incomplète bien sûr, mais elle fixera quelques idées et pourra être approfondie par la lecture de différents textes indiqués dans la bibliographie du chapitre.

2. — DÉFINITION

On appelle *système formel* S la donnée :

- d'un *alphabet* Σ_S , qui peut être fini, ou infini dénombrable, auquel cas on se donne une numérotation fixée une fois pour toutes de Σ_S .
- d'un sous-ensemble récursif F_S de l'ensemble de toutes les suites finies d'éléments de Σ_S :
$$F_S \subset \Sigma^*_S, F_S \text{ récursif.}$$

F_S est appelé ensemble des *formules bien formées de S*.

- (c) d'un sous-ensemble récursif A_S de F_S , appelé ensemble des *axiomes de S* :
 $A_S \subset F_S$, A_S récursif.
- d) d'un ensemble fini R_S de prédictables décidables définis sur F_S , appelés *règles d'inférence* :

$$R_S = \{r_1, r_2, \dots, r_n\}.$$

Au lieu de noter $r_i(f_1, f_2, \dots, f_\ell, g)$

on écrira : $f_1, f_2, \dots, f_\ell \vdash_{r_i} g$.

et on lira « à partir des formules f_1, f_2, \dots, f_ℓ on peut déduire la formule g par la règle r_i ».

Remarques :

- 1) L'ensemble des formules bien formées peut être défini par une grammaire de production ; dans ce cas, on prendra bien garde de ne pas confondre les règles de production de la grammaire et les règles d'inférence.
- 2) Les conditions de récursivité et de décidabilité imposées dans la définition signifient :
 - il existe un programme P_1 qui, pour la donnée de $f \in \Sigma_S^*$, indique (au bout d'un temps fini) si $f \in F_S$ ou si $f \notin F_S$.
 - il existe un programme P_2 qui, pour la donnée de $f \in F_S$, indique (au bout d'un temps fini) si $f \in A_S$ ou si $f \notin A_S$.
 - pour chaque règle r_i , il existe un programme Q_i qui, pour la donnée de $f_1, f_2, \dots, f_\ell, g$ indique (au bout d'un temps fini) si $f_1, f_2, \dots, f_\ell \vdash_{r_i} g$ ou non.

Exemple :

- $\Sigma_S = \{1, +, =\}$
- $F_S = \text{ensemble des formules composées d'un nombre fini non nul de "1" puis du symbole " + ", puis d'un nombre fini non nul de "1", puis du symbole " = ", puis d'un nombre fini non nul de "1", ce que nous noterons :}$

$$F_S = \{1^n + 1^m = 1^p \mid n \in \mathbb{N} - \{0\}, m \in \mathbb{N} - \{0\}, p \in \mathbb{N} - \{0\}\}$$

(en théorie des langages, on utilise parfois la notation :

$$F_S = (1^+ + 1^+ = 1^+)$$

• $A_S = \{1 + 1 = 11\}$, il y a donc un axiome unique.

• $R_S = \{r_1, r_2\}$

$$r_1 : 1^n + 1^m = 1^p \xrightarrow[r_1]{} 1^{n+1} + 1^m = 1^{p+1}$$

(à partir de la formule $1^n + 1^m = 1^p$, on peut déduire $1^{n+1} + 1^m = 1^{p+1}$).

$$r_2 : 1^n + 1^m = 1^p \xrightarrow[r_2]{} 1^n + 1^{m+1} = 1^{p+1}$$

(à partir de la formule $1^n + 1^m = 1^p$ on peut déduire $1^n + 1^{m+1} = 1^{p+1}$).

On appelle *déduction à partir de* h_1, h_2, \dots, h_n toute suite finie de formules f_1, f_2, \dots, f_p telle que, pour tout $i \in \{1, 2, \dots, p\}$:

(a) f_i est un axiome

ou bien :

(b) f_i est l'une des formules h_1, h_2, \dots, h_n

ou bien :

(c) f_i est obtenue par application d'une règle $r_k \in R_S$ à partir de formules $f_{i_0}, f_{i_1}, \dots, f_{i_l}$ placées avant f_i :

$$f_{i_0}, f_{i_1}, \dots, f_{i_l} \xrightarrow[r_k]{} f_i, i_0 < i, i_1 < i, \dots, i_l < i.$$

Dans une telle situation, on dit aussi que :

f_1, f_2, \dots, f_p est une *déduction de* f_p à partir des hypothèses h_1, h_2, \dots, h_n , ce que l'on note :

$$h_1, h_2, \dots, h_n \xrightarrow[S]{} f_p.$$

On appelle *théorème de S* toute formule t pour laquelle il existe une déduction à partir de \emptyset , c'est-à-dire telle que :

$$\xrightarrow[S]{} t.$$

L'ensemble des théorèmes de S est noté T_S .

Une déduction à partir de \emptyset est aussi appelée simplement *déduction* ; il s'agit donc d'une suite de formules f_1, f_2, \dots, f_p telle que, pour tout $i \in \{1, 2, \dots, p\}$:

(a) ou bien (c) .

Exemple :

La suite de formules :

$$f_1 : 1 + 11 = 11$$

$$f_2 : 1 + 111 = 111$$

$$f_3 : 11 + 111 = 1111$$

(application de r_2 à f_1)

(application de r_1 à f_2)

constitue une déduction de $11 + 111 = 1111$ à partir de la formule $1 + 11 = 11$ (qui n'est pas un axiome).

La suite de formules :

$f_1 : 1 + 1 = 11$	(axiome)
$f_2 : 1 + 11 = 111$	(application de r_2 à f_1)
$f_3 : 1 + 111 = 1111$	(application de r_2 à f_2)
$f_4 : 11 + 111 = 11111$	(application de r_1 à f_3)

constitue une déduction de $11 + 111 = 11111$ à partir de \emptyset

donc $11 + 111 = 11111$ est un théorème de S :

$$11 + 111 = 11111 \in T_S.$$

De manière à visualiser facilement certaines déductions on utilise parfois une représentation en arbre ou même en graphe. Par exemple, imaginons que dans un certain système formel S on ait la déduction :

- f_1 Axiome 1
- f_2 Axiome 2
- f_3 application de r_1 à f_1 et f_2
- f_4 application de r_2 à f_2
- f_5 application de r_3 à $f_2 f_3 f_4$

Cette déduction pourra être représentée par l'arbre suivant :

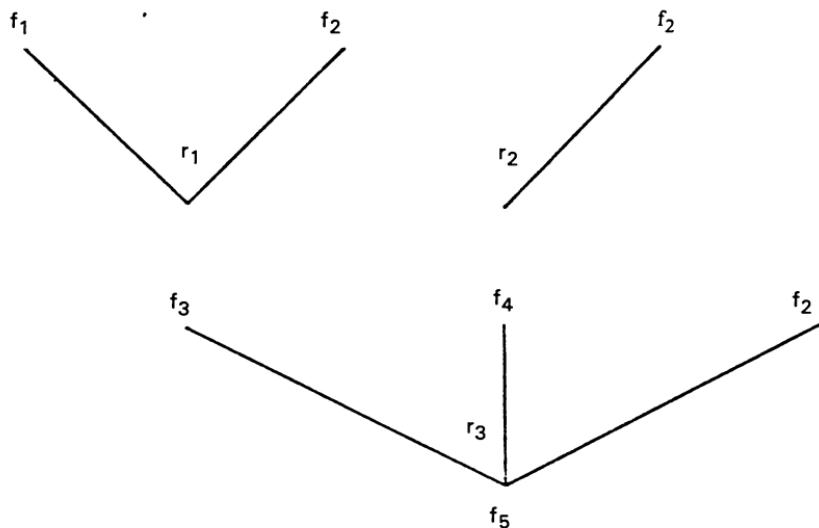


Fig. 3 — Arbre de dérivation de f_5

Chaque extrémité de l'arbre (on dit parfois chaque feuille) doit être un axiome ou une hypothèse.

La même déduction peut être représentée par un graphe sans cycle et ayant un nombre de nœuds égal à la longueur de la déduction.

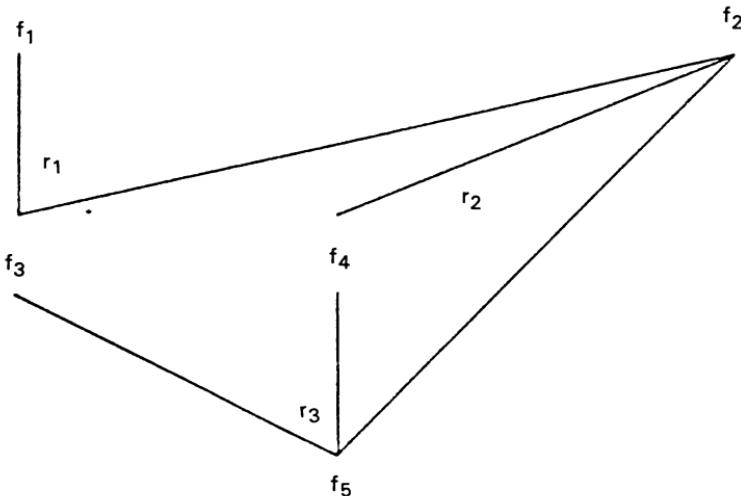


Fig. 4 — Graphe de dérivation de f_5

Cette représentation est parfois assez emmêlée.

Dans notre exemple, et dans tout système formel, les manipulations effectuées sur les formules sont purement syntaxiques et n'utilisent aucunement le sens qu'on peut attribuer aux formules.

Cependant assez naturellement, (et c'est ce qui fait l'intérêt des systèmes formels), une ou plusieurs *interprétations* possibles des formules s'imposent à l'esprit. Ces interprétations sont parfois utiles pour démontrer des résultats sur les systèmes formels, mais on doit faire attention à ne pas se laisser abuser par les interprétations, et par exemple, croire que ce que l'on sait être vrai est forcément un théorème. Il se peut que, justement, il n'y ait pas *adéquation* entre le formalisme, les axiomes et les règles d'inférence adoptées, et la vérité qu'on a voulu formaliser. Il se peut aussi que l'adéquation ne soit que partielle, dans le cas de notre exemple, on aimerait que : $1 + 1 + 1 = 111$ soit un théorème de S alors que ce n'est pas le cas, tout simplement parce que $1 + 1 + 1 = 111$ n'est pas une formule bien formée de S.

On dit qu'un système formel S est *cohérent* s'il existe des formules bien formées qui ne sont pas des théorèmes.

On dit qu'un système formel S comportant un symbole de négation (il est souvent noté « \neg » et se lit « non ») est *consistant* (ou *non-contradictoire*) s'il n'existe aucune formule bien formée $f \in F_S$ telle que f et $\neg f$ soient des théorèmes de S . Dans le cas contraire, on parle de système *inconsistant* (ou *contradictoire*).

On dit qu'un système formel S comportant un symbole de négation est *catégorique* si, pour toute formule bien formée $f \in F_S$ (en calcul des prédictats du premier ordre on ajoute « n'ayant pas de variables libres ») on a :

$$f \in T_S \text{ ou } \neg f \in T_S.$$

On dit qu'un système formel S est *saturé* si, pour toute formule bien formée $f \notin T_S$, le système formel S' obtenu en ajoutant f aux axiomes de S , est inconsistant.

On dit que les axiomes d'un système formel S sont *indépendants* si, à chaque fois que l'on enlève un axiome à S , on obtient un système formel ayant moins de théorèmes.

On appelle *problème de la décision pour S* , le problème de savoir si T_S est récursif ou pas ; autrement dit, le problème de savoir si le prédictat $P(t)$:

$$t \text{ est un théorème de } S$$

est décidable ou pas.

Lorsque le problème de la décision admet une réponse positive, on dit que *le système formel S est décidable*.

On dit qu'un système formel S est *finiment axiomatisable* (ou fini axiomatisable) si A_S est fini ou s'il existe un système formel ayant même alphabet, même ensemble de formules bien formées, même ensemble de règles d'inférence, même ensemble de théorèmes et ayant un ensemble fini d'axiomes.

Lorsqu'on a en vue la modélisation d'une situation particulière et qu'on voudrait obtenir comme ensemble de théorèmes de S un ensemble T fixé à l'avance, on dit que S est correct (« *sound* » en anglais) si $T_S \subseteq T$ et on dit que S est complet (« *complete* » en anglais) si $T_S = T$. Ce vocabulaire sera utilisé dans le chapitre sur la résolution.

Remarque :

D'un simple point de vue syntaxique, les deux problèmes les plus importants sont celui de la consistance et celui de la décision.

La notion de système formel a été introduite en logique mathématique avant de servir en informatique, et de nombreux résultats souvent assez déconcertants ont été démontrés. Le théorème de Gödel de 1931 indique, par exemple, qu'on ne peut établir la consistance d'un système formel de l'arithmétique qu'à

l'aide de moyens plus puissants que ceux du système en question, et que donc, en particulier, il est impossible d'établir la consistance de l'arithmétique par des moyens purement combinatoires en raisonnant sur les formules et les déductions du système. Ce résultat montra que les buts que s'étaient fixés l'école formaliste de HILBERT (la démonstration de la consistance de systèmes formels pouvant exprimer les mathématiques, à l'aide de moyens purement « finistes ») ne pourraient jamais être atteints.

D'un point de vue plus général, un autre problème fondamental concernant les systèmes formels est celui de leur adéquation au domaine qu'on veut modéliser en les définissant. Là encore, des résultats assez inattendus ont été prouvés. Toujours dans son article de 1931, GÖDEL montre, par exemple, qu'aucun système formel de l'arithmétique ne peut être catégorique et qu'en particulier, si un système formel de l'arithmétique est consistant et n'admet comme théorèmes que des formules vraies (c'est là une exigence minimale d'adéquation), alors il existe des formules d'arithmétique qui sont vraies et qui ne sont pas des théorèmes de ce système formel.

3. — RÉSULTATS GÉNÉRAUX

Proposition 1

$$\begin{array}{l} \text{Si } h_1, h_2, \dots, h_k \vdash_S g_1, \\ h_1, h_2, \dots, h_k \vdash_S g_2, \end{array}$$

...

$$h_1, h_2, \dots, h_k \vdash_S g_\ell,$$

$$\text{et } g_1, g_2, \dots, g_\ell \vdash_S n$$

$$\text{alors } h_1, h_2, \dots, h_k \vdash_S n.$$

En particulier :

$$\text{si } h \vdash_S g \text{ et } g \vdash_S n$$

$$\text{alors } h \vdash_S n.$$

Démonstration :

Pour tout $i \in \{1, 2, \dots, \ell\}$ soit $f_1^i, f_2^i, \dots, f_{P_i}^i$ une déduction de g_i à partir de h_1, h_2, \dots, h_k .

Soit aussi r_1, r_2, \dots, r_q une déduction de n à partir de g_1, g_2, \dots, g_ℓ .

On vérifie, en utilisant la définition de déduction, que :

$$f_1^1, f_2^1, \dots, f_{P_1-1}^1, f_1^2, f_2^2, \dots, f_{P_2-1}^2, \dots, f_1^\ell, f_2^\ell, \dots, f_{P_\ell-1}^\ell, r_1, r_2, \dots, r_q.$$

est une déduction de n à partir de h_1, h_2, \dots, h_k .

□

Proposition 2 — L'ensemble des déductions est récursif.

Démonstration :

Il s'agit de montrer qu'il existe un programme qui, pour toute suite finie f_1, f_2, \dots, f_n d'éléments de Σ^*_S indique au bout d'un temps fini si cette suite est une déduction ou pas.

Voici comment procéder pour construire ce programme :

- on commence par vérifier si chaque f_i est une formule bien formée (en utilisant le programme P_1).
- on repère ensuite les axiomes (en utilisant le programme P_2).
- pour chaque formule, qui n'est pas un axiome, on vérifie qu'elle peut s'obtenir à partir des formules qui la précèdent (en utilisant les programmes Q_1, Q_2, \dots, Q_ℓ).

Les programmes $P_1, P_2, Q_1, Q_2, \dots, Q_\ell$ sont ceux introduits dans la remarque qui suit la définition des systèmes formels.

□

Proposition 3 — L'ensemble T_S des théorèmes est récursivement énumérable.

Démonstration :

Il faut montrer qu'il existe un programme qui fait la liste de tous les théorèmes du système formel S .

Voici comment procéder pour construire ce programme .

- (a) on considère d'abord un programme qui fait la liste de toutes les suites finies de formules bien formées (la méthode utilisée pour énumérer tous les programmes de $P_{e_1 s_1}$ au chapitre 2 s'adapte sans difficulté).
- b) pour chacune des suites finies de formules bien formées, on utilise le programme défini dans la démonstration de la proposition 2, de manière à savoir s'il s'agit d'une déduction ou pas.
 - si la réponse est OUI, on imprime la dernière formule de la liste,
 - sinon, on passe à la liste suivante.

□

Proposition 4 — Il existe des systèmes formels S pour lesquels T_S n'est pas récursif.

Remarque :

Ce résultat est fondamental car il montre que, bien que tous les objets intervenant dans la définition d'un système formel soient récursifs, l'ensemble des théorèmes peut ne pas l'être. De plus, pour la plupart des systèmes formels un peu intéressants (ce qui n'est pas le cas de celui utilisé dans la démonstration, qui, lui, est tout à fait artificiel), T_S n'est pas récursif et donc :

alors qu'il existe toujours un moyen algorithmique, (c'est-à-dire un programme) qui détermine si une suite de formules est une déduction ou pas, il n'existe bien souvent aucun moyen algorithmique pour déterminer si une formule bien formée est un théorème ou pas.

Démonstration :

Considérons le système formel suivant :

- Σ_S : ensemble de symboles du langage de programmation qu'on utilise (voir chapitre 2), plus les symboles « [», «] », « | », « □ », « arrêté », « non arrêté ».
- F_S : ensemble de toutes les formules de l'une des formes suivantes :

$$[P_n \square \text{arrêté}]$$

$$[P_n \square |^m \square \text{non arrêté}]$$

où P_n désigne (comme au chapitre 2) le n -ième programme à une entrée entière et une sortie entière, et où $|^m$ désigne $\overbrace{| \dots |}^m$ fois

- A_S : ensemble des formules bien formées de la forme :

$$[P_n \square \square \text{ non arrêté}]$$
- $R_S = \{r_1, r_2\}$

$$r_1 : [P_n \square |^m \square \text{ non arrêté}] \xrightarrow[r_1]{} [P_n \square |^{m+1} \square \text{ non arrêté}]$$

si le programme P_n , pour la donnée n , n'a pas terminé à la fin de la $m + 1$ -ième étape (on suppose que le déroulement des programmes est découpé en étapes discrètes).

$$r_2 : [P_n \square |^m \square \text{ non arrêté}] \xrightarrow[r_2]{} [P_n \square \text{ arrêté}]$$

si le programme P_n pour la donnée n arrive à l'instruction arrêt à l'étape $(m + 1)$.

Ce système est spécialement conçu de manière à ce que :

$$[P_n \square |^m \square \text{ non arrêté}] \in T_S.$$

si et seulement si : le programme P_n pour la donnée n n'a pas terminé à l'étape m .

De plus :

$$[P_n \square \text{ arrêté}] \in T_S$$

si et seulement si : le programme P_n pour la donnée n finit par s'arrêter.

Donc, si l'ensemble des théorèmes de T_S était récursif, alors $\{n \mid P_n(n) \neq \perp\}$ serait aussi récursif, ce qui est faux (voir chapitre 2).

□

4. — ÉTUDE D'UN EXEMPLE

Reprendons le système formel défini au § 2 et montrons que :

$$T_S = \{1^n + 1^m = 1^{n+m} \mid n \in \mathbb{N} - \{0\}, m \in \mathbb{N} - \{0\}\}$$

- Toute formule de la forme $1^n + 1^m = 1^{n+m}$ est un théorème de S car :
 - c'est vrai si $n + m = 2$
 - si on suppose que c'est vrai pour tous les couples (n, m) avec $n + m < p$, alors en prenant un couple (n', m') tel que $n' + m' = p$, on a :

$$1^{n'-1} + 1^{m'} = 1^{p-1} \in T_S \text{ car } (n'-1) + m' < p$$

et donc, par application de r_1 :

$$1^{n'-1+1} + 1^m' = 1^{p-1+1} \in T_S,$$

c'est-à-dire :

$$1^{n'} + 1^m' = 1^p \in T_S.$$

- Toutes les formules intervenant dans une déduction sont de la forme $1^n + 1^m = 1^{n+m}$. En effet :
 - c'est vrai pour l'axiome,
 - si c'est vrai pour f et que $f \vdash_{r_1} g$
alors c'est vrai pour g ,
 - si c'est vrai pour f et que $f \vdash_{r_2} g$
alors c'est vrai pour g .

donc tous les théorèmes de S sont de la forme $1^n + 1^m = 1^{n+m}$.

Puisque nous avons réussi à caractériser tous les théorèmes de notre système formel S (ce qui est impossible dès qu'on traite de systèmes formels plus complexes) nous pouvons dire :

- le système formel S est cohérent (car par exemple la formule $1 + 1 = 111$ n'est pas un théorème)
- le système formel S est finiment-axiomatisable (dès le départ, on pouvait l'affirmer car S ne comporte qu'un seul axiome)
- T_S est récursif, c'est-à-dire le problème de la décision pour S admet une réponse positive. (Construire un programme qui pour la donnée de $f \in \Sigma^*$ indique au bout d'un temps fini si OUI ou NON, f est de la forme $1^n + 1^m = 1^{n+m}$ ne présente aucune difficulté)
- S est un système formel qui modélise correctement la notion d'addition entre deux entiers non nuls. Relativement à ce problème de modélisation il est donc correct et complet.

EXERCICES

Exercice sur l'interprétation d'un moteur d'inférences comme système formel.

Reprendre l'exemple utilisé au chapitre 1 pour décrire le chaînage avant et le chaînage arrière.

Décrire des systèmes formels associés à ce problème.

1^{re} méthode : Les faits donnent les axiomes ; la base de connaissances donne les règles d'inférence.

2^e méthode : Les faits et la base de connaissances donnent les axiomes ; la règle d'inférence est la règle de détachement (modus ponens).

3^e méthode : Les faits, la base de connaissances et la négation de la question donnent les axiomes ; la règle d'inférence est le modus tollens.

Pour chacune des trois méthodes, donner une déduction de H.

Exercice sur un système formel élémentaire

Soit le système formel S défini par :

$$\Sigma = \{a, b, c\} \text{ vocabulaire (ou alphabet)}$$

$$F = \{a^nbc^m \mid n, m \geq 0\} \text{ ensemble des formules bien formées}$$

$$A = \{a^{2i}bc^{2i} \mid i \geq 0\} \text{ ensemble des axiomes}$$

R : il y a une seule règle d'inférence décrite par :

$$r_1 : a^nbc^m, a^{n'}bc^{m'} \vdash a^{n+n'}bc^{m+m'}$$

- 1) Montrer que $a^6bc^2, a^{10}b$ sont des théorèmes.
- 2) Identifier l'ensemble des théorèmes T et montrer que chaque théorème peut être dérivé en au plus 3 étapes.
- 3) Montrer que si on enlève un seul axiome, alors T n'est plus le même.
- 4) Proposer un système formel (Σ', F', A', R') avec $\Sigma' = \Sigma$, $F' = F$ tel que A' contienne un seul élément et R' contienne deux règles et tel que $T = T'$.

Exercice sur le système formel MIU (D. HOFSTADTER)

Soit le système formel S défini par :

- $\Sigma = \{M, I, U\}$
- $F_S = \Sigma^*_S$ (c'est-à-dire que toute suite finie d'éléments de Σ_S est une formule bien formée)
- $A_S = \{MI\}$
- $R_S = \{r_1, r_2, r_3, r_4\}$

r_1 : à partir de f, on peut dériver fU :

$$f \vdash_{r_1} fU ,$$

$$r_2 : Mf \vdash_{r_2} Mff ,$$

$$r_3 : fIIIg \vdash_{r_3} fUg ,$$

$$r_4 : fUUg \vdash_{r_4} fg .$$

- (a) Montrer que $MUIU \in T_S$.
- (b) Montrer que si $t \in T_S$, alors t commence par M .
- (c) Est-ce que $MU \in T_S$?
- (d) Caractériser les théorèmes de S .

Exercice sur un système formel ayant une infinité d'axiomes

Soit le système formel S défini par :

- $\Sigma = \{=, +, 1, 2, \dots, n, \dots\}$
- $F = \{n_0 + n_1 + \dots + n_p = m_0 + m_1 + \dots + m_q \mid p \geq 0, q \geq 0, n_i, m_i \in \text{IN} - \{0\}\}$
- $A = \underbrace{1 + 1 + \dots + 1}_{p \text{ fois}} = p \mid p \in \text{IN} - \{0\}$
- $R = \{r_1\}$

$$r_1 : n_0 + \dots + n_p = m_0 + \dots + m_q, n'_0 + \dots + n'_{p'} = m'_0 + \dots + m'_{q'} \vdash r_1 \\ n_0 + \dots + n_p + n'_0 + \dots + n'_{p'} = m_0 + \dots + m_q + m'_0 + \dots + m'_{q'}$$

- 1) Trouver une dérivation pour la formule $1 + 1 + 1 + 1 = 2 + 2$.
Montrer que les formules $2 + 3 = 5, 1 + 1 + 1 + 1 = 1 + 2$ ne sont pas des théorèmes de S .
- 2) Trouver une règle d'inférence supplémentaire r_2 de façon à ce qu'on puisse dériver la formule $n + m = p$ quels que soient $n, m, p \in \text{IN} - \{0\}$ vérifiant « arithmétiquement » $n + m = p$.

Pouvez-vous dériver la formule :

$$n_1 + \dots + n_p = m_1 + \dots + m_q,$$

pour tout système d'entiers vérifiant « arithmétiquement » :

$$n_1 + \dots + n_p = m_1 + \dots + m_q ?$$

Pouvez-vous dériver d'autres formules ? (si OUI trouver une autre règle r_2).

- 3) Que se passe-t-il quand on enlève 1 axiome ?
- 4) Que se passe-t-il quand on ajoute aux axiomes :
 - (a) un théorème,
 - (b) la formule $1 + 1 = 1$?

Exercices sur la récursivité dans les systèmes formels

- 1) Montrer que dans tout système formel dont l'alphabet est fini, les ensembles suivants sont récursifs :
 - (a) $\{f \mid f \text{ est un théorème ayant une déduction de longueur inférieure à } k\}$, $k \in \mathbb{N}$, k fixé.
 - (b) $\{d \mid d \text{ est une déduction d'un théorème } t \text{ dont il n'existe aucune déduction plus courte}\}$.
- 2) Montrer qu'il existe des systèmes formels pour lesquels les ensembles suivants ne sont pas récursifs :
 - (a') $\{t \mid t \text{ est un théorème qui possède une déduction de longueur paire}\}$
 - (b') $\{f \mid f \text{ est une formule bien formée telle qu'aucune déduction contenant } f \text{ ne réutilise } f\}$.

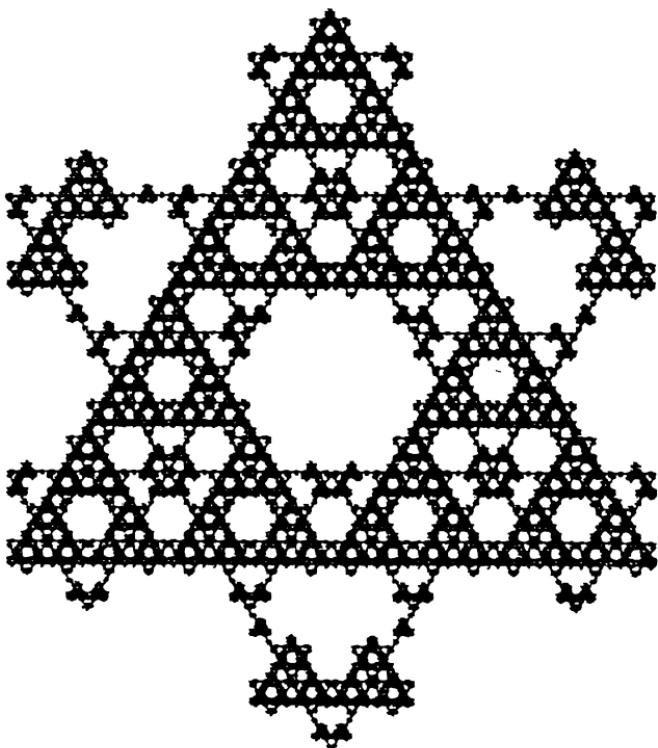
BIBLIOGRAPHIE

- [AZR 73] AZRA J.-P. et JAULIN B.
Récursivité.
Collection « Programmation », Gauthier-Villars, Paris, 1973.
{La deuxième partie du livre contient une étude assez complète des problèmes de décidabilité des théories mathématiques}
- [BOU 66] BOURBAKI N.
Eléments de Mathématique. Livre I. Théorie des Ensembles.
Hermann, Paris, 1966.
{On trouvera dans le chapitre 1 une présentation détaillée d'un système formel permettant d'exprimer toutes les mathématiques. C'est sur ce système formel que repose toute la suite du traité de Bourbaki}
- [BOU 69] BOURBAKI N.
Eléments d'Histoire des Mathématiques.
Collection Histoire de la Pensée.
Hermann, Paris, 1969.
{Ce livre qui regroupe les notes historiques du traité commence par un chapitre sur les fondements des mathématiques, qui parle en particulier, des systèmes formels et des résultats les plus importants obtenus à leur sujet}
- [COM 71] COMBES M.
Fondements des Mathématiques.
Presses Universitaires de France, Paris, 1971.
{Histoire et commentaires sur la « crise des mathématiques » du début du XX^e siècle, qui a conduit au développement des notions d'axiomatiques et de systèmes formels. Facile à lire et intéressant}
- [DIE 78] DIEUDONNÉ J. et Col.
Abrégé d'Histoire des Mathématiques.
Hermann 1978, Paris.
{Le chapitre « Axiomatique et Logique » de M. Guillaume est une très bonne présentation de l'histoire des fondements des Mathématiques}
- [HIL 71] HILBERT D.
Les fondements de la géométrie.
Edition critique préparée par Paul Rossier du célèbre ouvrage de Hilbert : *Grundlagen der geometrie*, 1899.
Dunod, Paris, 1971.
{Cette étude détaillée des axiomatiques de la géométrie révolutionna les fondements des mathématiques}
- [HOF 80] HOFSTADTER D. R.
Gödel, Escher, Bach : An Eternal Golden Braid.
Vintage Books Editions, New York, 1980. InterEditions, Paris, 1985.
{Le but principal de ce livre déjà cité aux chapitres 1 et 2 est l'étude et la réflexion non technique sur la puissance des systèmes formels. Il contient énormément de choses (des exemples simples, des considérations historiques, philosophiques, etc.) toujours présentées dans un langage clair et ne nécessitant pas de connaissances mathématiques approfondies}.

- [KLE 71] KLEENE S. C.
Logique Mathématique.
Librairie Armand Colin, Paris, 1971.
(Traduction de : Mathematical Logic, John Wiley and Sons, New York, 1967).
{Le paragraphe 37 introduit la notion de système formel, le paragraphe 38 donne une description complète d'un système formel de l'arithmétique, et le paragraphe 39 introduit d'autres exemples. De nombreux résultats sur ces systèmes formels se trouvent au chapitre 6}
- [MOS 53] MOSTOWSKI V. A., ROBINSON R. M. et TARSKI A.
Undecidable Theories.
North-Holland Publishing Company, Amsterdam, 1953.
{Ouvrage classique sur les problèmes de décidabilité des systèmes formels des théories mathématiques}
- [ROU 67] ROURE M. L.
Éléments de Logique Contemporaine.
Collection Sup. Presses Universitaires de France, Paris, 1967.
{En même temps que des considérations générales sur la logique, ce livre contient la description complète de divers systèmes formels avec des déductions détaillées. Facile à lire et intéressant}
- [SMU 61] SMULLYAN R. N.
Theory of Formal Systems.
Annals of Mathematics Studies n° 47, Princeton, 1961.
{Développement de la théorie de la récursivité à partir de la notion de système formel}
- [WAN 74] WANG H.
From Mathematics to Philosophy.
Routledge and Kegan Paul, Londres, 1974.
{Exposé des problèmes philosophiques liés aux mathématiques et en particulier de ceux qui ont conduit à l'introduction et à l'utilisation des systèmes formels}

CHAPITRE 4

LE CALCUL PROPOSITIONNEL



I. — INTRODUCTION

Le calcul propositionnel ou logique des propositions a pour objet l'étude des formes de raisonnement dont la validité est indépendante de la structure des propositions composantes et résulte uniquement de leurs propriétés d'être vraies ou fausses.

Comme en calcul des prédicts (que nous étudierons au chapitre suivant) deux aspects complémentaires doivent être pris en compte et liés l'un à l'autre :

- L'aspect *syntaxique* qui revient simplement à définir un système formel (au sens du chapitre 3) dans lequel les déductions qu'on peut faire conduisent à des théorèmes du calcul propositionnel.

Plusieurs méthodes sont possibles, nous avons choisi d'en exposer une qui a l'avantage de permettre une démonstration assez rapide du théorème de complétude (proposition 9), mais qui a l'inconvénient de limiter le langage aux connecteurs \neg et \rightarrow .

- L'aspect *sémantique* qui est l'interprétation des formules (et cette fois, nous accepterons les connecteurs v , \wedge , \iff) et qui consiste en l'analyse des « formules toujours vraies » appelées tautologies.

La liaison entre les deux aspects est la démonstration que les formules qui sont les tautologies (c'est-à-dire qui sont sémantiquement valables) sont les mêmes que les formules qui sont les théorèmes (c'est-à-dire qui sont syntaxiquement valables). Une conséquence de ce résultat de complétude est que le problème de la décision pour le calcul des propositions admet une réponse positive (corollaire 1 de la proposition 9).

2. — DÉFINITION DU CALCUL PROPOSITIONNEL P_0 ET EXEMPLES DE THÉORÈMES

On appelle calcul propositionnel P_0 le système formel défini par :

- $\Sigma_{P_0} = \{p_0, p_1, \dots, p_n, \dots\} \cup \{\neg, \rightarrow, (,)\}$

Les symboles p_i sont appelés variables propositionnelles ou propositions atomiques, ou atomes.

- $F_{P_0} =$ le plus petit ensemble de formules tel que :

— $\forall i : p_i \in F_{P_0}$

— $\forall A \in F_{P_0}, \forall B \in F_{P_0} : \neg A \in F_{P_0}, (A \rightarrow B) \in F_{P_0}$

- A_{P_o} = l'ensemble de toutes les formules de l'une des trois formes suivantes :

$$SA_1 : (A \rightarrow (B \rightarrow A))$$

$$SA_2 : ((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$$

$$SA_3 : (\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$$

avec $A, B, C \in F_{P_o}$.⁽¹⁾

les expressions SA_1, SA_2, SA_3 s'appellent des schémas d'axiomes, à chacune d'elles correspond une infinité d'axiomes. Par exemple, à SA_1 correspond :

$$(p_o \rightarrow (p_1 \rightarrow p_o)),$$

$$((p_2 \rightarrow p_o) \rightarrow ((p_1 \rightarrow p_o) \rightarrow (p_2 \rightarrow p_o))).$$

- $R_{P_o} = \{m.p.\}$; il y a une seule règle d'inférence appelée modus ponens :

$$\begin{array}{c} m.p. : A, (A \rightarrow B) \\ \hline m.p. \end{array}$$

Comme dans ce chapitre n'intervient que le système formel P_o à la place de A_1, A_2, \dots, A_n , nous écrirons :

$$\begin{array}{c} P_o \\ A_1, A_2, \dots, A_n \hline B. \end{array}$$

Exemple de déduction dans P_o :

$$f_1 : ((p_o \rightarrow ((p_1 \rightarrow p_o) \rightarrow p_o)) \rightarrow ((p_o \rightarrow (p_1 \rightarrow p_o)) \rightarrow (p_o \rightarrow p_o))) \quad (SA_2)$$

$$f_2 : (p_o \rightarrow ((p_1 \rightarrow p_o) \rightarrow p_o)) \quad (SA_1)$$

$$f_3 : ((p_o \rightarrow (p_1 \rightarrow p_o)) \rightarrow (p_o \rightarrow p_o)) \quad (m.p. f_1, f_2)$$

$$f_4 : (p_o \rightarrow (p_1 \rightarrow p_o)) \quad (SA_1)$$

$$f_5 : (p_o \rightarrow p_o) \quad (m.p. f_3, f_4)$$

Donc : $(p_o \rightarrow p_o)$ est un théorème de P_o :

$$(p_o \rightarrow p_o) \in T_{P_o} \text{ (ce que nous écrivons aussi } \vdash (p_o \rightarrow p_o)).$$

À la place de p_o et p_1 , on aurait pu prendre n'importe quelles formules bien formées A et B de P_o , donc :

Proposition 1. — Pour tout $A \in F_{P_o}$: $(A \rightarrow A) \in T_{P_o}$
(autrement dit : $\vdash (A \rightarrow A)$) .

Dans la suite du paragraphe, A, B, C, A_i, B_i, C_i désigneront des formules bien formées de P_o .

(1) Nous omettrons parfois la paire de parenthèses la plus extérieure d'une formule. Nous n'avons pas fait cet abus pour SA_1 et SA_2 , nous l'avons fait pour SA_3 .

Proposition 2. — Si : $A_1 A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B)$
alors : $A_1 A_2, \dots, A_{n-1}, A_n \vdash B$.

Démonstration :

Soit C_1, C_2, \dots, C_p une déduction de $(A_n \rightarrow B)$ à partir de A_1, A_2, \dots, A_{n-1} .
 Alors $C_1, C_2, \dots, C_p, A_n, B$ est une déduction de B à partir de A_1, A_2, \dots, A_n . En effet, la $(p+1)$ -ième formule de cette liste est la formule A_n qui est une hypothèse et la $(p+2)$ -ième formule résulte du m.p. à partir de la p -ième et de la $(p+1)$ -ième formule.

□

Moins évidente est la réciproque :

Proposition 3. (Appelée THÉORÈME DE DÉDUCTION)

Si : $A_1 A_2, \dots, A_{n-1}, A_n \vdash B$
alors : $A_1 A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B)$.

Démonstration :

Rappelons que $A_1 A_2, \dots, A_{n-1}, A_n \vdash B$ signifie qu'il existe une déduction de B à partir des hypothèses $A_1, A_2, \dots, A_{n-1}, A_n$. Soit k la longueur de cette déduction

- Si $k = 1$ alors trois cas sont possibles :

(a) *B est un axiome.* Alors :

$$\begin{aligned} f_1 &: B && (\text{axiome}) \\ f_2 &: (B \rightarrow (A_n \rightarrow B)) && (\text{SA}_1) \\ f_3 &: (A_n \rightarrow B) && (\text{m.p. } f_1, f_2) \end{aligned}$$

constitue une déduction de $(A_n \rightarrow B)$ et donc :

$$A_1 A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B)$$

(b) *B est l'une des hypothèses A_1, A_2, \dots , ou A_{n-1}*

Alors :

$$\begin{aligned} f_1 &: B && (\text{hypothèse}) \\ f_2 &: (B \rightarrow (A_n \rightarrow B)) && (\text{SA}_1) \\ f_3 &: (A_n \rightarrow B) && (\text{m.p. } f_1, f_2) \end{aligned}$$

constitue une déduction de $(A_n \rightarrow B)$ à partir de A_1, A_2, \dots, A_{n-1} et donc : $A_1 A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B)$

(c) B est l'hypothèse A_n . Alors, d'après la proposition 1, on sait que :

$\vdash (A_n \rightarrow B)$ et donc :

$$A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B).$$

- Supposons maintenant que la proposition 3 est vraie pour tout $k < k_0$ et montrons qu'elle est encore vraie pour k_0 (raisonnement par récurrence sur la longueur de la déduction).

Il y a maintenant quatre cas possibles :

- B est un axiome,
- B est l'une des hypothèses A_1, A_2, \dots, A_{n-1} ,
- B est l'hypothèse A_n ,
- B est obtenu par le m.p. à partir de $(C \rightarrow B)$ et C .

Les cas (a) (b) (c) se traitent comme pour $k = 1$.

Etudions le cas (d) :

$(C \rightarrow B)$ et C sont des formules de la déduction de B à partir des hypothèses A_1, A_2, \dots, A_n . Donc :

$$A_1, A_2, \dots, A_n \vdash (C \rightarrow B) \quad (\text{en moins de } k_0 \text{ étapes})$$

$$A_1, A_2, \dots, A_n \vdash C \quad (\text{en moins de } k_0 \text{ étapes})$$

L'hypothèse de récurrence donne que :

$$A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow (C \rightarrow B))$$

$$A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow C)$$

Grâce à SA₂, on peut écrire :

$$A_1, A_2, \dots, A_{n-1} \vdash ((A_n \rightarrow (C \rightarrow B)) \rightarrow ((A_n \rightarrow C) \rightarrow (A_n \rightarrow B)))$$

et donc, en appliquant le m.p. deux fois :

$$A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B)$$

□

Exemple d'utilisation de la proposition 3 :

Nous savons que :

$$(A_1, A_2, \dots, A_{n-1}, A_n \vdash B)$$

si et seulement si :

$$(A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B))$$

En particulier, pour établir que :

$$\vdash (A \rightarrow (B \rightarrow C)) \rightarrow (B \rightarrow (A \rightarrow C))$$

il faut et il suffit d'établir que :

$$(A \rightarrow (B \rightarrow C)) \vdash (B \rightarrow (A \rightarrow C))$$

et donc, il faut et il suffit d'établir que :

$$(A \rightarrow (B \rightarrow C)), B \vdash (A \rightarrow C)$$

et donc, il faut et il suffit d'établir que :

$$(A \rightarrow (B \rightarrow C)), B, A \vdash C$$

ce qui est immédiat par :

$$f_1 : A \rightarrow (B \rightarrow C) \quad (\text{hypothèse})$$

$$f_2 : B \quad (\text{hypothèse})$$

$$f_3 : A \quad (\text{hypothèse})$$

$$f_4 : B \rightarrow C \quad (\text{m.p. } f_1, f_3)$$

$$f_5 : C \quad (\text{m.p. } f_2 f_4)$$

L'utilisation du théorème de déduction évite d'avoir à écrire une déduction complète de la formule considérée, ce qui prendrait plusieurs dizaines de lignes.

Proposition 4. — Toutes les formules suivantes sont des théorèmes de P_o :

- 1) $((A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))) ,$
- 2) $(B \rightarrow ((B \rightarrow C) \rightarrow C)) ,$
- 3) $(\neg B \rightarrow (B \rightarrow C)) ,$
- 4) $(\neg \neg B \rightarrow B) ,$
- 5) $(B \rightarrow \neg \neg B) ,$
- 6) $((A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)) ,$
- 7) $(B \rightarrow (\neg C \rightarrow \neg (B \rightarrow C))) ,$
- 8) $((B \rightarrow A) \rightarrow ((\neg B \rightarrow A) \rightarrow A)) .$

Démonstration :

Laissée en exercice.

3. — LA NOTION D'INTERPRÉTATION POUR LES FORMULES DE P_1

Soit $\Sigma_{P_1} = \{p_0, p_1, \dots, p_n, \dots\} \cup \{\neg, \rightarrow, \vee, \wedge, \leftrightarrow, (,)\}$

Soit F_{P_1} le plus petit ensemble de formules tel que :

- $\forall i : p_i \in F_{P_1}$

- $\forall A \in F_{P_1}, \forall B \in F_{P_1} :$

$$\neg A \in F_{P_1}, (A \rightarrow B) \in F_{P_1}, (A \vee B) \in F_{P_1}, (A \wedge B) \in F_{P_1}$$

$$(A \iff B) \in F_{P_1}$$

Les symboles « \neg », « \rightarrow », « \vee », « \wedge », « \iff » se lisent respectivement : « non », « implique », « ou », « et », « équivaut à », et sont appelés des connecteurs.

On a bien sûr : $F_{P_0} \subset F_{P_1}$.

Dans toute la suite du paragraphe, les lettres A, B, C, A_i, B_i, C_i désigneront des formules de F_{P_1} .

On appelle *interprétation de F_{P_1}* (ou *valuation*, ou *réalisation*, ou *assiguation*) toute application :

$$i : \{p_0, p_1, \dots, p_n, \dots\} \rightarrow \{V, F\}$$

L'application i est alors étendue à tout F_{P_1} par les formules suivantes :

$$i(\neg A) = \neg [i(A)]$$

$$i((A \rightarrow B)) = \neg [i(A), i(B)]$$

$$i((A \vee B)) = V[i(A), i(B)]$$

$$i((A \wedge B)) = \wedge [i(A), i(B)]$$

$$i((A \iff B)) = \iff [i(A), i(B)]$$

où $\neg [.]$ (resp. $\rightarrow [., .]$, $\vee [., .]$, $\wedge [., .]$, $\iff [., .]$) sont les applications de $\{V, F\}$ dans $\{V, F\}$ (resp. de $\{V, F\} \times \{V, F\}$ dans $\{V, F\}$) définies par :

$$\neg[V] = F \quad \neg[F] = V$$

X, Y	$\rightarrow [X, Y]$	$\vee [X, Y]$	$\wedge [X, Y]$	$\iff [X, Y]$
V V	V	V	V	V
V F	F	V	F	F
F V	V	V	F	F
F F	V	F	F	V

Par exemple, si i est une interprétation de F_{P_1} telle que :

$i[p_0] = V, i[p_1] = F, i[p_2] = V$ alors :

$$\begin{aligned} i(((p_0 \rightarrow p_1) \vee \neg p_2)) &= \vee [i(p_0 \rightarrow p_1), i(\neg p_2)] \\ &= \vee [\neg [i(p_0), i(p_1)], \neg [i(p_2)]] \\ &= \vee [\neg [V, F], \neg [V]] \\ &= \vee [F, F] = F \end{aligned}$$

On appelle *tautologie* toute formule $A \in F_{P_1}$ telle que, pour toute interprétation i : $i[A] = V$.

On écrit alors : $\models A$.

On dit que la formule $B \in F_{P_1}$ est *conséquence de la formule* $A \in F_{P_1}$ si, à chaque fois que $i[A] = V$ alors $i[B] = V$. On écrit alors $A \models B$.

On dit que la formule $B \in F_{P_1}$ est *conséquence de l'ensemble des formules* $\mathcal{A} \subset F_{P_1}$ si, à chaque fois que $i[A] = V$ pour tout $A \in \mathcal{A}$ alors $i[B] = V$. On écrit alors $\mathcal{A} \models B$.

On dit que deux formules $A \in F_{P_1}$ et $B \in F_{P_1}$ sont *équivalentes* si $A \models B$ et $B \models A$.

On écrit alors : $A \equiv B$.

On dit qu'une formule $A \in F_{P_1}$ est *satisfiable* ou *consistante* s'il existe une interprétation i telle que : $i[A] = V$.

On dit que l'ensemble de formules $F \subset F_{P_1}$ est *satisfiable* ou *consistant* s'il existe une interprétation i telle que, pour tout $A \in F$, $i[A] = V$ (une telle interprétation i est appelée *modèle de* F).

On dit que deux ensembles de formules sont *équivalents* s'ils ont exactement les mêmes modèles.

On dit qu'une formule $A \in F_{P_1}$ est *insatisfiable* ou *inconsistante* si, pour toute interprétation i , on a : $i[A] = F$. On montre facilement que A est insatisfiable si et seulement si $\neg A$ est une tautologie.

On dit que l'ensemble de formules $F \subset F_{P_1}$ est *insatisfiable* ou *inconsistant* si, pour tout interprétation i , il existe $A \in F$ tel que $i[A] = F$ (autrement dit, s'il n'existe aucun modèle de F).

Ces notions de consistance et d'inconsistance sont liées à celles introduites au chapitre 3 par le résultat suivant :

L'ensemble de formules $F \subset F_{P_0}$ est consistant (au sens donné ici) si et seulement si le système formel S_F obtenu à partir de P_0 en ajoutant F à l'ensemble des axiomes est consistant (au sens du chapitre 3). (Ce résultat se démontre facilement à partir de ceux donnés au paragraphe 4).

Proposition 5. — Soient A et B deux formules de F_{P_1} :

- (a) $\models(A \rightarrow B)$ si et seulement si $A \models B$
- (b) $\models(A \leftrightarrow B)$ si et seulement si $A \equiv B$
- (c) Si $\models A$ et si $\models(A \rightarrow B)$ alors $\models B$
- (d) $\models(A \wedge B)$ si et seulement si $\models A$ et $\models B$
- (e) Si $\models A$ ou $\models B$ alors $\models A \vee B$

Démonstration :

Laissée en exercice.

Proposition 6. — Soit A une formule de F_{P_0} .

Si $\vdash A$ alors $\models A$.

Autrement dit, les formules de F_{P_0} qui sont des théorèmes (validité syntaxique) sont aussi des tautologies (validité sémantique).

Démonstration :

Soit $A \in F_{P_q}$ telle que $\vdash A$. Raisonnons par récurrence sur la longueur n de la déduction qui donne A.

Si $n = 1$, cela signifie que A est obtenu comme axiome. Comme tous les axiomes sont des tautologies (vérification immédiate), on en conclut que $\models A$. Supposons que le résultat soit vrai pour tous les théorèmes X ayant une déduction de longueur $< n$, et montrons que le résultat est vrai pour A.

La dernière étape de la déduction de longueur n qui donne A consiste :

- ou en une introduction d'axiome et alors $\models A$,
- ou en une utilisation du modus ponens à partir de deux formules $(B \rightarrow A)$ et (B) situées avant A dans la déduction de A. Ces deux formules sont donc des théorèmes qui admettent des déductions de longueur $< n$, donc sont des tautologies, donc, (d'après la proposition 5 (c)), on a : $\models A$.

□

Proposition 7. — Soit A une tautologie de F_{P_1} utilisant les variables propositionnelles p_0, p_1, \dots, p_n . Soient P_0, P_1, \dots, P_n des formules quelconques de F_{P_1} . La formule A' obtenue en remplaçant chaque p_i par P_i pour $i = 1, \dots, n$, est une tautologie.

Démonstration :

Soit i une interprétation quelconque de F_{P_1} . Pour calculer $i[A']$, on calcule d'abord $i[P_1], \dots, i[P_n]$, puis on reporte les valeurs trouvées dans A. Comme A est une tautologie, la valeur de $i[A']$ qu'on calcule alors ne dépend pas de ces valeurs et est toujours V. Donc pour toute interprétation i, on a $i[A'] = V$.

□

Proposition 8. — Soit A $\in F_{P_1}$. Il existe des formules B_1, B_2, B_3, B_4 et B_5 équivalentes à A et telles que :

- (a) B_1 n'utilise que les connecteurs \vee, \neg ;
- (b) B_2 n'utilise que les connecteurs \wedge, \neg ;
- (c) B_3 n'utilise que les connecteurs \rightarrow, \neg ;

- (d) B_4 n'utilise que les connecteurs \vee , \wedge , \neg et est de la forme $D_1 \wedge D_2 \wedge \dots \wedge D_m$ avec chaque D_i de la forme :
- $$p_{\ell_1} \vee p_{\ell_2} \vee \dots \vee p_{\ell_n} \vee \neg p_{q_1} \vee \neg p_{q_2} \vee \dots \vee \neg p_{q_r}$$
- $(B_4$ est appelé forme normale conjonctive de A) ;
- (e) B_5 n'utilise que les connecteurs \vee , \wedge , \neg et est de la forme $D_1 \vee D_2 \vee \dots \vee D_m$ avec chaque D_i de la forme
- $$p_{\ell_1} \wedge p_{\ell_2} \wedge \dots \wedge p_{\ell_n} \wedge \neg p_{q_1} \wedge \neg p_{q_2} \wedge \dots \wedge \neg p_{q_r}$$
- $(B_5$ est appelé forme normale disjonctive de A).

Démonstration :

- (a) L'utilisation répétée des formules :

$$(A \rightarrow B) \equiv (\neg A \vee B)$$

$$(A \wedge B) \equiv \neg(\neg A \vee \neg B)$$

$$(A \leftrightarrow B) \equiv ((A \rightarrow B) \wedge (B \rightarrow A)) \equiv ((A \wedge B) \vee (\neg A \wedge \neg B))$$

permet de transformer une formule quelconque en une formule équivalente n'utilisant que les connecteurs \vee et \neg .

- (b) (c) Même méthode.

- (d) On utilise en plus que :

$$\neg(A \vee B) \equiv (\neg A \wedge \neg B); \neg(A \wedge B) \equiv (\neg A \vee \neg B)$$

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C); A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$

$$\neg \neg A \equiv A$$

□

On appelle clause toute formule de la forme $p_{\ell_1} \vee p_{\ell_2} \vee \dots \vee p_{\ell_n} \vee \neg p_{q_1} \vee \neg p_{q_2} \vee \dots \vee \neg p_{q_r}$.

La partie (d) du théorème 8 permet donc d'énoncer que :

Tout ensemble de formules est équivalent à un ensemble de clauses.

Prenons par exemple la formule :

$$((p_0 \vee p_1) \rightarrow p_2) \wedge (p_0 \leftrightarrow p_3)$$

On obtient la suite de formules équivalentes suivantes :

$$(\neg(p_0 \vee p_1) \vee p_2) \wedge ((p_0 \rightarrow p_3) \wedge (p_3 \rightarrow p_0))$$

$$((\neg p_0 \wedge \neg p_1) \vee p_2) \wedge ((\neg p_0 \vee p_3) \wedge (\neg p_3 \vee p_0))$$

$$(\neg p_0 \vee p_2) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_0 \vee p_3) \wedge (\neg p_3 \vee p_0)$$

La formule de départ est donc équivalente à l'ensemble de clauses :

- $\neg p_0 \vee p_2$
- $\neg p_1 \vee p_2$
- $\neg p_0 \vee p_3$
- $\neg p_3 \vee p_0$

La mise sous forme de clauses est très importante et sera réutilisée au chapitre 6.

4. — RÉSULTATS

Proposition 9. — (Appelée théorème de complétude du calcul propositionnel P_o)

Pour toute formule $A \in F_{P_o}$:

si $\models A$ alors $\vdash A$.

Autrement dit : toutes les tautologies de F_{P_o} sont des théorèmes du système formel P_o .

Remarque :

Avec la proposition 6, on obtient donc que : $\models A$ si et seulement si $\vdash A$.

Ceci signifie simplement que le système formel défini au début du chapitre est une modélisation correcte pour le calcul des propositions. Il y a d'autres modélisations possibles (autres choix d'axiomes, autres choix de règles d'inférence) et à chaque fois la bonne adéquation du système proposé doit être démontrée.

Il faut bien comprendre que cette adéquation n'est pas du tout évidente. D'ailleurs, si on enlève un schéma d'axiomes au système formel P_o , on obtient d'autres systèmes formels pour lesquels il n'est plus vrai que toute tautologie est un théorème (la réciproque, par contre, reste vraie, car la démonstration de la proposition 6 est encore valable).

Lemme. — Soit $A \in F_{P_o}$ et soit $p_{j_1}, p_{j_2}, \dots, p_{j_k}$ les p_j apparaissant dans A . Soit i une interprétation de P_o .

Notons $B_\ell = p_{j_\ell}$ si $i[p_{j_\ell}] = V$

$B_\ell = \neg p_{j_\ell}$ si $i[p_{j_\ell}] = F$

Soit A' la formule définie par :

$A' = A$ si $i[A] = V$

$A' = \neg A$ si $i[A] = F$

Alors : $B_1, B_2, \dots, B_k \vdash A'$

(Illustration : $A = (p_1 \rightarrow p_2) \rightarrow p_3$, $i[p_1] = V$, $i[p_2] = F$,
 $i[p_3] = F$, $B_1 = p_1$, $B_2 = \neg p_2$, $B_3 = \neg p_3$;
le lemme affirme que :
 $p_1, \neg p_2, \neg p_3 \vdash (p_1 \rightarrow p_2) \rightarrow p_3$).

Démonstration du lemme :

Démontrons le lemme en raisonnant par récurrence sur le nombre de connecteurs de la formule A. Ce nombre n est appelé *complexité* de A.

- Si $n = 0$, alors $A = p_{i_1}$, deux cas sont possibles :
 - (a) $i[p_{i_1}] = V$ alors $B_1 = p_{i_1}$ et $A' = A$.
 Il est bien vrai que :
 $B_1 \vdash A'$
 - (b) $i[p_{i_1}] = F$ alors $B_1 = \neg p_{i_1}$ et $A' = \neg p_{i_1}$
 Il est bien vrai que :
 $B_1 \vdash A'$.
- Supposons le lemme vrai pour toute formule A de complexité $n < n_0$ et montrons que le lemme est vrai pour A de complexité n_0 .

Deux cas sont possibles :

 - (a) A est de la forme $\neg C$.
 Là encore deux cas sont possibles :
 - (a1) $i[A] = V$
 Alors $A' = A$, $i[C] = F$, $C' = \neg C = A = A'$.
 La formule C est de complexité $n_0 - 1$, donc :
 $B_1, B_2, \dots, B_k \vdash C'$
 Puisque ici $A = C'$:
 $B_1, B_2, \dots, B_k \vdash A'$
 - (a2) $i[A] = F$
 Alors $A' = \neg A$, $i[C] = V$, $C' = C$, et donc :
 $A' = \neg \neg C$. Par hypothèse :
 $B_1, B_2, \dots, B_k \vdash C$
 D'après la proposition 4 :
 $B_1, B_2, \dots, B_k \vdash (C \rightarrow \neg \neg C)$
 Donc, par application du modus ponens :
 $B_1, B_2, \dots, B_k \vdash \neg \neg C$
 c'est-à-dire :
 $B_1, B_2, \dots, B_k \vdash A'$.

(b) A est de la forme $(B \rightarrow C)$.

Là encore deux cas sont possibles :

(b1) $i[A] = F$

Alors $i[B] = V$ et $i[C] = F$ et donc :

$$C' = \neg C, B' = B, A' = \neg A = \neg(B \rightarrow C)$$

Par hypothèse :

$$B_1, B_2, \dots, B_k \vdash \neg C$$

$$B_1, B_2, \dots, B_k \vdash B$$

D'après la proposition 4 :

$$B_1, B_2, \dots, B_k \vdash (B \rightarrow (\neg C \rightarrow \neg(B \rightarrow C)))$$

et donc, en utilisant deux fois le modus ponens :

$$B_1, B_2, \dots, B_k \vdash A'$$

(b2) $i[A] = V$

Laissé en exercice.

Démonstration de la proposition 9 :

Soit A une tautologie, soit $p_{j_1}, p_{j_2}, \dots, p_{j_k}$ les p_j apparaissant dans A. Il y a 2^k interprétations différentes des $p_{j_1}, p_{j_2}, \dots, p_{j_k}$ et pour chacune d'elles, l'application du lemme nous donne une formule du type :

$$B_1, B_2, \dots, B_k \vdash A$$

En particulier, on a :

$$p_{j_1}, p_{j_2}, \dots, p_{j_{k-1}}, p_{j_k} \vdash A$$

$$p_{j_1}, p_{j_2}, \dots, p_{j_{k-1}}, \neg p_{j_k} \vdash A$$

qui correspondent aux interprétations :

$$i[p_{j_1}] = V, i[p_{j_2}] = V, \dots, i[p_{j_{k-1}}] = V, i[p_{j_k}] = V$$

et

$$i[p_{j_1}] = V, i[p_{j_2}] = V, \dots, i[p_{j_{k-1}}] = V, i[p_{j_k}] = F$$

On a donc (théorème de déduction) :

$$p_{j_1}, p_{j_2}, \dots, p_{j_{k-1}} \vdash (\neg p_{j_k} \rightarrow A)$$

$$p_{j_1}, p_{j_2}, \dots, p_{j_{k-1}} \vdash (p_{j_k} \rightarrow A)$$

D'après la proposition 4, on a aussi :

$$p_{j_1}, p_{j_2}, \dots, p_{j_{k-1}} \vdash ((p_{j_k} \rightarrow A) \rightarrow ((\neg p_{j_k} \rightarrow A) \rightarrow A))$$

Et donc, par application du modus ponens :

$$p_{j_1}, p_{j_2}, \dots, p_{j_{k-1}} \vdash A$$

De façon analogue, on trouve que :

- $p_{j_1}, p_{j_2}, \dots, p_{j_{k-2}} \vdash A$
-
- $p_{j_1} \vdash A$
- $\vdash A$.

C'est ce que nous cherchions à établir.

□

Corollaire 1 de la proposition 9

Le problème de la décision pour le calcul propositionnel P_o admet une réponse positive.

Autrement dit, T_{P_o} est récursif.

Autrement dit encore : il existe un programme qui, pour toute formule $A \in F_{P_o}$ indique (en un temps fini) si OUI ou NON : $\vdash A$

Démonstration :

Puisque $\vdash A$ est équivalent à $\models A$, il suffit de montrer qu'il existe un programme qui, pour toute formule $A \in F_{P_o}$ indique si OUI ou NON $\models A$. Voici la structure d'un tel programme :

- repérer quelles sont les variables propositionnelles $p_{i_1}, p_{i_2}, \dots, p_{i_n}$ qui interviennent dans A .
- pour chacune des 2^n interprétations possibles de $p_{i_1}, p_{i_2}, \dots, p_{i_n}$ calculer $i[A]$.
- Si, à chaque fois, on a obtenu $i[A] = V$, alors répondre OUI.
- Sinon, répondre NON.

□

Corollaire 2 de la proposition 9

Pour toutes formules B_1, B_2, \dots, B_n, A de F_{P_o} :

$B_1, B_2, \dots, B_n \models A$

si et seulement si

$B_1, B_2, \dots, B_n \vdash A$.

Démonstration :

Toutes les affirmations suivantes sont équivalentes :

- $B_1, B_2, \dots, B_{n-1}, B_n \models A$
- $B_1, B_2, \dots, B_{n-1} \models (B_n \rightarrow A)$ (utilisation des définitions)
-
- $\models (B_1 \rightarrow (B_2 \rightarrow \dots (B_n \rightarrow A) \dots))$ (utilisation des définitions)
- $\vdash (B_1 \rightarrow (B_2 \rightarrow \dots (B_n \rightarrow A) \dots))$ (proposition 9)
- $B_1 \vdash (B_2 \rightarrow (B_3 \rightarrow \dots (B_n \rightarrow A) \dots))$ (proposition 2)
-
- $B_1, B_2, \dots, B_n \vdash A$ (proposition 2)

□

Proposition 10. — (Théorème de compacité)

Soit F un ensemble de formules de F_{P_1} .

Si, pour toute famille finie $F' \subset F$ il existe une interprétation telle que :

$\forall A \in F' : i[A] = V$ alors il existe une interprétation i telle que :

$\forall A \in F : i[A] = V$

Démonstration :

Pour chaque formule $A \in F$, soit $I[A]$ l'ensemble des interprétations de F_{P_1} telles que $i[A] = V$. $I[A]$ est un ouvert de $\{V, F\}^{\{p_0, p_1, \dots, p_n, \dots\}}$ (muni de la topologie produit) car A ne fait intervenir qu'un nombre fini de p_i . C'est également un fermé car les interprétations qui ne satisfont pas A sont celles qui satisfont $\neg A$. L'hypothèse du théorème signifie que toute intersection finie de $I[A]$ pour $A \in F$ est non vide. Comme $\{V, F\}^{\{p_0, p_1, \dots, p_n, \dots\}}$ est compact, on en déduit que l'intersection de tous les $I[A]$ est non vide, c'est-à-dire qu'il existe une interprétation i telle que :

$$\forall A \in F : i[A] = V$$

□

Remarque :

Il existe des démonstrations de cette proposition ne faisant pas intervenir les notions de topologie utilisées.

Proposition 11. — (Théorème de finitude)

Soit F un ensemble de formules de F_{P_1} .

Soit $B \in F_{P_1}$.

Si : $F \models B$,

alors il existe $F' \subset F$, F' fini tel que :

$$F' \models B.$$

Démonstration :

Dire que $F \models B$ signifie qu'il n'existe pas d'interprétation i telle que :

$$\forall A \in F : i[A] = V \text{ et } i[\neg B] = V$$

D'après la proposition 10, appliquée à $F \cup \{\neg B\}$, il existe donc $G \subset F \cup \{\neg B\}$, G fini, tel qu'il n'existe pas d'interprétation i vérifiant :

$$\forall A \in G : i[A] = V$$

$G \cup \{\neg B\}$ peut s'écrire sous la forme :

$F' \cup \{\neg B\}$ avec $F' \subset F$ et F' fini.

Par construction, il n'existe pas d'interprétation i telle que :

$$\forall A \in F' : i[A] = V \text{ et } i[\neg B] = V$$

donc :

$$F' \models B.$$

□

Proposition 12

Soit $F \subset F_{P_0}$ et $A \in F_{P_0}$.

$F \models A$ si et seulement si $F \vdash A$.

(La notation $F \vdash A$ signifie, bien sûr : il existe une déduction de A utilisant des hypothèses prises dans l'ensemble des formules F).

Démonstration :

Toutes les affirmations suivantes sont équivalentes :

- $F \models A$
- il existe B_1, B_2, \dots, B_n dans F tels que $B_1, B_2, \dots, B_n \models A$ (proposition 11).
- il existe B_1, B_2, \dots, B_n dans F tels que $B_1, B_2, \dots, B_n \vdash A$ (corollaire de la proposition 9).
- $F \vdash A$
(définition de la notation $F \vdash A$).

EXERCICES

Exercice sur les propriétés du système formel P_o

Indiquer, en justifiant vos réponses, si le système formel P_o est :

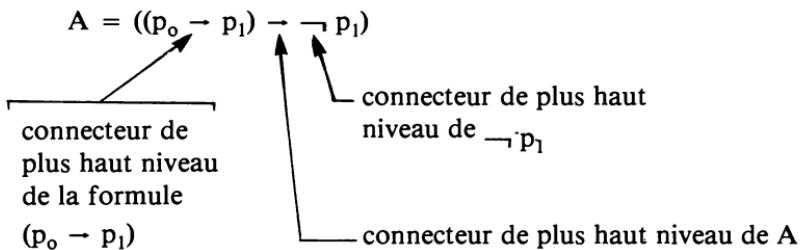
- (a) cohérent
- (b) consistant
- (c) catégorique
- (d) saturé
- (e) indépendant (les axiomes sont-ils indépendants ?)
- (f) finiment axiomatisable.

Exercice sur les tables de vérité

On appelle table de vérité d'une formule $A \in F_{P_1}$ le tableau obtenu de la façon suivante :

- sur la première ligne, on écrit A ,
- on considère les variables propositionnelles $p_{j_1}, p_{j_2}, \dots, p_{j_n}$ de A ,
- on réserve une ligne pour chacune des 2^n interprétations possibles de $p_{j_1}, p_{j_2}, \dots, p_{j_n}$,
- sur la ligne n° k , correspondant à la k -ième interprétation possible,
 - on écrit $i[p_{j_1}]$ sous chaque $p_{j_1}, \dots, i[p_{j_n}]$ sous chaque p_{j_n} .
 - on écrit $i[B_c]$ sous chaque connecteur c de A où B_c désigne la sous-formule de A dont c est le connecteur de plus haut niveau.

Exemple :



$$((p_0 \rightarrow p_1) \rightarrow \neg p_1)$$

V	V	V	<u>F</u>	F	V	← ligne pour l'interprétation i[p ₀] = V i[p ₁] = V
V	F	F	<u>V</u>	V	F	
F	V	V	<u>F</u>	F	V	
F	V	F	<u>V</u>	V	F	

↑ ↑ ↑

colonne donnant $i[\neg p_1]$

colonne donnant $i[A]$

colonne donnant $i[(p_0 \rightarrow p_1)]$

La colonne donnant $i[A]$ sera appelée colonne principale de la table de vérité de A.

- 1) Interpréter les notions de « tautologies », « conséquences », « formules équivalentes », « formules satisfiables », « formules insatisfiables » en utilisant la notion de « colonne principale d'une table de vérité ».
- 2) Faire les tables de vérité des schémas d'axiomes.
- 3) Démontrer la proposition 5.

Exercice sur la consistance

- 1) Soit A une formule quelconque de F_{p_0} . Considérons le système formel obtenu à partir de P_0 en ajoutant les deux axiomes A, $\neg A$. Montrer que pour ce système formel l'ensemble des théorèmes T est F_{p_0} (Utiliser la formule 3° de la proposition 4).
- 2) En déduire que tout système formel obtenu en ajoutant des axiomes à P_0 et qui n'est pas consistant, n'est pas non plus cohérent (voir les définitions de consistant et cohérent au chapitre 3).

Exercice concernant la proposition 8

- 1) Soit la formule

$$A = ((p_0 \rightarrow p_1) \rightarrow (p_0 \rightarrow p_2))$$

Trouver les formules B_1, B_2, B_3, B_4 et B_5 équivalentes à A et dont la proposition 8 énonce l'existence.

2) Même question avec :

$$A = ((\neg(p_0 \leftrightarrow p_1) \vee (p_1 \wedge p_2)) \rightarrow p_0)$$

Exercice sur un calcul propositionnel à un seul connecteur

On introduit un connecteur nouveau * dont la table est définie par :

*	F	V	V
F	F	V	
V	F	V	
V	V	F	

Montrer que toute formule de F_{P_1} est équivalente à une formule ne contenant que le connecteur *.

Exercice sur les problèmes de substitution

- 1) Donner un exemple de formule $F(p_0, p_1, \dots, p_n)$ qui ne soit pas une tautologie et qui soit telle qu'on puisse trouver des formules P_0, P_1, \dots, P_n faisant de $F(P_0, P_1, \dots, P_n)$ une tautologie.
- 2) Indiquer une condition nécessaire et suffisante sur F pour que ce qui est demandé en 1 soit possible.

Exercice sur la négation en calcul propositionnel

Soit $A \in F_{P_1}$ une formule n'utilisant que les connecteurs \neg, \vee, \wedge .

On considère la formule A' obtenue à partir de A

- en remplaçant chaque p_i par $\neg p_i$
- en remplaçant chaque \vee par \wedge et chaque \wedge par \vee
- en supprimant tous les $\neg\neg$.

Par exemple : $A = ((\neg p_0 \vee p_1) \wedge \neg p_2)$

devient : $A' = ((p_0 \wedge \neg p_1) \vee p_2)$

Montrer que A' est équivalente à $\neg A$.

Exercice sur la complétude : un système avec trop de théorèmes.

On considère le système formel obtenu en ajoutant un quatrième schéma

d'axiomes au système formel P_o :

$$SA_4 \ ((B \rightarrow (A \rightarrow B)) \rightarrow A).$$

- 1) Les axiomes donnés par ce nouveau schéma sont-ils tous des tautologies ?
- 2) Caractériser les théorèmes du nouveau système formel ainsi défini.
- 3) Quelle est la réponse au problème de la décision pour ce système formel.

Exercice sur la complétude : un système manquant de théorèmes

On considère le système formel obtenu en remplaçant dans le système formel P_o les trois schémas d'axiomes par un unique schéma SA'_1 :

$$SA'_1 \ ((A \rightarrow B) \rightarrow (A \rightarrow A))$$

On notera T' l'ensemble des théorèmes de ce système formel.

- 1) Est-ce que tout théorème $t \in T'$ est une tautologie ?
- 2) Montrer que :
$$T' = \{((A \rightarrow B) \rightarrow (A \rightarrow A)) \mid A, B \in F_{P_o}\}$$
$$\cup \{((A \rightarrow B) \rightarrow (A \rightarrow B)) \mid A, B \in F_{P_o}\}.$$
- 3) Est-ce que toute tautologie de F_{P_o} est dans T' ?
- 4) Le théorème de déduction est-il valable pour ce système formel ?
- 5) Quelle est la réponse au problème de la décision pour ce système formel ?

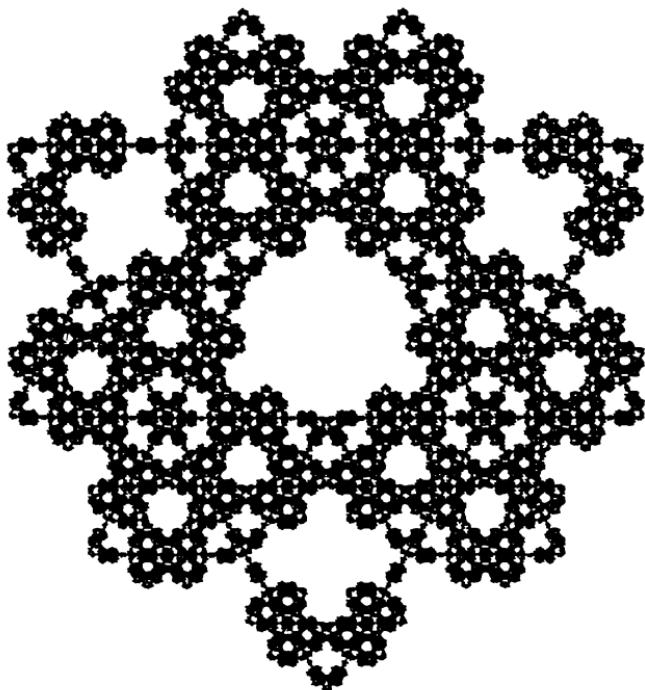
BIBLIOGRAPHIE

- [BOU 66] BOURBAKI N.
Eléments de Mathématique. Livre I. Théorie des Ensembles.
Hermann, Paris, 1966.
{Le paragraphe 3 du chapitre 1, présente une description complète d'un système formel pour le calcul propositionnel, différent de celui présenté ici}
- [CAL 80] CALAIS M.
Cours de Logique.
Polycopié, Université des Sciences et Techniques de Lille.
U.E.R. de Mathématique, 1980.
{Le chapitre 3 contient une étude des problèmes de validité sémantique, plus générale que celle du cours. En particulier plus de connecteurs sont envisagés. Des résultats plus précis de décomposition sont indiqués}
- [FRA 71] FRAISSE R.
Cours de logique Mathématique.
Gauthier-Villars, Paris, 1971.
{On trouvera au chapitre 2 du tome 1, la présentation d'une dizaine de systèmes formels pour le calcul des propositions et les théorèmes de complétude correspondants}
- [HOF 80] HOFSTADTER D. R.:
Gödel, Escher, Bach : An Eternal Golden Braid.
Vintage Books Editions, New York, 1980. InterEditions, Paris, 1985.
{Le calcul propositionnel présenté au chapitre 7 est très intéressant car il permet d'écrire des déductions de manière assez naturelle}
- [KAI 83] KAISER D.
Examen des diverses méthodes utilisées en représentation des connaissances.
Laboratoire de Recherche en Informatique, Université de Paris Sud, 91405 Orsay, 1983.
{Discussion et bibliographie de base sur : les logiques multivaluées, les logiques modales, les logiques non monotones}
- [KLE 71] KLEENE S. C.
Logique Mathématique.
Librairie Armand Colin, Paris, 1971.
(Traduction de : Mathematical Logic, John Wiley and Sons, New York, 1967).
{Présentation très détaillée et très commentée du calcul propositionnel avec un système formel tenant compte de tous les connecteurs classiques : \neg , \rightarrow , \vee , \wedge , \iff , $\{$ }
- [KRE 67] KREISEL G. et KRIVINE J.-L.
Eléments de Logique Mathématique : Théorie des Modèles.
Dunod, Paris, 1967.
{Débute par une formulation concise et élégante de la sémantique du calcul propositionnel, à laquelle nous avons emprunté la démonstration de notre proposition 10}

- [LAR 72] LARGEAULT J.
Logique Mathématique : Textes.
Librairie Armand Colin, Paris, 1972.
{Parmi les textes présentés, on notera celui de E. Post qui introduit les systèmes de vérité à m valeurs}
- [ROU 67] ROURE M.-L.
Eléments de Logique Contemporaine.
Collection Sup. Presses Universitaires de France, Paris, 1967.
{Le chapitre 2, en plus de la sémantique du calcul propositionnel et d'un système formel pour ce calcul, décrit le système connu sous le nom de « Méthode de Déduction Naturelle » dû à Gentzen}
- [TAR 71] TARSKI A.
Introduction à la logique.
Gauthier-Villars, Paris, 1971.
(Traduction de : Introduction to logic and to methodology of deductive sciences, 1941).
{Livre assez élémentaire, dont le chapitre 2 discute (entre autres choses) longuement de la justification des tables de vérités des connecteurs, qui posent parfois des problèmes à ceux qui les abordent pour la première fois}
- [TAR 72] TARSKI A.
Logique Sémantique ; Métamathématique.
Librairie Armand Colin, Paris, 1972.
{Traductions d'articles dont en particulier le fameux article de Lukasiewicz et Tarski sur les calculs propositionnels à n valeurs et à une infinité de valeurs}
- [YAS 71] YASUHARA A.
Recursive Function Theory and Logic.
Academic Press, New York, 1971.
{Nous avons repris le système formel et la démonstration du théorème de complétude du chapitre IX de ce livre}.

CHAPITRE 5

LE CALCUL DES PRÉDICATS DU PREMIER ORDRE



1. — INTRODUCTION

Dès qu'on veut manipuler des propriétés générales un peu compliquées et des relations entre objets, on est conduit à utiliser des énoncés dont la vérité dépend de variables, comme par exemple : « x donne y à z ». De tels énoncés s'appellent des prédictats et leur théorie, qui généralise le calcul propositionnel, s'appelle le calcul des prédictats.

Ce calcul a été introduit par les mathématiciens pour répondre à leurs propres besoins, et la preuve de son immense pouvoir d'expression est qu'il leur permet effectivement de représenter (par exemple par le biais d'une axiomatisation au premier ordre de la théorie des ensembles) tous les objets et notions dont ils se servent.

C'est à cause de son aptitude très générale pour la représentation et la manipulation des connaissances que le calcul des prédictats a intéressé les informatiens. En particulier de nombreux systèmes experts l'utilisent ainsi que plusieurs langages de l'Intelligence Artificielle comme Prolog ou Snark.

Nous avons choisi de présenter sous une forme assez générale le calcul des prédictats du premier ordre : utilisation de symboles de prédictats et de symboles fonctionnels d'arité quelconque. Après en avoir fixé la syntaxe d'une manière assez stricte (ce qui nous sera utile aux chapitres 6 et 7) nous procérons comme pour le calcul propositionnel : définition complète du système formel Pr (paragraphe 3), présentation de la sémantique (paragraphe 4), énoncé des liens entre les deux aspects (paragraphe 5). Les deux derniers paragraphes (6 et 7) montrent comment, à partir du système formel Pr, on peut introduire d'autres systèmes formels et énoncent quelques résultats à propos des problèmes de décision pour ces systèmes formels.

2. — TERMES, ATOMES, FORMULES, VARIABLES LIBRES ET LIÉES

(a) Eléments de base.

On se donne une fois pour toutes des ensembles dénombrables disjoints :

$$V ; C ; F_j, j \in \text{IN} - \{0\} ; P_j, j \in \text{IN}.$$

Les éléments de V sont appelés *variables* et sont toujours notés par les lettres x, y, z, u, v, w , éventuellement munies d'indices.

Les éléments de C sont appelés *constantes* et sont toujours notés par les lettres a, b, c, d, e , éventuellement munies d'indices.

Les éléments de $F_j, j \in \text{IN} - \{0\}$, sont appelés *symboles fonctionnels j-aires*, et sont notés par les lettres f, g, h , éventuellement munies d'indices.

Les éléments de P_j , $j \in \mathbb{N}$, sont appelés *symboles relationnels j-aires* (ou *symboles de prédictats j-aires*) et sont notés par les lettres p, q, r, s , éventuellement munies d'indices.

(b) Les termes

On appelle ensemble des *termes du calcul des prédictats de 1^{er} ordre* et on note « Terme » le plus petit ensemble de formules défini sur l'alphabet :

$$V \cup C \cup (\cup F_j) \cup \{\{\} \cup \{\}\} \cup \{\{\}$$

et tel que :

- $V \subset \text{Terme}$,
- $C \subset \text{Terme}$,
- si $t_1, t_2, \dots, t_j \in \text{Terme}$, si $f \in F_j$ alors $f(t_1, t_2, \dots, t_j) \in \text{Terme}$.

Exemple de termes :

$$x ; a ; f(a) ; g(a, x, f(x, a)) ; f(f(f(x_1, x_2), x_3), x_4).$$

Plus concrètement si on a un symbole de constante, a , représentant un personnage nommé Alain et si on a un symbole fonctionnel, f , s'interprétant comme « père de » alors :

$$f(a)$$

représente le personnage qui est le père de Alain et

$$f(f(a))$$

représente le grand-père paternel de Alain.

La lettre t , éventuellement munie d'indices, sera réservée pour désigner des termes.

(c) Les atomes

On appelle ensemble des *atomes (ou formules atomiques) du calcul des prédictats du premier ordre* et on note « Atome », le plus petit ensemble de formules défini sur l'alphabet :

$$V \cup C \cup (\cup F_j) \cup (\cup P_j) \cup \{\{\} \cup \{\}\} \cup \{\{\}$$

et tel que :

- si $t_1, t_2, \dots, t_j \in \text{Terme}$, si $p \in P_j$, alors $p(t_1, t_2, \dots, t_j) \in \text{Atome}$.

Exemples d'atomes :

$$p, q(f(x, y)), r(x, a, f(x, g(c))).$$

Plus concrètement, en poursuivant l'exemple commencé plus haut, si p désigne le prédictat unaire « est père » alors :

$$p(a)$$

représervera l'affirmation « Alain est père ».

Si p' désigne le prédicat binaire « est père de » alors :

$$p'(a, b)$$

représervera l'affirmation « Alain est le père de b » (b désignant par exemple le personnage Bernard).

Bien noter les trois utilisations différentes du mot « père » pour définir une fonction, un prédicat unaire, un prédicat binaire.

(d) Les formules

On appelle ensemble des *formules du calcul des prédictats du premier ordre* et on note F_{Pr} le plus petit ensemble de formules défini sur l'alphabet :

$$\Sigma_{Pr} = V \cup C \cup (\cup F_j) \cup (\cup P_j) \cup \{\{\} \cup \{\}\} \cup \{,\} \cup \{\neg, \rightarrow, \forall\}$$

et tel que :

- Atome $\subset F_{Pr}$
- si $A \in F_{Pr}$, $B \in F_{Pr}$, $x \in V$ alors :
 $\neg A \in F_{Pr}$, $(A \rightarrow B) \in F_{Pr}$, $\forall x A \in F_{Pr}$.

Exemples de formules :

$$p ; (p_1 \rightarrow p_2) ; (\neg q(x, f(x)) \rightarrow r(x)) ; \forall x \neg r(x) ;$$

$$\forall x \forall y (p(x) \rightarrow (r(y) \rightarrow \forall z s(x, y, z))) .$$

En reprenant encore l'exemple introduit plus haut, on a les formules suivantes :

$$\forall x p'(f(x), x)$$

qui signifie « pour tout x : le père de x est père de x »

$$(p'(a, b) \rightarrow p'(a, c))$$

qui signifie « si Alain est père de Bernard alors Alain est père de Christian » (affirmation qui par exemple, est conséquence de ce que Bernard et Christian sont frères).

Les lettres majuscules A, B, C éventuellement munies d'indices seront réservées pour désigner des formules.

(e) Numérotation de Σ_{Pr}

L'alphabet Σ_{Pr} qui d'après les hypothèses est dénombrable, est muni d'une numérotation fixée une fois pour toutes. On suppose que cette numérotation est telle que l'ensemble des numéros de chacun des ensembles suivants est récursif :

$$V ; C ; F_j, j \in \text{IN} - \{0\} ; P_j, j \in \text{IN}.$$

(f) Extension du vocabulaire

Pour écrire des formules de F_{Pr} , nous utiliserons parfois les symboles supplémentaires suivants :

\exists , \iff , v , \wedge . Il faut considérer ces symboles comme de simples raccourcis d'écriture :

- | | |
|----------------|---|
| $\exists x A$ | représente simplement : $\neg \forall x \neg A$ |
| $(A \vee B)$ | représente simplement : $(\neg A \rightarrow B)$ |
| $(A \wedge B)$ | représente simplement : $\neg (\neg A \rightarrow \neg B)$ |
| $(A \iff B)$ | représente simplement : $\neg ((A \rightarrow B) \rightarrow \neg (B \rightarrow A))$ |

Une petite analyse de ces formules montre qu'elles donnent bien au vocabulaire supplémentaire introduit son sens habituel. Par exemple, dire qu'il existe un x vérifiant la propriété A signifie exactement qu'il n'est pas vrai que tout x a la propriété non A .

(g) Possibilité de se ramener à un alphabet fini.

Il est possible de n'utiliser qu'un alphabet fini, en procédant de la manière suivante :

A la place de $V \cup C \cup (\cup F_j) \cup (\cup P_i)$ on prend l'ensemble fini de symboles : { v , c , f , p , I , $+$ } et on convient que :

les variables sont : v , vI , vII , $vIII$, etc.,

les constantes sont : c , cI , cII , $cIII$, etc.,

les symboles fonctionnels j -aires sont :

$\underbrace{f + + \dots +}_{j \text{ fois}}$, $\underbrace{f + + \dots + I}_{j \text{ fois}}$, $\underbrace{f + + \dots + II}_{j \text{ fois}}$, etc.

les symboles de prédicats j -aires sont :

$\underbrace{p + + \dots +}_{j \text{ fois}}$, $\underbrace{p + + \dots + I}_{j \text{ fois}}$, $\underbrace{p + + \dots + II}_{j \text{ fois}}$, etc.

Cette méthode (ou n'importe quelle autre méthode équivalente) rend possible le traitement informatique complet du calcul des prédicats. En particulier l'implémentation effective des algorithmes envisagés dans le texte est réalisable non pas seulement pour des morceaux du calcul des prédicats mais pour le calcul des prédicats dans toute sa généralité.

Remarquons encore que si on procède comme nous le suggérons, les conditions de récursivité envisagées plus haut pour la numérotation de Σ_{Pr} sont vérifiées en numérotant Σ_{Pr} par « longueur croissante, et ordre alphabétique si longueur égale ».

(h) Calculs d'ordres supérieurs

Dans le calcul des prédictats du premier ordre, il y a un seul type d'objets, et les quantifications portent toujours sur ces objets. Il est possible d'imaginer des *calculs de prédictats à plusieurs types d'objets*, ou des *calculs de prédictats d'ordre > 1* dans lesquels des quantifications peuvent être faites sur les objets et les prédictats comme par exemple dans la formule :

$$\forall p \exists q \forall x : p(x,x) \longleftrightarrow q(x)$$

Ce genre de calculs des prédictats sort de notre étude.

(i) Variables liées et variables libres

L'ensemble des variables d'une formule A est l'ensemble des éléments de V qui apparaissent dans A, on notera : var(A) cet ensemble.

L'ensemble des *variables liées* d'une formule A est noté varliée(A) et est défini par récurrence par :

- Si A ∈ Atome : varliée(A) = \emptyset
- Si A est de la forme (B → C) : varliée(A) = varliée(B) ∪ varliée(C)
- Si A est de la forme $\neg B$: varliée(A) = varliée(B)
- Si A est de la forme $\forall x B$: varliée(A) = varliée(B) ∪ {x}.

L'ensemble des *variables libres* d'une formule A est noté varlib(A) et est défini par récurrence par :

- Si A ∈ Atome : varlib(A) = var(A)
- Si A est de la forme (B → C) : varlib(A) = varlib(B) ∪ varlib(C)
- Si A est de la forme $\neg B$: varlib(A) = varlib(B)
- Si A est de la forme $\forall x B$: varlib(A) = varlib(B) - {x}.

Une formule sans variable libre est dite : *close* ou *fermée*.

Exemples :

$$A = (p(f(x,y)) \vee \forall z r(a,z))$$

$$\text{var}(A) = \{x, y, z\}$$

$$\text{varliée}(A) = \{z\}$$

$$\text{varlib}(A) = \{x, y\}$$

$$B = (\forall x p(x,y,z) \vee \forall z (p(z) \rightarrow r(z)))$$

$$\text{var}(B) = \{x, y, z\}$$

$$\text{varliée}(B) = \{x, z\}$$

$$\text{varlib}(B) = \{y, z\}$$

Remarquer que z est à la fois libre et liée dans B.

La formule :

$$\forall x \exists y (p(x,y) \rightarrow \forall z r(x,y,z))$$

est une formule close.

(j) Renommage et substitution

Soit $A(x)$ une formule contenant x comme variable libre ; soit t un terme. On notera $A(t)$ toute formule obtenue en remplaçant chaque x par t dans la formule $A(x)$, et ceci après avoir changé dans A les noms des variables liées de telle manière que x ne soit plus variable liée (si x l'était) et que plus aucune variable de t ne soit liée (s'il y en avait).

Lorsque $x \notin \text{varlib}(A)$ on convient que : $A = A(x) = A(t)$.

Exemple :

$$A(x) = (p(x) \vee \forall y \exists x r(x,y))$$

$$t = f(y,u)$$

Pour obtenir $A(t)$, on change d'abord le nom des variables liées y et x ; on obtient :

$$(p(x) \vee \forall z_1 \exists z_2 r(z_2,z_1))$$

puis on effectue la substitution, ce qui donne :

$$A(t) = (p(f(y,u)) \vee \forall z_1 \exists z_2 r(z_2,z_1)).$$

3. — LE SYSTÈME FORMEL Pr DU CALCUL DES PRÉDICATS DU PREMIER ORDRE

On appelle calcul des prédictats du premier ordre le système formel Pr défini par :

- Σ_{Pr} , l'alphabet décrit au paragraphe 2
- F_{Pr} , l'ensemble des formules décrit au paragraphe 2
- A_{Pr} , l'ensemble des formules F_{Pr} de l'une des formes suivantes :

$$SA_1: (A \rightarrow (B \rightarrow A))$$

$$SA_2: ((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$$

$$SA_3: ((\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A))$$

$$SA_4: (\forall x A(x) \rightarrow A(t))$$

$$SA_5: ((D \rightarrow B) \rightarrow (D \rightarrow \forall x B))$$

où A, B, C sont des formules quelconques de F_{Pr} ,

x une variable,

t un terme,

D une formule n'ayant pas x pour variable libre.

- $R_{P_r} = \{m.p., g.\}$
 m.p. : $A, A \rightarrow B \vdash B$ (modus ponens)
 m.p.
 $g. : A \vdash \forall x A$ (généralisation)
- pour toute formule A, B de F_{P_r} et pour toute variable x .
 Pour indiquer qu'il existe une déduction de B à partir des hypothèses A_1, A_2, \dots, A_n nous écrirons :
- $$A_1, A_2, \dots, A_n \vdash_{Pr} B \text{ ou } A_1, A_2, \dots, A_n \vdash B$$

Exemple de déduction :

Montrons que :

$$\forall x \forall y p(x,y) \vdash \forall z p(z,z)$$

$$f_1 : \forall x \forall y p(x,y) \quad (\text{hypothèse})$$

$$f_2 : (\forall x \forall y p(x,y) \rightarrow \forall y p(z,y)) \quad (SA_4)$$

$$f_3 : \forall y p(z,y) \quad (\text{m.p. avec } f_1, f_2)$$

$$f_4 : (\forall y p(z,y) \rightarrow p(z,z)) \quad (SA_4)$$

$$f_5 : p(z,z) \quad (\text{m.p. avec } f_3, f_4)$$

$$f_6 : \forall z p(z,z) \quad (g.)$$

L'ensemble des théorèmes de Pr (c'est-à-dire l'ensemble des formules qu'on peut obtenir avec une déduction sans hypothèse) est noté T_{P_r} .

Proposition 1. — Pour tout $A \in F_{P_r}$: $(A \rightarrow A) \in T_{P_r}$
 · (autrement dit : $\vdash (A \rightarrow A)$).

Proposition 2. — Si $A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B)$
 alors $A_1, A_2, \dots, A_{n-1}, A_n \vdash B$.

Proposition 3. — (appelée théorème de déduction)
 Soient A_1, A_2, \dots, A_n des formules closes.

Si $A_1, A_2, \dots, A_{n-1}, A_n \vdash B$
 alors $A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow B)$

Démonstrations :

Pour les propositions 1 et 2, il suffit de copier les démonstrations du chapitre 1. Pour la proposition 3, tout est comme au chapitre 3 sauf qu'en plus des cas (a) (b) (c) (d), il y a maintenant un cas (e).

(e) B est obtenue par application de g. sur une formule

$$B = \forall x C .$$

Par hypothèse :

$$A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow C) .$$

D'après SA₅ :

$$A_1, A_2, \dots, A_{n-1} \vdash ((A_n \rightarrow C) \rightarrow (A_n \rightarrow \forall x C))$$

donc, en appliquant le modus ponens :

$$A_1, A_2, \dots, A_{n-1} \vdash (A_n \rightarrow \forall x C) .$$

□

Soit C' ⊂ C ; F'_j ⊂ F_j, j ∈ IN - {0} ; P'_j ⊂ P_j, j ∈ IN .

On pose :

$$\Sigma_{P_r'} = V \cup C' \cup (\cup F'_j) \cup (\cup P'_j) \cup \{\{\} \cup \{\}\} \cup \{\{\} \cup \{\}\} \cup \{\neg, \rightarrow, \forall\}$$

Exactement comme on vient de le faire pour Terme, Atome, F_{P_r}, P_r, on définit :

Terme' : l'ensemble des termes sur l'alphabet Σ_{P_r'},

Atome' : l'ensemble des atomes sur l'alphabet Σ_{P_r'},

F_{P_r'} : l'ensemble des formules sur l'alphabet Σ_{P_r'},

P' : le système formel associé à l'alphabet Σ_{P_r'}.

Exemple : C' = C ; F'_j = ∅, j ∈ IN - {0} ; P'_j = P_j, donnent ce qu'on appelle le calcul des prédicats du premier ordre sans symbole de fonction.

4. — INTERPRÉTATIONS

(a) Définitions

Soit C' ⊂ C ; F'_j ⊂ F_j, j ∈ IN - {0} ; P'_j ⊂ P_j, j ∈ IN .

On appelle interprétation (ou réalisation) de

$$\Sigma_{P_r'} = V \cup C' \cup (\cup F'_j) \cup (\cup P'_j) \cup \{\{\} \cup \{\}\} \cup \{\{\} \cup \{\}\} \cup \{\neg, \rightarrow, \forall\}$$

la donnée :

- d'un ensemble non vide E, appelé ensemble de base de l'interprétation,
- d'un élément $\bar{c} \in E$ pour tout $c \in C'$

- d'une application $\bar{f} : E^j \rightarrow E$ pour tout $f \in F'_j$
- d'une application $\bar{p} : E^j \rightarrow \{V, F\}$ pour tout $p \in P'_j$.

A partir de ces données, on associe :

- une application \bar{t} de $E^j \rightarrow E$ à tout terme t ayant j variables, de la façon suivante :
 - si t est une variable, alors \bar{t} est l'application identique de E dans E ,
 - si t est une constante $c : \bar{t} = \bar{c}$,
 - si t est de la forme $f(t_1, t_2, \dots, t_j)$ alors $\bar{t} = \bar{f}(\bar{t}_1, \bar{t}_2, \dots, \bar{t}_j)$ (composition des applications)
- une application \bar{A} de $E^j \rightarrow \{V, F\}$ à toute formule ayant j variables libres ; de la façon suivante :
 - si A est un atome $A = p(t_1, t_2, \dots, t_n)$ alors $\bar{A} = \bar{p}(\bar{t}_1, \bar{t}_2, \dots, \bar{t}_n)$
 - si A est de la forme $\neg B$ alors pour tout $(e_1, e_2, \dots, e_j) \in E^j$:
 $\bar{A}(e_1, e_2, \dots, e_j) = V$ si $\bar{B}(e_1, e_2, \dots, e_j) = F$
 $\bar{A}(e_1, e_2, \dots, e_j) = F$ si $\bar{B}(e_1, e_2, \dots, e_j) = V$
 - si A est de la forme $(B \rightarrow C)$, alors pour tout $(e_1, e_2, \dots, e_j) \in E^j$:
 $\bar{A}(e_1, e_2, \dots, e_j) = F$ si $\bar{B}(e_1, e_2, \dots, e_j) = V$ et $\bar{C}(e_1, e_2, \dots, e_j) = \bar{F}$
 $\bar{A}(e_1, e_2, \dots, e_j) = V$ sinon.
 - si A est de la forme $\forall x B(x, y_1, y_2, \dots, y_j)$, alors pour tout $(e_1, e_2, \dots, e_j) \in E^j$:
 $\bar{A}(e_1, e_2, \dots, e_j) = V$ si, pour tout $e \in E$: $\bar{B}(e, e_1, e_2, \dots, e_j) = V$
 $\bar{A}(e_1, e_2, \dots, e_j) = F$ sinon.

Si i est une interprétation donnée, à toute formule close se trouve associé l'un des symboles V ou F .

A la place de la notation \bar{t} , \bar{A} nous utiliserons parfois la notation $i[t]$, $i[A]$ pour désigner les applications associées à des termes t , ou à des formules A .

Lorsque A contient des variables libres et que $i[A]$ est la fonction constante toujours égale à V on écrira $i[A] = V$.

(b) Exemple 1

$$C' = \emptyset \quad (\text{pas de symbole de constante})$$

$$F'_1 = \{f\}$$

$$F'_j = \emptyset \text{ pour } j \geq 2 \quad (\text{un seul symbole fonctionnel})$$

$$P'_2 = \{p\}$$

$$P'_j = \emptyset \text{ si } j \neq 2 \quad (\text{un seul symbole relationnel})$$

Parmi les formules de \Pr' , il y a :

$$\forall x(p(x, f(x)) \rightarrow p(f(x), x))$$

Considérons l'interprétation $i = (\bar{E}, \bar{f}, \bar{p})$

$$E = \{e_1, e_2, e_3\}$$

$$\bar{f} : e_1 \rightarrow e_2 ; e_2 \rightarrow e_3 ; e_3 \rightarrow e_1$$

$$\bar{p} : (e_1, e_2) \rightarrow V ; (e_2, e_1) \rightarrow V$$

$(x, y) \rightarrow F$ pour tous les autres couples $(x, y) \in E^2$.

Alors, à $A = p(x, f(x))$ est associée une fonction de E dans $\{V, F\}$ définie par :

- $\bar{A}(e_1) = \bar{p}(e_1, \bar{f}(e_1)) = \bar{p}(e_1, e_2) = V$
- $\bar{A}(e_2) = \bar{p}(e_2, \bar{f}(e_2)) = \bar{p}(e_2, e_3) = F$
- $\bar{A}(e_3) = \bar{p}(e_3, \bar{f}(e_3)) = \bar{p}(e_3, e_1) = F$

De même, à $B = p(f(x), x)$ est associée une fonction de E dans $\{V, F\}$ définie par :

- $\bar{B}(e_1) = V, \bar{B}(e_2) = F ; \bar{B}(e_3) = F$

À la formule $C = (p(x, f(x)) \rightarrow p(f(x), x))$ est associée la fonction $\bar{C} : E \rightarrow \{V, F\}$ définie par :

- $\bar{C}(e_1) = \bar{C}(e_2) = \bar{C}(e_3) = V$

Donc : $i[\forall x(p(x, f(x)) \rightarrow p(f(x), x))] = V$

Il est facile de voir que la formule proposée n'est pas vraie dans toute interprétation.

(c) Exemple 2

$C' = \{a, b, c\}$ (Trois constantes destinées à représenter trois personnages : Alain, Bernard, Christian).

$F'_1 = \{f\}$ (Un seul symbole de fonction destiné à représenter $F'_j = \emptyset$ pour $j \geq 2$ la fonction « père de »)

$P'_1 = \{p\}$ (Un symbole de prédicat unaire pour « est père » et $P'_2 = \{p'\}$ un symbole de prédicat binaire pour « est père de »).

$P'_j = \emptyset$ pour $j \geq 3$

Une interprétation de $\Sigma_{\Pr'}$, est la donnée d'une situation concrète dans laquelle, à chaque constante est associé un personnage et dans laquelle on explicite ce qui est vrai et ce qui est faux pour les personnages constituant la base de l'interprétation.

Par exemple considérons l'interprétation

$i = (E, \bar{a}, \bar{b}, \bar{c}, \bar{f}, \bar{p}, \bar{p}')$

définie par :

$$E = \{\text{Alain, Bernard, Christian, ancêtre}\}$$

$$\bar{a} = \text{Alain}, \bar{b} = \text{Bernard}, \bar{c} = \text{Christian}$$

$$\bar{f} : \bar{b} \rightarrow \bar{a}, \bar{c} \rightarrow \bar{a}, \bar{a} \rightarrow \text{ancêtre}, \text{ancêtre} \rightarrow \text{ancêtre}$$

$$\bar{p}(\bar{a}) = V, \bar{p}(\bar{b}) = F, \bar{p}(\bar{c}) = F, \bar{p}(\text{ancêtre}) = V$$

$$\bar{p}'(x, y) = V \text{ si et seulement si } (x = \bar{a} \text{ et } y = \bar{b}) \text{ ou } (x = \bar{a} \text{ et } y = \bar{c}) \text{ ou } (x = \text{ancêtre} \text{ et } y = \bar{a}) \text{ ou } (x = \text{ancêtre} \text{ et } y = \text{ancêtre}).$$

(d) Remarques

Une fois fixée, l'interprétation détermine l'attribution d'une valeur de vérité V ou F à n'importe quelle formule close, aussi compliquée soit-elle, et ceci en calculant sur des fonctions. Cette façon de ramener la recherche de la vérité d'une formule à un simple calcul permet le traitement rigoureux de la notion de vérité et en particulier permet de se poser les questions :

- est-ce que les formules que le système formel Pr donne comme théorème sont toujours vraies ?
- est-ce que les formules toujours vraies sont des théorèmes de Pr ?

Il ne faudrait cependant pas croire que la définition de vérité pour les formules de Pr qui se trouve ainsi formulée soit une définition arbitraire. Une analyse réfléchie montre au contraire qu'elle est la seule possible et on peut affirmer que les formules de Pr qui sont calculées vraies (pour toute interprétation) correspondent à ce que tout être raisonnable (utilisant les symboles \forall , \neg , \rightarrow dans leur sens usuel) considérera comme des formules vraies.

L'identité entre les formules calculées vraies dans toute interprétation (qu'on appelle tautologies) et les formules qui sont les théorèmes de Pr signifie donc l'adéquation parfaite du système formel Pr à la notion de vérité. Ce théorème (voir paragraphe 5) à l'inverse des théorèmes d'incomplétude marque la puissance de la notion de système formel.

(e) Tautologies, conséquences, etc.

On appelle *tautologie* (ou thèse) toute formule $A \in F_{Pr}$ telle que, pour toute interprétation i ; $i[A] = V$. On écrit alors $\models A$.

Exemples de tautologies :

$$\models (p(x) \vee \neg p(x))$$

$$\models (\forall x(p(x) \wedge q(x)) \longleftrightarrow (\forall y p(y) \wedge \forall z q(z)))$$

$$\models (\exists y \forall x r(z, x, y) \rightarrow \forall x \exists y r(z, x, y))$$

$$\models (\forall x(p(x) \rightarrow p(f(x))) \rightarrow \forall y(p(y) \rightarrow p(f(f(y)))))$$

On remarquera que, d'après la définition, la formule A contenant les variables libres x_1, x_2, \dots, x_n est une tautologie si et seulement si la formule close $\forall x_1 \forall x_2 \dots \forall x_n A$ est une tautologie.

Contrairement à ce qui se passe pour le calcul propositionnel le nombre d'interprétations différentes pour une formule donnée n'est pas fini. Il n'y a donc pas pour le calcul des prédictats d'équivalent à la notion de table de vérité. Pour savoir qu'une formule est une tautologie, il ne peut être question d'énumérer toutes les interprétations ; il faut raisonner.

On dit que la formule $B \in F_{Pr}$ est *conséquence* de la formule $A \in F_{Pr}$ si, pour toute interprétation i telle que $i[A] = V$, on a aussi $i[B] = V$. On écrit alors $A \models B$.

On dit que la formule $B \in F_{Pr}$ est *conséquence* de l'ensemble de formules $\mathcal{F} \subset F_{Pr}$ si pour toute interprétation i telle que pour tout $A \in \mathcal{F}$: $i[A] = V$, on a aussi : $i[B] = V$. On écrit alors : $\mathcal{F} \models B$.

On dit qu'une formule $A \in F_{Pr}$ est *satisfiable* ou *consistante* s'il existe une interprétation i telle que : $i[A] = V$.

On dit que l'ensemble de formules $\mathcal{F} \subset F_{Pr}$ est *satisfiable* ou *consistant* s'il existe une interprétation i telle que, pour tout $A \in \mathcal{F}$ on ait $i[A] = V$. L'interprétation i est alors appelée *modèle* de \mathcal{F} .

On dit qu'une formule $A \in F_{Pr}$ est *insatisfiable* ou *inconsistante* s'il n'existe aucune interprétation i telle que $i[A] = V$. Lorsque A est une formule close on montre facilement que : A est insatisfiable si et seulement si $\neg A$ est satisfiable.

On dit que l'ensemble de formules $\mathcal{F} \subset F_{Pr}$ est *insatisfiable* ou *inconsistant* s'il n'existe aucune interprétation i telle que $i[A] = V$ pour toute formule $A \in \mathcal{F}$.

Les théorèmes du paragraphe suivant établissent que les notions de consistance et d'inconsistance introduites dans ces définitions correspondent bien à celles introduites au chapitre 3.

Quelques exemples :

- La formule $\forall x p(x,x)$ est conséquence des formules :

$$\begin{aligned} & \forall x \forall y (q(x,y) \rightarrow p(x,y)) \\ & \forall z q(z,z) \end{aligned}$$
- L'ensemble de formules suivant est satisfiable.

$$\begin{aligned} & \forall x \forall y (p(x,x) \rightarrow (p(y,y) \rightarrow p(x,y))) \\ & \forall z p(z,z) \end{aligned}$$

On peut d'ailleurs montrer que pour tout ensemble non vide E il n'existe qu'un seul modèle de base E de cet ensemble de formules, qui est le modèle (E, \bar{p}) défini par $\bar{p}(\alpha, \beta) = V$ pour tout $\alpha \in E$ et $\beta \in E$.

- L'ensemble de formules suivant est insatisfiable :

$$\begin{aligned} & \exists a p(a) \\ & \forall x (p(x) \rightarrow \neg r(x)) \\ & \forall y (\neg r(y) \rightarrow q(y)) \\ & \forall z (q(z) \rightarrow \neg p(z)) \end{aligned}$$

alors que tout sous-ensemble de trois formules prises dans cet ensemble est satisfiable.

Proposition 4. — Soit $\mathcal{A} \subset F_{Pr}$ un ensemble de formules closes de P_r . Soit B une formule close de P_r . Alors : $\mathcal{A} \models B$ si et seulement si $\mathcal{A} \cup \{\neg B\}$ est insatisfiable.

Démonstration :

(a) Supposons que $\mathcal{A} \models B$.

Soit i une interprétation

- si $i[A] = V$ pour tout $A \in \mathcal{A}$, alors $i[B] = V$
donc $i[\neg B] = F$, et donc i n'est pas un modèle de $\mathcal{A} \cup \{\neg B\}$.
- si $i[A] = F$ pour une formule $A \in \mathcal{A}$, alors i n'est pas un modèle de $\mathcal{A} \cup \{\neg B\}$.

Dans aucun cas, i n'est modèle de $\mathcal{A} \cup \{\neg B\}$ donc $\mathcal{A} \cup \{\neg B\}$ est insatisfiable.

(b) Supposons que $\mathcal{A} \cup \{\neg B\}$ soit insatisfiable.

Soit i telle que $i[A] = V$ pour tout $A \in \mathcal{A}$

Puisque $\mathcal{A} \cup \{\neg B\}$ n'a pas de modèle, il est impossible que $i[\neg B] = V$, donc $i[B] = V$.

□

Cette proposition dont la démonstration n'est que l'utilisation immédiate des définitions est, malgré tout, importante car c'est elle qui justifie la méthode dite de « falsification » : pour établir que $\mathcal{A} \models B$, on établit que $\mathcal{A} \cup \{\neg B\}$ n'a pas de modèle.

Proposition 5. — Soit $A \in F_{Pr}$ une formule.

Si $\vdash A$, alors $\models A$

Autrement dit, les formules de F_{Pr} qui sont des théorèmes sont aussi des tautologies : la validité syntaxique implique la validité sémantique.

Démonstration :

La démonstration donnée au chapitre 4 de l'énoncé analogue se complète sans difficulté en tenant compte de la règle d'inférence de généralisation et des nouveaux schémas d'axiomes.

□

5. — RÉSULTATS

Les propositions 9, 10, 11 et 13 du chapitre 4 s'adaptent au calcul des prédictats, mais leurs démonstrations sont plus difficiles, nous ne les donnerons pas (voir [HUE 75], [KLE 51], [LAR 72] et [YAS 71]).

Proposition 6. — (Théorème de complétude)

*Pour toute formule $A \in F_{Pr}$
si $\models A$, alors $\vdash A$.*

Proposition 7. — (Théorème de compacité)

Soit F un ensemble de formules de F_{Pr} .

Si toute famille finie $F' \subset F$ est satisfiable, alors F aussi est satisfiable.

Proposition 8. — (Théorème de finitude)

Soit F un ensemble de formules de F_{Pr} ,

soit $B \in F_{Pr}$.

Si $F \models B$

alors il existe $F' \subset F$, F' fini, tel que : $F' \models B$.

Proposition 9. — (Théorème de complétude généralisé)

Soit $F \subset F_{Pr}$, et $B \in F_{Pr}$ alors :

$F \models B$ si et seulement si $F \vdash B$.

Proposition 10. — Soit $F \subset F_{Pr}$ et B une tautologie alors :

$F \vdash \neg\neg B$ si et seulement si F n'a pas de modèle.

Démonstration (de la proposition 10 à partir de la proposition 9) :

Si $F \vdash \neg\neg B$ alors : $F \models \neg B$, donc, à chaque fois que $i[A] = V$ pour tout $A \in F$, on a $i[\neg B] = V$. Comme, pour toute interprétation i , on a $i[B] = V$, on voit qu'il ne peut pas exister de i tel que $i[A] = V$ pour tout $A \in F$, ce qui signifie que F n'a pas de modèle.

Réciproquement, si F n'a pas de modèle $F \models \neg B$ donc $F \vdash \neg\neg B$.

□

Remarque :

Comme au chapitre 4 il est facile de voir que le système formel obtenu en ajoutant F aux axiomes de Pr est cohérent si et seulement si il est consistant (au sens du chapitre 3). De même il est cohérent si et seulement si il n'existe pas de tautologie B telle que $F \vdash \neg B$.

La proposition 10 montre donc que les notions de consistance des chapitres 3 et 5 correspondent bien.

Proposition 11. — (Théorème de Löwenheim-Skolem).

Soit $F \subset F_{Pr}$.

Si F admet des modèles, alors F admet au moins un modèle dénombrable (c'est-à-dire dont l'ensemble de base est dénombrable).

Ce théorème indique en particulier qu'il est impossible d'exprimer la propriété d'être dénombrable avec des formules du premier ordre.

6. — LA NOTION DE THÉORIE AXIOMATIQUE

(a) Définitions

On appelle *théorie axiomatique* tout système formel TA.

- ayant un alphabet de la forme $\Sigma_{Pr'}$ (voir paragraphe 3),
- ayant pour ensemble de formules bien formées, l'ensemble $F_{Pr'}$ (défini à partir de $\Sigma_{Pr'}$),
- dont l'ensemble d'axiomes contient tous ceux de Pr' (qui sont appelés axiomes logiques de la théorie), et dont les autres sont appelés axiomes non logiques de la théorie,
- dont les règles d'inférence sont m.p. et g.

On appelle modèle de TA toute interprétation i telle que, pour tout axiome $A : i[A] = V$.

(b) Applications des résultats du paragraphe 5

L'ensemble des théorèmes d'une théorie axiomatique TA, c'est, par définition, l'ensemble des formules qu'on peut déduire des axiomes (logiques et non logiques). D'après la proposition 9, c'est aussi l'ensemble des formules qui sont vraies dans toute interprétation qui satisfait les axiomes, c'est-à-dire dans tout modèle.

Cette équivalence (qui signifie que tout ce qui est vrai est déductible et que tout ce qui est déductible est vrai) n'est pas du tout évidente (rappelons que nous n'avons pas démontré la proposition 9) ; c'est elle qui justifie l'utilisation de la notion de théorie axiomatique formalisée dans le calcul des prédictats du

premier ordre. Outre qu'il y a plusieurs façons différentes de formaliser le calcul des prédicats pour les ordres > 1 , cette équivalence entre vérité syntaxique et vérité sémantique n'est plus vraie en général pour les ordres > 1 .

On peut résumer les différentes définitions et différents théorèmes concernant la consistance d'une théorie axiomatique TA dont l'ensemble d'axiomes non logiques est \mathcal{A} en disant que toutes les propriétés suivantes sont équivalentes :

- il n'existe pas de formules F de TA telles que :

$$\begin{array}{c} \vdash F \quad \text{et} \quad \vdash \neg F \\ \text{TA} \qquad \qquad \text{TA} \end{array}$$

- TA est consistante
- TA est non contradictoire
- TA est cohérente
- il existe des formules F de TA qui ne sont pas des théorèmes de TA
- \mathcal{A} est consistant
- \mathcal{A} est satisfiable
- \mathcal{A} admet des modèles
- \mathcal{A} admet des modèles dénombrables.

(c) Exemples

Exemple 1 :

Théorie des relations d'équivalence TEQ.

- Il y a un seul symbole de prédicat $p \in P_2$, pas de constante, pas de symbole de fonction.
- Les axiomes non logiques sont :
 $\forall x p(x,x)$
 $\forall x \forall y (p(x,y) \rightarrow p(y,x))$
 $\forall x \forall y \forall z ((p(x,y) \wedge p(y,z)) \rightarrow p(x,z))$

Un théorème de cette théorie est, par exemple :

$$\forall x \forall y \forall z \forall u (((p(x,y) \wedge p(y,z)) \wedge \neg p(x,u)) \rightarrow \neg p(z,u))$$

Un modèle de TEQ est simplement la donnée :

- d'un ensemble de base E
- d'une application $\bar{p} : E^2 \rightarrow \{V,F\}$
telle que la relation $\bar{p}(x,y) = V$ soit une relation d'équivalence au sens habituel sur l'ensemble E.

Exemple 2 :

Théorie associée à une situation concrète.

De manière à être plus lisible, nous ne respecterons pas pour cet exemple les conventions d'écriture fixées au début du chapitre (ce qui nous permet d'ailleurs d'utiliser des notations proches de celles de Prolog).

On voudrait « faire la théorie » d'une certaine famille de 11 personnes dont les liens de parenté sont exprimés par le schéma suivant :

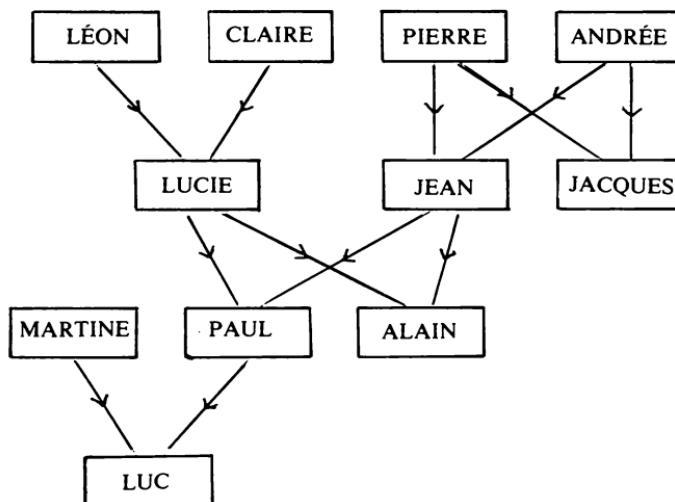


Fig. 5

Voici une façon de construire la théorie associée.

On introduit un symbole de constante pour chaque personnage donc :

$C' = \{\text{léon}, \text{claire}, \text{pierre}, \text{andrée}, \text{lucie}, \text{jean}, \text{jacques}, \text{martine}, \text{paul}, \text{alain}, \text{luc}\}$

On ne prend pas de symbole fonctionnel, mais on prend plusieurs symboles de prédicats :

masculin	} prédicats unaires
féminin	
est-père-de	} prédicats binaires
est-mère-de	
est-parent-de	} prédicats binaires
est-grand-père-de	
est-grand-mère-de	
est-grand-parent-de	

On pourrait en introduire d'autres si on voulait être plus complet.

Décrire le schéma et donner aux prédicts considérés leur sens habituel conduit à écrire des axiomes. On peut prendre par exemple les suivants :

a 1	masculin (léon)	a12	est-père-de (léon, lucie)
a 2	masculin (pierre)	a13	est-mère-de (claire, lucie)
a 3	masculin (jean)	a14	est-père-de (pierre, jean)
a 4	masculin (jacques)	a15	est-père-de (pierre, jacques)
a 5	masculin (paul)	a16	est-mère-de (andrée, jean)
a 6	masculin (alain)	a17	est-mère-de (andrée, jacques)
a 7	masculin (luc)	a18	est-mère-de (lucie, paul)
a 8	féminin (claire)	a19	est-mère-de (lucie, alain)
a 9	féminin (andrée)	a20	est-père-de (jean, paul)
a10	féminin (lucie)	a21	est-père-de (jean, alain)
a11	féminin (martine)	a22	est-mère-de (martine, luc)
		a23	est-père-de (paul, luc)

a24

$$\forall x \forall y ((\text{est-mère-de}(x,y) \vee \text{est-père-de}(x,y)) \iff \text{est-parent-de}(x,y))$$

a25

$$\forall x \forall y (\text{est-grand-père-de}(x,y) \iff \exists z (\text{est-père-de}(x,z) \wedge \text{est-parent-de}(z,y)))$$

a26

$$\forall x \forall y (\text{est-grand-mère-de}(x,y) \iff \exists z (\text{est-mère-de}(x,z) \wedge \text{est-parent-de}(z,y)))$$

a27

$$\forall x \forall y ((\text{est-grand-père-de}(x,y) \vee \text{est-grand-mère-de}(x,y)) \iff \text{est-grand-parent-de}(x,y))$$

On pourrait établir les théorèmes suivants :

F₁ • est-grand-parent-de (léon, paul)

F₂ • $\exists x \text{ est-parent-de(jean, } x)$

F₃ • $\forall x \forall y (\text{est-grand-parent-de}(x,y) \iff (\exists z (\text{est-père-de}(x,z) \wedge \text{est-père-de}(z,y)) \vee$

$\exists z (\text{est-père-de}(x,z) \wedge \text{est-mère-de}(z,y)) \vee$

$\exists z (\text{est-mère-de}(x,z) \wedge \text{est-père-de}(z,y)) \vee$

$\exists z (\text{est-mère-de}(x,z) \wedge \text{est-mère-de}(z,y))))$

et bien d'autres.

Ce qui signifie deux choses :

- pour chacune de ces formules F_i il existe une déduction de Pr n'utilisant que les axiomes de Pr et les axiomes a1-a27 et dont la dernière formule est F_i,
- quelle que soit l'interprétation i, si i[A] = V pour toutes les formules a1-a27 alors i[F₁] = V, i[F₂] = V et i[F₃] = V.

Une interprétation de cette théorie c'est la donnée :

- d'un ensemble de base E , qu'on peut imaginer comme un ensemble d'objets parmi lesquels il y aura des représentants de chaque constante, mais qui pourra comporter d'autres objets,
- d'une interprétation des constantes, c'est-à-dire du choix pour chaque constante, par exemple alain, d'un élément $\underline{\text{alain}} \in E$,
- d'une interprétation des prédictats, c'est-à-dire du choix pour chaque prédictat, par exemple masculin, et pour chaque $x \in E$ d'une valeur de vérité V ou F pour masculin (x).

Une interprétation sera un modèle de cette théorie si, par définition, tous les axiomes y sont vrais, c'est-à-dire si le calcul de $i[A]$ donne V pour chaque axiome.

(d) Théorie axiomatique égalitaire

On appelle *théorie axiomatique égalitaire* toute théorie axiomatique

- dont l'alphabet comporte le symbole de prédictat binaire $=$ (à la place de $=(x,y)$ on écrit $x = y$).
- et dont les axiomes comportent :
 $\forall x \ x = x$
 $(x = y \rightarrow (A(x) \leftrightarrow A(y)))$

pour toute formule A de la théorie.

(Dans le cas d'une théorie axiomatique égalitaire, on inclut ces axiomes parmi les axiomes logiques).

On peut montrer qu'il existe un modèle de la théorie axiomatique égalitaire TAE si et seulement si il existe un modèle de TAE dans lequel $=$ est interprété comme l'égalité (un tel modèle est appelé modèle égalitaire).

(c) Exemples de théories égalitaires

Exemple 1 :

La théorie des ensembles ordonnés TO

- Il y a un seul symbole de prédictat $p \in P_2$ (en plus de $=$) ; pas de constante ; pas de symbole de fonction. Les axiomes non logiques sont :

$$\forall x \ p(x,x)$$

$$\forall x \forall y \ ((p(x,y) \wedge p(y,x)) \rightarrow x = y)$$

$$\forall x \forall y \forall z \ ((p(x,y) \wedge p(y,z)) \rightarrow p(x,z))$$

Exemple 2 :

La théorie de l'arithmétique du premier ordre TAr :

- Il n'y a pas de symbole de prédictat (autre que $=$)

- Il y a un symbole de constante : 0
 - un symbole $s \in F_1$
 - deux symboles $+$, $\cdot \in F_2$
 - (à la place de $+(x,y)$, $\cdot(x,y)$ on écrit $(x+y)$, $(x.y)$)
- Les axiomes non logiques sont :
 - a1 : $x = y \rightarrow s(x) = s(y)$
 - a2 : $s(x) = s(y) \rightarrow x = y$
 - a3 : $\neg(s(x) = 0)$
 - a4 : $x + 0 = x$
 - a5 : $x + s(y) = s(x + y)$
 - a6 : $x.0 = 0$
 - a7 : $x.s(y) = (x.y) + x$
 - a8 : $((A(0) \wedge (\forall x (A(x) \rightarrow A(s(x)))) \rightarrow (\forall z A(z)))$
pour toute formule A ayant x comme variable libre.

Alors que a_1, \dots, a_7 sont des axiomes au sens strict, a_8 , lui, est un schéma d'axiomes. Ce schéma d'axiomes exprime le principe de récurrence.

Le fait que $(IN, 0, x+1, +, \cdot)$ ($x+1$ désigne la fonction successeur ; $+$ désigne l'addition usuelle des entiers ; \cdot désigne la multiplication usuelle des entiers) soit un modèle de TAr est naturel et évident (car chaque axiome est satisfait). Par contre, qu'il puisse exister des modèles dont l'ensemble de base soit non dénombrable est plus étonnant. C'est pourtant le cas et l'explication se trouve dans le fait que a_8 n'est pas assez fort : a_8 exprime que la propriété de récurrence doit être vérifiée pour toute partie de la base définissable par une formule, et non pas que la propriété de récurrence doit être vérifiée pour toute partie de la base (il y a une infinité dénombrable de formules, mais une infinité non dénombrable de parties de la base).

La théorie TAr n'est donc pas vraiment en adéquation avec l'objet qu'on voulait modéliser (IN muni de ses opérations). Cette inadéquation partielle de TAr n'est pas due à un mauvais choix d'axiomes, elle est beaucoup plus profonde et doit être mise en rapport avec les résultats de Gödel dont nous avons parlé au chapitre 3.

7. QUELQUES RÉSULTATS DE DÉCIDABILITÉ ET D'INDÉCIDABILITÉ

Nous allons maintenant indiquer, sans démonstration, un certain nombre de résultats concernant le problème de la décision pour des théories basées sur le calcul des prédictats :

- Le calcul des prédictats Pr est *indécidable*.

Ceci signifie : il n'existe aucun programme qui, pour toute formule $A \in F_{Pr}$, indique en un temps fini si OUI ou NON, A est un théorème de P_r , ou, ce qui revient au même, indique en un temps fini si OUI ou NON, A est une tautologie de Pr . Les résultats plus précis suivants délimitent assez bien la frontière entre le décidable et le non-décidable :

- La théorie axiomatique égalitaire ayant un seul symbole de prédicat binaire p (en plus de $=$) est *indécidable*.
- La théorie axiomatique égalitaire ayant un seul symbole fonctionnel binaire f (en plus du symbole du prédicat binaire $=$) est *indécidable*.
- La théorie axiomatique égalitaire ayant deux symboles fonctionnels unaires f,g (en plus du symbole de prédicat binaire $=$) est *indécidable*.
- Toute théorie axiomatique égalitaire ayant :
 - un nombre fini de symboles, de constantes c_1, c_2, \dots, c_n
 - un seul symbole fonctionnel unary f
 - un nombre fini de symboles de prédicats unaires p_1, p_2, \dots, p_m (en plus du symbole de prédicat binaire $=$) et n'ayant pas d'axiomes non logiques, est *décidable*.
- La théorie de l'arithmétique du premier ordre est *indécidable*.
- La théorie de l'arithmétique sans le symbole . est *décidable*. (Théorème de Presburger).

Beaucoup d'autres résultats ont été établis, on en trouvera de nombreux ainsi que la démonstration de ceux indiqués ici dans le livre de AZRA et JAULIN cité dans la bibliographie.

EXERCICES

Exercices sur la substitution avec renommage

Après avoir déterminé l'ensemble des variables libres et l'ensemble des variables liées des formules suivantes, effectuer les substitutions indiquées en respectant le principe de renommage préliminaire indiqué dans le cours.

- (a) Substituer $f(x,y)$ à t dans :
 $\forall x \ \forall y (p(x,a,t) \rightarrow \exists t p(x,y,t))$
- (b) Substituer $f(g(t,v),u)$ à x dans :
 $(\forall x \exists v (p(x,v) \rightarrow r(a)) \rightarrow \forall u \forall v p(t,r,f(x)))$

Exercices sur les déductions dans Pr

- 1) Etablir en détail que :
 $\vdash A$ si et seulement si $\vdash \forall x A$

2) Etablir en détail que :

$$\begin{aligned} \vdash (\forall x A(x) \rightarrow \forall y A(y)) \\ \vdash (\forall x A(x) \rightarrow \exists y A(y)) \end{aligned}$$

Exercice sur la notion de modèle

On considère l'ensemble \mathcal{F} composé des trois formules :

$$A1 : \forall x \forall y \forall z ((p(x,y) \wedge p(y,z)) \rightarrow p(x,z))$$

$$A2 : \forall x (p(a,x) \wedge p(x,b))$$

$$A3 : \forall x p(x, f(x))$$

1) Proposer un modèle i de \mathcal{F} c'est-à-dire :

- un ensemble de base E ,
- un élément $\bar{a} \in E$ et un élément $\bar{b} \in E$,
- une application \bar{f} de E dans E ,
- une application \bar{p} de $E \times E$ dans $\{V,F\}$

tels que $i[A1] = i[A2] = i[A3] = V$.

Calculer soigneusement étape par étape l'application $i[A]$ de $E \rightarrow \{V,F\}$ associée par i à la formule A :

$$\forall y (p(x,y) \rightarrow p(x,f(y)))$$

2) Montrer qu'il n'est pas vrai que :

$$A1, A2, A3 \models \exists x p(x,a)$$

Pour cela proposer un modèle i de \mathcal{F} tel que : $i[\exists x p(x,a)] = F$.

Un tel modèle s'appelle un contre-modèle pour la formule $\exists x p(x,a)$. Cette méthode a été souvent utilisée pour établir qu'une certaine formule n'est pas conséquence d'un ensemble donné de formules. En particulier c'est par cette méthode qu'on établit l'indépendance d'axiomes, et qu'a été démontrée l'impossibilité de déduire l'axiome des parallèles des autres axiomes de la géométrie plane, ou l'impossibilité de déduire l'axiome du choix des autres axiomes de la théorie des ensembles.

Exercices sur les tautologies

Indiquer pour chacune des formules suivantes s'il s'agit d'une tautologie ou non.

- (a) $(\forall x \exists y p(x,y) \rightarrow \exists x p(x,x))$
- (b) $(\exists x \forall y p(x,y) \rightarrow \exists x p(x,x))$
- (c) $((\exists y \forall x p(x,y) \rightarrow r(x)) \rightarrow \exists z r(z))$
- (d) $(\forall x \forall y (p(x,y) \vee p(y,x)) \rightarrow \forall x p(x,x))$
- (e) $(\forall x \exists y \forall z p(x,y,z) \rightarrow \exists y \forall z p(z,y,z))$
- (f) $((\forall x (p(x) \rightarrow r(x)) \wedge \exists y \neg r(y)) \rightarrow \exists z \neg p(z))$

Exercices sur des tautologies de base

- 1) Donner une démonstration détaillée de :

$$\models (\forall x A(x) \rightarrow \exists y A(y))$$

- 2) Montrer que :

$$\models (\forall x \forall y A(x,y) \leftrightarrow \forall y \forall x A(x,y))$$

$$\models (\exists x \exists y A(x,y) \leftrightarrow \exists y \exists x A(x,y))$$

$$\models (\exists x \forall y A(x,y) \rightarrow \forall y \exists x A(x,y))$$

- 3) En supposant que $x \notin \text{varlib}(A)$ montrer que :

$$\models (\forall x (A \vee B(x)) \leftrightarrow (A \vee \forall x B(x)))$$

$$\models (\forall x (A \wedge B(x)) \leftrightarrow (A \wedge \forall x B(x)))$$

$$\models (\exists x (A \vee B(x)) \leftrightarrow (A \vee \exists x B(x)))$$

$$\models (\exists x (A \wedge B(x)) \leftrightarrow (A \wedge \exists x B(x)))$$

- 4) Montrer que :

$$\models ((\forall x A(x) \vee \forall x (B(x)) \rightarrow \forall x (A(x) \vee B(x)))$$

$$\models ((\forall x A(x) \wedge \forall x B(x)) \leftrightarrow \forall x (A(x) \wedge B(x)))$$

$$\models ((\exists x A(x) \vee \exists x B(x)) \leftrightarrow \exists x (A(x) \vee B(x)))$$

$$\models (\exists x (A(x) \wedge B(x)) \rightarrow (\exists x A(x) \wedge \exists x B(x)))$$

- 5) En supposant que $x \notin \text{varlib}(B)$ montrer que :

$$\models ((\forall x A(x) \rightarrow B) \leftrightarrow \exists x (A(x) \rightarrow B))$$

$$\models ((\exists x A(x) \rightarrow B) \leftrightarrow \forall x (A(x) \rightarrow B))$$

$$\models ((B \rightarrow \forall x A(x)) \leftrightarrow \forall x (B \rightarrow A(x)))$$

$$\models ((B \rightarrow \exists x A(x)) \leftrightarrow \exists x (B \rightarrow A(x)))$$

Exercices sur la notion de formules équivalentes

Soient $A(x_1, x_2, \dots, x_n)$ et $B(x_1, x_2, \dots, x_n)$ deux formules dont les variables libres sont parmi x_1, x_2, \dots, x_n .

On dit que $A(x_1, x_2, \dots, x_n)$ et $B(x_1, x_2, \dots, x_n)$ sont équivalentes si pour toute interprétation i les deux applications $i[A]$ et $i[B]$ de E^n dans $\{V, F\}$ sont identiques. On note alors :

$$A(x_1, x_2, \dots, x_n) \equiv B(x_1, x_2, \dots, x_n)$$

- 1) Montrer que :

$$A(x_1, x_2, \dots, x_n) \equiv B(x_1, x_2, \dots, x_n)$$

si et seulement si

$$\models (A(x_1, x_2, \dots, x_n) \leftrightarrow B(x_1, x_2, \dots, x_n))$$

- 2) Montrer que si :

$$A(x_1, x_2, \dots, x_n) \equiv B(x_1, x_2, \dots, x_n) \text{ et}$$

$$A'(x_1, x_2, \dots, x_n) \equiv B'(x_1, x_2, \dots, x_n)$$

alors :

- $\forall x_i A(x_1, x_2, \dots, x_n) \equiv \forall x_i B(x_1, x_2, \dots, x_n)$
 - $\neg A(x_1, x_2, \dots, x_n) \equiv \neg B(x_1, x_2, \dots, x_n)$
 - $(A(x_1, x_2, \dots, x_n) \rightarrow A'(x_1, x_2, \dots, x_n)) \equiv (B(x_1, x_2, \dots, x_n) \rightarrow B'(x_1, x_2, \dots, x_n))$
- 3) Montrer que si dans une formule F comportant la sous-formule $A(x_1, x_2, \dots, x_n)$ on remplace $A(x_1, x_2, \dots, x_n)$ par la formule équivalente $B(x_1, x_2, \dots, x_n)$ alors on obtient une formule F' équivalente à F .

Exercices sur la négation

- 1) Montrer que pour toutes formules A et B :

$$\begin{aligned}\neg(A \rightarrow B) &\equiv (A \wedge \neg B) \\ \neg(A \vee B) &\equiv (\neg A \wedge \neg B) \\ \neg(A \wedge B) &\equiv (\neg A \vee \neg B) \\ \neg(A \leftrightarrow B) &\equiv ((A \wedge \neg B) \vee (\neg A \wedge B)) \\ \neg \forall x A &\equiv \exists x \neg A \\ \neg \exists x A &\equiv \forall x \neg A\end{aligned}$$

- 2) En utilisant ces équivalences et les résultats de l'exercice sur la notion de formules équivalentes, trouver pour chacune des formules suivantes une formule équivalente comportant le symbole de négation placé uniquement devant les symboles de prédictats.

$$\begin{aligned}\neg(\forall x p(x) \rightarrow (\exists y p(y) \rightarrow r(y) \vee \exists z r(z))) \\ \neg(\forall x (p(x) \leftrightarrow r(x)) \rightarrow \exists y (p(y) \wedge r(y))) \\ \neg(\forall x \exists y \forall z r(x,y,z) \rightarrow \exists t (p(t) \rightarrow q(t)))\end{aligned}$$

Exercices sur les modèles de base infinie

- 1) Formuler un système d'axiomes dont tout modèle ait une base infinie.
- 2) Soit n un entier non nul fixé. Formuler un système d'axiomes dont tout modèle ait une base ayant au moins n éléments.

Exercices sur le prolongement de modèles

Soit $(E, (\bar{f}_0, \bar{f}_1, \dots, \bar{f}_n, \dots), (\bar{p}_0, \bar{p}_1, \dots, \bar{p}_m, \dots))$ un modèle de l'ensemble de formules A .

Soit F tel que $F \cap E = \emptyset$.

- 1) Montrer qu'il existe un modèle de A de base $E \cup F$ de la forme

$$(E \cup F, (\bar{\bar{f}}_0, \bar{\bar{f}}_1, \dots, \bar{\bar{f}}_n), (\bar{\bar{p}}_0, \bar{\bar{p}}_1, \dots, \bar{\bar{p}}_m, \dots))$$

tel que les $\bar{\bar{f}}_i$ et les $\bar{\bar{p}}_i$ prolongent les \bar{f}_i et les \bar{p}_i , c'est-à-dire tel que :

$$\forall (e_1, e_2, \dots, e_j) \in E^j : \bar{\bar{f}}_i(e_1, e_2, \dots, e_j) = \bar{f}_i(e_1, e_2, \dots, e_j)$$

$$\forall (e_1, e_2, \dots, e_j) \in E^j : \bar{\bar{p}}_i(e_1, e_2, \dots, e_j) = \bar{p}_i(e_1, e_2, \dots, e_j)$$

- 2) En déduire qu'il n'existe pas de système d'axiomes A dont tout modèle ait une base finie (resp. dénombrable). En particulier décrire un modèle non dénombrable pour la théorie proposée à la question 1 de l'exercice précédent et décrire un modèle à $(n+1)$ éléments pour la théorie proposée à la question 2 du même exercice.

Exercice sur les modèles égalitaires

- 1) Soit n un entier non nul fixé. Formuler un système d'axiomes n'utilisant aucune constante, aucun symbole fonctionnel et le seul symbole de prédicat $=$, et tel que tout modèle égalitaire de ce système ait une base de n éléments exactement.
- 2) Formuler un système d'axiomes n'utilisant aucune constante, aucun symbole fonctionnel et le seul symbole de prédicat $=$ et vérifiant de plus que :
 - (a) tout ensemble E ayant au moins 2 éléments et au plus 5 éléments est la base d'un modèle égalitaire de ce système.
 - (b) tout modèle égalitaire de ce système a une base d'au moins 2 éléments et d'au plus 5 éléments.
- 3) Indiquer dans chaque cas si l'ensemble des formules de Pr , conséquences du système proposé est récursif ou non.

Exercice sur la récursivité d'ensembles de formules

Indiquer pour chacun des ensembles suivants s'il est récursif ou non :

$A = \{F \in F_{Pr} \mid F \text{ est une formule du calcul des prédicats du premier ordre qui n'est pas un théorème}\}$

$B = \{F \in F_{Pr} \mid F \text{ est une formule du calcul des prédicats du premier ordre telle que soit } F \text{ soit } \neg F \text{ est un théorème}\}$

$C = \{F \in F_{Pr} \mid F \text{ est une formule du calcul des prédicats du premier ordre dont il existe une déduction ayant moins de } 10^5 \text{ formules chacune de longueur inférieure à } 10^5\}$.

BIBLIOGRAPHIE

- [ARN 70] ARNAULD A. et NICOLE P.
La Logique ou l'Art de Penser.
Flammarion, Paris, 1970.
(A partir de l'édition de Guillaume Desprez de 1683).
{On trouvera un exposé de la façon dont on pouvait envisager la logique avant les progrès déterminants de la fin du XIX^e siècle et du début du XX^e siècle qui conduisirent à la conception « moderne » que seule nous avons considérée ici}
- [ARNO 70] ARNOLD A.
Les Mathématiques à la portée de l'ordinateur.
Dunod, Paris, 1970.
{Présentation d'un système formel permettant d'écrire des démonstrations mathématiques dans un style naturel et de les faire vérifier par un ordinateur}
- [AZR 73] AZRA J.-P. et JAULIN B.
Récursivité.
Collection « Programmation ». Gauthier-Villars, Paris, 1973.
{Contient, sous une présentation assez différente, tout ce qui est dans le chapitre, et beaucoup plus. En particulier les démonstrations des théorèmes de décidabilité et d'indécidabilité du paragraphe 7}
- [CHA 73] CHANG C.-L. et LEE R.-C.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, New York, 1973.
{Les trois premiers chapitres exposent de façon très efficace les éléments du calcul des prédicts que nous avons choisis de détailler et de formaliser un peu plus dans notre cours, et qui formeront la base utilisée dans la suite de leur livre pour présenter les techniques de démonstrations automatiques}
- [FRA 71] FRAISSE R.
Cours de Logique Mathématique.
Gauthiers-Villars, Paris, 1971.
{Très complet pour tout ce qui concerne l'aspect sémantique du calcul des prédicts}
- [HUE 75] HUET G.
Introduction à la Logique Mathématique.
Polycopié de Cours, 1975.
{Bonne présentation concise des résultats fondamentaux de la logique mathématique}
- [KLE 71] KLEENE S.-C.
Logique Mathématique.
Librairie Armand Colin, Paris, 1971.
(Traduction de : Mathematical Logic, John Wiley and Sons, New York, 1967).

{On trouvera une introduction très détaillée du calcul des prédictats, la démonstration du théorème de complétude ainsi que celles de l'indécidabilité du calcul des prédictats du premier ordre et de l'arithmétique. Aucune connaissance mathématique n'est nécessaire pour aborder cet ouvrage}

- [KRE 67] KREISEL G. et KRIVINE J.-L.
Eléments de Logique Mathématique : Théorie des Modèles.
Dunod, Paris, 1967.
{Cours de D.E.A. sur l'aspect sémantique du calcul des prédictats du premier ordre et du calcul des prédictats à plusieurs types d'objets. Excellent mais assez difficile}
- [KRI 72] KRIVINE J.-L.
Théorie Axiomatique des Ensembles.
Presses Universitaires de France, Paris, 1972.
{Très bonne introduction aux problèmes de la théorie des ensembles présentés dans le formalisme du calcul des prédictats du premier ordre (axiomes de Zermelo-Fraenkel)}
- [LAR 72] LARGEAULT J.
Logique Mathématique : Textes.
Librairie Armand Colin, Paris, 1972.
{Contient en particulier le texte traduit des articles de Lowenheim, Skolem, Gödel, Henkin, Beth qui traitent, dans des cadres divers, plus ou moins généraux, du problème de la complétude du calcul des prédictats}
- [LOI 82] LOI M.
Penser les Mathématiques.
Séminaire de Philosophie des Mathématiques de l'Ecole Normale Supérieure (J. Dieudonné, M. Loi, R. Thom). Editions du Seuil, 1982.
{Les appendices contiennent sous une forme condensée et facilement accessible la présentation de diverses notions et résultats récents, de logique mathématique. Les textes de J. Dieudonné, R. Fraisse apportent des compléments intéressants}
- [LOV 78] LOVELAND D.-W.
Automated Theorem Proving : A logical Basis.
North-Holland Publishing Company.
Amsterdam, 1978.
{Ce livre dont nous reparlerons plus loin donne dans son chapitre 1 une présentation du calcul des prédictats accompagnée de plusieurs exemples non triviaux intéressants}
- [ROB 79] ROBINSON J.-A.
Logic, Form and Function. The Mechanization of Deductive Reasoning.
Edinburgh University Press, Edinburgh, 1979.
{Cours sur le calcul des prédictats du premier ordre par celui qui est à l'origine du développement de la méthode de résolution dont nous parlerons au chapitre 7}
- [ROU 67] ROURE M.-L.
Eléments de Logique Contemporaine.
Collection Sup. Presses Universitaires de France, Paris, 1967.
{D'un abord facile, ce livre donne une présentation générale des éléments de

logique avec en particulier au chapitre 4 la définition (différente de la nôtre) d'un système formel pour le calcul des prédictats}

[TAR 53] TARSKI A., MOSTOWSKI A. et ROBINSON R.-M.

Undecidable Theories.

North Holland, Amsterdam, 1953.

{Ouvrage spécialement consacré aux problèmes de décidabilité des théories mathématiques}

[YAS 71] YASUHARA A.

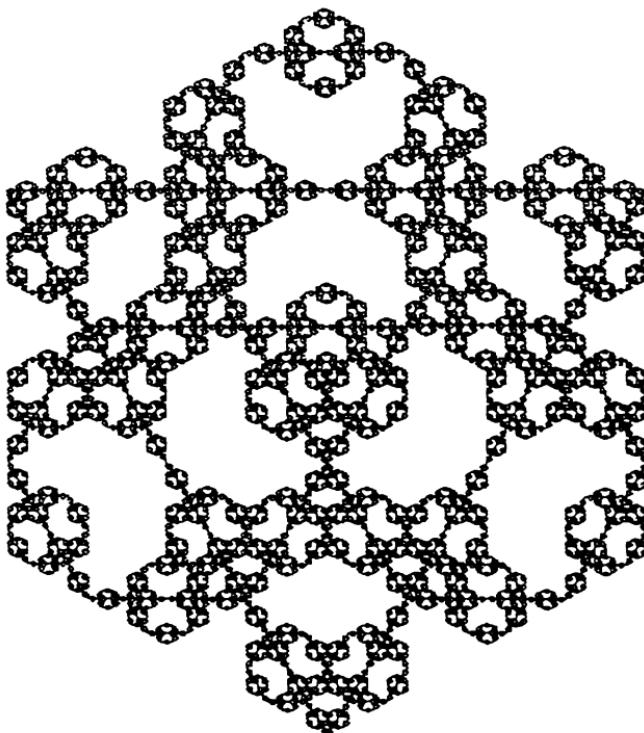
Recursive Function Theory and Logic.

Academic Press, New York, 1971.

{Contient sous une forme plus complète (sauf notre paragraphe 7) les notions et résultats de ce chapitre présentés dans un formalisme assez proche. Constitue donc un complément idéal}

CHAPITRE 6

PRÉPARATION DES FORMULES, MODÈLES DE HERBRAND



1. — INTRODUCTION

La mise sous forme prénexe (paragraphe 2) et la mise sous forme de Skolem (paragraphe 3) permettent de se débarrasser des quantificateurs d'un ensemble fini de formules du calcul des prédictats.

Le théorème de Herbrand (paragraphe 4) montre que lorsqu'on a un ensemble fini de formules (mis sous bonne forme) l'existence d'un modèle est équivalente à l'existence d'un « modèle syntaxique », c'est-à-dire basé uniquement sur les termes engendrés par les formules. Grâce à ce théorème, on peut proposer un algorithme de semi-décision pour l'ensemble des questions du type : B est-il conséquence de $\{B_1, B_2, \dots, B_n\}$? (paragraphe 5). Cet algorithme plus efficace qu'un simple algorithme d'énumération des déductions (quoique n'apportant théoriquement rien de plus) constitue le premier algorithme intéressant de démonstration automatique de théorème. C'est d'ailleurs un algorithme semblable qui en 1960 fut implémenté par Gilmore puis amélioré par Davis et Putnam (voir [CHA 73]). Bien sûr, cet algorithme ne peut traiter que des problèmes très simples et seule la méthode de résolution que nous présenterons au chapitre suivant permet (elle-même, ou une de ses multiples variantes) d'aborder des problèmes d'une taille plus grande.

2. — MISE SOUS FORME PRÉNEXE

Soit A une formule du calcul des prédictats, on dit que A est sous forme prénexe si A est de la forme :

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n B$$

où chaque Q_i est soit \forall , soit \exists et où B ne contient aucun quantificateur.

Proposition 1. — Pour toute formule A, il existe une formule A' sous forme prénexe équivalente à A.

Démonstration :

Rappelons (voir exercice, chapitre 5) que deux formules sont équivalentes si dans toute interprétation les fonctions qui leur sont associées sont identiques, c'est-à-dire si elles ont « même sens ».

On sait en particulier que si dans une formule F comportant la sous-formule A, on remplace A par une formule équivalente A', on obtient alors une formule F' équivalente à F.

Nous présentons maintenant deux méthodes permettant à partir d'une formule quelconque d'obtenir une formule équivalente sous forme prénexe. Le fait que ces méthodes se terminent et donnent effectivement une formule prénexe équivalente à celle prise au départ résulte de considérations élémentaires.

Méthode 1

- (a) Se débarrasser des \longleftrightarrow , \vee , \wedge en utilisant les équivalences suivantes de gauche à droite :
- a1 $(A \vee B) \equiv (\neg A \rightarrow B)$
 - a2 $(A \wedge B) \equiv \neg(\neg A \rightarrow \neg B)$
 - a3 $(A \longleftrightarrow B) \equiv \neg((A \rightarrow B) \rightarrow \neg(B \rightarrow A))$
- (b) Changer le nom de certaines variables liées de manière à n'avoir plus de variable quantifiée deux fois, en utilisant les équivalences suivantes :
- b1 $\forall x A(x) \equiv \forall y A(y)$
 - b2 $\exists x A(x) \equiv \exists y A(y)$
- (c) Faire remonter tous les quantificateurs en tête, en utilisant les équivalences suivantes de gauche à droite :

$x \notin \text{varlib}(C)$

- c1 $\neg \exists x A(x) \equiv \forall x \neg A(x)$
- c2 $\neg \forall x A(x) \equiv \exists x \neg A(x)$
- c3 $\neg \neg A \equiv A$
- c4 $C \rightarrow \forall x A(x) \equiv \forall x (C \rightarrow A(x))$
- c5 $C \rightarrow \exists x A(x) \equiv \exists x (C \rightarrow A(x))$
- c6 $(\forall x A(x) \rightarrow C) \equiv \exists x (A(x) \rightarrow C)$
- c7 $(\exists x A(x) \rightarrow C) \equiv \forall x (A(x) \rightarrow C)$

Exemple :

$$\begin{aligned} & (\forall x A(x) \vee \exists x (B(x) \wedge \exists t C(x,t))) \\ & \equiv (\neg \forall x A(x) \rightarrow \exists x (B(x) \wedge \exists t C(x,t))) & [a1] \\ & \equiv (\neg \forall x A(x) \rightarrow \exists x \neg (\neg B(x) \rightarrow \neg \exists t C(x,t))) & [a2] \\ & \equiv (\neg \forall x A(x) \rightarrow \exists y \neg (\neg B(y) \rightarrow \neg \exists t C(y,t))) & [b2] \\ & \equiv (\exists x \neg A(x) \rightarrow \exists y \neg (\neg B(y) \rightarrow \neg \exists t C(y,t))) & [c2] \\ & \equiv (\exists x \neg A(x) \rightarrow \exists y \neg (\neg B(y) \rightarrow \forall t \neg C(y,t))) & [c1] \\ & \equiv (\exists x \neg A(x) \rightarrow \exists y \neg \forall t (\neg B(y) \rightarrow \neg C(y,t))) & [c4] \\ & \equiv (\exists x \neg A(x) \rightarrow \exists y \exists t \neg (\neg B(y) \rightarrow \neg C(y,t))) & [c2] \\ & \equiv \forall x (\neg A(x) \rightarrow \exists y \exists t \neg (\neg B(y) \rightarrow \neg C(y,t))) & [c7] \\ & \equiv \forall x \exists y (\neg A(x) \rightarrow \exists t \neg (\neg B(y) \rightarrow \neg C(y,t))) & [c5] \\ & \equiv \forall x \exists y \exists t (\neg A(x) \rightarrow \neg (\neg B(y) \rightarrow \neg C(y,t))) & [c5] \end{aligned}$$

Il peut être intéressant pour limiter le nombre de quantificateurs de la formule finale de ne pas mener complètement l'étape (b) et, à l'étape (c), d'utiliser

l'équivalence supplémentaire :

$$c8 \quad (\forall x A(x) \rightarrow \exists x B(x)) \equiv \exists x (A(x) \rightarrow B(x))$$

Exemple :

$$\begin{aligned} & (\forall x A(x) \wedge \forall x B(x)) \\ \equiv & \neg(\forall x A(x) \rightarrow \neg \forall x B(x)) & [a2] \\ \equiv & \neg(\forall x A(x) \rightarrow \exists x \neg B(x)) & [c2] \\ \equiv & \neg \exists x (A(x) \rightarrow \neg B(x)) & [c8] \\ \equiv & \forall x \neg (A(x) \rightarrow \neg B(x)) & [c1] \end{aligned}$$

Sans l'utilisation de c8 le calcul aurait été :

$$\begin{aligned} & \forall x A(x) \wedge \forall x B(x) \\ \equiv & \neg(\forall x A(x) \rightarrow \neg \forall x B(x)) & [a2] \\ \equiv & \neg(\forall x A(x) \rightarrow \neg \forall y B(y)) & [b1] \\ \equiv & \neg(\forall x A(x) \rightarrow \exists y \neg B(y)) & [c2] \\ \equiv & \neg \exists x (A(x) \rightarrow \exists y \neg B(y)) & [c6] \\ \equiv & \neg \exists x \exists y (A(x) \rightarrow \neg B(y)) & [c5] \\ \equiv & \forall x \neg \exists y (A(x) \rightarrow \neg B(y)) & [c1] \\ \equiv & \forall x \forall y \neg (A(x) \rightarrow \neg B(y)) & [c1] \end{aligned}$$

Méthode 2

- (d) Se débarrasser des \rightarrow et \leftrightarrow en utilisant les équivalences suivantes de gauche à droite :
- d1 $(A \rightarrow B) \equiv (\neg A \vee B)$
d2 $(A \leftrightarrow B) \equiv ((A \wedge B) \vee (\neg A \wedge \neg B))$
- (b) Changer le nom de certaines variables liées de manière à n'avoir plus de variable quantifiée deux fois, en utilisant les équivalences (b1) et (b2).
- (e) Faire remonter tous les quantificateurs en tête en utilisant les équivalences suivantes de gauche à droite :

$$\begin{aligned} x \notin \text{varlib}(C) \\ e1 \quad \neg \exists x A(x) & \equiv \forall x \neg A(x) \\ e2 \quad \neg \forall x A(x) & \equiv \exists x \neg A(x) \\ e3 \quad \neg \neg A & \equiv A \\ e4 \quad (C \vee \forall x A(x)) & \equiv \forall x (C \vee A(x)) \\ e5 \quad (C \vee \exists x A(x)) & \equiv \exists x (C \vee A(x)) \\ e6 \quad (\forall x A(x) \vee C) & \equiv \forall x (A(x) \vee C) \\ e7 \quad (\exists x A(x) \vee C) & \equiv \exists x (A(x) \vee C) \\ e8 \quad C \wedge \forall x A(x) & \equiv \forall x (C \wedge A(x)) \\ e9 \quad C \wedge \exists x A(x) & \equiv \exists x (C \wedge A(x)) \end{aligned}$$

$$\begin{array}{ll} e10 \ (\forall x A(x) \wedge C) & \equiv \forall x (A(x) \wedge C) \\ e11 \ (\exists x A(x) \wedge C) & \equiv \exists x (A(x) \wedge C) \end{array}$$

Exemple :

$$\begin{aligned} & (\forall x A(x) \rightarrow (\exists t B(t) \vee \exists t C(t))) \\ \equiv & (\neg \forall x A(x) \vee (\exists t B(t) \vee \exists t C(t))) & [d1] \\ \equiv & (\neg \forall x A(x) \vee (\exists t B(t) \vee \exists y C(y))) & [b2] \\ \equiv & (\exists x \neg A(x) \vee (\exists t B(t) \vee \exists y C(y))) & [e2] \\ \equiv & \exists x (\neg A(x) \vee (\exists t B(t) \vee \exists y C(y))) & [e7] \\ \equiv & \exists x (\neg A(x) \vee \exists t (B(t) \vee \exists y C(y))) & [e7] \\ \equiv & \exists x \exists t (\neg A(x) \vee (B(t) \vee \exists y C(y))) & [e5] \\ \equiv & \exists x \exists t (\neg A(x) \vee \exists y (B(t) \vee C(y))) & [e5] \\ \equiv & \exists x \exists t \exists y (\neg A(x) \vee (B(t) \vee C(y))) & [e5] \end{aligned}$$

Pour limiter le nombre de quantificateurs de la formule finale, il peut être intéressant de ne pas mener complètement l'étape (b), et à l'étape (e) d'utiliser les deux équivalences supplémentaires :

$$e12 \ (\forall x A(x) \wedge \forall x B(x)) \equiv \forall x (A(x) \wedge B(x))$$

$$e13 \ (\exists x A(x) \vee \exists x B(x)) \equiv \exists x (A(x) \vee B(x))$$

En reprenant le même exemple :

$$\begin{aligned} & (\forall x A(x) \rightarrow (\exists t B(t) \vee \exists t C(t))) \\ \equiv & (\neg \forall x A(x) \vee (\exists t B(t) \vee \exists t C(t))) & [d1] \\ \equiv & (\neg \forall x A(x) \vee \exists t (B(t) \vee C(t))) & [e13] \\ \equiv & (\exists x \neg A(x) \vee \exists t (B(t) \vee C(t))) & [e2] \\ \equiv & \exists x (\neg A(x) \vee \exists x (B(x) \vee C(x))) & [b2] \\ \equiv & \exists x (\neg A(x) \vee (B(x) \vee C(x))) & [e13] \end{aligned}$$

□

3. — THÉORÈME DE SKOLEM

Soit $Q_1 x_1 Q_2 x_2 \dots Q_m x_m B(x_1, x_2, \dots, x)$ une formule A mise sous forme pré-nexe.

On appelle *forme de Skolem de A* et on note A^S la formule obtenue en enlevant tous les quantificateurs $\exists x_i$, en remplaçant chacune des variables x_i quantifiée avec un \exists par $f_i(x_{j_1}, x_{j_2}, \dots, x_{j_p})$ où $x_{j_1}, x_{j_2}, \dots, x_{j_p}$ sont les variables quantifiées par des \forall placés devant le $\exists x_i$. On suppose bien sûr que les symboles fonctionnels f_i introduits sont différents de tous ceux utilisés par ailleurs. Lorsqu'il n'y a aucun quantificateur \forall devant le $\exists x_i$, le symbole que l'on introduit est une constante (car une constante n'est rien d'autre qu'un symbole fonctionnel 0-aire).

Exemples :

La forme de Skolem de :

$$\exists x p(x, f(x)) \text{ est } p(a, f(a))$$

La forme de Skolem de :

$$\forall x \exists y p(x, f(y)) \text{ est } \forall x p(x, f(f_1(x)))$$

La forme de Skolem de :

$$\exists x_1 \forall x_2 \exists x_3 \exists x_4 (p(x_1, x_2) \rightarrow r(x_3, x_4))$$

est :

$$\forall x_2 (p(a, x_2) \rightarrow r(f_1(x_2), f_2(x_2)))$$

La forme de Skolem de :

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \exists x_5 p(x_1, x_2, x_3, x_4, x_5)$$

est :

$$\forall x_2 \forall x_4 p(a, x_2, f_1(x_2), x_4, f_2(x_2, x_4)).$$

Proposition 2. — (Théorème de Skolem)

Soit $\{A_1, A_2, \dots, A_n\}$ un ensemble fini de formules du calcul des prédictifs.

Soit $\{A_1^S, A_2^S, \dots, A_n^S\}$ l'ensemble des formes de Skolem de ces formules.

Alors $\{A_1, A_2, \dots, A_n\}$ admet un modèle de base E si et seulement si

$\{A_1^S, A_2^S, \dots, A_n^S\}$ admet un modèle de base E.

Démonstration :

Nous allons établir ce résultat dans le cas d'une unique formule A que nous supposerons de la forme : $\forall x \exists y p(x, y)$.

La démonstration complète du théorème de Skolem, (à des complications techniques près) est analogue.

Nous avons : $A^S = \forall x p(x, f(x))$

- (a) Soit $i = (E, \bar{p})$ avec $\bar{p} = E \times E \rightarrow \{V, F\}$ un modèle de A de base E.

Par définition :

$i[\exists y p(x, y)]$ est une application de E dans $\{V, F\}$ qui ne prend que la valeur V ; donc, pour tout $\alpha \in E$, il existe $\beta \in E$ tel que $\bar{p}(\alpha, \beta) = V$.

En introduisant f et en posant pour tout $\alpha \in E$ $\bar{f}(\alpha) = \beta$ où β est l'élément de E obtenu au-dessus, on obtient $i' = (E, \bar{p}, \bar{f})$ un modèle de la formule A^S .

Le modèle (E, \bar{p}) de A s'est donc « prolongé » en un modèle de (E, \bar{p}, \bar{f}) de A^S .

- (b) Réciproquement, si (E, \bar{p}, \bar{f}) est un modèle de A^S , on vérifie immédiatement que (E, \bar{p}) est un modèle de A .

□

Le problème général de la démonstration automatique des théorèmes peut se formuler par la question : est-il vrai que :

[la formule T est un théorème de la théorie dont l'ensemble d'axiomes non logiques est \mathcal{A} .] ?

Ce qui est équivalent à : est-il vrai que :

[la formule T est conséquence de \mathcal{A}] ?

ou encore, d'après la proposition 4 du chapitre 5 à : est-il vrai que :

[l'ensemble $\mathcal{A} \cup \{\neg T\}$ n'a pas de modèle] ?

On est donc ramené au problème de savoir si un certain ensemble de formules F_0 possède des modèles ou non. Pour le résoudre, on effectue une mise sous forme standard en opérant successivement les traitements suivants :

- mise sous forme prénexe des formules de F_0 , ce qui donne F_1 (voir paragraphe 2) ;
- mise sous forme de Skolem des formules de F_1 , ce qui donne F_2 ;
- suppression des quantificateurs universels des formules de F_2 , ce qui donne F_3 ;
- mise sous forme de clauses des formules de F_3 , ce qui donne F_4 (voir chapitre 4 proposition 8).

L'existence de modèles pour F_0 est équivalente à l'existence de modèles pour chaque F_i et donc en particulier à l'existence de modèles pour F_4 .

Exemple :

Posons-nous le problème :

La formule $T = \exists x \exists y q(x,y)$ est-elle un théorème de la théorie dont l'ensemble d'axiomes est \mathcal{A} :

$$\{\forall x (p(x) \rightarrow \exists y (r(y) \wedge q(x,y))), \\ \exists x p(x)\}$$

L'ensemble F_0 est alors :

$$F_0 = \{\forall x (p(x) \rightarrow \exists y (r(y) \wedge q(x,y))), \\ \exists x p(x), \\ \neg \exists x \exists y q(x,y)\}$$

On obtient successivement les ensembles de formules :

$$\begin{aligned} F_1 = & \{\forall x \exists y (\neg p(x) \vee (r(y) \wedge q(x,y))), \\ & \exists x p(x), \\ & \forall x \forall z \neg q(x,y)\} \end{aligned}$$

$$\begin{aligned} F_2 = & \{\forall x (\neg p(x) \vee (r(f(x)) \wedge q(x,f(x)))), \\ & p(a), \\ & \forall x \forall y \neg q(x,y)\} \end{aligned}$$

$$\begin{aligned} F_3 = & \{(\neg p(x) \vee (r(f(x)) \wedge q(x,f(x)))), \\ & p(a), \\ & \neg q(x,y)\} \end{aligned}$$

$$\begin{aligned} F_4 = & \{\neg p(x) \vee q(x,f(x)), \\ & \neg p(x) \vee r(f(x)), \\ & p(a), \\ & \neg q(x,y)\} \end{aligned}$$

C'est cet ensemble F_4 qu'on traite. Si F_4 ne possède pas de modèle, cela signifie que T est bien un théorème, si F_4 possède des modèles cela signifie que T n'est pas un théorème.

Il existe des procédures de mise sous forme standard limitant au maximum la complexité des formules de F_4 . Ces procédures utilisent le mieux possible les équivalences de factorisation (voir paragraphe 2), mélangeant les phases de mise sous forme prénexe et de mise sous forme de Skolem de manière à limiter le nombre d'arguments des fonctions de Skolem introduites (voir [LOV 78]) et simplifient quand c'est possible les clauses obtenues.

4. — THÉORÈME DE HERBRAND

A priori, pour savoir si un ensemble de formules du calcul des prédictats du premier ordre admet un modèle il faut réaliser une infinité d'essais : essayer s'il y a un modèle ayant une base à un élément, essayer s'il y a un modèle ayant une base à deux éléments,..., essayer s'il y a un modèle ayant une base à une infinité d'éléments. Chacun de ces essais se divise lui-même en un très grand nombre d'essais. L'intérêt du théorème de Herbrand que nous allons voir dans ce paragraphe est qu'il permet de se ramener à un seul essai : pour savoir si un ensemble de formules F possède des modèles, il suffit de savoir si F possède un « modèle syntaxique » c'est-à-dire construit de manière standard à partir du vocabulaire utilisé dans les formules de F . De plus, savoir si ce modèle syntaxique existe se ramène à l'étude d'ensembles de formules du calcul propositionnel (pour lequel nous savons que « tout est décidable »).

Soit $\{A_1, A_2, \dots, A_n\}$ un ensemble fini de formules du calcul des prédictats du premier ordre, résultat par exemple de la mise sous forme de clauses d'un ensemble de formules dont on cherche à savoir s'il admet des modèles.

Pour illustrer les définitions qui viennent, nous reprenons l'exemple du paragraphe précédent :

$$A_1 = \neg p(x) \vee q(x, f(x))$$

$$A_2 = \neg p(x) \vee r(f(x))$$

$$A_3 = p(a)$$

$$A_4 = \neg q(x, y)$$

On appelle *univers de Herbrand* associé à $\{A_1, A_2, \dots, A_n\}$ et on note $U_{\{A_1, \dots, A_n\}}$ l'ensemble de tous les termes sans variables construits à partir du vocabulaire des formules de $\{A_1, A_2, \dots, A_n\}$. De manière à n'avoir jamais d'univers de Herbrand vide, lorsque ce vocabulaire ne contient pas de constante, on en introduit une.

Pour notre exemple :

$$U_{\{A_1, A_2, \dots, A_n\}} = \{a, f(a), f(f(a)), \dots, f(f(\dots a \dots)), \dots\}$$

On remarquera que dès qu'il y a un symbole fonctionnel l'univers de Herbrand est infini.

On appelle *atomes de Herbrand* associés à $\{A_1, A_2, \dots, A_n\}$ et on note $A_{\{A_1, A_2, \dots, A_n\}}$ l'ensemble de tous les atomes sans variables construits à partir du vocabulaire des formules de $\{A_1, A_2, \dots, A_n\}$, c'est-à-dire de toutes les formules de la forme $p(t_1, t_2, \dots, t_r)$ où p est un symbole de prédicat de l'une des formules A_1, A_2, \dots, A_n et où t_1, t_2, \dots, t_r sont des éléments de $U_{\{A_1, A_2, \dots, A_n\}}$.

Pour notre exemple :

$$A_{\{A_1, A_2, \dots, A_n\}} = \{p(a), r(a), q(a, a), p(f(a)), r(f(a)), q(a, f(a)), q(f(a), a), q(f(a), f(a)), p(f(f(a))), \dots\}$$

(On écrit d'abord les atomes n'utilisant que le terme a ; puis ceux utilisant en plus, le terme $f(a)$; puis ceux utilisant en plus, le terme $f(f(a))$; etc.).

On appelle *système de Herbrand* associé à $\{A_1, A_2, \dots, A_n\}$ et on note $S_{\{A_1, A_2, \dots, A_n\}}$ l'ensemble de toutes les formules obtenues à partir des A_i en remplaçant dans les A_i les variables par des éléments de l'univers de Herbrand. Lorsque les A_i sont des clauses, les formules du système de Herbrand sont toutes des disjonctions d'atomes de Herbrand et de négations d'atomes de Herbrand.

Pour notre exemple :

$$\begin{aligned} S_{\{A_1, A_2, \dots, A_n\}} = & \{\neg p(a) \vee q(a, f(a)), \neg p(a) \vee r(f(a)), p(a), \\ & \neg q(a, a), \neg p(f(a)) \vee q(f(a), f(f(a))), \neg p(f(a)) \vee r(f(f(a))), \\ & \neg q(a, f(a)), \neg q(f(a), a), \neg q(f(a), f(a)), \dots\} \end{aligned}$$

(Comme pour les atomes de Herbrand, on écrit d'abord les formules obtenues à partir de A_1, A_2, \dots, A_n en substituant a aux variables ; puis celles obtenues en substituant a ou $f(a)$ aux variables ; etc.).

Proposition 3. — (Théorème de Herbrand)

Soit $\{A_1, A_2, \dots, A_n\}$ un ensemble fini de clauses. Les trois affirmations suivantes sont équivalentes :

- (a) $\{A_1, A_2, \dots, A_n\}$ possède un modèle,
- (b) $\{A_1, A_2, \dots, A_n\}$ possède un modèle dont la base est l'univers de Herbrand $U_{\{A_1, A_2, \dots, A_n\}}$,
- (c) Le système de Herbrand $S_{\{A_1, A_2, \dots, A_n\}}$ considéré comme un ensemble de formules du calcul propositionnel dont les atomes sont $A_{\{A_1, A_2, \dots, A_n\}}$, possède un modèle.

Remarque :

Les modèles dont il est question au (c) s'appellent des *modèles de Herbrand* de $\{A_1, A_2, \dots, A_n\}$. Se donner un tel modèle c'est fixer une application i de $A_{\{A_1, A_2, \dots, A_n\}}$ dans $\{V, F\}$ telle que $i[f] = V$ pour tout $f \in S_{\{A_1, A_2, \dots, A_n\}}$. La donnée d'une telle application est équivalente à la donnée du sous-ensemble I des atomes At de $A_{\{A_1, A_2, \dots, A_n\}}$ tels que $i[At] = V$. C'est pourquoi on trouve parfois définie la notion de modèle de Herbrand comme sous-ensemble de $A_{\{A_1, A_2, \dots, A_n\}}$.

Démonstration :

(a) \Rightarrow (c)

Soit i un modèle de $\{A_1, A_2, \dots, A_n\}$.

Pour chaque formule B de $S_{\{A_1, A_2, \dots, A_n\}}$ on a : $i[B] = V$.

Donc, en posant :

$i'[At] = i[At]$ pour chaque atome de Herbrand, on définit une interprétation i' du calcul propositionnel basé sur les atomes de Herbrand associés à A_1, A_2, \dots, A_n qui est un modèle de $S_{\{A_1, A_2, \dots, A_n\}}$.

(c) \Rightarrow (b)

Soit i' un modèle de $S_{\{A_1, A_2, \dots, A_n\}}$.

A partir de i' , on définit un modèle i de $\{A_1, A_2, \dots, A_n\}$ dont la base est l'univers de Herbrand U en procédant comme suit :

- chaque symbole fonctionnel est interprété de la façon naturelle : $f(t_1, t_2, \dots, t_p) = f(t_1, t_2, \dots, t_p)$
- pour chaque symbole de prédicat, on pose :

$$\bar{p}(t_1, t_2, \dots, t_p) = i'[p(t_1, t_2, \dots, t_p)]$$

L'interprétation $i = [U, \bar{f}, \bar{g}, \dots, \bar{p}, \bar{r}, \dots]$ ainsi définie à partir de i' est un modèle de $\{A_1, A_2, \dots, A_n\}$ car i' était un modèle de $S_{\{A_1, A_2, \dots, A_n\}}$.

(b) \Rightarrow (a)

Evident car si $\{A_1, A_2, \dots, A_n\}$ admet un modèle dont la base est l'univers de Herbrand, alors $\{A_1, A_2, \dots, A_n\}$ admet un modèle. \square

Numérotons les formules de $S_{\{A_1, A_2, \dots, A_n\}} = \{F_0, F_1, \dots, F_m, \dots\}$

Proposition 4. — L'ensemble $\{A_1, A_2, \dots, A_n\}$ est inconsistante (c'est-à-dire n'admet pas de modèle) si et seulement si une des formules $F_0 \wedge F_1 \wedge \dots \wedge F_m$ est insatisfiable.

Démonstration :

Résulte immédiatement de la proposition 3 et du théorème de compacité du calcul propositionnel.

□

Illustrons le théorème de Herbrand en reprenant notre exemple.

Nous avons la numérotation :

$$\begin{array}{ll} F_0 = \neg p(a) \vee q(a, f(a)) & , \quad F_1 = \neg p(a) \vee r(f(a)), \\ F_2 = p(a) & , \quad F_3 = \neg q(a, a), \\ F_4 = \neg p(f(a)) \vee q(f(a), f(f(a))) & , \quad F_5 = \neg p(f(a)) \vee r(f(f(a))), \\ F_6 = \neg q(a, f(a)) & , \quad F_7 = \neg q(f(a), a), \text{ etc.} \end{array}$$

Pour savoir si l'ensemble de formules $\{A_1, A_2, A_3, A_4\}$ que nous avions au départ possède un modèle, nous étudions successivement $F_0, F_0 \wedge F_1, F_0 \wedge F_1 \wedge F_2, \dots, F_0 \wedge F_1 \wedge F_2 \wedge \dots \wedge F_n, \dots$ en cherchant à chaque étape à savoir si la formule est satisfiable ou non. Jusqu'à $F_0 \wedge F_1 \wedge F_2 \wedge \dots \wedge F_5$, on constate que oui (par exemple grâce à l'interprétation $i[p(a)] = V, i[q(a, f(a))] = V, i[r(f(a))] = V, i[q(a, a)] = F, i[p(f(a))] = F$).

Une fois arrivé à $F_0 \wedge F_1 \wedge F_2 \wedge \dots \wedge F_6$, on constate qu'il est impossible de construire un modèle car F_0, F_2 et F_6 ne peuvent pas être satisfaites simultanément. On en conclut qu'il n'existe pas de modèle de Herbrand, donc que $\{A_1, A_2, A_3, A_4\}$ n'a pas de modèle et donc (voir paragraphe 3) que la formule $T = \exists x \exists y q(x, y)$ est un théorème de la théorie dont les axiomes non logiques sont :

$$\forall x (p(x) \rightarrow \exists y (r(y) \wedge q(x, y)))$$

$$\exists x p(x).$$

5. — UN ALGORITHME DE DÉMONSTRATION AUTOMATIQUE

Le théorème de Herbrand ainsi que la proposition 4 suggère l'algorithme de démonstration automatique suivant :

- transformer $\{B_1, B_2, \dots, B_n, \neg B\}$ en un ensemble de clauses équivalent $\{A_1, A_2, \dots, A_p\}$ (voir paragraphe 3)
- si $S_{\{A_1, A_2, \dots, A_p\}}$ est fini alors faire
 - si $S_{\{A_1, A_2, \dots, A_p\}}$ est satisfiable alors faire
 - imprimer « B n'est pas conséquence de B_1, \dots, B_n »
 - arrêt
 - sinon faire
 - imprimer « B est conséquence de B_1, \dots, B_n »
 - arrêt
 - fin-de-si
 - sinon faire
 - $r := 0$
 - tant-que les r premières formules de $S_{\{A_1, A_2, \dots, A_p\}}$ sont satisfiables faire
 - $r := r + 1$
 - fin-de-tant-que
 - imprimer « B est conséquence de B_1, \dots, B_n »
 - fin-de-si
- arrêt.

Exception faite du cas où le système de Herbrand est fini, cet algorithme s'arrêtera si B est conséquence de $\{B_1, B_2, \dots, B_n\}$ et ne s'arrêtera jamais sinon. D'un point de vue théorique, il ne fait rien de plus que l'algorithme qui passerait en revue toutes les déductions à partir de B_1, B_2, \dots, B_n et ne s'arrêterait que lorsqu'il a trouvé une déduction de B . Malgré tout, d'un point de vue pratique, l'algorithme suggéré par le théorème de Herbrand est plus facile à mettre en œuvre et peut s'appliquer facilement à des problèmes simples. C'est en quelque sorte le premier algorithme praticable de démonstration automatique.

Pour vérifier si un ensemble fini de formules du calcul propositionnel est satisfiable la méthode la plus simple est de calculer la table de vérité de la conjonction des formules et de regarder si il y a au moins un V dans la colonne principale (ce V indique l'existence d'une interprétation qui satisfait toutes les formules de l'ensemble qu'on étudie, et montre donc que cet ensemble est satisfiable). Il existe bien d'autres méthodes et par exemple celle utilisée par Gilmore en 1960 pour le premier démonstrateur de théorèmes (basé sur le théorème de Herbrand) était celle de la mise sous forme normale disjonctive avec simplification.

La méthode des tables de vérité et celle de Gilmore sont très inefficaces, c'est pourquoi Davis et Putnam mirent au point une autre méthode (voir exercice).

Lorsqu'on travaille à la main on utilise souvent la méthode des arbres sémantiques que nous allons décrire à l'aide d'un exemple.

Imaginons qu'on cherche à savoir si la formule :

$$B = (\exists x \neg q(x) \rightarrow \forall y p(y))$$

est conséquence des deux formules :

$$B_1 = (\exists x p(x) \rightarrow \forall y p(y))$$

$$B_2 = \forall x (p(x) \vee q(x))$$

La négation de B est :

$$(\exists x \neg q(x) \wedge \exists y \neg p(y))$$

La mise sous forme de clauses de $\{B_1, B_2, \neg B\}$ donne :

$$A_1 = \neg p(x) \vee p(y)$$

$$A_2 = p(x) \vee q(x)$$

$$A_3 = \neg q(a)$$

$$A_4 = \neg p(b)$$

(la mise sous forme de Skolem a introduit deux constantes a et b).

L'univers de Herbrand est :

$$U_H = \{a, b\}$$

Les atomes de Herbrand sont :

$$A_H = \{q(a), p(a), q(b), p(b)\}$$

Le système de Herbrand est :

$$\begin{aligned} S_H = & \{\neg p(a) \vee p(a), \neg p(a) \vee p(b), \neg p(b) \vee p(b), \\ & \neg p(b) \vee p(a), p(a) \vee q(a), p(b) \vee q(b), \neg q(a), \\ & \neg p(b)\} = \{F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8\}. \end{aligned}$$

On dessine alors un arbre binaire, chaque niveau correspondant à un atome de Herbrand et chaque branche entre la racine et un nœud, correspondant à une interprétation de tous les atomes de Herbrand des niveaux par où passe cette branche.

La branche renforcée sur le schéma, par exemple, correspond à l'interprétation

$$i[q(a)] = F ; i[p(a)] = V ; i[q(b)] = V ; i[p(b)] = F$$

Dans cet arbre, on interrompt le développement dès que l'interprétation correspondant à la branche contredit une formule de S_H . On marque alors le nœud avec la formule qui a donné la contradiction.

Trois cas peuvent se produire :

- toutes les branches sont interrompues et marquées, cela signifie qu'il n'existe aucun modèle de S_H ;

- l'arbre est fini et l'une des branches n'a pas été marquée. L'interprétation correspondant à cette branche est un modèle de S_H qui est donc satisfiable ;
- l'arbre est infini il existe alors une branche infinie (lemme de König) à laquelle correspond une interprétation de S_H , qui est donc satisfiable.

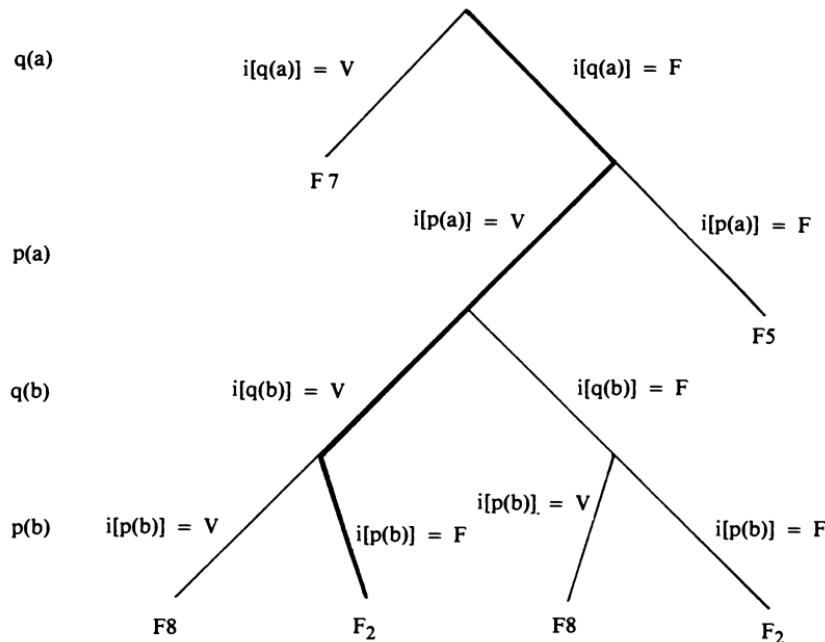


Fig. 6

Nous venons de voir un cas où l'arbre était fini et entièrement marqué ce qui donné comme conclusion que la formule B est conséquence de B_1, B_2 . Voyons maintenant un cas où l'arbre est infini.

$$B = \forall x p(x)$$

$$B_1 = \forall x p(f(x))$$

B est-il conséquence de B_1 ?

La mise sous forme de clauses donne :

$$\{ \neg p(a) \\ p(f(x))$$

On a donc :

$$U_H = \{a, f(a), f(f(a)), \dots\}$$

$$A_H = \{p(a), p(f(a)), p(f(f(a))), \dots\}$$

$$\begin{aligned} S_H &= \{\neg p(a), p(f(a)), p(f(f(a))), \dots\} \\ &= \{F_1, F_2, F_3, \dots\} \end{aligned}$$

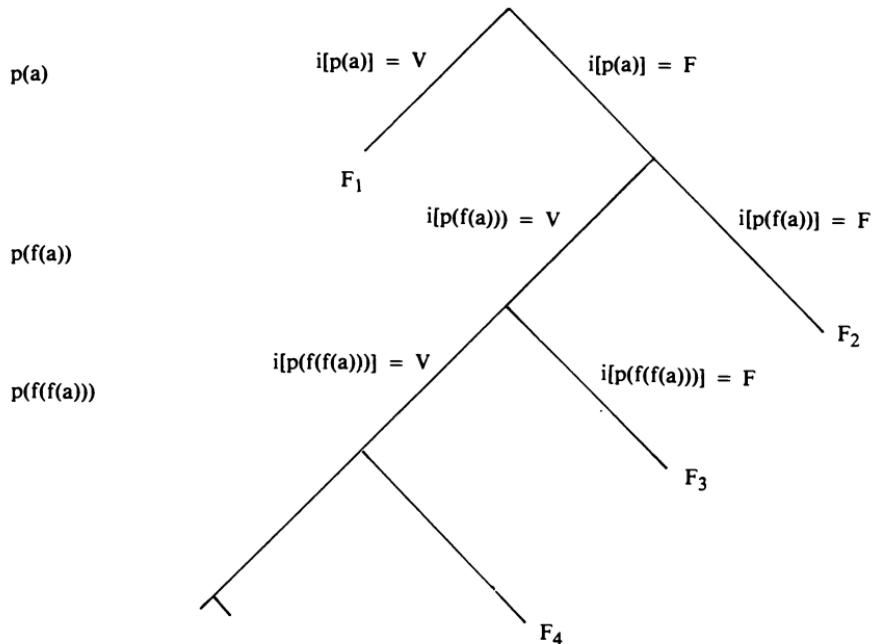


Fig. 7

L'arbre se poursuit indéfiniment et donc B n'est pas conséquence de B_1 . Remarquons d'ailleurs que la branche infinie définit une interprétation i de S_H qui elle-même donne une interprétation sur l'univers de Herbrand dans laquelle $i[B] = F$ et $i[B_1] = V$.

Cette interprétation sur U_H constitue donc un modèle de $\{B_1, \neg B\}$, c'est ce qu'on appelle un contre-modèle pour : $B_1 \vdash B$. Cette situation est générale : lorsqu'il y a des branches infinies chacune d'elles donne un contre-modèle à la conséquence qu'on cherchait à établir.

EXERCICES

Exercices sur la préparation des formules

- 1) En utilisant la méthode 1, puis la méthode 2 mettre sous forme prénexe les formules suivantes :
 - (a) $((\exists x p(x) \rightarrow (r(x) \vee \forall y p(y))) \wedge \forall x \exists y (r(y) \rightarrow p(x)))$
 - (b) $((p_1(x_1) \rightarrow \exists x_2 p_2(x_2)) \rightarrow \exists x_3 p_3(x_3)) \rightarrow \exists x_4 p_4(x_4))$
- 2) Réaliser la mise sous forme de clauses pour les ensembles F_0 suivants
 - (a) $F_0 = \{\forall x (p(x) \wedge \exists y q(x,y)), (\forall x p(x) \rightarrow \exists y r(y))\}$
 - (b) $F_0 = \{\exists y (r(y) \wedge \forall x \forall z p(x,y,z)), \forall x_1 \exists x_2 r(x_1 x_2), \forall x_1 \exists x_2 \forall x_3 \exists x_4 (p(x_1, x_2, x_3) \rightarrow r(x_1, x_4))\}$

Exercice sur le théorème de Herbrand

- 1) Après avoir transformé le problème sous bonne forme, utiliser le théorème de Herbrand pour savoir si la formule :
$$T = \forall x (r(x) \rightarrow p(x))$$
est conséquence des deux formules :
$$A_1 = \forall x (p(x) \rightarrow (q(x) \vee r(x)))$$
$$A_2 = \forall y (q(y) \rightarrow r(y)).$$
- 2) Même question avec :
$$T = \forall x (p(x) \rightarrow r(x))$$
$$A_1 = \forall x (p(x) \rightarrow (q(x) \vee r(x)))$$
$$A_2 = \forall x (q(x) \rightarrow r(x)).$$
- 3) Même question avec :
$$T = \forall z q(z)$$
$$A_1 = \forall x \exists y p(x,y)$$
$$A_2 = \forall x \forall y (p(x,y) \rightarrow q(x)).$$
- 4) Même question avec :
$$T = \exists z q(z)$$
$$A_1 = \forall x (p(x) \rightarrow q(x))$$
$$A_2 = \exists x p(x).$$

Exercice sur l'algorithme de démonstration du paragraphe 5

Soit $\{A_1, A_2, \dots, A_n\}$ un ensemble de clauses sans quantificateur et sans symbole fonctionnel.

Soit B une formule de la forme :

$$\forall x_1 \forall x_2 \dots \forall x_n \exists y_1 \exists y_2 \dots \exists y_m F$$

où F ne comporte aucun quantificateur ni symbole fonctionnel.

Montrer que l'algorithme du paragraphe 5 utilisé pour savoir si B est conséquence de $\{A_1, A_2, \dots, A_n\}$ s'arrête en un temps fini.

Exercice sur l'algorithme de démonstration du paragraphe 5

On se donne des formules A_1, A_2, \dots, A_n et B ayant chacune les propriétés suivantes :

- aucun symbole fonctionnel n'est utilisé
- les symboles \rightarrow , \leftrightarrow et \neg ne sont pas utilisés
- tous les quantificateurs sont des \forall ou tous sont des \exists .

Montrer que l'algorithme du paragraphe 5 utilisé pour savoir si B est conséquence de $\{A_1, A_2, \dots, A_n\}$ s'arrête en un temps fini.

Exercices sur les arbres sémantiques

1) Appliquer la méthode des arbres sémantiques pour savoir si la formule :

$$\forall y \exists x p(x,y)$$

est conséquence des formules :

$$\forall x \forall y (p(x,y) \rightarrow p(y,x))$$

$$\forall x \exists y p(x,y).$$

On s'arrangera pour ordonner les atomes de Herbrand de manière à avoir un arbre de taille raisonnable.

2) Reprendre les exercices sur le théorème de Herbrand et leur appliquer la méthode des arbres sémantiques.

Exercice sur la procédure de Davis et Putnam

Soit une clause :

$$At_1 \vee At_2 \vee \dots \vee At_n \vee \neg At'_1 \vee \neg At'_2 \vee \dots \vee \neg At'_m$$

Les atomes At_1, At_2, \dots, At_n sont appelés littéraux positifs de la clause, et les sous-formules $\neg At'_1, \neg At'_2, \dots, \neg At'_m$ sont appelés littéraux négatifs de la clause.

Si ℓ est un littéral d'une clause, on note ℓ^c le complémentaire de ℓ c'est-à-dire :

— At si $\ell = \text{At}$

At si $\ell = \neg \text{At}$

La clause vide notée \square est insatisfiable par convention⁽¹⁾. L'ensemble de clauses vide (c'est-à-dire l'ensemble de clauses ne comportant aucune clause) noté \emptyset par contre lui est satisfiable (car par exemple n'importe quelle interprétation en est un modèle).

Procédure de Davis et Putnam

Soit un ensemble de clauses sans variables.

Appliquer les règles suivantes jusqu'à ce qu'aucune ne puisse plus s'appliquer ; lorsque plusieurs règles peuvent s'appliquer on choisit celle de plus petit numéro.

Règle 1. — Enlever les tautologies (c'est-à-dire les clauses contenant un littéral et son complémentaire).

Règle 2. — Si l'une des clauses ne possède qu'un seul littéral ℓ , enlever toutes les clauses contenant ce littéral ℓ et enlever dans les autres clauses toutes les occurrences de ℓ^c .

Règle 3. — Si le littéral ℓ apparaît dans certaines clauses et que le littéral ℓ^c n'apparaît pas, enlever toutes les clauses contenant ℓ .

Règle 4. — Si une clause C a tous ses littéraux présents dans une clause C', enlever C'.

Règle 5. — Si le littéral ℓ ainsi que son complémentaire ℓ^c sont présents dans l'ensemble de clauses, remplacer celui-ci par deux ensembles de clauses :

- le premier obtenu en enlevant toutes les clauses contenant ℓ et toutes les occurrences de ℓ^c
- le second obtenu en enlevant toutes les clauses contenant ℓ^c et toutes les occurrences de ℓ .

(1) Cette convention n'est pas arbitraire, une façon par exemple de la justifier consiste à dire : puisqu'une clause est d'autant plus facile à satisfaire qu'elle a de nombreux littéraux, il est logique que la clause n'ayant aucun littéral soit impossible à satisfaire.

Exemple 1

En partant de l'ensemble de clauses :

$$\{ H \vee \neg H \vee G \vee A, G, \neg G \vee D \vee \neg E, \neg G \vee D \vee E, \neg G \vee A \vee \neg B, \\ \neg A \vee B, \neg A \vee B \vee C, A \vee \neg B \vee \neg C \}$$

on obtient, par la règle 1

$$\{ G, \neg G \vee D \vee \neg E, \neg G \vee D \vee E, \neg G \vee A \vee \neg B, \neg A \vee B, \\ \neg A \vee B \vee C, A \vee \neg B \vee \neg C \}$$

par la règle 2 :

$$\{ D \vee \neg E, D \vee E, A \vee \neg B, \neg A \vee B, \neg A \vee B \vee C, A \vee \neg B \vee \neg C \}$$

par la règle 3 :

$$\{ A \vee \neg B, \neg A \vee B, \neg A \vee B \vee C, A \vee \neg B \vee \neg C \}$$

par la règle 4 :

$$\{ A \vee \neg B, \neg A \vee B, A \vee \neg B \vee \neg C \}$$

par la règle 3 :

$$\{ A \vee \neg B, \neg A \vee B \}$$

par la règle 5 :

$$\{ B \}, \{ \neg B \}$$

par la règle 2 :

$$\emptyset, \emptyset$$

Remarquer que lorsqu'on enlève toutes les occurrences de ℓ à la clause ℓ on obtient la clause \square (ce qui n'est pas la même chose que supprimer la clause).

Exemple 2

$$\{ A \vee B, \neg A \vee B, \neg B \}$$

donne par la règle 2 (avec $\ell = \neg B$) :

$$\{ A, \neg A \}$$

puis par la règle 2 encore (avec $\ell = A$) :

$$\{ \square \}$$

1) Appliquer la procédure de Davis et Putnam à

$$S_1 = \{ A \vee \neg B \vee \neg C, \neg A \vee \neg B \vee C, A \vee B \vee \neg C \}$$

$$S_2 = \{ A \vee \neg B \vee \neg C \vee E, A \vee \neg B \vee C \vee \neg C \vee \neg E, \\ A \vee \neg E, B \vee \neg A \vee \neg E, A \vee B \vee \neg C \}$$

2) Montrer qu'un ensemble de clause est satisfiable si et seulement si l'un des ensembles de clauses déduits par la procédure de Davis et Putnam est satisfiable.

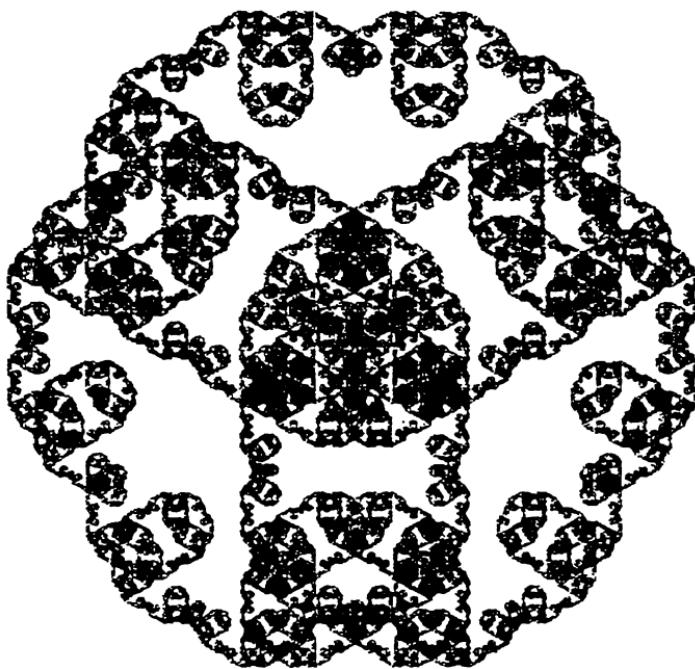
3) Appliquer le résultat de la question 2 aux ensembles de clauses de l'exemple 1, de l'exemple 2 et à S_1 et S_2 . Retrouver vos conclusions par une autre méthode.

BIBLIOGRAPHIE

- [CHA 73] CHANG C. L. et LEE R. C.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, New York, 1973.
{Souvent présentés de manières différentes ce livre et les suivants contiennent des résultats équivalents à ceux de ce chapitre}
- [KLE 71] KLEENE S. C.
Logique Mathématique.
Librairie Armand Colin, Paris, 1971.
(Traduction de : Mathematical logic, John Wiley and Sons, New York, 1967).
- [KRE 67] KREISEL G. et KRIVINE J. L.
Eléments de Logique Mathématique : Théorie des Modèles.
Dunod, Paris, 1967.
- [LOV 78] LOVELAND D. W.
Automated Theorem Proving : A Logical Basis.
North-Holland Publishing Company, Amsterdam, 1978.
- [NIL 80] NILSSON N.
Principles of Artificial Intelligence.
Tioga Publishing Company. Palo Alto, 1980.
- [RIC 83] RICH E.
Artificial Intelligence.
Mac Graw-Hill Inc., New York, 1983.
- [ROB 79] ROBINSON J. A.
Logic : Form and Function. The mechanization of Deductive Reasoning.
Edinburgh University Press, Edinburgh, 1979.
- [YAS 71] YASUHARA A.
Recursive Function Theory and Logic.
Academic Press, New York, 1971.

CHAPITRE 7

UNIFICATION, RÉSOLUTIONS



1. — INTRODUCTION

Nous avons vu que le problème de savoir si une formule B résultait d'un ensemble fini de formules A_1, A_2, \dots, A_n pouvait se ramener au problème de la consistance d'un ensemble de formules sans quantificateur. Les algorithmes de résolution sans variable et de résolution avec variables par l'unification ont pour but de traiter le plus efficacement possible ce problème. Il s'agit là de méthodes absolument essentielles qui, sous des formes diverses, interviennent dans presque tous les démonstrateurs de théorèmes et en particulier dans Prolog que nous étudierons au chapitre 8.

2. — RÉSOLUTION SANS VARIABLE

Un ensemble $\{A_0, A_1, \dots, A_n\}$ de formules du calcul propositionnel étant donné, on cherche à savoir s'il est satisfiable ou non (c'est-à-dire s'il existe ou non une interprétation i telle que $i[A_0] = V, \dots, i[A_n] = V$).

La procédure naturelle et simple qui consiste à faire la table de vérité de $A_0 \wedge A_1 \wedge \dots \wedge A_n$ puis à regarder s'il y a au moins un V dans la colonne principale de cette table, assure d'obtenir une réponse en un temps fini. La durée du calcul est de l'ordre de 2^n (où n est le nombre d'atomes intervenant dans les A_i). Cette procédure est donc vite inutilisable⁽¹⁾.

La méthode que nous présentons (appelée résolution sans variable) est plus rapide, très simple et généralisable aux formules du calcul des prédictats du premier ordre (paragraphe 4).

Nous en donnons une présentation en terme de système formel de façon à mettre en évidence sa simplicité vis-à-vis du système formel du calcul propositionnel présenté au chapitre 4.

(1) En fait le problème de la satisfiabilité d'un ensemble de clauses appartient à la classe des problèmes NP-complets (voir [LEW 80] [SIM 84]). Ce résultat dû à Cook (1973) peut s'interpréter (si, ainsi que tout le monde le pense, les problèmes NP-complets sont réellement non polynomiaux) comme indiquant qu'aucun algorithme ne pourra être efficace sur tous les problèmes de satisfiabilité d'ensembles de clauses. Ceci n'empêche pas qu'il y a des algorithmes pour ce problème qui sont plus efficaces que d'autres, ni même que certains puissent être efficaces dans presque tous les cas.

Soit le système formel RSV (résolution sans variable) défini par :

$$\Sigma_{RSV} = \{p_0, p_1, \dots, p_n, \dots\} \cup \{\neg, v\}$$

F_{RSV} = ensemble de toutes les clauses construites à partir des variables propositionnelles $p_0, p_1, \dots, p_n, \dots$; c'est-à-dire ensemble de toutes les formules de la forme :

$$p_{i_1} \vee p_{i_2} \vee \dots \vee p_{i_n} \vee \neg p_{j_1} \vee \neg p_{j_2} \vee \dots \vee \neg p_{j_m}$$

Les sous-formules p_{i_h} et $\neg p_{j_k}$ sont appelées littéraux de la clause. On ne considérera que les clauses sans répétition, c'est-à-dire ne contenant pas deux fois le même p_{i_h} ou deux fois le même p_{j_k} .

Lorsque f et g sont deux clauses, on notera $f \vee g$ la clause dont les littéraux sont ceux de f et de g écrits sans répétition. On ne tiendra pas compte de l'ordre des littéraux dans une clause.

La clause n'ayant aucun littéral est notée \square . Elle fait partie de F_{RSV} , et aucune interprétation ne peut la satisfaire.

$$A_{RSV} = \emptyset. Il n'y a aucun axiome.$$

$$R_{RSV} = \{\text{cut}\}. Il n'y a qu'une seule règle d'inférence :$$

$$f \vee p_i, g \vee \neg p_i \vdash_{\text{cut}} f \vee g$$

Exemple :

On a :

$$p_0 \vee p_1 \vee p_2, \neg p_0 \vee p_1 \vee p_2, \neg p_1 \vee p_2 \vdash_{RSV} p_2$$

grâce à la déduction :

$$f_1 : p_0 \vee p_1 \vee p_2 \quad (H1)$$

$$f_2 : \neg p_0 \vee p_1 \vee p_2 \quad (H2)$$

$$f_3 : p_1 \vee p_2 \quad (\text{cut avec } f_1 \text{ et } f_2)$$

$$f_4 : \neg p_1 \vee p_2 \quad (H3)$$

$$f_5 : p_2 \quad (\text{cut avec } f_3 \text{ et } f_4)$$

On a :

$$p_0 \vee p_1, p_0 \vee \neg p_1, \neg p_0 \vee p_1, \neg p_0 \vee \neg p_1 \vdash_{RSV} \square$$

grâce à la déduction :

$$f_1 : p_0 \vee p_1 \quad (H1)$$

$$f_2 : p_0 \vee \neg p_1 \quad (H2)$$

$$f_3 : p_0 \quad (\text{cut avec } f_1, f_2)$$

- | | |
|--------------------------------|------------------------|
| $f_4 : \neg p_0 \vee p_1$ | (H3) |
| $f_5 : \neg p_0 \vee \neg p_1$ | (H4) |
| $f_6 : \neg p_0$ | (cut avec f_4, f_5) |
| $f_7 : \square$ | (cut avec f_3, f_6) |

Les arbres de dérivation (voir chapitre 3) associés à ces deux déductions sont :

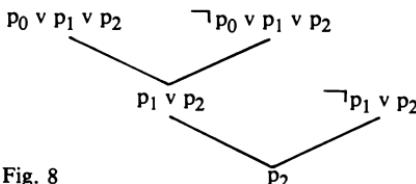


Fig. 8

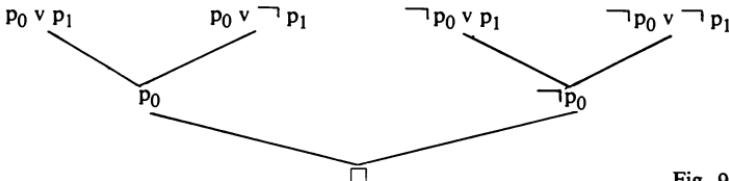


Fig. 9

Remarque :

La règle d'inférence de résolution sans variable généralise la règle d'inférence du modus ponens.

En effet, exprimée avec des clauses, celle-ci est :

$$\neg p_i \vee p_j, p_i \vdash \neg p_j \\ \text{m.p.}$$

ce qui est bien le cut avec $f = p_j$ et $g = \square$.

La règle d'inférence de résolution sans variable généralise aussi le modus tollens. En effet, celui-ci est :

$$p_i \rightarrow p_j, \neg p_j \vdash \neg p_i \\ \text{m.t.}$$

ce qui, écrit avec des clauses, donne :

$$\neg p_i \vee p_j, \neg p_j \vdash \neg p_i$$

ce qui est bien le cut avec : $f = \square$ et $g = \neg p_i$,

On vérifie sans difficulté que la règle d'inférence de transitivité de l'implication :

$$p_i \rightarrow p_j, p_j \rightarrow p_k \vdash p_i \rightarrow p_k$$

t.i.

est aussi un cas particulier de la règle de résolution sans variable.

De même encore, une fois écrit avec des clauses, on voit que plusieurs applications de la résolution sans variable donnent les règles suivantes :

$$p_{i_1}, p_{i_2}, \dots, p_{i_n}, (p_{i_1} \wedge p_{i_2} \wedge \dots \wedge p_{i_n}) \rightarrow p_j \vdash p_j$$

$$p_j \rightarrow (p_{i_1} \vee p_{i_2} \vee \dots \vee p_{i_n}), \neg p_{i_1}, \neg p_{i_2}, \dots, \neg p_{i_n} \vdash \neg p_j$$

C'est justement cette grande généralité qui fait la force de la résolution. Avec elle, on peut faire du chaînage avant, du chaînage arrière ou les deux.

Proposition 1. — Un ensemble F de clauses (du calcul propositionnel) est insatisfiable si et seulement si :

$$F \vdash \square$$

RSV

Démonstration :

(a) On peut supposer que F est fini, car quand F est infini :

• F est insatisfiable si et seulement si il existe un sous-ensemble fini de F qui est insatisfiable (théorème de compacité).

• $F \vdash \square$ si et seulement si il existe un sous-ensemble fini F' de F tel

que $F' \vdash \square$

(évident car une déduction n'utilise qu'un nombre fini d'hypothèses).

(b) Si $F \vdash \square$ alors F est insatisfiable.

En effet, si $F \vdash \square$, c'est que F contient p_i et $\neg p_i$ pour un certain i, et donc que F est insatisfiable.

(c) Si F est satisfiable, alors il est impossible que :

$$F \vdash \square$$

RSV

Cela résulte de (b) et du lemme suivant :

Lemme 1. — Si $F_1 \vdash_{\text{cut}} C$ alors $F_1 \models C$.

Démonstration du lemme 1 :

F_1 contient deux clauses $f \vee p_i, g \vee \neg p_i$ et $C = f \vee g$.

Soit i une interprétation satisfaisant F_1 :

- si $i[p_i] = V$, alors $i[\neg p_i] = F$, donc :
 $i[g] = V$, donc $i[f \vee g] = V$.
- si $i[p_i] = F$ alors $i(f) = V$ donc :
 $i[f \vee g] = V$.

□

(d) Il reste à établir que si F est insatisfiable, alors $F \vdash_{\overline{\text{RSV}}} \square$

On dit que F est un ensemble *insatisfiable minimal* si tout sous-ensemble strict de F est satisfiable.

Il est clair que tout ensemble de clauses F insatisfiable admet un, et le plus souvent plusieurs, sous-ensembles insatisfiables minimaux.

Pour conclure, il suffit donc d'établir le lemme suivant :

Lemme 2. — Si F est un ensemble de clauses insatisfiable, fini et minimal, alors :

$F \vdash_{\overline{\text{RSV}}} \square$.

Démonstration du lemme 2 :

On appelle *nombre de littéraux en excès* de l'ensemble de clauses F , et on note $\text{nle}(F)$, l'entier défini par :

$$\text{nle}(F) = (\text{nombre de littéraux de toutes les clauses}) - (\text{nombre de clauses})$$

Par exemple :

$$\text{nle}(\{p_1 \vee p_2, p_2 \vee \neg p_3\}) = 4 - 2 = 2.$$

La démonstration du lemme 2 se fait par récurrence sur $\text{nle}(F)$.

- Si $\text{nle}(F) = 0$ et que F est insatisfiable et minimal, F est de la forme $\{p_i, \neg p_i\}$ et donc $F \vdash_{\overline{\text{RSV}}} \square$.

- Supposons que pour tout F insatisfiable minimal tel que $nle(F) < n_0$ on ait :
 $F \vdash \square$, et montrons que si F est insatisfiable minimal tel que
RSV
 $nle(F) = n_0$, alors on a aussi $F \vdash \square$.

On peut, bien sûr, supposer $n_0 > 0$. Soit donc une clause $f \in F$ ayant plus d'un littéral. Soit ℓ un littéral de f . Notons g la clause obtenue en supprimant ℓ dans f .

Considérons $F_1 = (F - \{f\}) \cup \{g\}$.

F_1 est insatisfiable (car une interprétation qui satisfait F_1 satisfait F).

Soit F_1^* un sous-ensemble de F_1 insatisfiable et minimal,

Puisque $nle(F_1^*) < n_0$, d'après l'hypothèse de récurrence, on sait qu'il existe une déduction f_1, f_2, \dots, f_k de \square à partir de F_1^* .

Si aucun f_i n'est g alors f_1, f_2, \dots, f_k est une déduction de \square à partir de F et donc, on a terminé.

On peut donc supposer que l'un des f_i est g . En ajoutant ℓ à chaque clause f_i intervenant après g dans f_1, f_2, \dots, f_k , on obtient une déduction f'_1, f'_2, \dots, f'_k de ℓ à partir de F .

Soit maintenant $F_2 = (F - \{f\}) \cup \{\ell\}$. Comme pour F_1 on voit que F_2 est insatisfiable. Soit F_2^* un sous-ensemble minimal insatisfiable de F_2 . On a nécessairement $\ell \in F_2^*$ (car sinon $F_2^* \subset F - \{f\}$ ce qui est satisfiable puisque F est minimal insatisfiable).

Puisque $nle(F_2^*) < n_0$, il existe une déduction de \square à partir de F_2^* . Soit g_1, g_2, \dots, g_h cette déduction. On a donc la situation suivante :

$F \vdash \ell$ grâce à f'_1, f'_2, \dots, f'_k
RSV

$F \cup \{\ell\} \vdash \square$ grâce à g_1, g_2, \dots, g_h
RSV

ce qui permet d'écrire :

$F \vdash \square$ grâce à $f'_1, f'_2, \dots, f'_k, g_1, g_2, \dots, g_h$
RSV

\square

Remarques :

- Si le système formel RSV permet de savoir pour tout ensemble F de clauses s'il est insatisfiable ou pas, ce système formel est malgré tout moins puissant que le système formel P_0 du chapitre 3,
 - d'une part il n'accepte que des clauses,

- d'autre part, alors que pour P_0 :

$$(h_1, h_2, \dots, h_n \vdash_{P_0} t \Leftrightarrow (h_1 \wedge h_2 \wedge \dots \wedge h_n \models t))$$

ceci n'est plus vrai pour RSV car, par exemple :

$$p_0 \models p_0 \vee p_1 \text{ bien qu'on n'ait pas } p_0 \vdash_{RSV} p_0 \vee p_1.$$

- 2) La procédure de DAVIS-PUTNAM (voir exercice chapitre 6) est, en général, plus rapide que la résolution sans variable pour savoir si un ensemble de clauses est insatisfiable ou non.

3. — UNIFICATION

L'algorithme décrit au chapitre précédent pour déterminer si une formule B est conséquence d'un ensemble $\{A_1, A_2, \dots, A_n\}$ de formules est inutilisable dès qu'on doit traiter des problèmes comportant plus d'une dizaine de formules. Cela est dû à ce qu'on essaie de trouver une contradiction avec des atomes de Herbrand obtenus en remplaçant les variables par des termes de façon absolument non systématique. Par exemple, il est maladroit d'essayer de savoir si $\neg p(x) \vee q(y)$, $p(f(z)) \vee q(t)$ et $\neg q(v)$ va donner une contradiction en remplaçant x par a , y par a , v par a , z par $f(a)$, t par $f(a)$; alors qu'essayer de trouver une contradiction en remplaçant x par $f(a)$, y par a , z par a , t par a , v par a (ce qui donne $\neg p(f(a)) \vee q(a)$, $p(f(a)) \vee q(a)$, $\neg q(a)$) conduit immédiatement à la contradiction voulue et montre que les 3 formules en question n'ont pas de modèle.

Cette idée de faire coïncider des atomes les uns avec les autres de manière à trouver plus rapidement des contradictions (c'est-à-dire à montrer l'insatisfiabilité des formules traitées) c'est l'idée de base de la résolution avec variables, que nous verrons au paragraphe 4.

La mise en coïncidence des atomes par un bon choix des termes substitués aux variables, c'est l'unification. Il s'agit là d'une idée qui est assez ancienne en logique mathématique puisqu'elle apparaît dans la thèse de Herbrand, en 1930. Cependant, son utilisation comme élément fondamental d'un algorithme de démonstration automatique de théorèmes et comme outil privilégié de la programmation logique est due à J.-A. ROBINSON (1965).

(a) Composants de substitution ; substitutions

On appelle *composant de substitution* toute expression de la forme $(x \mid t)$ où x est une variable et t un terme quelconque du calcul des prédictats.

Si A est une formule du calcul des prédictats, on note $(x|t)A$ la formule obtenue en remplaçant toutes les occurrences libres de x dans A par t . Par exemple :

$$\begin{aligned} & (x|f(y,g(a))) (p(u) \rightarrow r(x)) \\ &= (p(u) \rightarrow r(f(y,g(a)))) \end{aligned}$$

Une substitution est une application σ de F_{Pr} dans F_{Pr} de la forme :

$$\sigma : A \mid \rightarrow c_1 c_2 \dots c_k A$$

où c_1, c_2, \dots, c_k sont des composants de substitution.

La suite finie c_1, c_2, \dots, c_k est appelée décomposition de la substitution Φ ; ce que l'on note :

$$\Phi = [c_1 c_2 \dots c_k].$$

La substitution « identique » sera notée ϵ ou $[]$.

L'ensemble des substitutions sera noté Sub.

Remarque :

Le renommage préliminaire que nous faisions au chapitre 5 (voir page 99) n'est plus nécessaire ici, car nous n'envisageons que des formules sans quantificateur.

Exemples :

$\sigma = [(x|f(a))(y|f(x))]$ est une substitution et :

$$\begin{aligned} \sigma p(x,y) &= (x|f(a))(y|f(x)) p(x,y) \\ &= (x|f(a)) p(x,f(x)) \\ &= p(f(a),f(f(a))) \end{aligned}$$

On remarquera que :

- Il n'est pas vrai en général que $[c_1 c_2] = [c_2 c_1]$.

Par exemple :

$$\begin{aligned} (y|f(x)) (x|f(a)) p(x,y) &= (y|f(x)) p(f(a),y) \\ &= p(f(a)|f(x)) \\ &\neq \sigma p(x,y) \end{aligned}$$

- La décomposition d'une substitution en composants de substitutions n'est pas unique en général :

$$\begin{aligned} [(x|y)(z|y)] &= [(z|y)(x|y)] \\ [(x|t)(y|t)(z|t)] &= [(x|t)(y|x)(z|x)] \end{aligned}$$

Soit $S = \{A_1, A_2, \dots, A_n\}$ un ensemble fini de formules atomiques du calcul des prédictats.

On appelle *unificateur de S* toute substitution σ telle que :

$$\sigma A_1 = \sigma A_2 = \dots = \sigma A_n.$$

Pour $S_1 = \{A_1, A_2, A_3\}$ avec :

$$A_1 = p(x, z) \quad A_2 = p(f(y), g(a)) \quad A_3 = p(f(u), z)$$

la substitution $\sigma_1 = [(x|f(u))(y|u)(z|g(a))]$ est un unificateur car :

$$\sigma_1 A_1 = \sigma_1 A_2 = \sigma_1 A_3 = p(f(u), g(a)).$$

La substitution $\sigma_2 = [(u|f(a))]$ σ_1 , est aussi un unificateur de S_1 car :

$$\sigma_2 A_1 = \sigma_2 A_2 = \sigma_2 A_3 = p(f(f(a)), g(a)).$$

Il est d'ailleurs bien clair que toute substitution de la forme $\alpha\sigma$ sera un unificateur de S .

Il se peut qu'aucun unificateur n'existe. Par exemple, pour :

$$S_2 = \{p(x, y), r(f(t), y)\}$$

c'est évident car les prédicts sont différents. Pour :

$$S_3 = \{p(x, f(x)), p(f(y), y)\}$$

il n'y a pas non plus d'unificateur.

Notons U_S l'ensemble des unificateurs de S . Si σ est une substitution de U_S telle que :

$$\forall \alpha \in U_S \exists \beta \in \text{Sub} : \alpha = \beta\sigma$$

on dit que σ est *un plus grand unificateur de S* (ou unificateur le plus général de S).

La substitution σ_1 est un plus grand unificateur pour S_1 . Ce n'est pas le seul,

$$\sigma_3 = [(x|f(v))(y|v)(z|g(a))(u|v)]$$

en est un autre et on a :

$$\sigma_1 = (v|u) \sigma_3$$

$$\sigma_3 = (u|v) \sigma_1$$

Intuitivement, un plus grand unificateur d'un ensemble de formules (quand il en existe) est une substitution aussi générale que possible faisant coïncider les formules.

Algorithme d'unification de deux atomes A et B

- $\Theta := \epsilon$
- Tant que $\Theta A \neq \Theta B$, faire
 - déterminer le symbole le plus à gauche de ΘA qui soit différent du symbole de même rang de ΘB .

- déterminer t_1, t_2 les sous-termes de ΘA et de ΘB qui commencent à ce symbole.
- si « aucun n'est une variable » ou « l'un est une variable contenue dans l'autre »
 - alors imprimer « A et B ne sont pas unifiables » ; arrêt.
 - sinon faire
 - déterminer x une variable parmi t_1, t_2
 - déterminer t celui de t_1, t_2 qui n'est pas x
 - $\Theta := (x|t)\Theta$
 - fin-de-si
- fin-de-tant-que
- imprimer Θ « est le plus grand unificateur de A et B »
- arrêt.

Exemple 1 :

$\Theta(A)$	$\Theta(B)$	
$p(x, f(x), a)$	$p(u, w, w)$	$\Theta = \epsilon$
↑	↑	
$p(x, f(x), a)$	$p(u, w, w)$	$\Theta = [(x u)]$
↑	↑	
$p(u, f(u), a)$	$p(u, w, w)$	$\Theta = [(w f(u))(x u)]$
↑	↑	
$p(u, f(u), a)$	$p(u, f(u), f(u))$	échec car ni a ni $f(u)$ ne sont des variables
↑	↑	

Exemple 2 :

$p(x, f(g(x)), a)$	$p(b, y, z)$	$\Theta = \epsilon$
↑	↑	
$p(x, f(g(x)), a)$	$p(b, y, z)$	$\Theta = [(x b)]$
↑	↑	
$p(b, f(g(b)), a)$	$p(b, y, z)$	$\Theta = [(y f(g(b)))(x b)]$
↑	↑	
$p(b, f(g(b)), a)$	$p(b, f(g(b)), z)$	$\Theta = [(z a)(y f(g(b)))(x b)]$
↑	↑	

succès, A et B sont unifiables et un plus grand unificateur est :
 $[(z|a)(y|f(g(b)))(x|b)]$.

Exemple 3 :

$p(x, f(x))$	$p(f(y), y)$	$\Theta = \epsilon$
↑	↑	
$p(x, f(x))$	$p(f(y), y)$	$\Theta = [(x f(y))]$
↑	↑	
$p(f(y), f(f(y)))$	$p(f(y), y)$	échec car y est une variable du terme $t = f(f(y))$
↑	↑	

On trouvera dans le livre de LOVELAND une preuve de l'algorithme d'unification de deux atomes. Cette preuve établit en particulier que si deux atomes admettent un unificateur alors il existe un plus grand unificateur pour ces deux atomes et que tous les plus grands unificateurs ne diffèrent que par des changements de noms de variables. Cette propriété est vraie même pour plus de deux atomes et un algorithme général d'unification peut être défini à partir de l'algorithme d'unification de deux atomes.

Algorithme d'unification de A_1, A_2, \dots, A_n .

- Pour i allant de 1 à $n-1$ faire
 - si « $\sigma_{i-1} \sigma_{i-2} \dots \sigma_1 A_i$ et $\sigma_{i-1} \sigma_{i-2} \dots \sigma_1 A_{i+1}$ sont unifiables »
 - alors déterminer σ_i un plus grand unificateur
 - sinon imprimer « A_1, A_2, \dots, A_n ne sont pas unifiables » ; arrêt.
 - fin' de - si
 - fin - de - pour
- imprimer $\sigma_{n-1} \dots \sigma_1$ « est un plus grand unificateur de A_1, A_2, \dots, A_n »
- arrêt.

Exemple :

$$S = \{p(x, y), p(f(z), x), p(w, f(x))\}$$

- $\sigma_1 = (y | f(z) \ (x | f(z))$ est un plus grand unificateur de $p(x, y), p(f(z), x)$
- $\sigma_1 p(f(z), x)) = p(f(z), f(z))$
- $\sigma_1 p(w, f(x)) = p(w, f(f(z)))$
- échec car $p(f(z), f(z))$ et $p(w, f(f(z)))$ ne sont pas unifiables.

Il faut bien prendre garde, lorsqu'on unifie par cette méthode, à ne pas oublier d'appliquer $\sigma_{i-1} \sigma_{i-2} \dots \sigma_1$ aux atomes non encore unifiés avant de les unifier à leur tour. Dans l'exemple précédent, si on avait oublié d'appliquer σ_1 à $p(w, f(x))$, la deuxième étape d'unification aurait été possible (car $p(f(z), f(z))$ et

$p(w, f(x))$ s'unifient en $p(f(z), f(z))$) et on aurait donc conclu faussement que S était unifiable.

Un grand nombre de variantes et d'améliorations de l'algorithme d'unification de Robinson ont été imaginées et sont utilisées. Ces algorithmes ont, en particulier, pour but d'augmenter la rapidité de l'unification et de diminuer l'espace nécessaire pour représenter les atomes. (Voir bibliographie).

4. — RÉSOLUTION AVEC VARIABLES

Ce que nous avons vu au § 2 pour la résolution sans variable s'étend au calcul des prédictats.

La méthode de résolution obtenue fournit divers algorithmes analogues à celui du chapitre précédent pour déterminer si une formule résulte d'un ensemble fini donné de formules. Mais, cette fois-ci, l'efficacité des méthodes permet une utilisation réelle de ces algorithmes pour la démonstration automatique de théorèmes.

De très nombreuses stratégies différentes d'utilisation de la résolution existent. Nous en présentons quelques-unes au paragraphe 5.

Soit le système formel RAV (résolution avec variables) défini par :

$\Sigma_{RAV} = \Sigma_{P_1}$ (l'alphabet du système formel RAV est le même que celui du calcul des prédictats du premier ordre du chapitre 5).

$F_{RAV} =$ ensemble des clauses du calcul des prédictats du premier ordre, c'est-à-dire l'ensemble de toutes les formules de la forme :

$$a_1 \vee a_2 \vee \dots \vee a_n \vee \neg b_1 \vee \neg b_2 \vee \dots \vee \neg b_m$$

où les a_i et les b_j sont des formules atomiques.

On fait les mêmes conventions que pour F_{RSV} .

$A_{RAV} = \emptyset$ Il n'y a aucun axiome.

$R_{RAV} = \{\text{rés., dim.}\}$
(résolution et diminution)

Il y a deux règles d'inférence que nous décrivons maintenant en détail.

Règle d'inférence de résolution

$$\frac{f, g \vdash h}{\text{res}}$$

si et seulement si :

f est de la forme $a \vee f_1$

g est de la forme $\neg b \vee g_1$

h est de la forme $\sigma(\Theta f_1 \vee g_1)$

où Θ est une substitution telle que Θf et g n'aient aucune variable libre commune (Θ est appelée substitution de renommage),

et où σ est un plus grand unificateur de Θa et b.

La clause h est appelée résolvante de f et g.

Exemple 1 :

$p(x,c) \vee r(x), \neg p(c,c) \vee q(x)$

$\vdash r(c) \vee q(x)$

res

avec : $a = p(x,c)$, $b = p(c,c)$

$\Theta = (x|y)$ $\Theta a = p(y,c)$

$\sigma = (y|c)$.

Exemple 2 :

$p(x,f(x)), \neg p(a,y) \vee r(f(y))$

$\vdash r(f(f(a)))$

res

avec : $a = p(x,f(x))$, $b = p(a,y)$

$\Theta = \epsilon$, $\sigma = (y|f(a))(x|a)$.

Règle d'inférence de diminution

$f \vdash h$
dim

si et seulement si :

f est de la forme $a \vee b \vee f_1$

h est de la forme $\sigma a \vee \sigma f_1$

où σ est un plus grand unificateur de a et de b.

Exemple :

$p(x,g(y)) \vee p(i(c),z) \vee r(x,y,z)$

$\vdash r(i(c),y,g(y)) \vee p(i(c),g(y))$

dim

avec $\sigma = (x|i(c))(z|g(y))$

Exemple de déduction dans le système formel RAV.

On cherche à déduire \square des trois formules :

$$\neg p(x) \vee p(f(x)), p(a), \neg p(f(z))$$

(car, comme au § 2, le fait de pouvoir déduire \square , signifie que les formules considérées n'ont pas de modèles, et donc que la formule $\exists z p(f(z))$ résulte des formules :

$$\forall x(p(x) \rightarrow p(f(x))) \text{ et } \exists y p(y)).$$

Voici cette déduction :

$f_1 : \neg p(x) \vee p(f(x))$	(hypothèse 1)
$f_2 : p(a)$	(hypothèse 2)
$f_3 : p(f(a))$	(résolution à partir de f_1 et f_2 avec $\Theta = (x a)$)
$f_4 : \neg p(f(z))$	(hypothèse 3)
$f_5 : \square$	(résolution à partir de f_3 et f_4 avec $\Theta = (z a)$)

En regardant cet exemple, on peut comprendre en quoi la résolution est naturelle.

L'hypothèse : $\neg p(x) \vee p(f(x))$ qui doit se comprendre comme $\forall x (p(x) \rightarrow p(f(x)))$, donne en particulier que : $p(a) \rightarrow p(f(a))$.

Avec l'hypothèse $p(a)$, on en déduit $p(f(a))$ (car si $p(a) \rightarrow p(f(a))$ est vrai et que $p(a)$ est vrai, alors $p(f(a))$).

L'hypothèse $\neg p(f(z))$ qui doit se comprendre comme $\forall z \neg p(f(z))$, donne $\neg p(f(a))$ et donc on obtient une contradiction, c'est-à-dire \square .

Proposition 2. — Un ensemble F de clauses (du calcul des prédictats) est insatisfiable si et seulement si :

$$\begin{array}{c} F \vdash \square \\ \text{RAV} \end{array} .$$

Esquisse de démonstration :

(Voir le livre de LOVELAND pour une démonstration détaillée).

On procède comme pour la proposition 1 :

- (a) On se ramène au cas où F est fini.
- (b) On montre que si $F \vdash \square$ en une étape, alors F est insatisfiable.

$$\begin{array}{c} F \vdash \square \\ \text{RAV} \end{array}$$
- (c) On montre que :
 - si $F_1 \vdash F_2$ et si M est un modèle de F_1 , alors M est aussi un modèle de F_2 (ce qui est immédiat par récurrence). On a donc

$$\begin{array}{c} F \vdash \square \\ \text{RAV} \end{array}$$
implique F insatisfiable.

(d) Pour montrer que :

F insatisfiable implique F $\vdash \square$
RAV

- On utilise le théorème de Herbrand qui nous fournit un ensemble insatisfiable fini F' de formules sans variable obtenues par substitution de termes à des variables dans des formules de F .
- On construit une déduction de \square à partir de F' dans RSV grâce à la proposition 1.
- on traduit cette déduction en une déduction de RAV.

□

Exemples de déduction dans le système formel RAV.

Exemple 1 (D'après [LOV 78]) :

On considère les axiomes suivants de la théorie des groupes.

Pour éviter d'utiliser le prédicat d'égalité, on introduit un prédicat $p(x,y,z)$ à 3 variables qui s'interprète comme $x.y = z$.

$A_1 \bullet \forall x \forall y \exists z p(x,y,z)$
(« opération interne »)

$A_2 \bullet \forall x \forall y \forall z \forall u \forall v \forall w$
 $(p(x,y,u) \wedge p(y,z,v)) \rightarrow (p(x,v,w) \longleftrightarrow p(u,z,w))$
(associativité)

$A_3 \bullet \exists x [\forall y p(x,y,y) \wedge \forall z \exists u p(u,z,x)]$
(existence de l'élément neutre à gauche et existence d'un inverse à droite).

Posons-nous maintenant la question de l'existence d'un inverse à gauche :

$B \bullet \exists x [\forall y p(x,y,y) \wedge \forall z \exists u p(z,u,x)]$

La négation de B est :

$\forall x [\exists y \neg p(x,y,y) \vee \exists z \forall u \neg p(z,u,x)]$

On considère donc l'ensemble de formules $\{A_1, A_2, A_3, \neg B\}$ et on l'écrit sous forme de clauses.

On obtient :

$a_1 \quad p(x,y,f(x,y))$

$a_2 \quad \neg p(x,y,u) \vee \neg p(y,z,v) \vee \neg p(x,v,w) \vee p(u,z,w)$

$a_3 \quad \neg p(x,y,u) \vee \neg p(y,z,v) \vee \neg p(u,z,w) \vee p(x,v,w)$

- $a_4 \quad p(e, y, y)$
 $a_5 \quad p(g(z), z, e)$
 $a_6 \quad \neg p(x, h(x), h(x)) \vee \neg p(k(x), u, x)$

Déduction de la clause vide :

- $f_1 : p(x, y, f(x, y)) \quad (a_1)$
 $f_2 : \neg p(x, y, u) \vee \neg p(y, z, v) \vee \neg p(x, v, w) \vee p(u, z, w) \quad (a_2)$
 $f_3 : \neg p(x, y, u) \vee \neg p(y, z, v) \vee \neg p(u, z, w) \vee p(x, v, w) \quad (a_3)$
 $f_4 : p(e, y, y) \quad (a_4)$
 $f_5 : p(g(z), z, e) \quad (a_5)$
 $f_6 : \neg p(x, h(x), h(x)) \vee \neg p(k(x), z, x) \quad (a_6)$
 $f_7 : \neg p(k(e), z, e) \quad (r., f_4, f_6)$
 $f_8 : \neg p(x, y, k(e)) \vee \neg p(y, z, v) \vee \neg p(x, v, e) \quad (r., f_2d, f_7)$
 $f_9 : \neg p(g(v), y, k(e)) \vee \neg p(y, z, v) \quad (r., f_5, f_8c)$
 $f_{10} : \neg p(g(v), e, k(e)) \quad (r., f_4, f_9b)$
 $f_{11} : \neg p(g(v), y, u) \vee \neg p(y, z, e) \vee \neg p(u, z, k(e)) \quad (r., f_3d, f_{10})$
 $f_{12} : \neg p(g(v), y, e) \vee \neg p(y, k(e), e) \quad (r., f_4, f_{11}c)$
 $f_{13} : \neg p(g(v), g(k(e)), e) \quad (r., f_5, f_{12}b)$
 $f_{14} : \square \quad (r., f_5, f_{13})$

Le graphe de dérivation associé est le suivant :

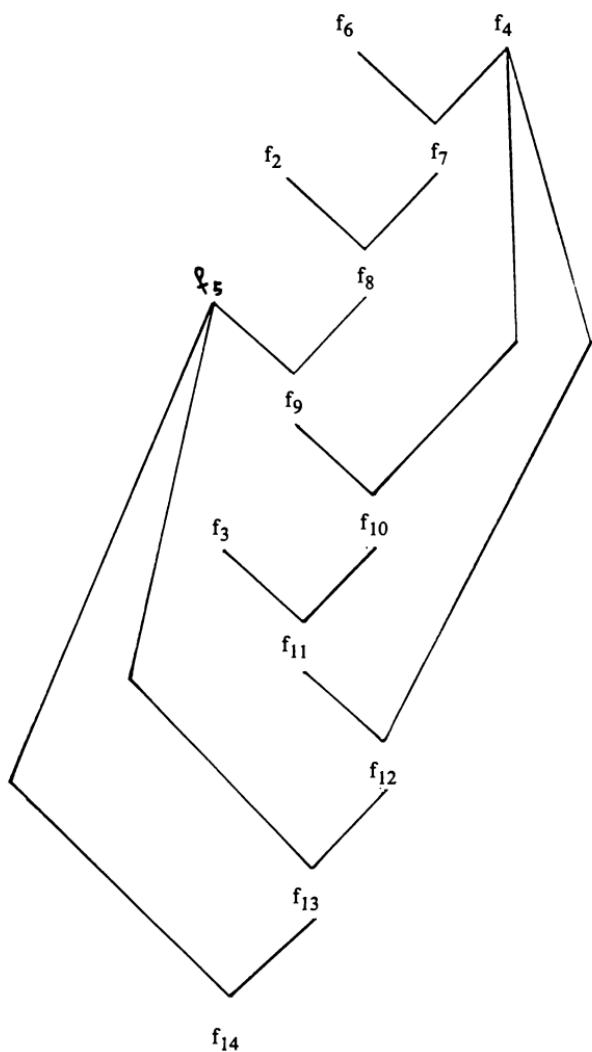


Fig. 10

Exemple 2 :

Nous reprenons l'exemple utilisé au chapitre 6.

$$\begin{aligned} A_1 &= \neg p(x) \vee q(x, f(x)) \\ A_2 &= \neg p(x) \vee r(f(x)) \\ A_3 &= p(a) \\ A_4 &= \neg q(x, y) \end{aligned}$$

Déduction de la clause vide :

$$\begin{array}{lll} f_1 : & \neg p(x) \vee q(x, f(x)) & (A_1) \\ f_2 : & p(a) & (A_3) \\ f_3 : & q(a, f(a)) & (\text{res. avec } f_1 \text{ et } f_2) \\ f_4 : & \neg q(x, y) & (A_4) \\ f_5 : & \square & (\text{res. avec } f_3 \text{ et } f_4) \end{array}$$

Exemple 3 :

Pour illustrer le progrès réalisé par la résolution par rapport à la méthode de Herbrand, considérons les deux clauses suivantes (n est un entier fixé) :

$$\begin{aligned} &p(a, x_2, f(x_2), x_4, f(x_4), \dots, x_{2n}, f(x_{2n})) \\ &\neg p(x_1, f(x_1), x_3, f(x_3), x_5, \dots, f(x_{2n-1}), x_{2n+1}) \end{aligned}$$

La résolution donne la clause vide en une étape.

La méthode de Herbrand, en considérant que les formules sont classées selon le principe utilisé au chapitre 6 (toutes celles utilisant a, puis celles utilisant a et f(a), etc.) ne donnerait la contradiction cherchée qu'avec $F_0 \wedge F_1 \wedge \dots \wedge F_m$

pour $m = 2 \times (2n)^{2n}$ au mieux.

En prenant $n = 20$ on obtient une valeur de m qui dépasse tout ce qu'on peut espérer traiter avec des machines, même dans un avenir lointain.

Exemple 4 :

On voudrait à partir des énoncés suivants :

- 1 • pour tout crime, il y a quelqu'un qui l'a commis,
- 2 • seuls les gens malhonnêtes commettent des crimes,
- 3 • ne sont arrêtés que les gens malhonnêtes,
- 4 • les gens malhonnêtes arrêtés ne commettent pas de crimes,
- 5 • il y a des crimes,

établir que :

- B Il y a des gens malhonnêtes non arrêtés.

Introduisons les prédictats suivants :

- | | |
|---------|--------------------|
| ar(y) | : y est arrêté |
| ma(y) | : y est malhonnête |
| co(y,x) | : y commet x |
| cr(x) | : x est un crime |

La traduction des énoncés 1, 2, 3, 4, 5 et de la négation de B donne :

- 1 $\forall x (cr(x) \rightarrow \exists y co(y,x))$
- 2 $\forall y \forall x ((cr(x) \wedge co(y,x)) \rightarrow ma(y))$
- 3 $\forall y (ar(y) \rightarrow ma(y))$
- 4 $\forall y ((ma(y) \wedge ar(y)) \rightarrow \neg \exists x (cr(x) \wedge co(y,x)))$
- 5 $\exists x cr(x)$
- 6 $\neg \exists y (ma(y) \wedge \neg ar(y))$

La mise sous forme de clauses donne :

- a1 : $\neg cr(x) \vee co(f(x),x)$
a2 : $\neg cr(x) \vee \neg co(y,x) \vee ma(y)$
a3 : $\neg ar(y) \vee ma(y)$
a4 : $\neg ma(y) \vee \neg ar(y) \vee \neg cr(x) \vee \neg co(y,x)$
a5 : $cr(a)$
a6 : $\neg ma(y) \vee ar(y)$

Pour établir que B est bien conséquence des énoncés 1, 2, 3, 4, 5, on construit une déduction de la clause vide dans RAV.

On commence par $f_1 = a_1$, $f_2 = a_2$, $f_3 = a_3$, $f_4 = a_4$, $f_5 = a_5$, $f_6 = a_6$ et on continue par :

- | | | |
|---|---|-------------------------------|
| $f_7 : co(f(a),a)$ | [rés. f_5, f_1] | quelqu'un a commis le crime |
| $f_8 : \neg cr(a) \vee ma(f(a))$ | [rés. f_7, f_2] | ce quelqu'un est malhonnête |
| $f_9 : ma(f(a))$ | [rés. f_8, f_5] | |
| $f_{10} : \neg ma(f(a)) \vee \neg ar(f(a))$ | [rés. f_7, f_4]
$\vee \neg cr(a)$ | |
| $f_{11} : \neg ma(f(a)) \vee \neg ar(f(a))$ | [rés. f_{10}, f_5] | |
| $f_{12} : \neg ar(f(a))$ | [rés. f_{11}, f_9] | ce quelqu'un n'est pas arrêté |
| $f_{13} : \neg ma(f(a))$ | [rés. f_{12}, f_6] | |
| $f_{14} : \square$ | [rés. f_{13}, f_6] | ce qui contredit 6 |

Le graphe de dérivation est le suivant :

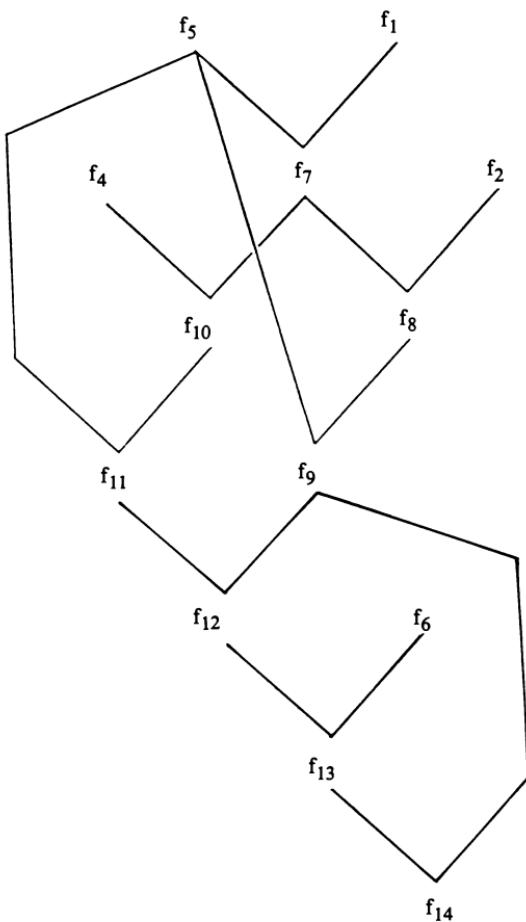


Fig. 11

Exemple 5 :

La règle de diminution n'a pour l'instant jamais été utilisée, cependant elle est bien indispensable.

Considérons en effet l'ensemble de clauses S :

$$S = \{p(x) \vee p(y), \neg p(x) \vee \neg p(y)\}$$

Il est impossible d'en déduire la clause vide en utilisant seulement la règle d'inférence de résolution (car la résolution appliquée à deux clauses de deux littéraux donne une clause à deux littéraux).

Par contre avec la règle de diminution, on a la déduction suivante :

- $f_1 : p(x) \vee p(y)$ (hypothèse)
 $f_2 : p(x)$ (diminution à partir de f_1)
 $f_3 : \neg p(x) \vee \neg p(y)$ (hypothèse)
 $f_4 : \neg p(y)$ (résolution avec f_2 et f_3)
 $f_5 : \square$ (résolution avec f_2 et f_4)

Au paragraphe suivant, nous verrons que dans certaines situations, la règle de diminution peut être inutile.

5. — STRATÉGIES D'UTILISATION DE LA RÉSOLUTION

Il y a deux techniques fondamentalement différentes d'utilisation de la résolution dans un système de démonstration automatique ; chacune de ces techniques donnant lieu à de nombreuses variantes.

Il y a ce qu'on peut appeler :

- (a) les techniques de gestion d'ensembles de clauses
et
(b) les techniques d'exploration de l'arbre des déductions.

Pour le premier type de techniques, on part de l'ensemble de clauses S_0 dont on cherche à savoir s'il est satisfiable ou non, on l'enrichit (par résolution et diminution) et on le simplifie. Tout cela géré par une stratégie qu'on détermine à l'avance et jusqu'à obtenir la clause vide ou jusqu'à savoir qu'on ne peut pas l'obtenir.

Pour le second type de techniques, on « se déplace » dans l'arbre des déductions dont on a condamné certaines branches jusqu'à atteindre la clause vide ou jusqu'à savoir qu'elle n'y est pas.

- (a) *Les techniques de gestion d'ensembles de clauses*

- (a1) *La stratégie de saturation*

De toutes les stratégies, c'est sans doute la plus simple et la plus naturelle :

- On part d'un ensemble de clauses S_0 dont on cherche à savoir s'il est satisfiable ou non.
- On effectue toutes les résolutions possibles et toutes les diminutions possibles à partir des clauses de S_0 , ce qui donne de nouvelles clauses qu'on ajoute à S_0 pour obtenir S_1 .

- On obtient de même S_2 à partir de S_1
etc.
- On s'arrête dès que \square est présent dans S_i ou dès que : $S_i = S_{i+1}$.

Exemple :

$$\begin{aligned}S_0 &= \{A, \neg A \vee B, \neg B \vee A, \neg B\} \\S_1 &= \{B, B \vee \neg B, \neg A \vee A, \neg A, A\} \cup S_0 \\S_2 &= \{A, \square, \dots\} \cup S_1\end{aligned}$$

Il s'agit là d'une technique totalement inefficace car les ensembles S_i augmentent de façon exponentielle. Cependant la proposition 2 garantit que si S_0 est insatisfiable, alors on le saura en un temps fini (car la clause vide \square est dans l'un des S_i). On dit que cette stratégie est correcte (elle ne conduit jamais à considérer comme insatisfiable un ensemble de clauses qui est satisfiable) et complète (si S_0 est insatisfiable, on finit par le savoir). D'un point de vue théorique, on ne gagne rien par rapport à l'algorithme du chapitre 6, car si S_0 est satisfiable, il est possible que S_{i+1} soit différent de S_i pour tout i , et donc que l'algorithme ne s'arrête jamais. C'est le cas par exemple avec : $S_0 = \{p(a), \neg p(x) \vee p(f(x))\}$.

(a 2) Stratégies de saturation avec simplification

Avant de définir cette stratégie, précisons les notions de clauses tautologiques et de clauses subsumées.

On vérifie facilement que les clauses qui sont des tautologies (c'est-à-dire vraies dans toute interprétation) sont les clauses de la forme :

$$A \vee \neg A \vee C$$

avec A un atome et C une clause.

Par exemple :

$$p(f(x)) \vee q(x) \vee \neg r(x) \vee \neg p(f(x))$$

est une tautologie.

On dit que la clause C *subsume* la clause D s'il existe une substitution σ telle que :

$$D = \sigma C \vee f \text{ avec } f \text{ une clause.}$$

Par exemple :

$$\begin{aligned}p(x) &\text{ subsume } p(f(a)) \vee q(a) \\p(x) \vee q(a) &\text{ subsume } p(a) \vee q(a) \vee r(f(x)) \vee r(b)\end{aligned}$$

Si C subsume D , on dit que D est *subsumée* par C . On a l'équivalence suivante :

[l'ensemble de clauses S est satisfiable]

si seulement si :

[l'ensemble de clauses S' obtenu à partir de S en enlevant toutes les tautologies et toute clause de S subsumée par une autre clause de S , est satisfiable]

On voit en effet immédiatement que si i est une interprétation qui satisfait S' , i satisfait aussi S , et réciproquement.

D'où l'idée très simple de la stratégie de saturation avec simplification :

On procède comme en (a1) mais, avant chaque étape de saturation, on simplifie S_i en enlevant toutes ses tautologies et toutes les clauses C qui sont subsumées par une autre clause. Il s'agit là d'une stratégie correcte (elle ne donne la clause vide que si S_0 est insatisfiable) et complète (si S_0 est insatisfiable elle donne la clause vide) (voir [CHA 73]).

Exemple :

$$S_0 = \{r(a) \vee p(a), p(x) \vee \neg r(b), r(b), \neg p(b), r(a) \vee r(c) \vee \neg r(a)\}$$

$$S'_0 = \{r(a) \vee p(a), p(x) \vee \neg r(b), r(b), \neg p(b)\}$$

$$S_1 = \{r(a) \vee p(a), p(x) \vee \neg r(b), r(b), \neg p(b), p(x), \neg r(b)\}$$

$$S'_1 = \{r(b), \neg p(b), p(x), \neg r(b)\}$$

$$S_2 = \{\square, \dots\}$$

Ce calcul est beaucoup plus simple que celui qu'aurait donné la stratégie de saturation de base.

Indiquons encore que l'utilisation de prédictats évaluables permet d'autres simplifications (voir [NIL 80]).

(a3) Stratégies « préférence des clauses simples »

Plutôt que de construire d'un seul coup toutes les clauses qui peuvent être obtenues à partir de S_i , on peut n'en construire qu'une seule, bien choisie, la rajouter, puis recommencer.

Il y a autant de manières qu'on veut de « bien choisir » la clause qu'on ajoute pour passer de S_i à S_{i+1} .

On peut procéder au hasard (« random strategy »), on peut choisir celle obtenue en premier, quand on a défini un ordre sur toutes les résolutions et diminutions possibles (par exemple en numérotant les clauses, ou en numérotant les prédictats).

Mais parmi ce type de stratégies, la plus naturelle est sans doute celle qui pour passer de S_i à S_{i+1} ajoute la clause la plus courte (en nombre de littéraux) parmi celles qu'on peut obtenir avec S_i . Ceci dans l'idée que, puisqu'on cherche à obtenir la clause vide, plus les clauses obtenues sont simples plus on se rapproche du but.

Exemple :

$$S_0 = \{r(x) \vee \neg p(x), \neg r(b), \neg r(c) \vee q(x) \vee r(f(a)), p(b)\}$$

$$S_1 = S_0 \cup \{\neg p(b)\}$$

$$S_2 = S_1 \cup \{\square\}$$

Cette stratégie utilisée de manière brutale n'est pas complète, en effet prenons l'exemple :

$$S_0 = \{\neg p(x) \vee p(f(x)), p(a), \neg p(a) \vee \neg p(b) \vee \neg p(c), p(b), p(c)\}$$

on a successivement :

$$S_1 = S_0 \cup \{p(f(a))\}$$

$$S_2 = S_1 \cup \{p(f(f(a)))\}$$

etc.

On n'obtient jamais la clause vide, alors que les quatre dernières clauses de S_0 , à condition de passer par une clause à 2 littéraux, donnent la clause vide en trois étapes.

Pour rendre cette stratégie complète, on peut par exemple introduire de temps en temps une étape de saturation.

Parmi d'autres variantes possibles, citons encore la stratégie de « préférence des clauses unités » où on opérera en priorité toutes les résolutions faisant intervenir des clauses à un seul littéral (clauses appelées « clauses unités »). Là encore cette stratégie utilisée brutalement n'est pas complète.

Plusieurs stratégies peuvent être combinées ; par exemple en introduisant des étapes de simplification, ou en adaptant la stratégie au problème traité. Toutefois il faut prendre garde que la complétude des stratégies (c'est-à-dire l'assurance d'arriver à la clause vide quand S_0 est insatisfiable) risque de ne plus être vérifiée, et que, même lorsqu'elle l'est, cela est parfois difficile à démontrer (voir [CHA 73], [LOV 78]).

(b) Techniques d'exploration de l'arbre des déductions.

On appelle arbre des déductions de l'ensemble de clauses \mathcal{C} , l'arbre (le plus souvent infini) dont la racine est \bullet et dont toute branche partant de la racine passe par des nœuds f_1, f_2, \dots, f_n constituant une déduction au sens du système formel RAV (ou RSV).

Par exemple, avec $\mathcal{C} = \{A, \neg A \vee B, \neg B\}$
 on a l'arbre de déduction suivant :

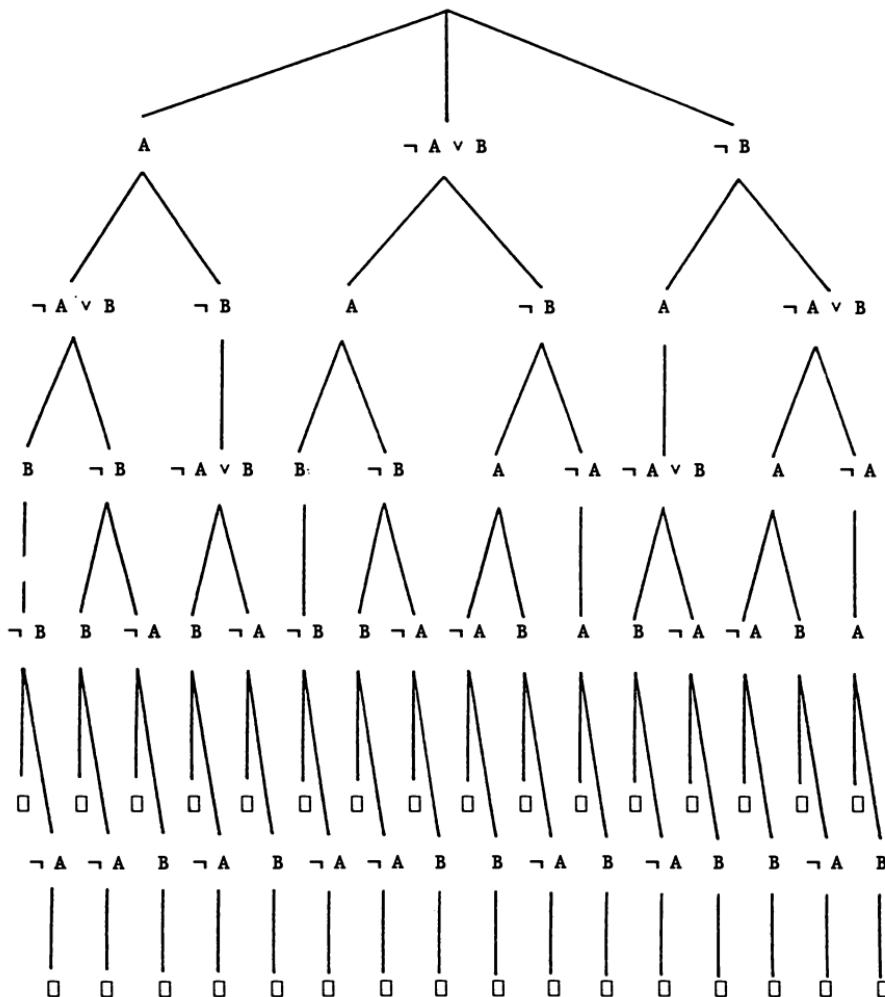


Fig. 12

Chacune des branches est bien une déduction. Par exemple, la branche la plus à gauche est :

- | | |
|-----------------------|------------------------------------|
| $f_1 : A$ | (introduction d'hypothèse) |
| $f_2 : \neg A \vee B$ | (introduction d'hypothèse) |
| $f_3 : B$ | (résolution entre f_1 et f_2) |
| $f_4 : \neg B$ | (introduction d'hypothèse) |
| $f_5 : \square$ | (résolution entre f_3 et f_4) |

On s'est limité aux déductions qui ne contiennent pas, deux fois la même formule, car si dans une déduction, il y a deux fois la même formule c'est que la déduction est inutilement longue.

Cette limitation ne fait perdre aucun théorème, de même que la limitation qui ne considèrerait que les déductions commençant par toutes les introductions d'hypothèses dans un ordre fixé, et qui aurait donné l'arbre de déduction suivant :

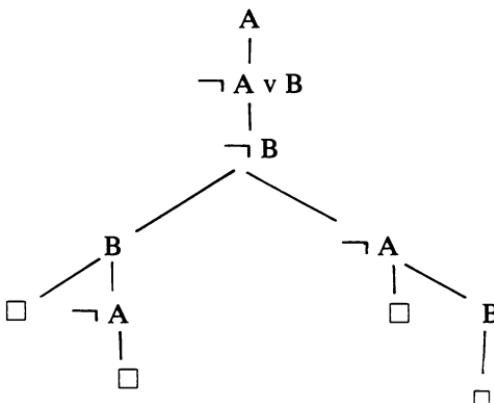


Fig. 13

De telles limitations qui ne font perdre aucun théorème (et surtout qui ne font pas perdre la clause vide si elle est présente) permettent de définir des stratégies d'exploration de l'arbre des déductions.

De manière plus précise, une stratégie d'exploration de l'arbre des déductions est fondée sur deux choix :

- le choix d'un sous-arbre de l'arbre des déductions, ce sous-arbre étant défini par une certaine limitation sur les déductions qu'on prend en compte.
- le choix d'un procédé d'exploration de ce sous-arbre.

Pour le second des choix qui détermine la façon dont le sous-arbre sélectionné est parcouru, nous n'envisagerons que les deux méthodes bien connues :

- exploration « en profondeur d'abord avec retours en arrière (backtracking) »
- exploration « en largeur d'abord »

« En profondeur d'abord
avec retours en arrière »

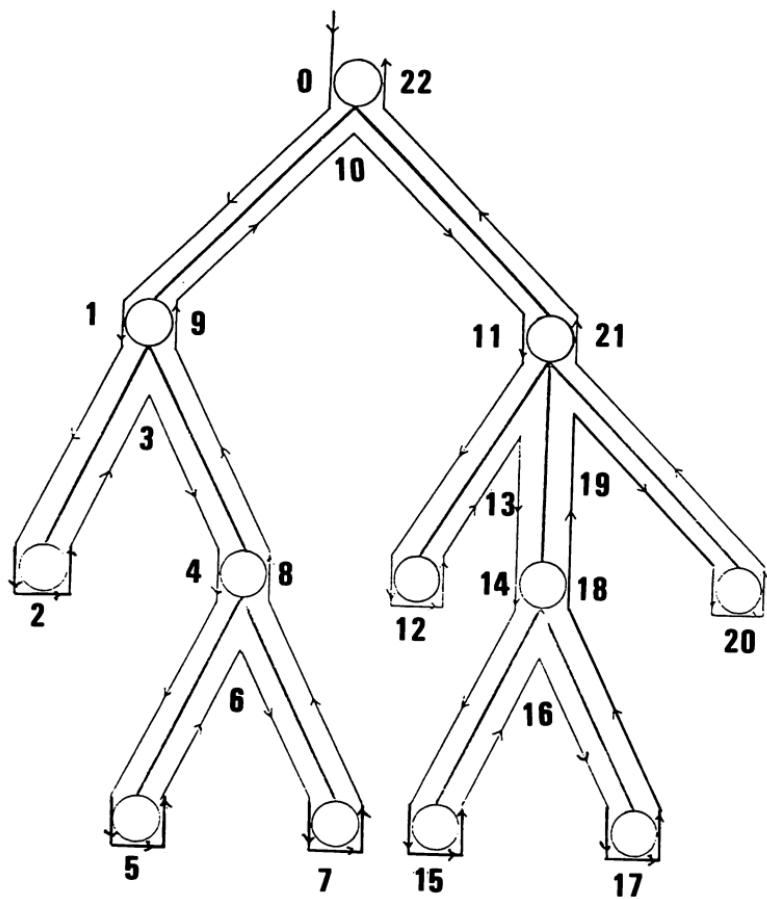


Fig. 14

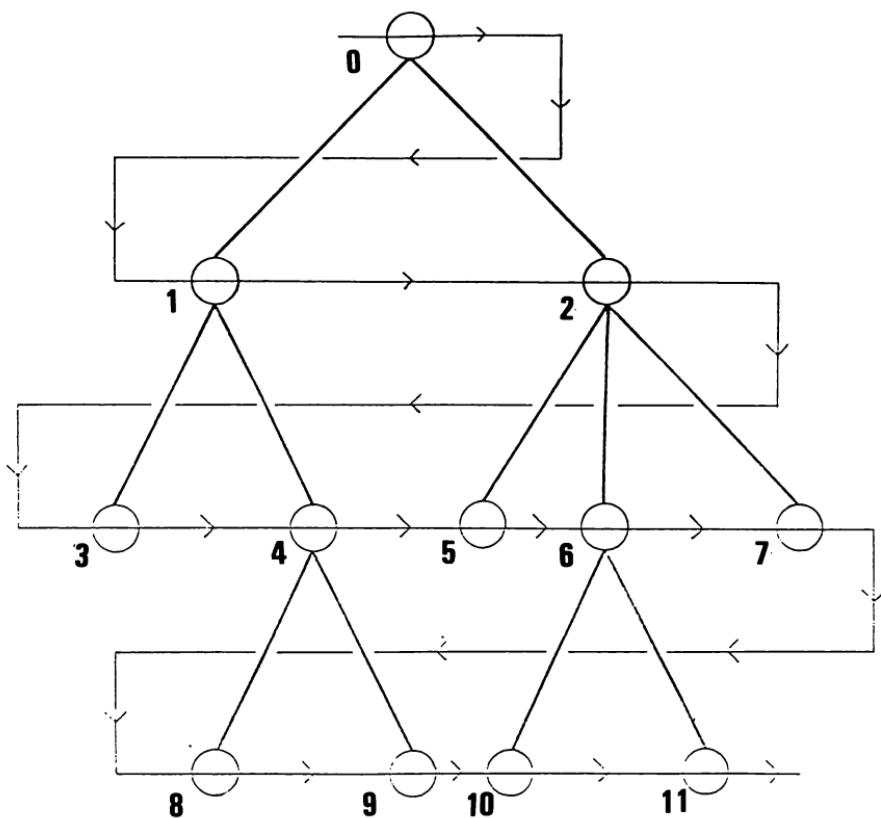


Fig. 15

Exploration d'arbres

L'avantage de la stratégie d'exploration « en largeur d'abord » est qu'elle permet (lorsqu'on a un arbre dont chaque nœud n'a qu'un nombre fini de fils, ce qui est toujours le cas ici) de parcourir l'arbre en entier même s'il est infini (ce qui est fréquent pour nous).

L'avantage principal de la stratégie d'exploration « en profondeur d'abord avec retours en arrière » est que lorsqu'on recherche dans l'arbre un objet qui y est plusieurs fois, mais assez profondément (ce qui est le type des situations que nous avons) on peut trouver rapidement cet objet. Un autre avantage de cette stratégie est qu'elle se programme assez facilement et ne nécessite pas trop d'espace mémoire.

L'inconvénient principal de la stratégie d'exploration « en largeur d'abord » est que lorsqu'on a un arbre dont chaque nœud a un grand nombre de fils (on parle de facteur de branchement important) alors il est très long de descendre profondément dans l'arbre.

L'inconvénient principal de la stratégie d'exploration « en profondeur d'abord avec retours en arrière » est qu'elle se perd dans la première branche infinie rencontrée et, qu'en conséquence, elle ne permet l'exploration entière que des arbres finis.

Si chaque nœud a k fils, il y a k nœuds à la profondeur 1, k^2 nœuds à la profondeur 2, ..., k^n nœuds à la profondeur n .

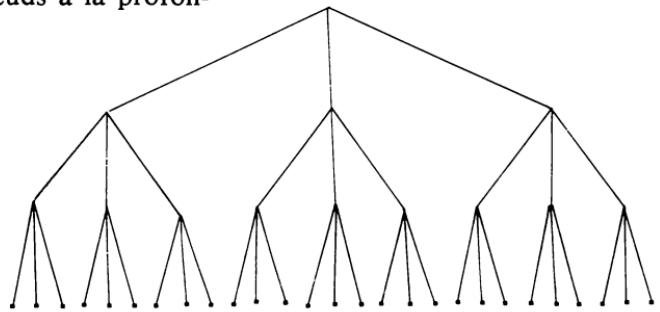


Fig. 16

Si l'arbre comporte une branche infinie, l'exploration « en profondeur d'abord avec retours en arrière » s'y enfonce sans passer partout.

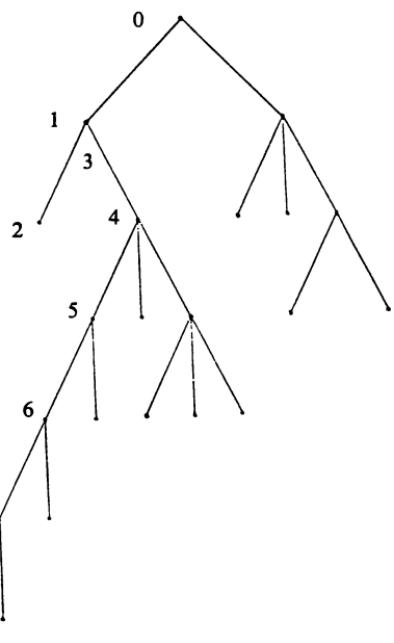


Fig. 17

Lorsqu'on a un ensemble fini de clauses $\mathcal{C} = \{c_0, c_1, \dots, c_n\}$, nous avons dit précédemment que nous ne perdions rien en nous limitant aux seules déductions commençant par $c_0 c_1 \dots c_n$ dans cet ordre. C'est pourquoi à partir de maintenant, nous ne considérerons que de telles déductions, pour lesquelles nous utiliserons la notation $\mathcal{C} f_0 f_1 \dots f_m$ pour désigner la déduction $c_0 c_1 \dots c_n f_0 f_1 \dots f_m$.

Pour reprendre l'exemple envisagé plus haut où $\mathcal{C} = \{A, \neg A \vee B, \neg B\}$, cette notation nous permet de représenter l'arbre des déductions par :

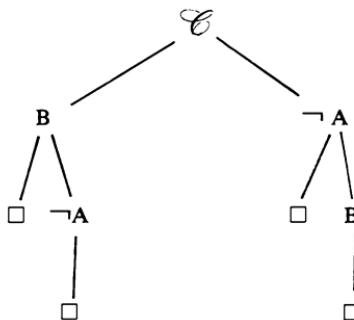


Fig. 18

Toutes les stratégies que nous envisagerons à partir de maintenant sont correctes (elles découvrent la clause vide seulement si \mathcal{C} est insatisfiable) pour la simple raison qu'elles explorent un sous-arbre de l'arbre de toutes les déductions et que si \mathcal{C} est satisfiable l'arbre tout entier (et donc le sous-arbre) ne contient pas la clause vide. Le problème de la complétude par contre se posera à chaque fois.

(b1) Stratégies générales

Les stratégies générales sont analogues à la stratégie de saturation pour ce type de méthode. Elles consistent simplement à explorer l'arbre de toutes les déductions.

Si l'exploration a lieu avec la stratégie « en largeur d'abord » on obtient une stratégie complète et correcte. Si l'exploration a lieu avec la stratégie « en profondeur d'abord avec retours en arrière » on obtient une stratégie correcte mais non complète.

Ces stratégies sont très inefficaces, encore plus que la stratégie par saturation car elles considèrent comme différentes des déductions qui ne changent que par l'ordre des formules.

(b2) Stratégies linéaires

On appelle déduction linéaire de racine c_0 à partir de l'ensemble de clauses (contenant c_0) toute déduction $\mathcal{C} f_0 f_1 \dots f_m$ telle que :

- f_0 est obtenue par résolution ou diminution à partir de clauses dont l'une est c_0 ,

- f_i , $i > 0$, est obtenue par résolution ou diminution à partir de clauses dont l'une est f_{i-1} .

Exemple :

$$\mathcal{C} = \{\neg A \vee \neg B, A \vee \neg C, C, B \vee \neg D, D \vee B\}$$

$$c_0 = \neg A \vee \neg B$$

La déduction :

\mathcal{C}

$$f_0 : \neg B \vee \neg C \text{ (résolution entre } c_0 \text{ et } A \vee \neg C)$$

$$f_1 : \neg B \quad \text{(résolution entre } f_0 \text{ et } C)$$

$$f_2 : \neg D \quad \text{(résolution entre } f_1 \text{ et } \neg B \vee \neg D)$$

$$f_3 : B \quad \text{(résolution entre } f_2 \text{ et } D \vee B)$$

$$f_4 : \square \quad \text{(résolution entre } f_3 \text{ et } f_1)$$

est une déduction linéaire de racine c_0 .

Le graphe de dérivation qui lui correspond est :

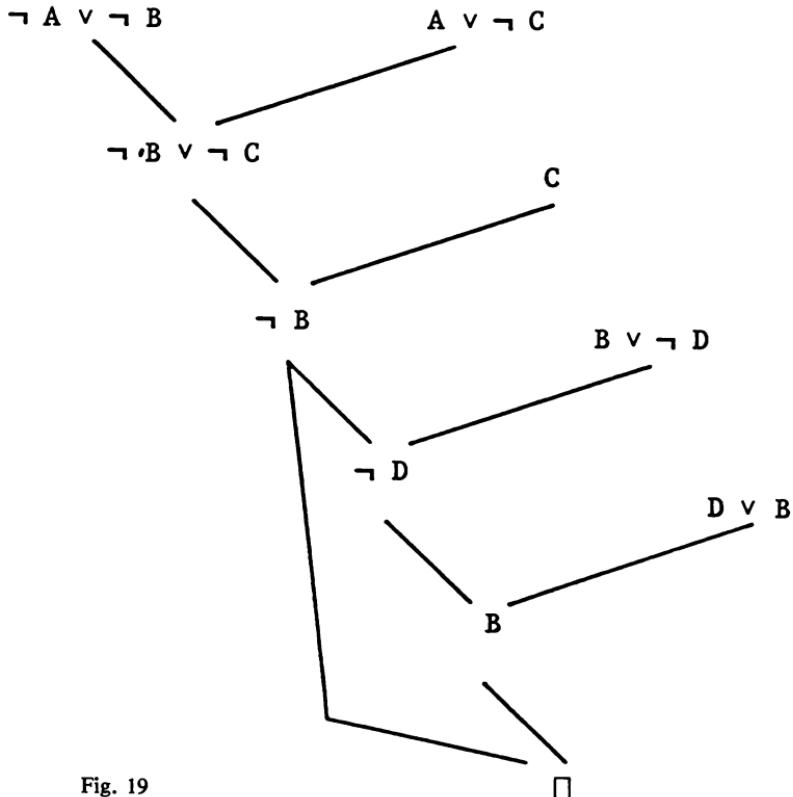
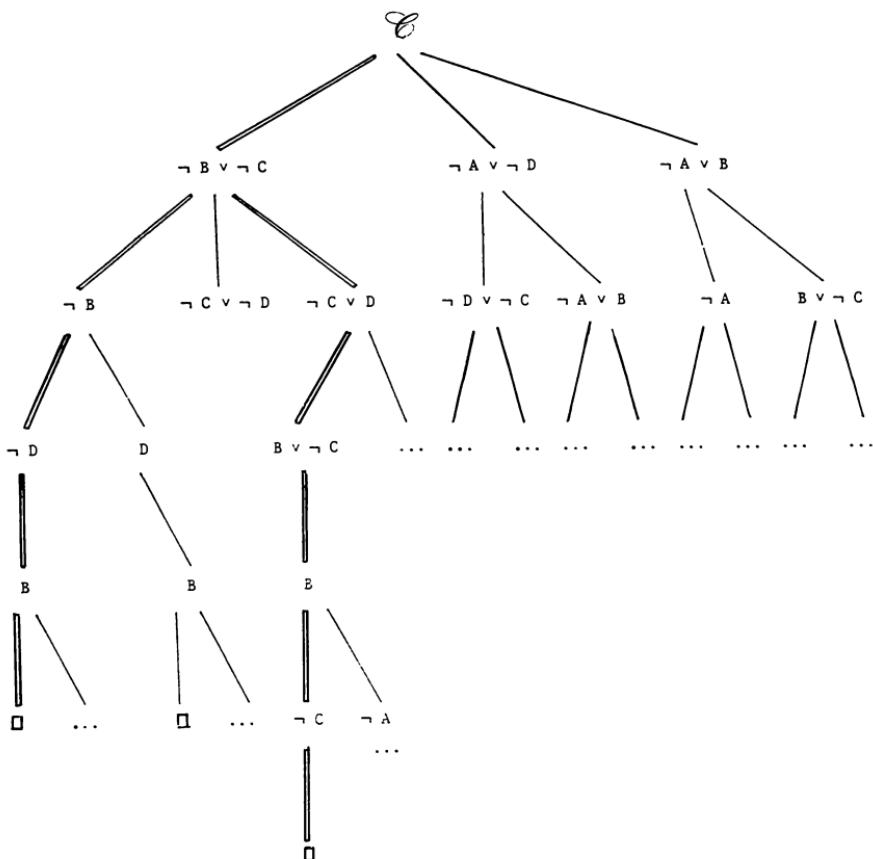


Fig. 19

Ce n'est pas la seule déduction linéaire de racine c_0 , car il y a aussi :

- | | |
|-----------------------------|---|
| $f'_0 : \neg B \vee \neg C$ | (résolution entre c_0 et $A \vee \neg C$) |
| $f'_1 : \neg C \vee D$ | (résolution entre f'_0 et $D \vee B$) |
| $f'_2 : B \vee \neg C$ | (résolution entre f'_1 et $B \vee \neg D$) |
| $f'_3 : B$ | (résolution entre f'_2 et C) |
| $f'_4 : \neg C$ | (résolution entre f'_3 et f'_0) |
| $f'_5 : \square$ | (résolution entre f'_4 et C) |

L'arbre de toutes les déductions linéaires de racines c_0 est déjà assez grand, en voici une partie :



On remarquera les deux branches renforcées qui donnent la déduction linéaire $\mathcal{C} f_0 f_1 f_2 f_3 f_4$ et la déduction linéaire $\mathcal{C} f'_0 f'_1 f'_2 f'_3 f'_4 f'_5$.

On établit (voir [CHA 73] [LOV 78]) que s'il existe une déduction de la clause vide à partir de \mathcal{C} alors il existe une déduction linéaire de la clause vide à partir de \mathcal{C} .

De plus, si $\mathcal{C} = \mathcal{C}' \cup \{c_0\}$ avec \mathcal{C}' satisfiable et \mathcal{C} insatisfiable alors il existe une déduction linéaire de la clause vide à partir de \mathcal{C} et ayant pour racine c_0 .

Ceci signifie que lorsqu'on cherche à savoir si T résulte de \mathcal{C} , il suffit de parcourir l'arbre des déductions linéaires de racine $c_0 = \neg T$ et d'y rechercher la clause vide.

Si on utilise une stratégie d'exploration « en largeur d'abord » on est sûr de trouver la clause vide, si elle y est (stratégie complète). Par contre, si on utilise une stratégie « en profondeur d'abord avec retours en arrière » on peut s'enfoncer dans une branche infinie, et donc, ne pas trouver la clause vide alors qu'elle y est, comme cela se produit sur l'exemple suivant :

$$\mathcal{C} = \{\neg p(x) \vee p(f(x)), \neg p(a), p(x)\}$$

$$c_0 = p(x)$$

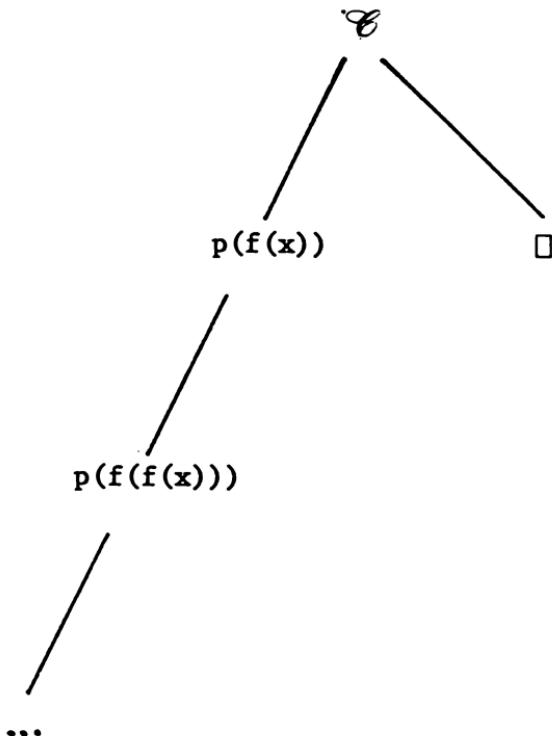


Fig. 21

L'exploration de la première branche de l'arbre des déductions linéaires ne se termine jamais, alors que la deuxième branche donne immédiatement une déduction de la clause vide.

(b3) Stratégies input

On appelle déduction input de racine c_0 à partir de l'ensemble de clauses \mathcal{C} (contenant c_0) toute déduction $\mathcal{C} f_0 f_1 \dots f_m$ telle que :

- f_0 est obtenue par résolution ou diminution à partir de clauses dont l'une est c_0
- f_i , $i > 0$, est obtenue par résolution ou diminution à partir de clauses dont l'une est dans \mathcal{C} .

Il est évident que dans une telle déduction si à un moment donné f_i est obtenue sans utiliser f_{i-1} , c'est qu'une partie de la déduction ne servira pas. On voit donc qu'en imposant (ce que nous ferons) à toute déduction input d'être aussi une déduction linéaire, on ne perd aucun théorème de plus.

Exemple :

$$\mathcal{C} = \{\neg A, A \vee \neg B, A \vee \neg C \vee \neg D, C, D \vee \neg C\}$$

$$c_0 = \neg A$$

La déduction :

$$\begin{aligned}\mathcal{C} \\ f_0 : \neg C \vee \neg D & \quad (\text{résolution entre } c_0 \text{ et } A \vee \neg C \vee \neg D) \\ f_1 : \neg D & \quad (\text{résolution entre } f_0 \text{ et } C) \\ f_2 : \neg C & \quad (\text{résolution entre } f_1 \text{ et } D \vee \neg C) \\ f_3 : \square & \quad (\text{résolution entre } f_2 \text{ et } C)\end{aligned}$$

est une déduction input (et linéaire).

L'arbre de toutes les déductions input est :

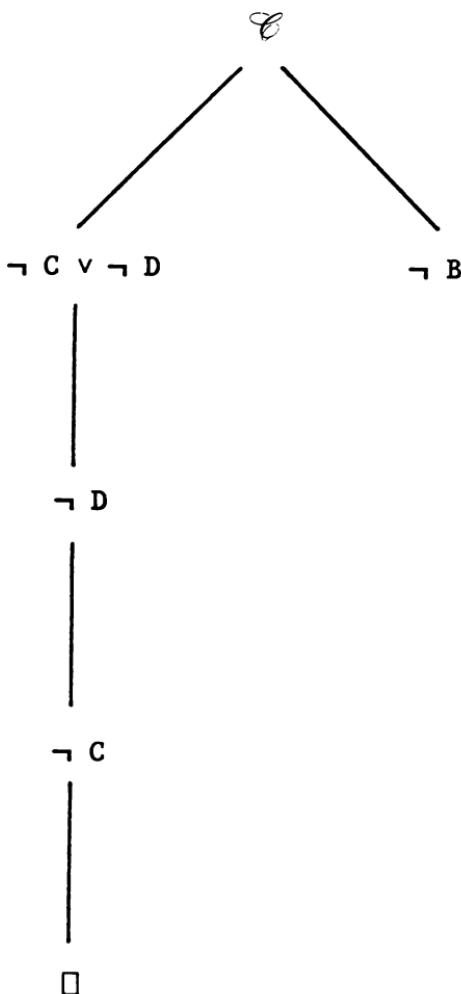


Fig. 22

Comme on le constate, l'une des branches se bloque ne conduisant pas à la clause vide. Ceci signifie qu'une stratégie d'exploration « en profondeur d'abord » même dans les cas simples où l'arbre des déductions input est fini, doit faire des « retours en arrière » (« backtracking ») pour trouver la clause vide.

Ici, contrairement au cas (b2), même une stratégie d'exploration « en largeur d'abord » n'est pas complète : il existe des situations comme par exemple $\mathcal{C} = \{A \vee B, A \vee \neg B, \neg A \vee B, \neg A \vee \neg B\}$ où l'arbre des déductions input ne contient pas la clause vide alors que l'arbre des déductions la contient. Formulé autrement : même si \mathcal{C} est insatisfiable, il se peut qu'aucune déduction input ne donne la clause vide.

On a quand même le très intéressant résultat suivant :

Si $\mathcal{C}' = \mathcal{C} \cup \{c_0\}$ est insatisfiable et que c_0 est une clause ne comportant que des littéraux négatifs (une telle clause s'appelle *clause négative*) et que \mathcal{C}' ne comporte que des clauses ayant chacune exactement un littéral positif (de telles clauses s'appellent des *clauses de Horn*)

alors :

il existe une déduction input de racine c_0 conduisant à la clause vide, et de plus, cette déduction n'utilise jamais la règle d'inférence de diminution.

(voir [HEN 74]).

Exemples d'utilisation :

Prenons :

$$\mathcal{C}' = \{\neg p(x) \vee p(f(x)), \neg p(f(y)) \vee \neg p(y) \vee r(y), p(a)\}$$

$$c_0 = \neg r(z)$$

Ce qui correspond au problème :

peut-on déduire : $\exists z r(z)$, des trois axiomes :

$$\forall x (p(x) \rightarrow p(f(x)))$$

$$\forall y ((p(f(y)) \wedge p(y)) \rightarrow r(y))$$

$$p(a)$$

On trouve la déduction input de racine c_0 :

$$\mathcal{C}' \cup \{c_0\}$$

$$f_0 : \neg p(f(y)) \vee \neg p(y) \quad (\text{résolution avec } c_0 \text{ et } \neg p(f(y)) \vee \neg p(y) \vee r(y))$$

$$f_1 : \neg p(f(a)) \quad (\text{résolution avec } f_0 \text{ et } p(a))$$

$$f_2 : \neg p(a) \quad (\text{résolution avec } f_1 \text{ et } \neg p(x) \vee p(f(x)))$$

$$f_3 : \square \quad (\text{résolution avec } f_2 \text{ et } p(a))$$

La conséquence essentielle du résultat cité plus haut, est que pour la classe des problèmes de la forme $\mathcal{C} = \mathcal{C}' \cup \{c_0\}$, avec \mathcal{C}' un ensemble de clauses de Horn et c_0 une clause négative la stratégie d'exploration « en largeur d'abord » de l'arbre des déductions input de racine c_0 est une stratégie correcte et complète.

La stratégie d'exploration « en profondeur d'abord avec retours en arrière », du même arbre, elle, est correcte et complète pour tous les problèmes cités plus haut engendrant un arbre de déduction input fini (et donc pour tous les problèmes sans variable), mais n'est pas complète en général.

Il est remarquable de voir que le mécanisme de chaînage arrière tel que nous l'avons envisagé au chapitre 1 (voir page 14) est en fait un cas particulier de stratégie d'exploration « en profondeur d'abord avec retours en arrière » de l'arbre des déductions input de racine c_0 (la négation du but). La complétude du mécanisme décrit au chapitre 1 est donc conséquence du résultat plus général que nous venons d'énoncer. En effet :

- les règles que nous envisagions étaient des règles de la forme

$$A_1, A_2, \dots, A_n \rightarrow B$$

c'est-à-dire :

$$(A_1 \wedge A_2 \wedge \dots \wedge A_n) \rightarrow B$$

ou encore :

$$\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n \vee B,$$

il s'agit bien de clauses de Horn.

- le but C dont on écrit la négation $\neg C$ est bien une clause négative,
- le raisonnement par chaînage arrière qui fait passer du but C à la liste de buts A_1, A_2, \dots, A_n (c'est-à-dire de $\neg C$ à $\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n$) quand la règle $A_1, A_2, \dots, A_n \rightarrow C$ est utilisée, est bien une étape de déduction input.

(b4) Stratégies ordonnées

Il existe de nombreuses façons (voir [LOV 78] et [CHA 73]) de tenir compte de l'ordre des littéraux des clauses pour limiter les déductions prises en considération dans l'arbre des déductions.

Nous allons décrire l'une d'elles. Pour cela, il nous faut d'abord définir ce que nous appellerons la résolution ordonnée :

la résolution *ordonnée* entre deux clauses ordonnées⁽¹⁾ (sans variables communes)

$$\begin{array}{ll} C = \ell_1 \vee \ell_2 \vee \dots \vee \ell_n & \ell_1 \text{ littéral positif} \\ C' = \ell'_1 \vee \ell'_2 \vee \dots \vee \ell'_m & \ell'_1 \text{ littéral négatif} \end{array}$$

a pour résultat (quand elle est possible) la clause ordonnée

$$\Theta(\ell_2 \vee \dots \vee \ell_n \vee \ell'_2 \vee \dots \vee \ell'_m)$$

(1) Jusqu'à présent l'ordre des littéraux d'une clause était indifférent (les clauses $A \vee B$ et $B \vee A$ étaient considérées comme identiques). Dans ce paragraphe l'ordre des littéraux est pris en compte.

où Θ est le plus grand unificateur de ℓ_1 et de $\neg \ell'_1$ (on impose que la résolution se fasse sur les littéraux de tête de chacune des clauses, on impose que le résultat soit ordonné en mettant d'abord les littéraux de la clause qui contenait le littéral positif, puis les littéraux de l'autre).

Par exemple la résolution ordonnée entre :

$p(x) \vee r(x) \vee q(x,y)$ et $\neg p(a) \vee \neg r(f(a))$

est possible et donne la clause ordonnée :

$r(a) \vee q(a,y) \vee \neg r(f(a))$

La résolution ordonnée entre :

$p(x) \vee r(x) \vee q(x,y)$ et $\neg r(f(a)) \vee \neg p(a)$

est impossible.

On appellera *déduction ordonnée* toute déduction n'utilisant que des résolutions ordonnées.

Exemple :

$$\mathcal{C} = \{A \vee B, \neg A \vee C, \neg B \vee D, \neg C, \neg D\}$$

La déduction :

$$\begin{aligned} \mathcal{C} \\ f_0 &: B \vee C && (\text{résolution ordonnée entre } A \vee B \text{ et } \neg A \vee C) \\ f_1 &: C \vee D && (\text{résolution ordonnée entre } f_0 \text{ et } \neg B \vee D) \\ f_2 &: D && (\text{résolution ordonnée entre } f_1 \text{ et } \neg C) \\ f_3 &: \square && (\text{résolution ordonnée entre } D \text{ et } \neg D) \end{aligned}$$

est une déduction ordonnée.

Pour nous, le résultat le plus important concernant les déductions ordonnées sera le suivant (qui précise le résultat indiqué en (b3)) :

Si $\mathcal{C} = \mathcal{C}' \cup \{c_0\}$ est insatisfiable, que c_0 est une clause négative, et que \mathcal{C}' ne contient que des clauses de Horn ordonnées en plaçant le littéral positif en tête, alors :

il existe une déduction à la fois input et ordonnée de racine c_0 conduisant à la clause vide. (Voir [APT 82] [LLO 84])

Exemple d'utilisation :

$$\begin{aligned} \mathcal{C}' &= \{p(x) \vee \neg r(x), p(x) \vee \neg q(f(x)) \vee \neg q(x), q(f(x)) \vee \neg q(x), q(a)\} \\ c_0 &= \neg p(a) \end{aligned}$$

L'arbre des déductions input et ordonnées de racine c_0 est le suivant :

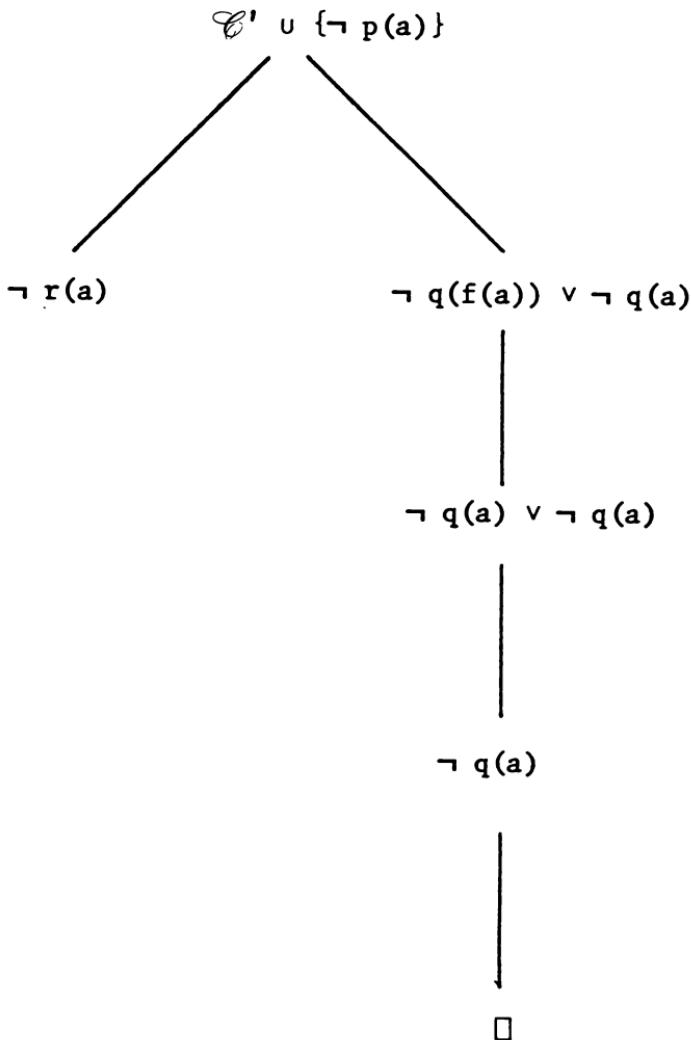


Fig. 23

Cet arbre est fini, relativement réduit et contient effectivement la clause vide conformément au résultat indiqué (car $C = C' \cup \{c_0\}$ est insatisfiable).

Comme en (b3), nous déduisons du résultat cité que : pour la classe des problèmes de la forme $C = C' \cup \{c_0\}$ avec, C' un ensemble de clauses de Horn et c_0 une clause négative, la stratégie d'exploration « en largeur d'abord » de l'arbre des déductions input et ordonnées de racine c_0 est une stratégie correcte et complète.

La stratégie d'exploration « en profondeur d'abord avec retours en arrière » du même arbre, elle, est correcte et complète pour tous les problèmes cités plus haut engendrant un arbre de déductions input et ordonnées qui est un arbre fini.

Le langage Prolog est fondé exactement sur ce principe : on ne peut écrire que des clauses de Horn et quand une question est posée, se met en route un algorithme d'exploration « en profondeur d'abord avec retours en arrière » de l'arbre des déductions input et ordonnées de racine la négation de la question posée. Si cet arbre est fini on est assuré, par le résultat précédent, d'obtenir ou bien la clause vide (c'est-à-dire un succès) si la question posée admet une réponse positive, ou bien un échec (quand tout l'arbre est parcouru sans qu'on y trouve la clause vide) si la question posée admet une réponse négative. Autrement dit, la stratégie d'utilisation de la résolution de Prolog est incomplète en général et complète quand l'arbre de déductions particulier engendré est fini.

(b5) Stratégies input ordonnées et extraction de réponses

Considérons l'exemple suivant :

$$\begin{aligned}\mathcal{C}' &= \{p(a), p(b), r(f(x)) \vee \neg p(x), q(y) \vee \neg r(y), q(c)\} \\ c_0 &= \neg q(z)\end{aligned}$$

On a l'arbre des déductions input ordonnées suivant :

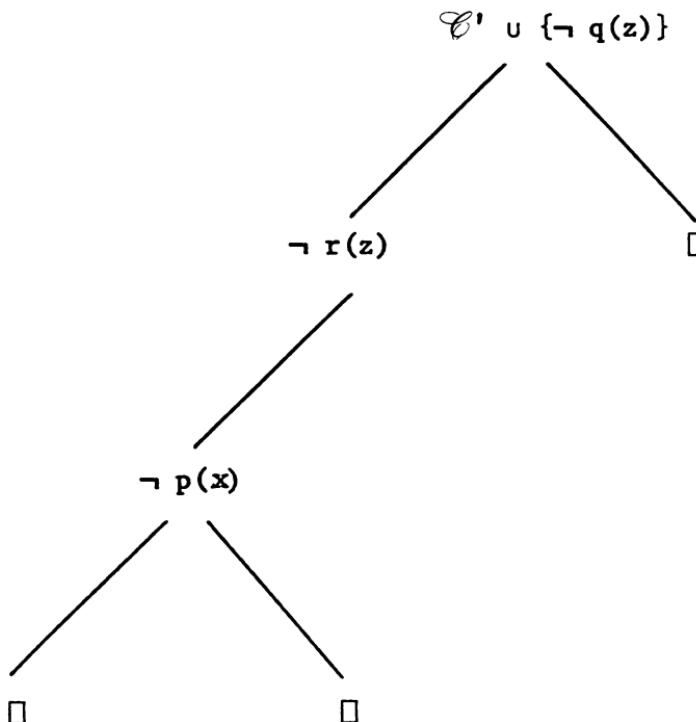


Fig. 24

Il y a trois déductions qui conduisent à la clause vide.

La première est :

$$\mathcal{C}' \cup \{\neg q(z)\}$$

$f_0 : \neg r(z)$ (résolution entre $\neg q(z)$ et $q(y) \vee \neg r(y)$ en faisant la substitution $(y|z)$)

$f_1 : \neg p(x)$ (résolution entre $r(f(x)) \vee \neg p(x)$ et $\neg r(z)$ en faisant la substitution $(z|f(x))$)

$f_2 : \square$ (résolution entre $\neg p(x)$ et $p(a)$ en faisant la substitution $(x|a)$)

Dans le déroulement de la déduction, la variable z a été remplacée par $f(x)$, puis x par a et donc z a été remplacée par $f(a)$.

On voit alors que :

$\mathcal{C}' \cup \{\neg q(f(a))\}$ aurait aussi conduit à la clause vide par la déduction :

$$\mathcal{C}' \cup \{\neg q(f(a))\}$$

$f_0 : \neg r(f(a))$

$f_1 : \neg p(a)$

$f_2 : \square$

Ce qui signifie que $q(f(a))$ résulte de \mathcal{C}' .

De même $q(f(b))$ résulte de \mathcal{C}' par transformation de la deuxième déduction, de même $q(c)$ résulte de \mathcal{C}' par transformation de la troisième déduction.

On constate donc que lorsqu'on considère des clauses avec variables (comme $\neg q(z)$), chaque déduction de la clause vide donne un objet t (de l'univers de Herbrand) tel que $q(t)$ résulte de \mathcal{C}' .

Ceci est en fait général, et on a le résultat suivant [APT 82] [LLO 84]).

Si $\mathcal{C} = \mathcal{C}' \cup \{c_0\}$ est insatisfiable et que c_0 est une clause négative, $c_0 = \neg C(x_1, x_2, \dots, x_n) = \neg \ell_1(x_1, x_2, \dots, x_n) \vee \neg \ell_2(x_1, x_2, \dots, x_n) \vee \dots \vee \neg \ell_r(x_1, x_2, \dots, x_n)$ contenant les variables x_1, x_2, \dots, x_n et que \mathcal{C}' ne comporte que des clauses de Horn ordonnées en plaçant le littéral positif en tête alors : chaque déduction input ordonnée se terminant par la clause vide et dont les substitutions sont $(x_1|t_1), (x_2|t_2), \dots, (x_n|t_n)$ définit des objets de l'univers de Herbrand de \mathcal{C}' tels que :

$C(t_1, t_2, \dots, t_n) = \ell_1(t_1, t_2, \dots, t_n) \wedge \ell_2(t_1, t_2, \dots, t_n) \wedge \dots \wedge \ell_r(t_1, t_2, \dots, t_n)$
est conséquence de \mathcal{C}' .

Réciproquement pour tout n -uplet (t_1, t_2, \dots, t_n) dans l'univers de Herbrand de \mathcal{C}' , tel que $C(t_1, t_2, \dots, t_n)$ soit conséquence de \mathcal{C}' , il existe une déduction input ordonnée de racine $\neg C(x_1, x_2, \dots, x_n)$ se terminant par la clause vide et

dont les substitutions $(x_1|t'_1), (x_2|t'_2), \dots, (x_n|t'_n)$ vérifient que $C(t_1, t_2, \dots, t_n)$ est une instantiation de $C(t'_1, t'_2, \dots, t'_n)$.⁽¹⁾

Ceci signifie que pour savoir quels sont les objets t pour lesquels $q(t)$ est conséquence de \mathcal{C} (ou ce qui est équivalent $q(t)$ est vrai dans tout modèle de Herbrand de \mathcal{C}), il suffit d'explorer l'arbre des déductions input ordonnées par une stratégie « en largeur d'abord », (ou par une stratégie « en profondeur d'abord avec retours en arrière » si on est sûr que cet arbre est fini) en notant à chaque clause vide rencontrée l'objet t qui lui est associé.

C'est là le principe des méthodes d'extraction des réponses utilisées dans de nombreux systèmes de démonstration automatique (voir [NIL 80]) et dans certains langages d'interrogation de bases de données. En particulier, le langage Prolog (qui ainsi que nous l'avons dit, utilise une stratégie d'exploration « en profondeur d'abord avec retours en arrière ») est basé sur ces résultats. On peut donc en dire qu'il calcule ce qui se passe dans le plus petit modèle de Herbrand de \mathcal{C} , l'ensemble des clauses qui définissent le programme ([LLO 84]).

EXERCICES

Exercices sur la résolution sans variable

- 1) Trouver une déduction (de RSV) de la clause vide montrant que \mathcal{C} est insatisfiable :

$$\mathcal{C} = \{A \vee \neg B \vee C, \neg A \vee C, \neg E, A \vee B \vee E, \neg C \vee E\}$$

- 2) En utilisant la proposition 1 établir que \mathcal{C} est satisfiable :

$$\mathcal{C} = \{A \vee B \vee C, A \vee \neg B \vee C \vee D, \neg A \vee B \vee D, A \vee \neg B \vee \neg C \vee D\}$$

Préciser une interprétation qui satisfasse \mathcal{C} .

Exercices sur l'unification

Unifier les atomes suivants, quand c'est possible :

- 1) At₁ = p(x, f(x), g(f(x)), x)
At₂ = p(z, f(f(a)), g(f(g(a, z))), v))
- 2) At₁ = p(x', y', z')
At₂ = p(f(g(x, y)), g(v, w), y)
At₃ = p(f(z), x, f(x))

(1) C'est-à-dire est obtenue à partir de $C(t'_1, t'_2, \dots, t'_n)$ en remplaçant certaines variables par des termes.

- 3) $At_1 = p(x, f(x), f(f(x)))$
 $At_2 = p(f(f(y)), y, f(y))$
- 4) $At_1 = p(x, y, z, t)$
 $At_2 = p(f(y), f(z), f(t), f(x))$
 $At_3 = p(g(z), g(x), g(y), g(z))$

Exercice sur l'unification

Caractériser les applications $\alpha : \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$ telles que :

$$\begin{aligned} &p(x_1, x_2, \dots, x_n) \\ &p(x_{\alpha(1)}, x_{\alpha(2)}, \dots, x_{\alpha(n)}) \end{aligned}$$

soient unifiables.

Exercices sur l'unification

Soient $\begin{cases} At_1 = p(b, x_1, x_1) \\ At'_1 = p(x'_1, x'_1, b) \end{cases}$

$$\begin{cases} At_2 = p(f(b, x_1, x_1), x_2, x_2) \\ At'_2 = p(x'_2, x'_2, f(x'_1, x'_1, b)) \end{cases}$$

$$\begin{cases} At_3 = p(f(f(b, x_1, x_1), x_2, x_2), x_3, x_3) \\ At'_3 = p(x'_3, x'_3, f(x'_2, x'_2, f(x'_1, x'_1, b))) \end{cases}$$

etc.

- 1) Calculer l'unifié de At_1 et At'_1 , de At_2 et At'_2 , de At_3 et At'_3 etc.
- 2) On appelle longueur de l'atome At le nombre de symboles qui le compose à l'exclusion des virgules et parenthèses (par exemple : $\text{longueur}(At_1) = 4$). Montrer qu'il existe deux constantes C_1, C_2 telles que :

$$C_1 > 0, C_2 > 1$$

pour tout n , il existe deux atomes At et At' de longueur $> n$ vérifiant l'inégalité :

$$[\text{longueur de l'unifié de } At \text{ et } At'] > [C_1 \times C_2^{\text{longueur}(At)} + \text{longueur}(At')]$$

Exercices sur l'unification

- 1) Démontrer en détail et sans utiliser l'algorithme d'unification que :
 $p(x, f(x))$ et $p(f(y), y)$
ne sont pas unifiables.

2) De même pour :

$$p(x,y,z) \text{ et } p(f(z),x,y).$$

Exercices sur l'utilisation du système formel RAV

1) Montrer en utilisant la proposition 2 (après avoir convenablement transformé le problème conformément aux techniques du chapitre 6), que :

$$F = \forall u \ q(u)$$

est conséquence de :

$$A_1 = \forall x \exists y \ p(x,y)$$

$$A_2 = \forall z_1 \forall z_2 (p(z_1,z_2) \rightarrow q(z_1))$$

2) Même exercice avec :

$$F = \forall x (p(x) \rightarrow p(f(f(x))))$$

$$A_1 = \forall x (p(x) \rightarrow r(f(x)))$$

$$A_2 = \forall x (r(x) \rightarrow p(f(x)))$$

3) Même exercice avec :

$$F = \exists z \ q(z)$$

$$A_1 = \exists y \forall x \ p(x,y)$$

$$A_2 = (\forall x \exists y \ p(x,y) \rightarrow \forall z \ q(z))$$

4) Même exercice avec :

$$F = \exists x \ s(x)$$

$$A_1 = \exists x \forall y (p(x,y) \vee p(y,x))$$

$$A_2 = \forall x (p(x,x) \rightarrow (q(x) \vee r(x)))$$

$$A_3 = \forall z (q(z) \rightarrow \neg s(z))$$

$$A_4 = \forall u (r(u) \rightarrow q(u))$$

5) Même exercice avec :

$$F = p(s(s(s(s(a)))))$$

$$A_1 = p(a)$$

$$A_2 = \forall x (p(x) \rightarrow q(s(x)))$$

$$A_3 = \forall x (q(x) \rightarrow p(s(x)))$$

Donner un sens à a, p, q et s qui permette de bien comprendre ce qui se passe.

Etablir que :

$$F' = \forall x \ p(x)$$

ne résulte pas de $\{A_1, A_2, A_3\}$

6) Même exercice avec :

$$F = p(b, c, d, a)$$

$$A_1 = \forall x \forall y \forall z \forall t (p(x, y, z, t) \rightarrow p(y, x, z, t))$$

$$A_2 = \forall x \forall y \forall z \forall t (p(x, y, z, t) \rightarrow p(z, y, x, t))$$

$$A_3 = \forall x \forall y \forall z \forall t (p(x, y, z, t) \rightarrow p(t, y, z, x))$$

$$A_4 = \forall x \forall y \forall z \forall t (p(x, y, z, t) \rightarrow p(t, x, y, z))$$

$$A_5 = p(a, b, c, d)$$

Exercice sur l'utilisation de la résolution dans un cas concret

On considère le vocabulaire suivant :

Constantes	: l	: un livre
	: e	: un éditeur
	: m	: monsieur Dupont
Prédicats	: rl(x,y)	: x reçoit le livre y
	: ca(x)	: x a un compte bancaire approvisionné
	: fc(x,y)	: x fait un chèque à y
	: eel(z,y,x)	: l'éditeur z envoie le livre y à x
	: vl(x,y)	: x veut recevoir le livre y
	: lde(y,z)	: le livre y est disponible chez l'éditeur z

1) Interpréter les axiomes :

$$a1 : ca(m)$$

$$a2 : lde(l,e).$$

$$a3 : vl(m,l)$$

$$a4 : eel(z,y,x) \rightarrow rl(x,y)$$

$$a5 : (fc(x,z) \wedge lde(y,z)) \rightarrow eel(z,y,x)$$

$$a6 : (vl(x,y) \wedge ca(x)) \rightarrow fc(x,z)$$

2) Montrer que $rl(m,l)$ résulte des axiomes par résolution. Transcrire chaque étape de résolution en langage courant de manière à obtenir un raisonnement naturel.

Exercices sur les clauses de Horn

L'ensemble de trois clauses suivant :

$$S = \{p(x) \vee q(x) \vee r(x), \neg p(x) \vee \neg q(x) \vee r(x), \neg p(x) \vee q(x) \vee \neg r(x)\}$$

n'est pas un ensemble de clauses de Horn (à cause de la première clause).

En introduisant les symboles de prédicat q' , r' à qui on fait jouer le rôle de $\neg q$, $\neg r$, on peut réécrire S sous la forme :

$$S = \{p(x) \vee \neg q'(x) \vee \neg r'(x), \neg p(x) \vee q'(x) \vee \neg r'(x), \neg p(x) \vee \neg q'(x) \vee r'(x)\}$$

qui cette fois, est un ensemble de clauses de Horn.

- 1) Montrer qu'un tel renommage des prédicats ne permet pas toujours de transformer en clauses de Horn un ensemble de clauses quelconques.
- 2) Montrer que si S est un ensemble de clauses sans variable insatisfiable minimal, alors il est possible de renommer ses prédicats pour en faire un ensemble de clauses de Horn et de clauses négatives. [HEN 74].

Exercice sur les clauses de Horn

- 1) Montrer que tout ensemble de clauses de Horn est satisfiable.
 - 2) Montrer que si \mathcal{C}' est un ensemble de clauses de Horn sans variable et que si c_0 est une clause sans variable négative telle que $\mathcal{C} = \mathcal{C}' \cup \{c_0\}$ soit insatisfiable alors il existe une déduction input de racine c_0 qui donne la clause vide.
 - 3) Montrer que si \mathcal{C}' est un ensemble de clauses de Horn sans variable et que i_1 et i_2 sont deux modèles de \mathcal{C}' alors i_3 défini par
 $i_3(A) = V \Leftrightarrow (i_1(A) = V \text{ et } i_2(A) = V)$
est aussi un modèle de \mathcal{C}' .
(l'intersection de deux modèles d'un ensemble de clauses de Horn en est encore un).
- Ce résultat est-il vrai sans l'hypothèse que les clauses sont de Horn ?

Exercice sur la résolution input ordonnée

Dessiner l'arbre des déductions input ordonnées de racine $c_0 = \neg r(a)$, pour $\mathcal{C}' = \{r(x) \vee \neg r(f(x)), r(x) \vee \neg r(g(x)), r(f(g(f(a))))\}$

Combien a-t-il de noeuds de profondeur n ?

Que donne une stratégie d'exploration « en profondeur d'abord avec retours en arrière », et une stratégie d'exploration « en largeur d'abord ».

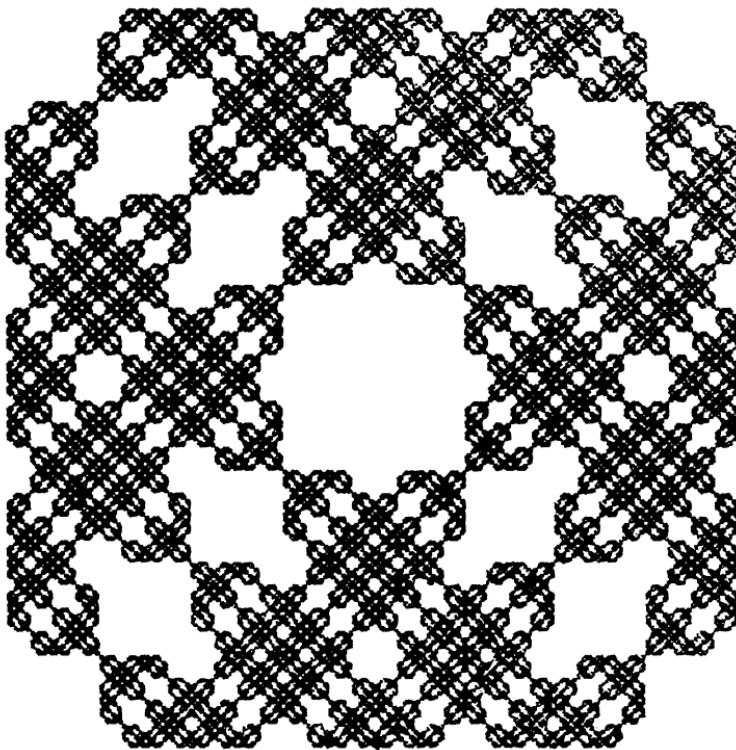
BIBLIOGRAPHIE

- [APT 82] APT K. R. et VAN EMDEN M. H.
Contribution to the Theory of Logic Programming.
Journal of the Association for Computing Machinery, Vol. 29, n° 3, 1982,
pp. 841-862.
{article fondamental concernant les stratégies input ordonnées, appelées
SLD dans ce document}.
- [BIB 82] BIBEL W.
Automated Theorem Proving.
Wieweg 85 ohm Braunschweig Wiesbaden 1982
{Présentation unifiée des méthodes de démonstration automatique}
- [CHA 73] CHANG C.-L. et LEE R.-C.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, New York, 1973.
{La méthode de résolution ainsi que de nombreuses variantes sont présentées et étudiées en détail de manière claire et précise}.
- [COM 83] COMYN G. et DAUCHET M.
Equations, Systèmes de Réécriture, Logique de Premier Ordre.
Polycopié de cours, Université de Lille I, U.E.R. d'I.E.E.A.
Villeneuve-d'Ascq 59655, 1983.
{Parmi d'autres choses, on trouvera exposées, l'unification et la
résolution}.
- [HEN 74] HENSCHEN L. et WOS L.
Unit Refutations and Horn Sets.
Journal of the Association for Computing Machinery. Vol. 21, n° 4, 1974,
pp. 590-605.
{Résultats importants sur les déductions input (voir paragraphe 5)}.
- [HUE 76] HUET G.
Résolution d'Equations dans des Langages d'ordre 1, 2, ..., ω
Thèse d'Etat, Université de Paris VII, 1976.
{Contient une théorie détaillée de l'unification}
- [KOW 79] KOWALSKI R.
Logic for Problem Solving.
Elsevier North Holland, Inc., New York, 1976.
{Présentation originale de la logique, avec pour but la résolution de problèmes et la programmation logique}
- [LEW 80] LEWIS H. R. et PAPADIMITRIOU C. H.
L'Efficacité des Algorithmes.
Dans « Les Progrès des Mathématiques ». Bibliothèque Pour la Science.
Belin, Paris, 1980.
{Article sur les problèmes de complexité des algorithmes}

- [LLO 84] LLOYD J. W.
Foundations of Logic Programming.
 Springer Verlag, Berlin, 1984.
 {Sur les aspects théoriques de la programmation logique. Destiné essentiellement aux chercheurs}
- [LOV 78] LOVELAND D. W.
Automated Theorem Proving : A Logical Basis.
 North-Holland Publishing Company, Amsterdam, 1978.
 {Le plus complet des ouvrages disponibles sur la démonstration automatique de théorèmes}
- [NIL 80] NILSSON N.
Principles of Artificial Intelligence.
 Tioga Publishing Company, Palo Alto, 1980.
 {Les chapitres 4 et 5 présentent de manière informelle et à l'aide de nombreux exemples la résolution et quelques-unes de ses variantes}
- [RIC 83] RICH E.
Artificial Intelligence.
 Mac Graw Hill, Inc., 1983.
 {Au chapitre 5, on trouvera un exposé de la résolution, concis, informel et fondé sur des exemples originaux}
- [ROB 65] ROBINSON J. A.
A Machine Oriented Logic Based on the Resolution Principle.
 Journal of the Association for Computing Machinery Vol. 12, 1965,
 pp. 227-234.
 {L'article à la base de tous les développements ultérieurs de la méthode de résolution}
- [ROB 79] ROBINSON J. A.
Logic : Form and Function, The Mechanization of Deductive Reasoning.
 Edinburgh University Press, Edinburgh, 1979.
 {Présentation du calcul des prédictats, de l'unification et de la résolution par celui qui fut à l'origine de toutes les recherches sur la résolution}
- [ROB 83] ROBINSON J. A.
Logic Programming-Past, Present and Future.
 New Generation Computing, 1983, pp. 107-124.
 {Très intéressante étude sur l'histoire et les perspectives de la Programmation Logique}
- [SIE 82] SIEKMANN J., SZABO P.
Universal Unification.
 6 th German Workshop on A.I., Springer Verlag, Berlin, 1982.
 {Article de synthèse sur les problèmes d'unification}
- [SIM 84] SIMON J.-C.
La reconnaissance des formes par algorithmes.
 Masson, Paris, 1984.
 {Le chapitre 1 présente les notions de problèmes NP complets et diverses autres notions concernant la complexité des algorithmes}

CHAPITRE 8

LE LANGAGE PROLOG



1. — INTRODUCTION

Inventé à MARSEILLE, au début des années soixante-dix, par Alain COLMERAUER et Philippe ROUSSEL, le langage PROLOG est maintenant universellement connu et utilisé.

De très nombreuses versions existent, parfois assez différentes les unes des autres. Nous avons choisi de nous référer à celle nommée PROLOG II qui a été développée à MARSEILLE à la suite du premier PROLOG.

Comme on va le voir, PROLOG peut s'utiliser de deux façons. La première, qui peut être qualifiée de naïve, consiste à énoncer des règles et des faits sur un domaine donné, qu'on groupe sans tenir compte de l'ordre, en un programme qui peut alors être interrogé. Cette première méthode d'utilisation atteint vite ses limites et ne permet de traiter que des problèmes simples. La seconde méthode nécessite la conscience du mécanisme de résolution PROLOG et permet, grâce à ce qu'on peut appeler une « double pensée », d'écrire des règles et des faits sur un domaine donné tout en envisageant l'utilisation qui va en être faite lors de la résolution, ce qui conduit à introduire les éléments du programme, dans un certain ordre, sous une certaine forme, en garantissant le bon fonctionnement.

2. — UN EXEMPLE

(a) Ecriture d'un programme en PROLOG II

Nous reprenons l'exemple déjà utilisé au chapitre 5.

On voudrait décrire la structure d'une famille de 11 personnes :

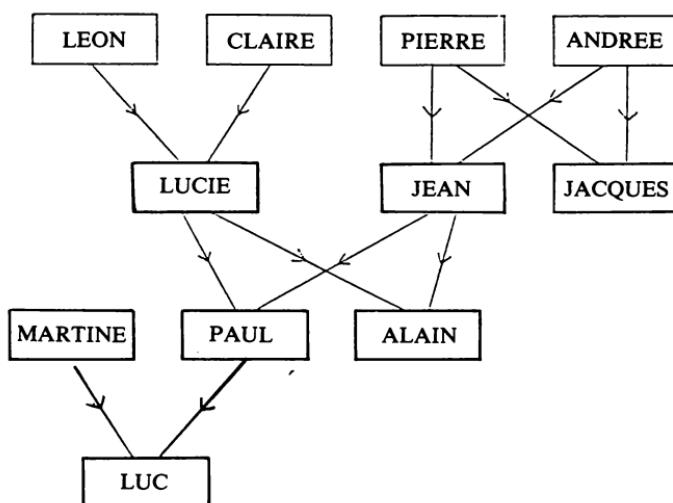


Fig. 25

Puis on voudrait que le programme nous donne la réponse à des questions du genre :

- Qui est le père de Jean ?
- Quels sont les grands-parents de Paul ?
- Alain a-t-il des oncles ?
- etc.

PROGRAMME

```
masculin(jean) ->;
masculin(paul) ->;
masculin(alain) ->;
masculin(pierre) ->;
masculin(leon) ->;
masculin(luc) ->;
masculin(jacques) ->;

feminin(lucie) ->;
feminin(claire) ->;
feminin(andree) ->;
feminin(martine) ->;

est-pere(jean,paul) ->;
est-pere(jean,alain) ->;
est-pere(pierre,jean) ->;
est-pere(leon,lucie) ->;
est-pere(paul,luc) ->;
est-pere(pierre,jacques) ->;

est-mere(claire,lucie) ->;
est-mere(andree,jean) ->;
est-mere(andree,jacques) ->;
est-mere(lucie,paul) ->;
est-mere(lucie,alain) ->;
est-mere(martine,luc) ->;

est-parent(x,y) -> est-pere(x,y);
est-parent(x,y) -> est-mere(x,y);

est-enfant(x,y) -> est-parent(y,x);
```

```

est-grand-pere(x,y) -> est-pere(x,z) est-parent(z,y);
est-grand-mere(x,y) -> est-mere(x,z) est-parent(z,y);
est-grand-parent(x,y) -> est-grand-pere(x,y);
est-grand-parent(x,y) -> est-grand-mere(x,y);
est-frere(x,y) -> masculin(x) est-pere(z,x) est-pere(z,y) dif(x,y);
est-soeur(x,y) -> feminine(x) est-pere(z,x) est-pere(z,y) dif(x,y);
est-petit-enfant(x,y) -> est-grand-parent(y,x);
est-petit-fils(x,y) -> masculin(x) est-petit-enfant(x,y);
est-petit-fille(x,y) -> feminine(x) est-petit-enfant(x,y);
est-oncle(x,y) -> est-frere(x,z) est-parent(z,y);
est-tante(x,y) -> est-soeur(x,z) est-parent(z,y);

```

Chaque ligne du programme correspond à une affirmation. Il y a les affirmations élémentaires (les faits), par exemple :

- `mASCULIN(jean)` —> ;
- `EST-PÈRE(jean,paul)` —> ;

qui doivent se comprendre comme :

- jean est de sexe masculin
- jean est le père de paul

Il y a les affirmations contenant des variables (appelées règles, ou axiomes) par exemple :

- `EST-PARENT(X,Y)` —> `EST-PÈRE(X,Y)` ;
- `EST-GRAND-PÈRE(X,Y)` —> `EST-PÈRE(X,Z)` `EST-PARENT(Z,Y)` ;

qui doivent se comprendre comme :

- si `x` est-père de `y` alors `x` est parent de `y`
- `x` est le grand-père de `y` s'il existe un `z` tel que
`x` est le père de `z` et `z` est parent de `y`.

Z

La flèche «—>» de PROLOG signifie donc « si »,
c'est-à-dire la réciproque de l'implication.

« A —> B » ; signifie « A si B »
ou « B implique A »
ou « si B alors A »

Chaque affirmation contenant des variables doit être comprise comme quantifiée avec des \forall placés devant (comme en calcul des prédictats).

$\text{est-grand-père}(x,y) \rightarrow \text{est-père}(x,z) \text{ est-parent}(z,y)$

signifie :

$\forall x \forall y \forall z ((\text{est-père}(x,z) \wedge \text{est-parent}(z,y)) \rightarrow \text{est-grand-père}(x,y))$

(on utilise « \rightarrow » pour l'implication du calcul des prédictats).

c'est-à-dire (voir chapitre 6 formule c8)) :

$\forall x \forall y ((\exists z (\text{est-père}(x,z) \wedge \text{est-parent}(z,y))) \rightarrow \text{est-grand-père}(x,y))$

On peut donc retenir le principe suivant :

- si la variable x n'apparaît qu'avant le symbole « \rightarrow »
 $A \rightarrow B$; signifie : $(\forall x A)$ si B
- si la variable x n'apparaît qu'après le symbole « \rightarrow »
 $A \rightarrow B$; signifie : A si $(\exists x B)$
- si la variable x apparaît avant et après le symbole « \rightarrow »
 $A \rightarrow B$; signifie : $\forall x (A \text{ si } B)$

En PROLOG, on ne peut écrire que des formules de la forme :

$A \rightarrow B_1 B_2 \dots B_n ;$

$n \geq 0$

ce qui correspond à :

$(B_1 \wedge B_2 \wedge \dots \wedge B_n) \rightarrow A$

ou encore à :

$A \vee \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_n.$

On ne peut donc écrire que des clauses ayant un littéral positif.

De telles clauses s'appellent des clauses de Horn.

On sait que tout ensemble fini de formules du calcul des prédictats du premier ordre peut s'écrire sous la forme d'un ensemble de clauses, c'est-à-dire de formules du type :

$A_1 \vee A_2 \vee \dots \vee A_m \vee \neg B_1 \vee \dots \vee \neg B_n$

Par contre, il n'est pas vrai que tout ensemble de formules puissent s'écrire sous la forme de clauses de Horn ($A \vee B \vee \neg C$ par exemple n'est équivalent à aucun ensemble de clauses de Horn).

Nous avons là une première limitation du langage PROLOG : il n'est pas possible d'exprimer n'importe quelles formules dans un programme PROLOG.

On constatera qu'en pratique, on arrive presque toujours à exprimer ce qu'on désire (ceci bien sûr, nécessite parfois certaines astuces)⁽¹⁾.

(1) Voir les exercices sur les clauses de Horn du chapitre 7.

Indiquons que les interpréteurs PROLOG exigent en général que dans un programme, toutes les règles dont la tête est composée à partir d'un prédicat donné soient regroupées ensemble.

Le programme :

```
pp(aa) —> ;  
rr(bb) —> ;  
pp(x) —> rr(x) ;
```

est incorrect, il faut écrire :

```
pp(aa) —> ;  
pp(x) —> rr(x) ;  
rr(bb) —> ;
```

Dans PROLOG II, toute lettre seule (suivie éventuellement par ' ou des chiffres) représente une variable. Exemple : a,x,m,X,T,x3,y5,x',x''.

Les constantes, les symboles de fonctions et de prédicats sont représentés par des identificateurs commençant par deux lettres au moins. Exemples : aa,XX,paul,Paul,Jean2,Jean-Luc,est-parent.

La syntaxe complète est décrite dans le manuel d'utilisation de PROLOG II, mais ce qui vient d'être dit et ce qui sera indiqué au paragraphe suivant est suffisant pour beaucoup de programmes et en particulier pour tous ceux de ce chapitre.

(b) *Utilisation d'un programme*

Un programme PROLOG, en soi, ne fait rien car c'est simplement une suite d'affirmations.

On utilise un programme PROLOG en l'interrogeant.

Les questions posées peuvent ne pas comporter de variables comme par exemple :

est-père(jean,alain) ;

La réponse est alors OUI ou NON.

Dans PROLOG II, on sait que la réponse est OUI ou NON en regardant où se place le symbole « > » une fois la question posée :

OUI correspond à : 1 ligne sautée, puis « > ».

NON correspond à : pas de ligne sautée (le symbole « > » s'affiche juste sous la question).

Le NON de PROLOG ne doit pas s'interpréter comme l'affirmation « il n'est pas vrai que... », mais comme l'affirmation « il ne résulte pas du programme que... ».

Le deuxième type de question qui peut être posé comporte des variables :
est-père(pierre,x) ;

qui signifie : « de quel x pierre est-il le père ? » ou encore « quels sont les x tels que est-père(pierre,x) résulte des affirmations du programme ? ».

La réponse est alors :

x = jean
x = jacques
>

ce qui signifie : « de ce qui est affirmé dans le programme, on peut déduire que :

est-père(pierre,jean)
est-père(pierre,jacques)

Une question peut comporter plusieurs variables.

Une question peut comporter plusieurs prédictats.

UTILISATION DU PROGRAMME

```
est-pere(jean,paul);
```

>

```
est-pere(jean,luc);
```

>

```
est-grand-pere(jean,luc);
```

>

```
est-oncle(alain,luc);
```

>

```
est-pere(x,jean);  
x=pierre  
>
```

```
est-pere(jean,x);
x=jean
x=alain
>
```

```
mASCULIN(x);
x=jean
x=paul
x=alain
x=pierre
x=leon
x=luc
x=jacques
>
```

```
fEMININ(x) est-grand-parent(x,y);
x=lucie y=luc
x=claire y=paul
x=claire y=alain
x=andree y=paul
x=andree y=alain
>
```

```
fEMININ(x) mASCULIN(x);
>
```

```
est-pere(pierre,x);
x=jean
x=jacques
>
```

```
est-pere(x,y);
x=jean y=paul
x=jean y=alain
x=pierre y=jean
x=leon y=lucie
x=paul y=luc
x=pierre y=jacques
>
```

```
est-grand-pere(pierre,x);
x=paul
x=alain
>
```

```
est-frere(alain,x);
x=paul
>
```

```
est-frere(x,alain);
x=paul
>
```

```
est-frere(x,y);
x=jean  y=jacques
x=paul  y=alain
x=alain  y=paul
x=jacques  y=jean
>
```

```
est-grand-parent(x,paul) est-parent(x,jacques);
x=pierre
x=andree
>
```

```
est-oncle(x,y);
x=alain  y=luc
x=jacques  y=paul
x=jacques  y=alain
>
```

```
est-pere(x,y) est-grand-pere(y,z);
x=pierre  y=jean  z=luc
>
```

3. — LA RÉSOLUTION PROLOG

(a) Bases de la résolution PROLOG

Lorsqu'on pose une question à l'interpréteur PROLOG, on met en route un algorithme (déterministe) qui permet, si tout se passe bien, d'obtenir la réponse souhaitée.

Cet algorithme est basé sur la résolution de ROBINSON mais il utilise le fait que les clauses d'un programme sont des clauses de Horn et détermine les règles à utiliser à chaque pas de résolution en tenant compte de l'ordre dans lequel elles sont écrites dans le programme.

De manière précise : l'algorithme de résolution Prolog est l'algorithme d'exploration « en profondeur d'abord avec retours en arrière » de l'arbre des déductions input ordonnées, associé à l'ensemble des clauses du programme (des clauses de Horn) et à la négation de la question (une clause négative).

Cet algorithme a déjà été envisagé au chapitre 7, mais nous allons le reconstruire en détail.

Si B et C sont unifiables par un plus grand unificateur α , la résolution entre :

$$\neg B \text{ et } C \vee \neg D_1 \vee \dots \vee \neg D_k$$

donne :

$$\neg \alpha(D_1) \vee \neg \alpha(D_2) \vee \dots \vee \neg \alpha(D_k)$$

De manière plus générale, si B_1 et C sont unifiables par un plus grand unificateur α , la résolution entre :

$$\neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_n \text{ et } C \vee \neg D_1 \vee \dots \vee \neg D_k$$

donne :

$$\neg \alpha(D_1) \vee \neg \alpha(D_2) \vee \dots \vee \neg \alpha(D_k) \vee \neg \alpha(B_2) \vee \dots \vee \neg \alpha(B_n).$$

Exprimé en terme de buts et avec les notations PROLOG, ceci signifie que :

si B_1 et C sont unifiables par un plus grand unificateur α , alors :

de la liste des buts $B_1 B_2 \dots B_n$

et de la règle $C \rightarrow D_1 \dots D_k$

on tire la nouvelle liste de buts :

$\alpha(D_1) \dots \alpha(D_k) \alpha(B_2) \dots \alpha(B_n)$

C'est cette idée, exploitée de façon systématique qui constitue le moteur de la résolution PROLOG.

(b) Description de la résolution PROLOG

La résolution PROLOG peut être envisagée comme un parcours d'arbre⁽¹⁾.

Chaque nœud de l'arbre de parcours est étiqueté par :

- une liste de buts $B_1 B_2 \dots B_n$
- une substitution Θ

Au départ, on est au nœud étiqueté par :

- la question posée
- la substitution identité.

Etape du parcours de l'arbre

• On se trouve au nœud :

$$\left\{ \begin{array}{l} B_1 B_2 \dots B_n \\ \Theta \end{array} \right\}$$

• Si la liste B_i est vide, alors

- imprimer Θ (car on a un succès)
- remonter au nœud précédent

• Sinon

- prendre B_1 le premier but
- rechercher parmi les règles non encore explorées à ce nœud et dans l'ordre où elles sont écrites, la première règle :

$$C \rightarrow D_1 D_2 \dots D_k$$

dont la tête peut s'unifier avec B_1

- s'il n'y a pas de telles règles, alors

- si on est au nœud de départ, alors arrêt
- sinon, remonter au nœud précédent
- fin-de-si

- sinon

- soit α un plus grand unificateur pour C et B_1
- descendre en créant un nouveau nœud étiqueté

$$\left\{ \begin{array}{l} \alpha(D_1) \dots \alpha(D_k) \alpha(B_2) \dots \alpha(B_n) \\ \alpha\Theta \end{array} \right\}$$

- fin-de-si

- fin-de-si

(1) L'arbre dont il est question ici est exactement l'arbre des déductions input ordonnées du chapitre 7 dans lequel à la place d'écrire la clause négative $\neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_n$, on écrit simplement $B_1 B_2 \dots B_n$ et dans lequel on précise à chaque nœud les substitutions réalisées lors des étapes de résolution qui y ont conduit.

De manière plus parlante, on peut exprimer « le mécanisme du raisonnement de Prolog » en termes de liste de buts, effacement de but, mise en coïncidence (unification) et de choix en suspens : Prolog essaie de satisfaire (ou d'effacer) une liste de buts. Au départ, ce sont les éléments de la question, dans le cours du raisonnement ce peut être une liste quelconque.

Pour cela, il prend le premier but de la liste et recherche en parcourant le programme de haut en bas, la première des règles non encore utilisées pour cette liste et dont la tête peut être mise en coïncidence avec ce but. Dès qu'il a trouvé une telle règle, il enlève le premier but de la liste des buts et le remplace par la queue de la règle en ayant soin d'effectuer (sur toute la liste de buts) les substitutions qui étaient nécessaires à la mise en coïncidence. Lorsque la queue de la règle est vide (règle de la forme A —> ;) cela revient simplement à effacer le premier but de la liste de buts (en effectuant les substitutions).

Les règles qui n'ont pas été utilisées et qui auraient pu l'être (mise en coïncidence possible) constituent ce qu'on appelle des choix en suspens, qu'il aura à considérer lorsqu'il reviendra à cette liste de buts.

Si la liste de buts obtenue est vide, c'est qu'il y a un succès. Alors, si la question de départ ne comportait pas de variable, une ligne est sautée, et s'il y avait des variables les « valeurs » de ces variables sont affichées à l'écran.

Après un succès, ou lorsque aucune mise en coïncidence n'a été possible, il revient en arrière à la liste de buts précédente. Si pour cette liste, il y a des choix en suspens alors il effectue la mise en coïncidence correspondant au premier de ces choix en suspens et il progresse vers une nouvelle liste de buts. Sinon il remonte encore.

Lorsqu'il ne peut plus remonter (car il est revenu à la question de départ et que toutes les règles dont la tête pouvait être mise en coïncidence avec le premier but ont été prises en considération) il s'arrête.

Remarque :

En fait, la présentation que nous donnons de la résolution PROLOG est un peu idéalisée. D'une part, l'algorithme d'unification de PROLOG ne vérifie pas que x n'est pas présent dans le terme t quand il recherche un plus grand unificateur, d'autre part, un certain nombre de moyens de contrôle (dont le / décrit plus loin) compliquent l'algorithme. Pour être précis, on doit dire aussi que tout, dans PROLOG II, se ramène à des manipulations d'arbres ; la distinction entre symbole de fonctions et symbole de prédictats qui est essentielle en logique du premier ordre, n'est pas explicite dans PROLOG II ; l'unification est ramenée à la résolution de systèmes d'équations et d'inéquations avec des arbres ; certains arbres infinis sont autorisés.

La présentation de PROLOG que nous donnons dans les termes du calcul des prédictats a l'avantage de bien mettre en avant la justification du mécanisme de résolution qui n'est en fait que l'adaptation de la résolution générale de ROBINSON à des clauses de Horn.

De plus, cette présentation permet de penser la résolution d'une façon naturelle et souvent assez simple (comme dans l'exemple donné juste ci-dessous).

Une présentation complète de PROLOG II uniquement en termes d'arbres peut être trouvée dans le Manuel de référence théorique et pratique de PROLOG II de A. COLMERAUER.

(c) Exemple détaillé de résolution PROLOG

Prenons comme exemple le programme suivant :

règle 1 bb —> ;
règle 2 vv —> ;
règle 3 cc(jj) —> ee ;
règle 4 cc(jj) —> bb vv ;
règle 5 mm(aa) —> ;
règle 6 mm(jj) —> ;
règle 7 gg(x) —> mm(x) cc(x) ;

Interprétation possible :

« qui va gagner à la roulette aujourd'hui ? »

gg(x) : x gagne
mm(x) : x est malin
cc(x) : x a de la chance
bb : il fait beau
vv : on est le vendredi
'ee : on est en été

règle 1 il fait beau
règle 2 on est le vendredi
règle 3 jj a de la chance si on est en été
règle 4 jj a de la chance s'il fait beau et si on est le vendredi
règle 5 aa est malin
règle 6 jj est malin
règle 7 x gagne si x est malin et si x a de la chance

Si on pose la question :

gg(x) ;

qui peut se traduire par : « qui gagne ? » alors on obtient la réponse :

x = jj
>

qui signifie : « jean gagne aujourd'hui et d'après le programme, il est le seul ».

Sur la figure 26 est dessiné l'arbre de résolution Prolog, qui indique, pas par pas, le « raisonnement » suivi pour arriver à cette réponse.

Résolution PROLOG

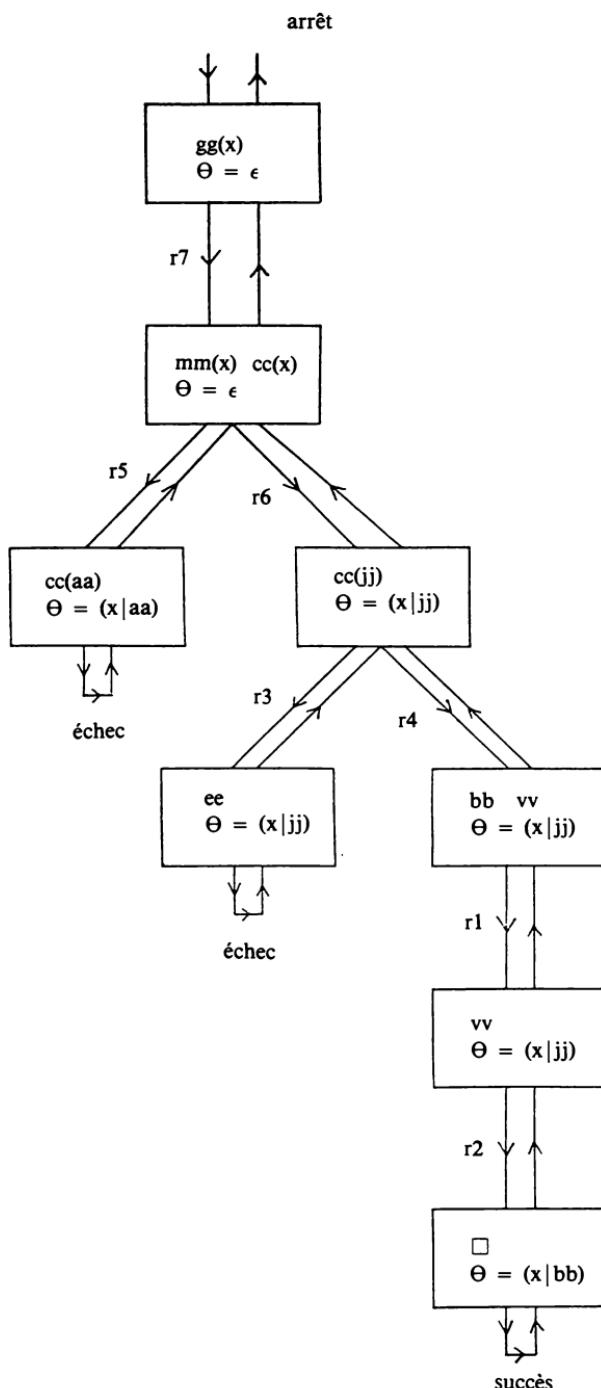


Fig. 26

Il est possible de traduire entièrement le parcours de l'arbre de résolution en un raisonnement logique élémentaire assez naturel. Voici par exemple une façon de le faire :

Pour savoir quels sont les x qui vont gagner aujourd'hui, j'utilise d'abord la règle 7 qui me dit que pour gagner il faut : être malin et avoir de la chance. Je cherche donc à savoir qui est malin et a de la chance. La règle 5 me dit que aa est malin. Je cherche donc à savoir s'il a de la chance. Rien dans les règles ne permet de conclure que aa a de la chance, je reviens donc à mon but antérieur : savoir qui est malin. La règle 6 indique que jj l'est. A-t-il de la chance ? La règle 3 me dit que jj a de la chance en été. Le programme ne dit pas qu'on est en été. La règle 3 ne pouvant me permettre d'avancer, j'essaie la règle 4 qui affirme que jj a de la chance s'il fait beau et qu'on est le vendredi. Il fait beau (règle 1), on est vendredi (règle 2) donc jj a de la chance aujourd'hui et donc (puisque je sais déjà qu'il est malin) jj va gagner. Aucune possibilité n'a été laissée en cours de route, donc mon raisonnement peut s'arrêter avec la conclusion : les informations contenues dans le programme indiquent que jj va gagner et qu'il est le seul.

En utilisant le mode trace de l'interpréteur, on peut vérifier que la résolution s'effectue bien selon l'algorithme décrit. La trace donnée par l'interpréteur PROLOG est constituée de toutes les têtes des règles qui sont utilisées avec les instanciations faites au moment où elles sont utilisées (On appelle instanciation du prédicat pp le prédicat obtenu par application d'une substitution à pp ; on parle aussi de variable instanciée, pour une variable x remplacée par un terme).

PROGRAMME

```
bb ->;  
vv ->;  
cc(jj) -> ee;  
cc(jj) -> bb vv;  
  
mm(aa) ->;  
mm(jj) ->;  
  
gg(x) -> mm(x) cc(x);
```

UTILISATION DU PROGRAMME

```
gg(x);  
x=jj  
>
```

```
mm(x);  
x=aa  
x=jj  
>
```

```
cc(x);  
x=jj  
>
```

DEMANDE DE TRACE

```
trace;  
>
```

```
gg(x);  
gg(x)  
mm(aa)  
mm(jj)  
cc(jj)  
cc(jj)  
bb  
vv  
x=jj  
>
```

```
mm(x);  
mm(aa)  
x=aa  
mm(jj)  
x=jj  
>
```

```

cc(x);
cc(jj)
cc(jj)
bb
vv
x=jj
>

```

(d) Limitations de la résolution PROLOG

Le fait que la résolution PROLOG s'effectue selon l'algorithme indiqué a pour conséquence que :

- l'ordre des clauses du programme est important ;
- certains programmes peuvent boucler indéfiniment ;
- même si un fait simple est conséquence d'un programme, le programme peut boucler sans le découvrir.

Les résultats du chapitre 7 (voir aussi [APT 82]) permettent malgré tout d'affirmer que :

- lorsque le parcours de l'arbre s'arrête, on est certain d'avoir toutes les solutions à la question posée.

(Cette dernière propriété n'est plus vraie si on utilise le /, voir paragraphe suivant).

Illustrons tout cela avec quelques exemples élémentaires :

(1) Le programme :

aa —> aa ;

boucle pour la question :

aa ;

En effet, l'arbre de résolution est :

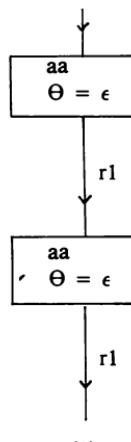


Fig. 27

Que l'algorithme de résolution PROLOG puisse boucler quand on lui demande si t résulte du programme P, alors que t ne résulte pas du programme P, ne doit pas nous surprendre. En effet, nous savons que le problème de la décision pour le calcul des prédictats admet une réponse négative et qu'en conséquence il ne peut exister aucun algorithme qui, pour tout P et t, indique en un temps fini si t est conséquence de P ou pas (le fait que P soit écrit en clauses de Horn ne change pas ce résultat).

Mais, en fait, avec l'algorithme de résolution PROLOG, les choses sont plus graves : même si t résulte de P, il se peut que l'algorithme boucle indéfiniment.

(2) Le programme :

aa —> aa ;
aa —> ;

boucle pour la question :

aa ;

alors que aa résulte du programme.

En écrivant les choses sous la forme :

aa —> ;
aa —> aa ;

alors, maintenant, on obtient bien la bonne réponse pour la question :

aa ;

Ceci montre que l'ordre des règles est important quand on écrit un programme PROLOG.

D'une manière générale, on peut dire qu'il vaut mieux mettre les faits avant les règles.

Malgré tout, ce n'est pas toujours suffisant pour éviter les boucles (et d'ailleurs le programme juste au-dessus donne la bonne réponse, mais ne s'arrête pas).

(3) Le programme :

dd —> ;
aa —> bb ;
bb —> cc ;
cc —> aa ;
cc —> dd ;

boucle pour la question :

aa ;

alors que aa résulte du programme.

En permutant les deux dernières règles, on obtient :

dd —>;
aa —> bb ;
bb —> cc ;
cc —> dd ;
cc —> aa ;

Alors le programme répond correctement⁽¹⁾ à la question :

aa ;

On peut, bien sûr, construire des exemples plus compliqués où il ne sera plus aussi évident de « réparer » le programme.

- (4) Le fait qu'un programme boucle ne provient pas forcément d'un circuit dans les atomes comme dans les exemples (1), (2), (3). Cela peut provenir de ce que l'arbre d'exploration est parfois infini et que la stratégie utilisée étant « en profondeur d'abord avec retours en arrière » il se peut qu'on s'enfonce dans une mauvaise branche (voir chapitre 7).

Par exemple le programme :

tre(ff(gg(jj))) —> ;
tre(x) —> tre(ff(x)) ;
tre(x) —> tre(gg(x)) ;

boucle pour la question :

tre(jj) ;

alors que, bien sûr, tre(jj) résulte des trois axiomes composant le programme.

(1) Ce programme ne s'arrête pas pour la question aa ; nous verrons plus loin qu'en posant la question sous la forme aa / ; on obtient une réponse positive et l'arrêt.

Examinons ce qui se passe :

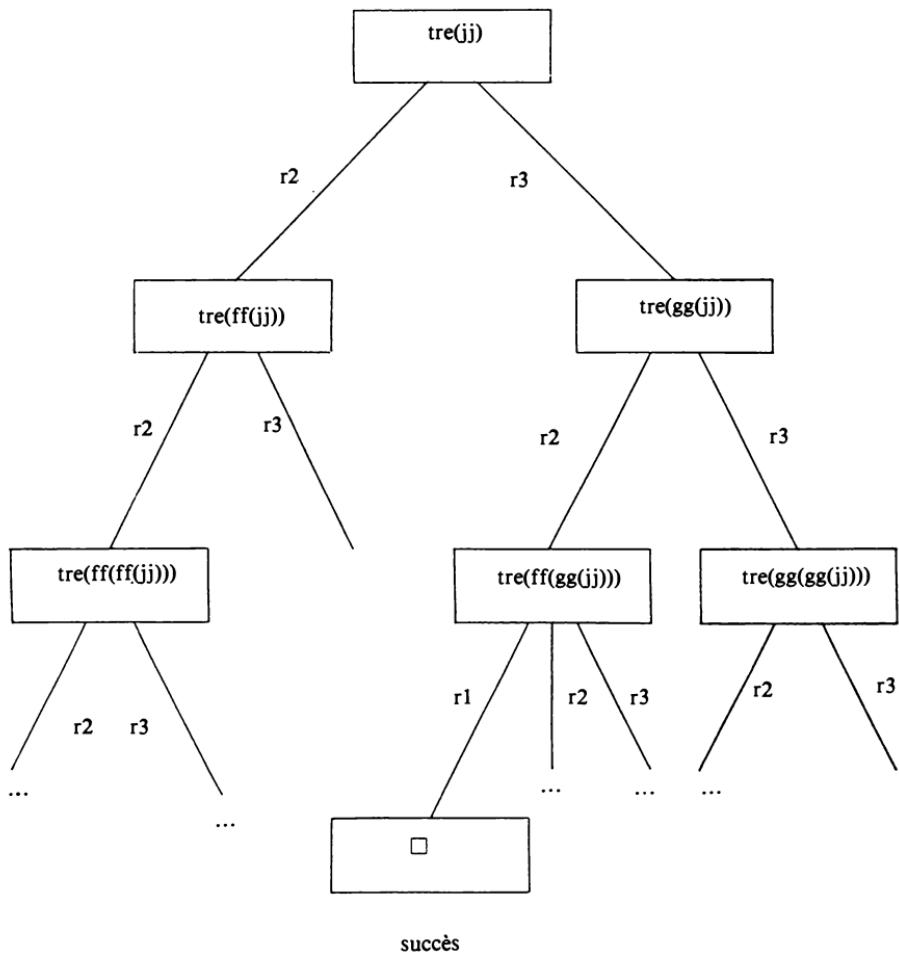


Fig. 28

La stratégie d'exploration de l'arbre des déductions input ordonnées étant la stratégie d'exploration « en profondeur d'abord avec retours en arrière », la branche la plus à gauche est empruntée et puisqu'elle est infinie, Prolog ne découvre pas le succès unique qui correspond à la branche r3, r2, r1.

Devant cet exemple, on est amené à penser qu'il faut modifier l'algorithme de résolution PROLOG de telle manière que l'exploration se fasse « en largeur d'abord » plutôt que « en profondeur d'abord ».

Malheureusement « l'explosion combinatoire » bien visible sur l'exemple (4), rend tout à fait irréaliste l'utilisation pratique d'un tel algorithme.

La solution proposée par les concepteurs et les défenseurs de PROLOG consiste à :

- introduire un peu de contrôle dans l'écriture des programmes (le / par exemple est présent dans toutes les versions de PROLOG),
- conseiller de bien écrire les programmes (!!).

Tout cela fait qu'il est faux de dire que PROLOG est un langage naturel et facile ; au contraire PROLOG est un langage délicat à manipuler et qui, comme tous les langages de programmation, nécessite un apprentissage et la compréhension des concepts impliqués.

4. — CONTRÔLE, ARITHMÉTIQUE, TRAITEMENT DE LISTES

Nous avons vu, au paragraphe 3, la nécessité de disposer de moyens de contrôle pour le parcours de l'arbre de résolution. L'un de ces moyens (le principal sans doute) est le « cut » qui, dans PROLOG II, est noté /. Le « cut » sert en particulier à définir la négation, mais en fait, il se révèle aussi, tout à fait indispensable, dès qu'on veut écrire des programmes récursifs (pour des calculs arithmétiques ou du traitement de listes par exemple) ou dès qu'on veut faire de la programmation impérative ce qui, même en Prolog, est parfois inévitable.

(a) Le « / »

Le fonctionnement de « / » peut être défini de la manière suivante :

- / est toujours vrai,
- pendant le parcours de l'arbre de résolution, si on fait du retour en arrière et si on arrive à un nœud dont la liste des buts commence par /, alors on remonte jusqu'au nœud correspondant au but précédent le but qui a déclenché l'appel de la règle contenant le /.

Autrement dit, il faut compléter l'algorithme de parcours décrit au paragraphe 3 et à chaque fois qu'il y a :

remonter au nœud précédent

il faut ajouter :

si la liste des buts du nœud précédent commence par / alors remonter jusqu'au nœud correspondant au but précédent le but qui a déclenché l'appel de la règle contenant le /.

Soit le programme :

```
rr(aa) —> ;  
rr(bb) —> ;  
rr(cc) —> ;  
pp(x) —> rr(x) / ;
```

A la question :

```
pp(x) ;
```

on obtient la réponse :

```
x = aa  
>
```

car l'arbre de parcours est :

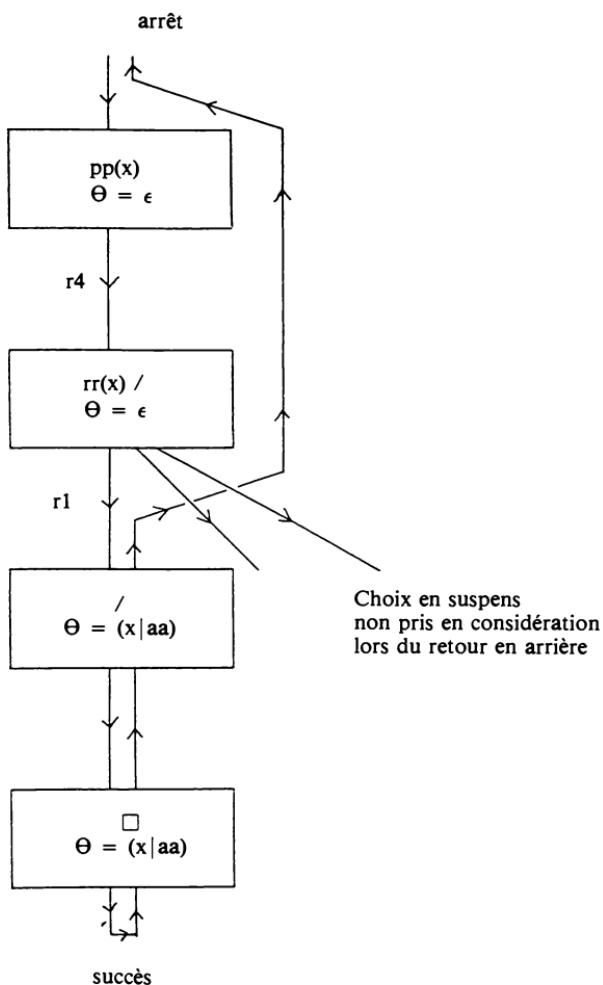


Fig. 29

Comme on le voit sur cet exemple, le / peut être utilisé pour n'obtenir qu'une réponse à un problème qui en comporte plusieurs. Il signifie, en quelque sorte :

« couper tous les choix en suspens entre le but qui appelle la règle contenant le / et le / lui-même ».

Dans les cas simples, et en particulier dans les questions comportant le /, dès qu'on remonte à un noeud de l'arbre dont la liste de buts commence par /, le parcours de l'arbre de résolution s'arrête.

Soit le programme :

```
rr(aa) —> ;  
rr(bb) —> ;  
rr(cc) —> ;  
ss(bb) —> ;  
pp(x) —> rr(x) ;
```

Si on pose la question :

```
pp(x) / ;
```

on obtient :

```
x = aa ;  
>
```

Si on pose la question :

```
/ pp(x) ;
```

on obtient :

```
x = aa  
x = bb  
x = cc  
>
```

Si on pose la question :

```
pp(x) / ss(x) ;
```

on obtient :

```
> (pas de solution)
```

Si on pose la question :

```
ss(x) / pp(x) ;
```

on obtient :

```
x = bb  
>
```

PROGRAMME

```
rr(aa) ->;  
rr(bb) ->;  
rr(cc) ->;  
  
pp(x) -> rr(x) /;
```

UTILISATION DU PROGRAMME

```
pp(x);  
x=aa  
>
```

```
rr(x);  
x=aa  
x=bb  
x=cc  
>
```

DEMANDE DE TRACE

```
trace;  
>
```

```
pp(x);  
pp(x)  
rr(aa)  
x=aa  
>
```

```
rr(x);
rr(aa)
x=aa
rr(bb)
x=bb
rr(cc)
x=cc
>
```

PROGRAMME

```
rr(aa) ->;
rr(bb) ->;
rr(cc) ->;

ss(bb) ->;
pp(x) -> rr(x);
```

UTILISATION DU PROGRAMME

```
pp(x);
x=aa
x=bb
x=cc
>
```

```
pp(x) /;
x=aa
>

/ pp(x);
x=aa
x=bb
x=cc
>
```

```
pp(x) / ss(x);  
>
```

```
ss(x) pp(x);  
x=bb  
>
```

Le programme suivant est un peu plus compliqué à démêler, en particulier pour la question :

```
ff(x) ;
```

En effet, c'est $hh(x)$ qui déclenche la règle contenant le / et donc quand on remonte et qu'on tombe sur le / après avoir donné la réponse $x = aa$, on reconnaît le but $gg(x)$ ce qui conduit à la réponse $x = bb$. (Voir trace et schéma).

PROGRAMME

```
gg(aa) ->;  
gg(bb) ->;  
  
hh(x) -> gg(x) /;  
  
ff(x) -> gg(x) hh(x);
```

UTILISATION DU PROGRAMME

```
gg(x);  
x=aa  
x=bb  
>
```

```
hh(x);  
x=aa  
>
```

```
ff(x);  
x=aa  
x=bb  
>
```

DEMANDE DE TRACE

```
trace;
```

```
>
```

```
gg(x);  
gg(aa)  
x=aa  
gg(bb)  
x=bb  
>
```

```
hh(x);  
hh(x)  
gg(aa)  
x=aa  
>
```

```
ff(x);  
ff(x)  
gg(aa)  
hh(aa)  
gg(aa)  
x=aa  
gg(bb)  
hh(bb)  
gg(bb)  
x=bb  
>
```

```
ff(aa);  
ff(aa)  
gg(aa)  
hh(aa)  
gg(aa)  
>
```

```
ff(bb);  
ff(bb)  
gg(bb)  
hh(bb)  
gg(bb)  
>
```

```
hh(aa);  
hh(aa)  
gg(aa)  
>
```

```
hh(bb);  
hh(bb)  
gg(bb)  
>
```

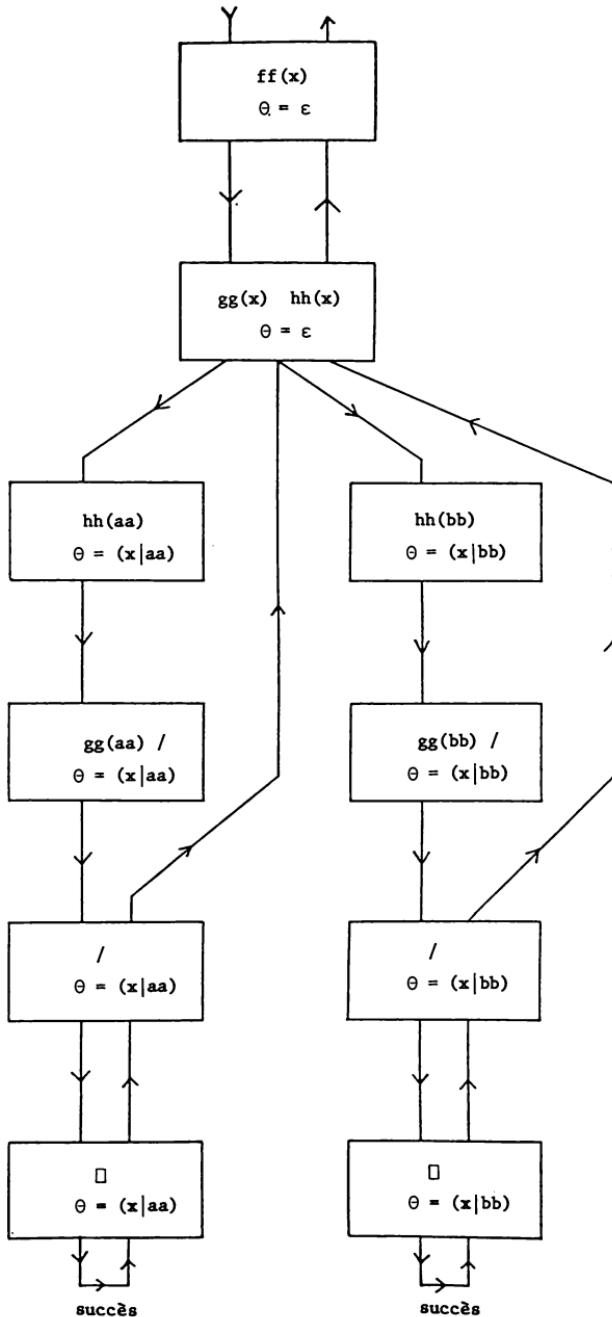


Fig. 30

(b) La négation

Un programme étant fixé, par exemple le programme des relations entre une famille de onze personnes, défini au paragraphe 2, on voudrait poser des questions avec des négations, du type :

Quels sont les grands-parents de Paul qui ne sont pas parents de Lucie ?

Pour cela, on ajoute simplement au programme initial :

non(p) —> p / impasse ;
non(p) —> ;

et on pose la question :

est-grand-parent(x,paul) non(est-parent(x,lucie)) ;

ce qui donne bien les réponses attendues :

x = pierre
x = andrée
>

Analysons ce qui se passe :

La question non(pp) déclenche la première règle de tête non, on obtient donc les 3 buts :
pp / impasse.

Si pp n'est pas satisfait, on revient en arrière et la deuxième règle de tête non remplace le but non(pp) par \square . On a un succès, puis arrêt.

Par contre, si pp est satisfait, les nouveaux buts sont : / impasse. Puis simplement : impasse.

Comme le prédicat impasse n'a pas été utilisé ailleurs dans le programme, le but impasse n'est pas satisfait, donc on remonte, on tombe sur le /, on s'arrête⁽¹⁾.

En résumé :

si p n'est pas satisfait : succès, arrêt.
si p est satisfait : arrêt.

Le non fonctionne donc comme voulu avec tout prédicat pp instancié.

En analysant ce qui se passe avec un prédicat non instancié, on voit que la question :

non(pp(x)) ;

conduit à un échec s'il existe x tel que pp(x)
et conduit à un succès sinon.

(1) N'importe quel prédicat sans variable conduisant à un échec pourrait être utilisé à la place de impasse dans la définition du non.

Il en résulte que :

non(pp(x)) ;

doit être compris comme :

non($\exists x \ pp(x)$)

c'est-à-dire :

$\forall x \ \text{non } pp(x)$

Il ne faut donc pas croire que la question non(pp(x)) ; donnera tous les x tels que pp(x) n'est pas vrai.

PROGRAMME

```
mASCULIN(jean) ->;  
.  
.  
.  
.  
.  
est-tante(x,y) -> est-soeur(x,z) est-parent(z,y);  
  
non(p) -> p / impasse;  
non(p) ->;
```

UTILISATION DU PROGRAMME

```
est-grand-parent(x,paul) non(est-parent(x,lucie));  
x=pierre  
x=andree  
>
```

```
non(est-parent(x,lucie));  
>
```

```
non(est-parent(x,leon));  
>
```

```
non(non(est-parent(x,lucie)));
```

```
>
```

```
non(non(est-parent(x,leon)));
```

```
>
```

```
mASCULIN(x) non(est-parent(x,alain));
```

```
x=paul
```

```
x=alain
```

```
x=pierre
```

```
x=leon
```

```
x=luc
```

```
x=jacques
```

```
>
```

```
fEMININ(x) non(mASCULIN(x));
```

```
x=lucie
```

```
x=claire
```

```
x=andree
```

```
x=martine
```

```
>
```

DEMANDE DE TRACE

```
trace;
```

```
>
```

```
non(est-parent(jean,alain));
```

```
non(est-parent(jean,alain))
```

```
est-parent(jean,alain)
```

```
est-pere(jean,alain)
```

```
>
```

```
non(est-parent(x,lucie));  
non(est-parent(x,lucie))  
est-parent(x,lucie)  
est-pere(leon,lucie)  
>
```

```
non(est-parent(x,leon));  
non(est-parent(x,leon))  
est-parent(x,leon)  
est-parent(x,leon)  
non(est-parent(x,leon))
```

```
>
```

```
non(est-frere(paul,alain));  
non(est-frere(paul,alain))  
est-frere(paul,alain)  
masculin(paul)  
est-pere(jean,paul)  
est-pere(jean,alain)  
dif(paul,alain)  
>
```

```
non(est-frere(paul,jean));  
non(est-frere(paul,jean))  
est-frere(paul,jean)  
masculin(paul)  
est-pere(jean,paul)  
non(est-frere(paul,jean))
```

```
>
```

(b) Récursivité et arithmétique élémentaire

PROLOG permet de résoudre assez simplement certains problèmes d'arithmétique élémentaire, ayant éventuellement plusieurs solutions. A titre d'exemple, nous allons construire un programme qui donnera toutes les solutions de l'équation :

$$x + y + z = 10 \quad x,y,z, \text{ entiers non nuls.}$$

Et nous verrons que le programme écrit pour cela permet de résoudre bien d'autres questions.

Volontairement, nous n'allons pas utiliser l'arithmétique de PROLOG II que nous verrons en (d).

On définit d'abord le prédictat `successeur(x,y)` (qui doit se comprendre « `x` est le successeur de `y` ») pour tous les entiers plus petits que 10 :

```
successeur(2,1) —> ;  
successeur(3,2) —> ;  
...  
successeur(10,9) —> ;
```

On définit ensuite la somme de deux nombres : `somme2(x, y, z)` (qui doit se comprendre « `x` est la somme de `y` et `z` ») :

```
somme2(x,1,y) —> successeur(x,y) ;  
somme2(x,y,z) —> successeur(y,y')  
                          somme2(x',y',z)  
                          successeur(x,x') ;
```

On définit ensuite la somme de trois nombres :

```
somme3(x,y,z,t)  
(qui doit se comprendre « x est la somme de y,z et t ») :  
somme3(x,y,z,t) —> somme2(x',y,z)  
                          somme2(x,x',t) ;
```

Ceci constitue le programme auquel on pose la question :

```
somme3(10,y,z,t) ;
```

Bien sûr, le même programme permet de résoudre des équations plus compliquées. Par exemple, pour le système d'équations :

$$\begin{cases} x = 2 + y + t \\ 9 = x + t \end{cases} \quad \text{x,y,t entiers non nuls}$$

il suffit de poser la question :

```
somme3(x,2,y,t) somme2(9,x,t) ;
```

Le programme définissant `somme2` est dit *récursif* car, dans la définition de `somme2`, il est fait appel à `somme2`.

Nous verrons plus loin que pour la manipulation des listes, l'usage de la récursivité est aussi très utile.

PROGRAMME

```
successeur(2,1) ->;
successeur(3,2) ->;
successeur(4,3) ->;
successeur(5,4) -> . 
successeur(6,5) ->;
successeur(7,6) ->;
successeur(8,7) ->;
successeur(9,8) ->;
successeur(10,9) ->; 

somme2(x,1,y) -> successeur(x,y);
somme2(x,y,z) ->
    successeur(y,y')
    somme2(x',y',z)
    successeur(x,x');

somme3(x,y,z,t) ->
    somme2(x',y,z)
    somme2(x,x',t);
```

UTILISATION DU PROGRAMME

```
somme2(x,2,2);
```

```
x=4
```

```
>
```

```
somme3(x,2,3,2);
```

```
x=7
```

```
>
```

```
somme2(4,2,x);
```

```
x=2
```

```
>
```

```
somme2(5,x,y);
x=1  y=4
x=2  y=3
x=3  y=2
x=4  y=1
>
```

```
somme3(10,x,y,z);
x=1  y=1  z=8
x=1  y=2  z=7
x=1  y=3  z=6
x=1  y=4  z=5
x=1  y=5  z=4
x=1  y=6  z=3
x=1  y=7  z=2
x=1  y=8  z=1
x=2  y=1  z=7
x=2  y=2  z=6
x=2  y=3  z=5
x=2  y=4  z=4
x=2  y=5  z=3
x=2  y=6  z=2
x=2  y=7  z=1
.
.
.
.
.
>
```

```
somme3(x,2,y,t)  somme2(9,x,t);
x=6  y=1  t=3
x=7  y=3  t=2
x=8  y=5  t=1
>
```

Arbre de résolution partiel pour la question somme2(3,y,z)

(Les notations ont été simplifiées)

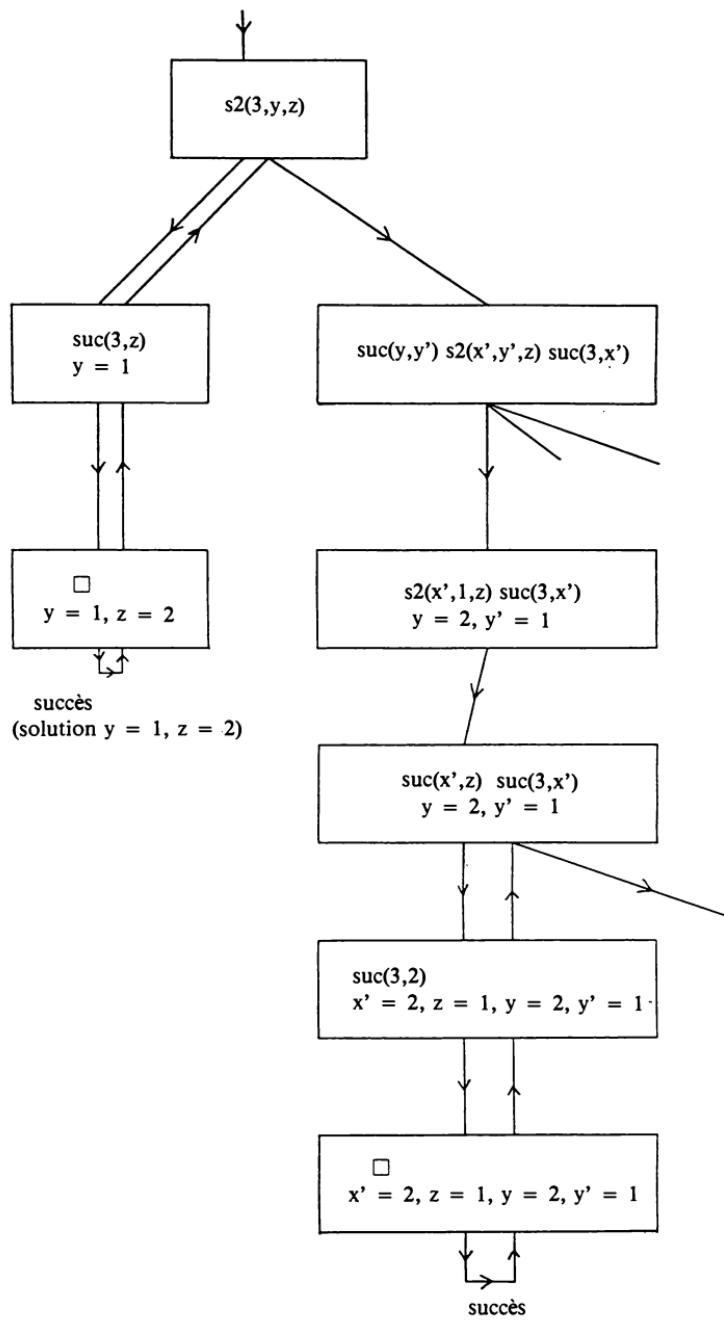


Fig. 31

(solution $y = 2, z = 1$)

(d) L'arithmétique prédéfinie de PROLOG II

PROLOG II permet la manipulation des entiers sans qu'il soit nécessaire de procéder comme en (c). On utilise le prédictat `val` et les fonctions évaluables prédéfinies.

Par exemple, on obtient le prédictat `carre(x,y)` (qui doit se comprendre « `x` est le carré de `y` ») avec la règle :

`carre(x,y) —> val(mu1(y,y),x) ;`

Le prédictat `val(t1,t2)` évalue le terme `t1` et ajoute la contrainte `valeur(t1) = t2`. Le terme `t1` ne doit pas contenir de variables non instanciées. Les fonctions évaluables les plus utiles sont :

`add, sub, mu1, div, mod`

pour l'addition, la soustraction, la multiplication, la division, la valeur modulo.

`inf(x,y)` donne 1 ou 0 selon que $x \leq y$ ou $x > y$

`eq(x,y)` donne 1 ou 0 selon que $x = y$ ou $x \neq y$

`si(t,t1,t2)` donne : t_1 si $t = 0$
 t_2 sinon.

Le programme suivant définit les prédictats :

<code>cube(x,y)</code>	: x est le cube de y
<code>puissance(x,y,z)</code>	: x est y à la puissance z
<code>sigma1(x,y)</code>	: x est la somme des entiers i pour i allant de 1 à y
<code>sigma2(x,y)</code>	: x est la somme du carré des entiers i pour i allant de 1 à y
<code>sigmage(x,y,z)</code>	: x est la somme des i à la puissance y pour i allant de 1 à z
<code>inférieur(x,y)</code>	: x est inférieur à y .

PROGRAMME

```
carre(x,y) —> val(mu1(y,y),x);  
cube(x,y) —> carre(x',y) val(mu1(x',y),x);
```

```

puissance(0,0,y) -> /;
puissance(y,y,1) -> /;
puissance(x,y,z) ->
    val(sub(z,1),z')
    puissance(x',y,z')
    val(mul(x',y),x);

sigmal(0,0) -> /;
sigmal(x,y) ->
    val(sub(y,1),y')
    sigmal(x',y')
    val(add(x',y),x);

sigma2(0,0) -> /;
sigma2(x,y) ->
    val(sub(y,1),y')
    sigma2(x',y')
    carre(z,y)
    val(add(x',z),x);

sigmage(0,j,0) -> /;
sigmage(x,j,n) ->
    val(sub(n,1),n')
    sigmage(x',j,n')
    puissance(t,n,j)
    val(add(x',t),x);

inferieur(x,y) -> val(inf(x,y),1);

```

UTILISATION DU PROGRAMME

```

carre(x,3);
x=9
>
```

```

cube(x,2);
x=8
>
```

```
carre(4,x);
```

TERME INCORRECT DANS EVALUER

>

```
puissance(x,0,5);
x=0
>
```

```
puissance(x,4,1);
x=4
>
```

```
puissance(x,2,3);
x=8
>
```

```
puissance(x,3,2);
x=9
>
```

```
sigmal(x,0);
x=0
>
```

```
sigma2(x,6);
x=91
>
```

```
sigimage(x,1,3);
x=6
>
```

```
sigimage(x,2,3);
x=14
>
```

```
sigimage(x,3,2);
```

```
x=9
```

```
>
```

```
sigimage(x,4,5);
```

```
x=979
```

```
>
```

```
inferieur(2,3);
```

```
>
```

```
inferieur(3,2);
```

```
>
```

(e) Manipulation des listes.

Les listes en PROLOG II sont notées simplement en séparant les différents éléments par des points.

AA.BB.CC.DD par exemple, est une liste.

Chaque liste est implicitement parenthésée de gauche à droite et donc AA.BB.CC.DD est le même objet que AA.(BB.(CC.DD)) et est différent de l'objet ((AA.BB).CC).DD.

On a illustré cela dans le programme qui suit en posant les questions

```
eg(AA.BB.CC,AA.(BB.CC)) ; et
```

```
eg(AA.BB.CC,(AA.BB).CC) ;
```

Le prédictat eg(x,y) est prédéfini, il est vrai si $x = y$, faux sinon. La règle qui le définit s'écrit simplement :

```
eg(x,x) —> ;
```

Il est souvent intéressant de convenir de terminer toutes les listes par « vide » et de ne considérer comme élément de la liste que les éléments différents de vide. Les éléments de AA.BB.vide seront simplement AA et BB,

Cette convention permet de repérer la fin de la liste et est très utile dans les définitions récursives. Dans le programme qui suit, nous avons adopté cette convention.

Le prédictat liste(x) signifie « x est une liste » (donc avec notre convention « x est vide ou x est de la forme $x_1 . x_2 . \dots . x_n . \text{vide}$ »)

Le prédictat élément(x,y) signifie « x est un élément de la liste y ».

Il y a plusieurs manières de définir le prédicat de concaténation de listes.
Nous en proposons deux :

conc1(x,y,z) , conc2(x,y,z)

qui doivent se comprendre « x est la liste obtenue par concaténation de y et de z ».

On remarquera que conc2 fonctionne correctement aussi bien pour des questions du type :

conc2(x,AA.BB.vide,CC.vide) ;

que pour des questions du type :

conc2(AA.BB.CC.vide,x,y) ;

ce qui n'est pas le cas de conc1.

Plusieurs définitions sont parfois possibles et selon l'utilisation qui va être faite du prédicat à définir, l'une est meilleure ou c'est une autre.
C'est ce qui se passe pour le prédicat inverse.

Inverse1(x,y) et inverse2(x,y) qui doivent se comprendre « x est la liste renversée de y » fonctionnent correctement pour y instancié, x variable, et ne fonctionnent pas pour y variable et x instancié (le premier répond NON, le second boucle). Il serait possible de définir un inverse3 qui fonctionnerait pour x instancié et y variable⁽¹⁾.

Le prédicat insertion(x,y,z) doit se comprendre « x est une liste obtenue par insertion de y dans la liste z ».

Le prédicat permutation(x,y) doit se comprendre « x est une liste obtenue par permutation des éléments de la liste y ».

(1) L'utilisation des prédicats prédéfinis pris(x), libre(x) et du / permet de regrouper des prédicats fonctionnant dans des situations différentes, pour en définir un qui fonctionne toujours.

PROGRAMME

```
liste(vide) ->;
liste(x.y) -> liste(y);

element(x,vide) -> / impasse;
element(x,x.y) ->;
element(x,y.z) -> element(x,z);

concl(x,vide,x) -> /;
concl(x.y,x.z,t) -> concl(y,z,t);

conc2(x.y,x.z,t) -> conc2(y,z,t);
conc2(x,vide,x) ->;

inverse1(vide,vide) -> /;
inverse1(z,x.y) -> inverse1(y',y) conc2(z,y',x.vide);

inverse2(z,x.y) -> inverse2(y',y) conc2(z,y',x.vide);
inverse2(vide,vide) ->;

insertion(e.x,e,x) ->;
insertion(f.y,e,f.x) -> insertion(y,e,x);

permutation(vide,vide) ->;
permutation(z,e.x) -> permutation(y,x) insertion(z,e,y);
```

UTILISATION DU PROGRAMME

```
liste(vide);
>

liste(LL.II.SS.TT.EE.vide);
>
```

```
liste(jean);
>

liste(AA.SS(ee,ff).BB.vide);
```

```
>

liste(x.y.vide);
```

```
liste(x);
x=vide
x=x4.vide
x=x4.x7.vide
x=x4.x7.x9.vide
x=x4.x7.x9.x12.vide
x=x4.x7.x9.x12.x14.vide
x=x4.x7.x9.x12.x14.x17.vide
.
.
.
```

```
liste(AA.(BB.CC.DD).EE.vide);
>
```

```
liste(AA.(BB.CC.(EE.FF).GG).HH.((II)).vide);
>
```

```
element(AA,vide);
>

element(EE,AA.BB.CC.DD.EE.FF.GG.vide);
>

element(EE,AA.BB.CC.EE.DD.FF.GG.vide);
>

element(CC,AA.(BB.CC.DD).EE.vide);
>

element(CC,AA.(BB.(CC.DD.vide)));
>

eg(AA,AA);
>

eg(AA.BB.CC,AA.(BB.CC));
>

eg(AA.BB.CC,(AA.BB).CC);
>

conc1(x,AA.DD.vide,AA.AA.AA.AA.vide);
x=AA.DD.AA.AA.AA.AA.vide
>

conc1(AA.BB.CC.vide,x,y);
x=vide  y=AA.BB.CC.vide
>
```

```
conc2(x,AA.BB.CC.vide,DD.EE.FF.vide);
x=AA.BB.CC.DD.EE.FF.vide
>

conc2(AA.BB.CC.DD.vide,x,y);
x=AA.BB.CC.DD.vide y=vide
x=AA.BB.CC.vide y=DD.vide
x=AA.BB.vide y=CC.DD.vide
x=AA.vide y=BB.CC.DD.vide
x=vide y=AA.BB.CC.DD.vide
>

conc2(AA.(BB.CC.DD).EE.vide,x,y);
x=AA.(BB.CC.DD).EE.vide y=vide
x=AA.(BB.CC.DD).vide y=EE.vide
x=AA.vide y=(BB.CC.DD).EE.vide
x=vide y=AA.(BB.CC.DD).EE.vide
>

inversel(z,AA.BB.CC.DD.vide);
z=DD.CC.BB.AA.vide
>

inversel(z,AA.(BB.CC.DD).EE.vide);
z=EE.(BB.CC.DD).AA.vide
>

inversel(AA.BB.vide,x);
>

inverse2(z,AA.BB.CC.vide);
z=CC.BB.AA.vide
>

inverse2(z,AA.BB.(CC.DD.EE).FF.vide);
z=FF.(CC.DD.EE).BB.AA.vide
>
```

```
inverse1(x.y.z,AA.BB.CC.DD.EE.vide);
x=EE  y=DD  z=CC.BB.AA.vide
>
```

```
insertion(x,AA,BB.CC.vide);
x=AA.BB.CC.vide
x=BB.AA.CC.vide
x=BB.CC.AA.vide
>
```

```
insertion(AA.BB.CC.DD.vide,x,AA.BB.DD.vide);
x=CC
>
```

```
permutation(z,AA.BB.CC.vide);
z=AA.BB.CC.vide
z=BB.AA.CC.vide
z=BB.CC.AA.vide
z=AA.CC.BB.vide
z=CC.AA.BB.vide
z=CC.BB.AA.vide
>
```

```
permutation(AA.BB.CC.vide,z);
z=AA.BB.CC.vide
z=BB.AA.CC.vide
z=CC.AA.BB.vide
z=AA.CC.BB.vide
z=BB.CC.AA.vide
z=CC.BB.AA.vide
>
```

EXERCICES

Exercices sur le premier programme du chapitre

- 1) Dessiner l'arbre de résolution Prolog correspondant à la question :
est-grand-parent(lucie,x) ;
- 2) Même question pour :
est-frère (paul,alain) ;
est-oncle (jacques,paul) ;
- 3) Compléter le programme en définissant les prédictats supplémentaires suivants :
est-neveu (x,y)
est-nièce (x,y)
a-des-enfants (x)
a-des-frères (x)
a-des-sœurs (x)
est-ancêtre (x,y)
- 4) Compléter le programme en ajoutant d'autres personnages avec leurs relations familiales. Définir ensuite les prédictats :
a-eu-des-enfants-avec (x,y)
est-demi-frère (x,y)

Exercice sur un exemple du chapitre 1

Ecrire un programme correspondant à l'exemple étudié pour décrire les moteurs d'inférences des systèmes experts au chapitre 1.

Dessiner l'arbre de résolution Prolog obtenu pour la question H.

La première règle du programme sera :

FF — > BB DD EE ;

Exercice sur la négation

A l'aide d'arbres détaillés de résolution Prolog, étudier ce que donnent les questions :

non (p) ;
non (non (p)) ;
non (non (non (p))) ;
...

lorsque p est un prédictat entièrement instancié et lorsque p contient des variables.

En tirer une règle générale.

Exercice sur la redéfinition de l'arithmétique

Compléter le programme du paragraphe 4 (c) et définir des prédictats pour la multiplication (`multi2(x,y,z)`), pour la puissance (`puissance(x,y,z)`), pour le calcul de factoriel (`fact (x,y)`). De manière à disposer d'un peu plus d'entiers que dans le cours ajouter :

```
successeur(11,10) ;  
successeur(12,11) ;  
.....  
successeur(50,49) ;
```

Exercices sur l'arithmétique et Prolog

- 1) Définir le prédictat :

pour(`x, y, z, t`)

qui pour la question :

pour(`x, 4, 9, 2`) :

donne :

```
x = 4  
x = 6  
x = 8  
>
```

et qui de manière générale à chaque fois que `y, z` et `t` sont instanciés par des entiers donne comme solution tous les `x` entre `y` et `z` par pas de `t`.

- 2) Définir le prédictat :

premier(`x`)

qui pour la question :

premier(`5`) ;

donne un succès ; qui pour la question

premier(`25`) ;

donne un échec et qui de manière générale donne un succès pour `x` instancié par un nombre entier premier et donne un échec pour `x` instancié par un nombre entier non premier.

- 3) Définir le prédictat :

liste-premier(`x`)

qui pour la question :

liste-premier(`100`) ;

donne tous les entiers premiers inférieurs à 100 et qui de manière générale lorsque `x` est instancié par un entier donne successivement tous les entiers premiers inférieurs à cet entier.

4) Définir le prédictat :

premiers-entre-eux (x, y)

qui à chaque fois que x et y sont instanciés par des entiers indique par un succès ou un échec si ces entiers sont premiers entre eux ou non.

5) Définir le prédictat :

solution (x, y, z)

qui donne tous les triplets x, y, z d'entiers positifs vérifiant :

$$\text{et } \begin{cases} x^2 + y^2 + z^2 \leq 1000 \\ (x + y)^2 = (x + z)^2 \\ \text{ou} \\ (x^2 = y^2) \end{cases}$$

et $0 \leq x \leq 20$

et $0 \leq y \leq 20$

et $0 \leq z \leq 20$

Exercices sur les listes

1) Définir le prédictat :

min (x, ℓ)

qui lorsque ℓ est instancié par une liste d'entiers donne à x la valeur minimum des éléments de la liste.

Par exemple

min ($x, 2.5.3.1.2. \text{ vide}$) ;

doit donner

$x = 1$

>

2) Même exercice pour le maximum.

3) Définir le prédictat :

mélange (x, ℓ_1, ℓ_2)

qui lorsque ℓ_1 et ℓ_2 sont instanciés par des listes (par exemple AA.BB.vide et LL.MM.NN.OO.vide) donne à x la valeur correspondant à la liste obtenue en mélangeant ℓ_1 et ℓ_2 (dans notre exemple $x = AA.LL.BB.MM.NN.OO.vide$).

BIBLIOGRAPHIE

- [APT 82] APT K. R. et VAN EMDEN M. H.
Contribution to the Theory of Logic Programming.
Journal of the Association for Computing Machinery, Vol. 29, n° 3, 1982,
pp. 841, 862.
{Sur la sémantique de Prolog}
- [CLA 78] CLARK K. L.
Negation as Failure.
Logic and Data Bases Gallaire et Minker eds., Plenum, New York, 1978.
{Etude détaillée du problème de la négation en programmation logique, et
en Prolog}
- [CLA 85] CLARK K. L. et Mc CABE F. G.
Micro-Prolog : Programmer en Logique.
Editions Eyrolles, Paris, 1985.
(Traduction de : *Micro-Prolog : Programming in Logic*).
{Manuel pour le grand public, expliquant l'utilisation de la version pour
ordinateurs familiaux du langage Prolog}
- [CLO 85] CLOCKSIW. F. et MELLISH C. S.
Programmer en Prolog.
Editions Eyrolles, Paris, 1985.
(Traduction de : *Programming in Prolog*. Springer Verlag, Berlin, 1981).
{Manuel de programmation en Prolog, pour le Prolog dit « Prolog d'Edin-
burgh »}
- [COE 80] COELHO H., COTTA J. C., PEREIRA L. M.
How to Solve it with Prolog.
Laboratório Nacional de Enghenharia Civil, Lisboa, 1980.
{Manuel d'exemples de programmes en Prolog}
- [COL 82] COLMERAUER A.
Prolog II. Manuel de Référence : Théorique et Pratique.
Groupe Intelligence Artificielle, E.R.A. C.N.R.S. 363. Faculté des Sciences
de Luminy 13288, Marseille Cedex 9, 1982.
{Modèle théorique de Prolog II fondé sur les notions d'arbres finis et
infinis}
- [COL 82] COLMERAUER A.
Prolog and Infinite Trees.
Logic Programming K. L. Clark et S. A. Tarnlund eds. Academic Press,
Londres, 1982.
{Reprend et complète le manuel théorique cité juste au-dessus}
- [COL 83] COLMERAUER A., KANQUI H. et VAN CANEGHEM M.
Prolog, bases théoriques et développements actuels.
Techniques et Sciences Informatiques, 1983, pp. 271-311.
{Présentation assez complète et très dense de Prolog II}

- [COM 83] COMYN G.
Notions de Prolog.
Groupe de Programmation Enseignement et Recherche I.U.T. « A » de Lille, Villeneuve-d'Ascq, 1983.
{Exposé général sur Prolog}
- [DON 84] DONZ P. et HURTADO R.
Le Langage D-Prolog.
Editest, Paris, 1984.
{Manuel de présentation et d'utilisation de la version D-Prolog}
- [GAL 83] GALLAIRE H.
A study of Prolog
Computer Program Synthesis Methodologies.
Birmann A.-W. et Guiho G. eds.
Reidel Publishing Company, Dordrecht (Holland), 1983, pp. 173-212.
{Excellenté présentation générale de Prolog}
- [GIA 85] GIANNESINI F., KANOUI H., PASERO R. et VAN CANEGHEM M.
Prolog
InterEdition, Paris, 1985.
{Par l'équipe de Marseille, très bon manuel de programmation en Prolog II}
- [GRO 85] GROUPE PROLOG DE L'AFCET
Prolog : Fondements et Applications.
Dunod, Paris, 1985.
- [KAN 82] KANOUI H.
Prolog II. Manuel d'Exemples.
Groupe Intelligence Artificielle E.R.A. C.N.R.S. 363.
Faculté des Sciences de Luminy 13288, Marseille, Cedex 9, 1982.
- [VAN 82] VAN CANEGHEM M.
Prolog II. Manuel d'Utilisation.
Groupe Intelligence Artificielle E.R.A. C.N.R.S. 363.
Faculté des Sciences de Luminy, 13288 Marseille, Cedex 9, 1982.
- [VAN 84] VAN EMDEN M. H. et LLOYD J. W.
A Logical Reconstruction of Prolog II.
The Journal of Logic Programming. Vol. 1 n° 2, 1984.

INDEX

A

Algorithme d'unification, 152.
Alphabet, 54.
Arbre de dérivation, 57.
Arbre sémantique, 134.
Arithmétique, 112.
Assignation, 77.
Atome, 72, 94.
Atome de Herbrand, 131.
Axiome, 55.

B

Base de connaissance, 13.
Base de données, 185.
Base de faits, 13.
Basic, 34.
Bijection, 30.
But, 15.

C

Calcul des prédicts, 93
Calcul propositionnel, 71.
Calculable, 38.
Chaînage arrière, 14.
Chaînage avant, 14.
Clause, 80.
Clause de Horn, 179, 183, 188, 189.
Clause négative, 179.
Clause unité, 167.
Close, 98.
Coefficient de vraisemblance, 13, 18.
Cohérent, 59.

Cohérente, 109.
Colmerauer, 194.
Complet, 59.
Complexité d'une formule, 82.
Connecteur, 77.
Consistant, 59, 105, 109.
Constante, 94.
Conséquence, 78, 106.
Contradictoire, 59, 109.
Correct, 59.

D

Davis et Putnam, 139.
Décidable, 45, 59.
Décision, 59.
Déduction, 56.
Déduction input, 177.
Déduction linéaire, 173.
Déduction ordonnée, 181.
Démonstration automatique, 124, 129.
Dendral, 12.
Diminution, 156.
Dreyfus, 9.

E

Echecs, 8.
Ensemble fini, 31.
Ensemble infini, 31.
Ensemble non dénombrable, 31.
Ensemble récursif, 41.
Ensemble récursivement énumérable, 41.
Equivivalence, 106.

Exploration d'arbres, 169.
Extraction de réponses, 183.

F

Falsification, 106.
Finiment axiomatisable, 59.
Fonction non récursive, 39.
Fonction récursive, 38.
Forme de Skolem, 127.
Forme normale conjonctive, 80.
Forme normale disjonctive, 80.
Forme prénexe, 124.
Formule, 96.
Formule close, 98.
Formules bien formées, 55.
Formules équivalentes, 116.
Fortran, 34.
Fractal, 1.

G

Généralisation, 100.
Gödel, 9, 59, 113.
Graphe de dérivation, 57.

H

Herbrand, 130.
Hilbert, 59.
Horn, 179.
Hypothèse, 56.

I

Inconsistant, 59, 78, 105.
Indépendant, 59.
Infini dénombrable, 31.
Infini non dénombrable, 31.
Insatisfiable, 78.
Interprétation, 58, 76, 101.

J

Jeu de l'imitation, 6.
Jeux, 8.

K L

Langues naturelles, 8.
Largeur d'abord, 171.
Liste en Prolog, 213.
Littéral, 145.

M

Machine de Turing, 30, 40.
Modèle, 78, 105.
Modèle de Herbrand, 132.
Modèle égalitaire, 112.
Modus ponens, 7, 73.
Modus tollens, 17.
Moteur d'inférences, 13, 47, 64.
Mycin, 12.

N O

Négation, 81, 117.
Négation en Prolog, 222.
Non contradictoire, 59.
NP-complet, 144.
Numérotation, 31.

P

Pascal, 34.
Plus grand unificateur, 152.
Prédicat, 4.
Prédicat décidable, 45.
Prédicat semi-décidable, 45.
Préférence des clauses simples, 166.
Pénexe, 124.
Problème de la décision, 59.
Profondeur d'abord, 170.
Programme universel, 36.
Programme à une entrée, 34.
Prolog, 47, 183, 185, 195.
Proposition atomique, 72.
Prospector, 12.

Q R

Réalisation, 77.
Reconnaissance des formes, 8.
Récursif, 41.
Récursivement énumérable, 41.
Règle d'inférence, 55.
Renommage, 99.
Résolution avec variables, 155.
Résolution Prolog, 202.
Résolution sans variable, 144.
Résolvante, 156.
Roussel, 194.

S

Satisfiable, 78, 105, 109.
Saturation, 164.
Saturation avec simplification, 165.
Saturé, 59.
Schémas d'axiomes, 73.
Sémantique, 72.
Semi-décidable, 45.
Skolem, 127.
Stratégie complète, 165.
Stratégie correcte, 165.
Stratégie input, 177.
Stratégie linéaire, 173.
Stratégie ordonnée, 180.
Substitution, 180, 89, 99, 150.
Subsumer, 165.
Symbole de prédicat, 95.
Symbole fonctionnel, 94.
Symbole relationnel, 95.
Système de Herbrand, 131.
Systèmes experts, 8, 13.
Systèmes formels, 54.

T U V

Table de vérité, 87.
Tautologie, 77, 104.

Terme, 94.
Test de Turing, 6.
Théorie axiomatique, 108.
Théorie égalitaire, 112.
Théorème, 56.
Théorème de compacité, 85, 107.
Théorème de complétude, 81, 107.
Théorème de déduction, 74, 100.
Théorème de finitude, 85, 107.
Théorème de Herbrand, 132.
Théorème de Löwenheim-Skolem, 108.
Théorème de Skolem, 127.
Thèse, 104.
Thèse de Church, 38.
Traduction automatique, 8.
Turing, 6, 9, 37.
Unificateur, 152.
Unification, 152.
Univers de Herbrand, 131.
Validité syntaxique, 79, 106.
Validité sémantique, 79, 106.
Valuation, 77.
Variable, 94.
Variable libre, 98.
Variable liée, 98.
Variable propositionnelle, 72.

Imprimerie de la Manutention à Mayenne
Dépôt légal : décembre 1986
N° d'Editeur : 4593