# Langage de spécifications

Michel Caplain

# THESE

*présentée à*

## Institut National Polytechnique de Grenoble

*pour obtenir le grade de*
DOCTEUR ES SCIENCES
Informatique

*par*

**Michel CAPLAIN**

# LANGAGE de SPECIFICATIONS

Thèse soutenue le 20 décembre 1978 devant la Commission d'Examen :

Président : B. VAUQUOIS

Examinateurs : J.R. ABRIAL
L. BOLLIET
A. COLMERAUER
P. DESVERGNES
A. GUILLON
P. JORRAND

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1977-1978

Président : M. Philippe TRAYNARD

Vice-présidents : M. René PAUTHENET

M. Georges LESPINARD

---

## PROFESSEURS TITULAIRES

| | | |
|---|---|---|
| MM. | BENOIT Jean | Electronique - automatique |
| | BESSON Jean | Chimie minérale |
| | BLOCH Daniel | Physique du solide - cristallographie |
| | BONNETAIN Lucien | Génie chimique |
| | BONNIER Etienne | Métallurgie |
| * | BOUDOURIS Georges | Electronique - automatique |
| | BRISSONNEAU Pierre | Physique du solide - cristallographie |
| | BUYLE-BODIN Maurice | Electronique - automatique |
| | COUMES André | Electronique - automatique |
| | DURAND Francis | Métallurgie |
| | FELICI Noël | Electronique - automatique |
| | FOULARD Claude | Electronique - automatique |
| | LANCIA Roland | Electronique - automatique |
| | LONGEQUEUE Jean-Pierre | Physique nucléaire corpusculaire |
| | LESPINARD Georges | Mécanique |
| | MOREAU René | Mécanique |
| | PARIAUD Jean-Charles | Chimie - physique |
| | PAUTHENET René | Electronique - automatique |
| | PERRET René | Electronique - automatique |
| | POLOUJADOFF Michel | Electronique - automatique |
| | TRAYNARD Philippe | Chimie - physique |
| | VEILLON Gérard | Informatique fondamentale et appliquée |

* en congé pour études

## PROFESSEURS SANS CHAIRE

| | | |
|---|---|---|
| MM. | BLIMAN Samuël | Electronique - automatique |
| | BOUVARD Maurice | Génie mécanique |
| | COHEN Joseph | Electronique - automatique |
| | GUYOT Pierre | Métallurgie physique |
| | LACOUME Jean-Louis | Electronique - automatique |
| | JOUBERT Jean-Claude | Physique du solide - cristallographie |

.../...

| | | |
|---|---|---|
| MM. | ROBERT André | Chimie appliquée et des matériaux |
| | ROBERT François | Analyse numérique |
| | ZADWORNY François | Electronique - automatique |

MAITRES DE CONFERENCES

| | | |
|---|---|---|
| MM. | ANCEAU François | Informatique fondamentale et appliquée |
| | CHARTIER Germain | Electronique - automatique |
| | CHIAVERINA Jean | Biologie, biochimie, agronomie |
| | IVANES Marcel | Electronique - automatique |
| | LESIEUR Marcel | Mécanique |
| | MORET Roger | Physique nucléaire - corpusculaire |
| | PIAU Jean-Michel | Mécanique |
| | PIERRARD Jean-Marie | Mécanique |
| | SABONNADIERE Jean-Claude | Informatique fondamentale et appliquée |
| Mme | SAUCIER Gabrielle | Informatique fondamentale et appliquée |
| M. | SOHM Jean-Claude | Chimie Physique |

CHERCHEURS DU C.N.R.S. (Directeur et Maîtres de Recherche)

| | | |
|---|---|---|
| M. | FRUCHART Robert | Directeur de Recherche |
| MM. | ANSARA Ibrahim | Maître de Recherche |
| | BRONOEL Guy | Maître de Recherche |
| | CARRE René | Maître de Recherche |
| | DAVID René | Maître de Recherche |
| | DRIOLE Jean | Maître de Recherche |
| | KLEITZ Michel | Maître de Recherche |
| | LANDAU Ioan-Doré | Maître de Recherche |
| | MATHIEU Jean-Claude | Maître de Recherche |
| | MERMET Jean | Maître de Recherche |
| | MUNIER Jacques | Maître de Recherche |

Personnalités habilitées à diriger des travaux de recherche (décision du Conseil Scientifique)

E.N.S.E.E.G.

| | | |
|---|---|---|
| MM. | BISCONDI Michel | Ecole des Mines St. Etienne (dépt. Métallurgie) |
| | BOOS Jean-Yves | Ecole des Mines St. Etienne (Métallurgie) |
| | DRIVER Julian | Ecole des Mines St. Etienne (Métallurgie) |

.../...

| | | |
|---|---|---|
| MM. | KOBYLANSKI André | Ecole des Mines St. Etienne (Métallurgie) |
| | LE COZE Jean | Ecole des Mines St. Etienne (Métallurgie) |
| | LESBATS Pierre | Ecole des Mines St. Etienne (Métallurgie) |
| | LEVY Jacques | Ecole des Mines St. Etienne (Métallurgie) |
| | RIEU Jean | Ecole des Mines St. Etienne (Métallurgie) |
| | SAINFORT | C.E.N. Grenoble (Métallurgie) |
| | SOUQUET | U.S.M.G. |
| | CAILLET Marcel | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | COULON Michel | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | GUILHOT Bernard | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | LALAUZE René | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | LANCELOT Francis | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | SARRAZIN Pierre | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | SOUSTELLE Michel | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | THEVENOT François | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | THOMAS Gérard | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | TOUZAIN Philippe | Ecole des Mines St. Etienne (Chim. Min. Ph.) |
| | TRAN MINH Canh | Ecole des Mines St. Etienne (Chim. Min. Ph.) |

**E.N.S.E.R.G.**

| | | |
|---|---|---|
| MM. | BOREL | Centre d'études nucléaires de Grenoble |
| | KAMARINOS | Centre national recherche scientifique |

**E.N.S.E.G.P.**

| | | |
|---|---|---|
| M. | BORNARD | Centre national recherche scientifique |
| Mme | CHERUY | Centre national recherche scientifique |
| MM. | DAVID | Centre national recherche scientifique |
| | DESCHIZEAUX | Centre national recherche scientifique |

28 Novembre 1978

# - REMERCIEMENTS -

Je tiens à remercier tout d'abord les personnes et organismes sans lesquels le travail présenté ici n'aurait jamais vu le jour :

- le CEA, et tout particulièrement Monsieur AMOUYAL,
- le ARET (DRME) et ses représentants : Messieurs RAMSAY, LA ROSA et DESVERGNES,
- Monsieur BOLLIET qui m'a accueilli à l'ENSIMAG, a fourni mon sujet de thèse et dirigé mes travaux.

Cette thèse a été influencée par mes discussions avec Bernard AMY et a beaucoup bénéficié des nombreuses remarques de Messieurs ABRIAL (rapporteur) et COLMERAUER.

Je remercie également les autres membres du jury : Monsieur VAUQUOIS qui en a accepté la présidence, Monsieur JORRAND (rapporteur) et Monsieur GUILLON de la Société GIXI.

Je remercie enfin Madame DIAZ qui a assuré la frappe de ce document.

M. G. Lan.

# RESUME

- *Le problème qui est à l'origine de cette thèse est de fournir aux industriels les moyens d'obtenir des programmes certifiés conformes à des spécifications.*

*L'insuffisance des techniques classiques de vérification a posteriori (chapitre 0) me conduit à proposer un système de Programmation Assistée par Ordinateur garantissant la production de programmes corrects par une méthode de transformations de spécifications: il s'agit de passer de la spécification au programme par une suite de transformations dont chacune est prouvée valide. Donc le système oblige et aide le concepteur à documenter et justifier chacun de ses choix d'implémentation.*

*Le principal ingrédient de ce système est un langage permettant d'exprimer à la fois: la spécification initiale (Intention), les spécifications intermédiaires (Documentation), le programme final et les raisonnements justifiant chaque transformation.*

*Ce langage de spécifications doit satisfaire à quatre exigences apparemment contradictoires:*

- *être clair, pour que le texte de la garantie soit lisible, crédible et convaincant ;*
- *tolérer les ambiguités, puisqu'on ne peut jamais les exclure complètement (chapitre I) ;*
- *être défini de façon suffisamment simple et rigoureuse pour se prêter à des preuves formelles ;*
- *être d'un emploi facile pour l'utilisateur.*

*Un tel langage peut évidemment aussi servir à la Conception Assistée de produits autres que des programmes, et à un vaste sous-ensemble de l' "Intelligence Artificielle".*

*On trouvera ici une présentation aussi intuitive que possible du langage de spécifications que je propose.*

- *Syntaxe du langage:* *Pour accommoder l'infinie diversité des applications et la simplicité d'utilisation, le langage doit être "dynamiquement extensible" afin d'épouser progressivement la forme d'expression choisie par l'utilisateur.*

*La grammaire est donc évolutive ; elle comporte à tout instant un ensemble d'opérateurs dits "généralisés" parce que leurs arguments et résultat sont délimités par des "types" qui - au lieu d'être déterminés - peuvent être seulement liés par une relation (et chaque opérateur a une syntaxe propre*

*L'ensemble de ces types est lui-même évolutif et défini à tout instant par des types de base et des constructeurs de types. Cet ensemble est de plus, muni d'un ordre partiel correspondant à l'inclusion (chapitre I).*

*Chaque identifieur ou expression apparaissant dans le langage a un type qui, s'il n'est pas exactement connu, peut être approché par des bornes inférieure et supérieure (ambiguité de types).*

*L'analyse syntaxique (chapitre II) a pour but d'établir pour chaque phrase de l'utilisateur un arbre syntaxique conforme à la syntaxe propre de chaque opérateur. S'il y a zéro ou plusieurs arbres, le système pose des questions dont les réponses permettront de déduire les extensions nécessaires (apprentissage, par le système, du langage de l'utilisateur).*

*L'utilisateur est sensé ne rien connaître de la grammaire et des types ; son seul rôle est de s'exprimer en respectant quelques conventions, de répondre aux questions, et de changer certains symboles si le système le lui demande.*

- *Signification du langage*

*L'utilisation du langage suppose qu'on puisse prouver qu'une spécification est conforme à (c'est-à-dire implique) une autre. Pour ce faire, on définit par des "règles d'inférence" une sémantique déductive du langage (chapitre III) qui, à tout ensemble de phrases (même ambigües) associe les phrases qui en sont des conséquences certaines.*

*Un démonstrateur de théorèmes est donc nécessaire pour au moins approximer cette sémantique.*

*Parmi les divers critères de qualité applicables à tout démonstrateur, on montre que le plus important est l' "autonomie" (difficulté des théorèmes qu'il peut prouver en un temps donné) ; en effet, en améliorant l'autonomie, on diminue d'autant l'intervention requise de l'utilisateur. Malheureusement, l'autonomie ne sera jamais suffisante pour que le démonstrateur puisse être entièrement automatique: il sera nécessairement interactif.*

*Le démonstrateur proposé est fondé pour l'essentiel sur une technique de "pattern matching" très efficace qui pourrait être appliquée aussi à d'autres domaines: l'information est codée sous forme de "spectres", et la déduction est dirigée par la ressemblance (résonance) des spectres des diverses hypothèses.*

*Avenir du langage*

*Seule une expérimentation en vraie grandeur permettra de décider du bien-fondé de cette approche, même s'il existe déjà des indices favorables, notamment:*

- *simulation à la main ;*
- *fondements semblables à ceux du langage LCF de Milner, dont le succès est indiscutable ;*
- *contournement de difficultés classiques de l'Intelligence Artificielle.*

*La principale difficulté d'implémentation sera de déterminer l'état initial du langage (types, opérateurs, axiomes et théorèmes initiaux) permettant aux utilisateurs de s'en servir facilement.*

*Enfin, l'intérêt et l'étendue des applications du langage justifient largement le coût estimé (relativement élevé) de son implémentation.*

SOMMAIRE

# INTRODUCTION

# RESUME DU CHAPITRE 0

*On pose ici le problème de la certification des programmes.*

*Dans un premier temps, on établit qu'il est quasi-impossible de valider un programme par la seule observation de son comportement externe: les tests de programmes sont d'indispensables outils de mise au point souvent très efficaces, mais ils ne sauraient suffire à assurer de hautes fiabilités.*

*Dans un deuxième temps, on montre que même la connaissance du texte complet d'un programme ne permet pas, dans la pratique, de prouver sa conformité avec une spécification lisible et convaincante: les méthodes existantes sont tellement difficiles à utiliser et améliorer qu'on ne sait prouver que des petits programmes ressemblant trop à leurs spécifications.*

*Plutôt que de s'acharner à prouver directement la validité du texte d'un programme, je suggère donc de prouver la validité de la construction du programme à partir de ses spécifications. Pour ce faire, un système de Programmation Assistée par Ordinateur est proposé pour aider à construire et prouver simultanément les programmes: le passage d'une spécification à un programme est le résultat d'une suite de transformations. Ces transformations peuvent être suggérées par le système ou à l'initiative complète de l'utilisateur ; elles sont toujours prouvées à l'aide du système. Ce système permet de certifier non seulement les programmes mais également leurs maintenance, transport et modifications.*

*La totalité des dialogues avec l'utilisateur s'effectue dans ce qui est appelé ici "Langage de Spécifications". Celui-ci doit être assez souple pour exprimer facilement toute la variété des problèmes et des raisonnements sur ces problèmes (langage pour l'Intelligence Artificielle).*

*Toute la suite de ce document propose l'ébauche d'une solution aux difficultés que présente la réalisation d'un tel langage.*

# CHAPITRE 0

# INTRODUCTION

# PREUVES DE PROGRAMMES et "INTELLIGENCE" ARTIFICIELLE

---

## I - VALIDATION DE PROGRAMMES

A l'origine, le langage présenté ici a été prévu expressément pour permettre la construction de programmes certifiés corrects. Cette section a pour but de montrer par quel raisonnement on peut déterminer la nécessité d'un langage de spécifications pour prouver les programmes.

### 1. Intérêt des preuves par rapport aux tests de programmes

Considérons un programme qui, depuis sa dernière modification, n'a jamais rencontré de cas d'erreur et a ainsi fonctionné (en simulation, test ou exploitation) pendant une durée totale: t . Que peut-on conclure sur sa fiabilité, et en particulier, avec quelle crédibilité peut-on estimer qu'il fonctionnera encore correctement pendant au moins une durée donnée: t' ?

Exemple numérique: Soit un programme ayant toujours fonctionné normalement, et ce depuis 1000 heures (réelles ou simulées):
(1) est-il vraisemblable qu'une erreur survienne avant 3 secondes?
(2) avec quelle certitude peut-on affirmer qu'il ne surviendra aucune erreur dans les 1000 heures à venir?

Réponse: Le modèle probabiliste proposé en ANNEXE I permet d'estimer les réponses avec les crédibilités respectives:
(1) $8.3\ 10^{-7}$ (donc très invraisemblable)
(2) 0.50 (donc loin d'être certain).

Plus généralement, ce modèle permet d'établir les formules et abaques ci-dessous Il faut noter qu'elles donneront des estimations trop optimistes si elles sont appliquées à des programmes soumis à des tests autres que *aléatoires*, c'est-à-dire respectant la distribution statistique des données* prévues en exploitation:

---

* par "données" il faut entendre tout ce qui est susceptible d'influencer le déroulement du programme, y compris éventuellement les pannes matérielles.

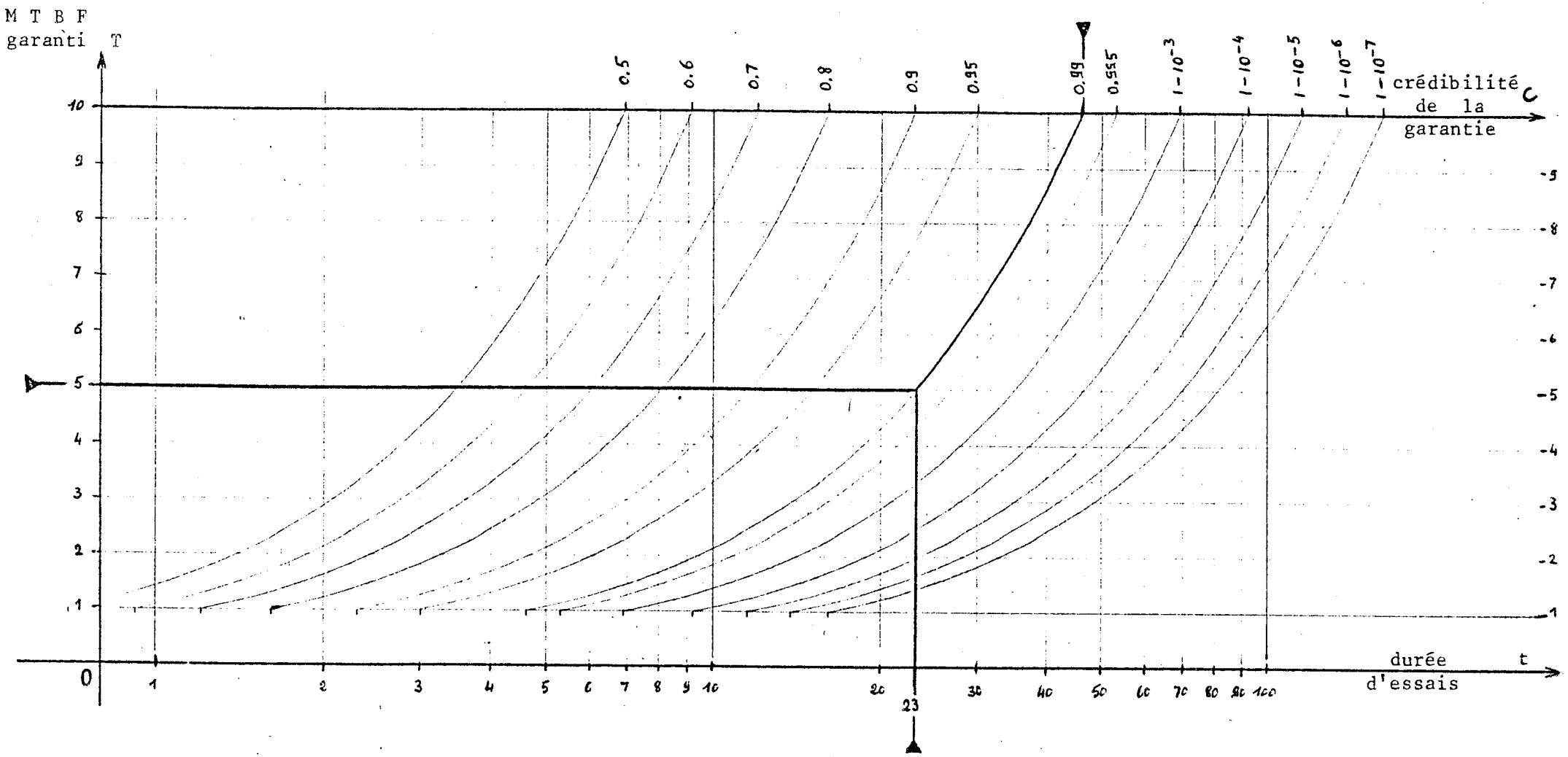



FIGURE 1 : ABAQUE ( Garantie de MTBF x Crédibilité de la garantie x durée d'essais )

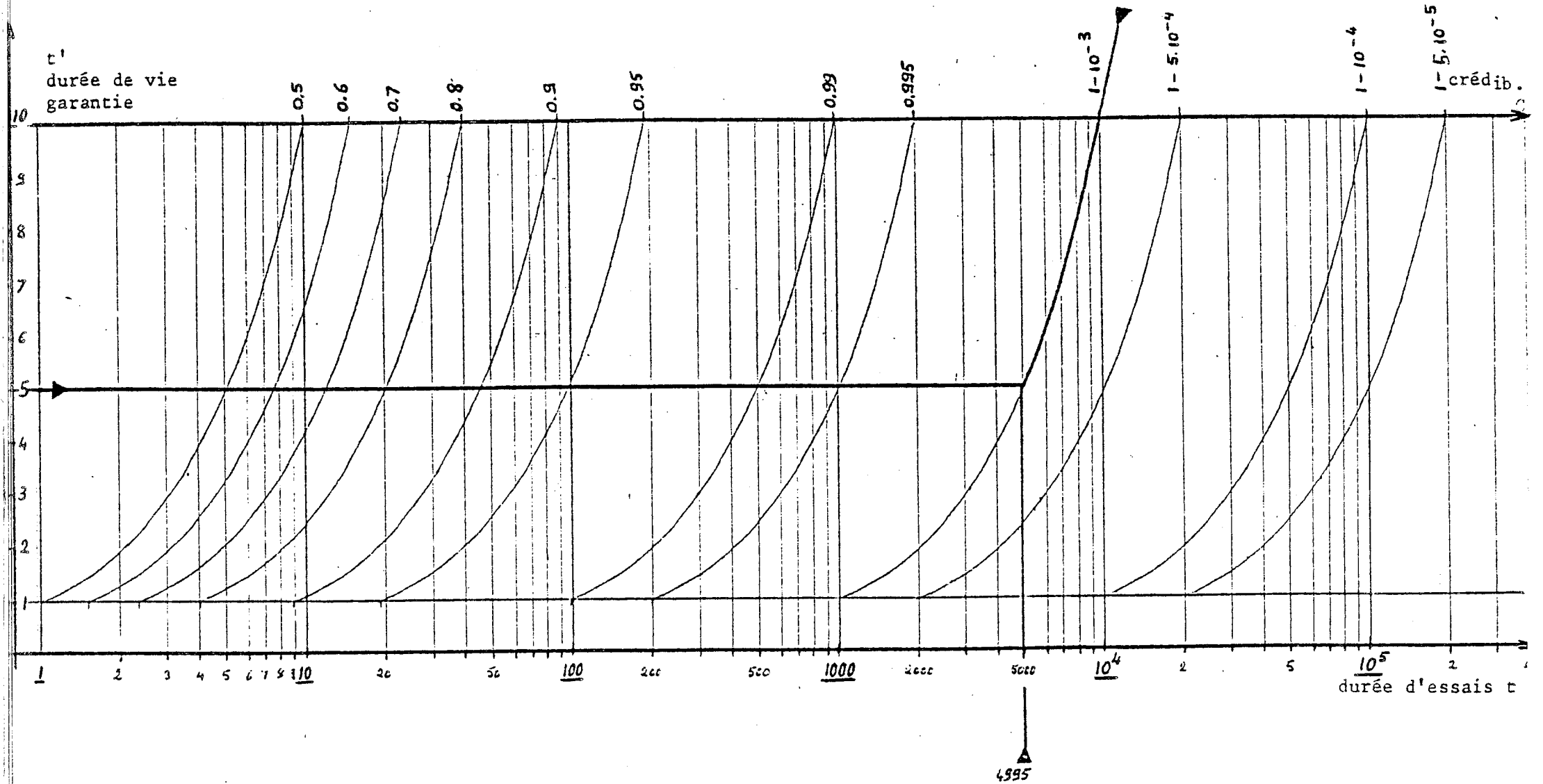



FIGURE 2 : ABAQUE ( Garantie de durée de vie x Crédibilité de la garantie x durée d'essais )

Estimation du MTBF (durée moyenne de vie T):
Si le programme a vécu un temps t et n'a jamais connu d'erreur, la *crédibili*
d'un MTBF≥T est définie par la formule:

$$1 - e^{-\frac{t}{T}}$$

exprimant la probabilité pour que, si le MTBF est T, le programme ne vive pas plus longtemps que t.

Estimation de durée de vie garantie t' :
La crédibilité de cette garantie est définie par la formule:

$$\frac{t}{t+t'}$$

Ces estimations se traduisent par les *abaques* ci-jointes (figures 1 et 2).

Exemple d'utilisation (2ème abaque)
(Centrale nucléaire) si l'on veut qu'un programme fonctionne sans accident pendant 50 ans avec une crédibilité 0.999, la simulation doit porter sur 49 950 ans (encore faudrait-il être sûr du simulateur).

En résumé, les tests de programmes restent d'indispensables outils de mise au point pour détecter rapidement *la plupart* des erreurs, mais ils ne peuvent pas suffire à justifier de très hautes fiabilités. Il existe d'ailleurs des exemples de programmes réputés corrects (ex. Compilateur FORTRAN) où des erreurs se sont révélées après plusieurs années d'exploitation.

Donc une technique de validation plus efficace que les tests s'impose.

## Preuves par récurrence

L'idéal serait la preuve par test exhaustif qui garantirait une durée de vie illimitée. Ceci étant évidemment irréalisable, on est obligé de trouver des *principes de récurrence* qui permettent de prouver en un temps raisonnable que le programme sera correct dans tous les cas possibles.

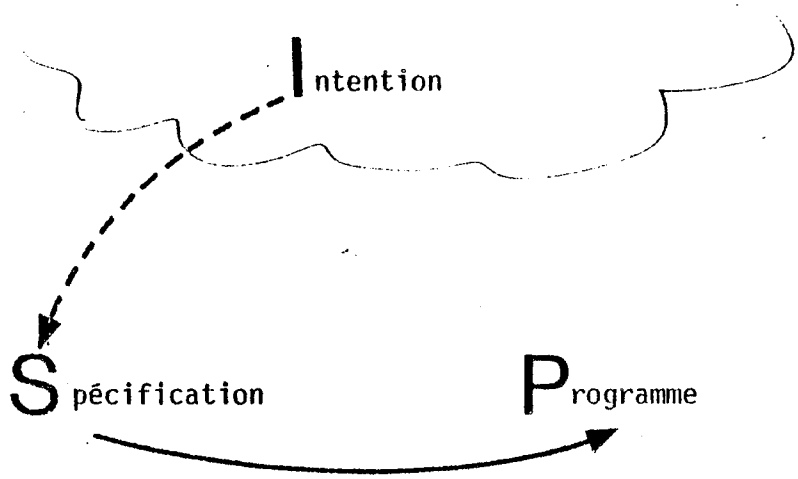
Certains de ces principes de récurrence bien connus ont donné lieu à des implémentations, mais aucune jusqu'à présent ne peut prétendre (et il s'en faut de beaucoup) à une quelconque utilité industrielle. Il convient donc d'examiner maintenant ce qui empêche ces méthodes d'être effectivement utilisables dans la pratique.

## 2. Difficultés des preuves de programmes: un diagnostic

Les principaux reproches formulés à l'encontre des meilleurs systèmes de preuves [20,30,38] concernent:

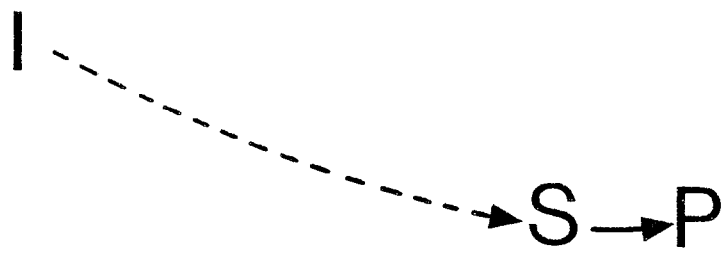
- leur faible portée: seuls de très petits programmes peuvent être "prouvés",
- leur ésotérisme: seuls les auteurs de ces systèmes et quelques initiés parviennent à s'en servir,
- leur manque de crédibilité: on est obligé de fournir des spécifications souvent encore moins lisibles que les programmes eux-mêmes.

Pour expliquer cet état de faits, remarquons que l'idéal serait de prouver qu'un Programme est conforme à l'Intention intuitive du programmeur. Comme c'est impossible, on substitue à l'Intention une Spécification écrite (énoncé du problème):




Il reste à prouver que le Programme est conforme à la Spécification (S→P) et à espérer que la Spécification est conforme à l'Intention (I→S). La crédibilité de la preuve (I→P) se ramène donc à la crédibilité du passage de l'Intention à la Spécification qui doit être facile et intuitif: S doit être proche de I.

Mais en raison de l'apparente difficulté de prouver la transition S→P les systèmes de preuves exigent que la spécification S soit très proche du programme P ; ceci se traduit par une tendance à élever le niveau des langages de programmation, ce qui est bien, mais aussi par un appauvrissement du langag de spécifications qui ne peut plus exprimer l'intention de façon crédible:




Dans ces conditions, la preuve (S→P) ne sert plus à grand chose puisque la plupart des erreurs logiques sont commises en amont de la spécification (transition IS) et se retrouvent donc aussi bien dans la spécification que dans le programme.

Par exemple: au lieu de spécifier le produit de matrices [C] ← [A].[B] par la formule $\forall ij\ C(ij) = \sum_k A(i,k)\ B(k,j)$ que le programme ne fera que paraphraser, il est plus utile de le spécifier par la *raison* pour laquelle on en a besoin, à savoir que $C\ \vec{x} \equiv A\ (B\ \vec{x})$, mais évidemment la preuve sera plus délicate.

On ne pourra pas remédier à cet état de faits tant que l'on ne saura prouver que des transitions (S→P) triviales, il faut donc comprendre pourquoi les démonstrateurs sont si inefficaces.

Les démonstrateurs utilisés pour la preuve de programmes ont trois défauts essentiels:

### 1er défaut: Inaptitude à utiliser des principes de récurrence

La preuve de tout programme itératif ou récursif donne lieu à des théorèmes qui ne peuvent être prouvés que par récurrence. Mais ceci nécessite la découverte d'hypothèses de récurrence suffisantes (exemple: invariants de boucles) La seule étude systématique de ce processus est très incomplète [6] et les quelques heuristiques disponibles sont le plus souvent insuffisantes [16,26,60] En général la découverte d'invariants suffisants ne peut se faire qu'à la main et nécessite beaucoup de flair et de patience.

2ème défaut: **Communication avec un démonstrateur interactif**

On sait maintenant que la démonstration automatique ne peut s'appliquer avec succès qu'à des domaines très restreints [46] . Les bons démonstrateurs sont donc nécessairement interactifs, c'est-à-dire capables de demander de l'aide et de suivre les indications de l'utilisateur. Malheureusement, cette communication démonstrateur ↔ utilisateur est, dans un sens comme dans l'autre, à peu près impossible:

. Les démonstrateurs "ne savent pas" expliquer leurs difficultés autrement qu'en donnant une trace indéchiffrable de ce qu'ils ont essayé ;

. L'utilisateur ne peut actuellement influencer le démonstrateur que s'il en connait bien le fonctionnement interne et à l'aide d'un langage de commandes généralement ésotérique.

3ème défaut: **Principes de démonstration**

Les démonstrateurs reposent sur des principes simples mais rigides (ex.: Résolution) qui n'acceptent que des propositions écrites dans un langage très pauvre (ex.: Logique du 1er ordre). De plus, ces démonstrateurs s'empressent de tenter une mise sous forme canonique (ex.: forme normale conjonctive skolemisée) qui est irréversible, et le plus souvent impropre à la communication avec l'utilisateur.

Les différentes formulations d'une même proposition logique (ex.: $A \supset B$, $\neg A \vee B$, si A alors B, $\neg B \supset \neg A$) indiquent des préférences sur la façon de s'en servir: information heuristique précieuse que les démonstrateurs s'empressent de supprimer avant même de lancer le mécanisme de la déduction. Nous reviendrons en détail (chapitre III) sur les critères d'évaluation des démonstrateurs.

## 3. Remède : Construction et Preuve par transformation de spécifications

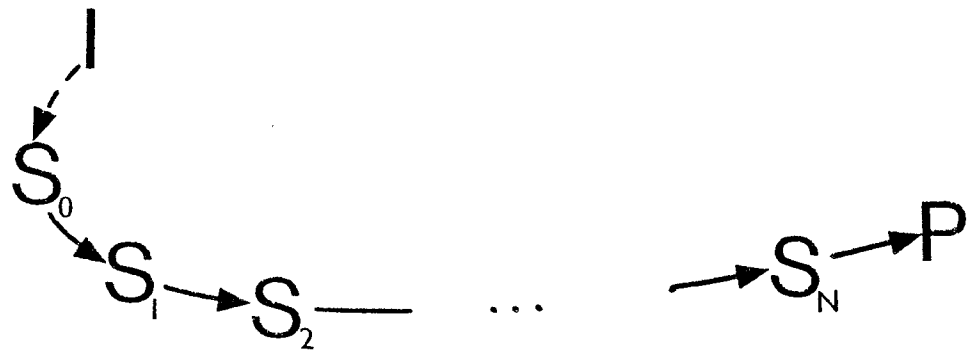
L'examen de ce diagnostic montre qu'on ne pourra pas certifier de logiciel tant qu'on ne disposera pas d'un bon *langage de spécifications* satisfaisant notamment à deux exigences:

(1) il doit être suffisamment évolué pour permettre de formuler facilement des spécifications lisibles de n'importe quel problème afin de rendre la transition I→S très sûre ;

(2) il doit permettre la preuve (interactive) de la transition S→P.

[ Ce n'est pas la première fois que la nécessité d'un tel langage a été soulignée [96] , on peut donc se demander pourquoi si peu de recherches ont été consenties dans cette direction ; j'y vois essentiellement deux raisons:
- l'impression fausse mais répandue qu'un bon démonstrateur (interactif) serait encore plus difficile à réaliser pour un langage évolué que pour le langage de la logique du premier ordre ;
- des exigences trop mal délimitées pour se prêter à une approche théorique, donc noble et confortable ;
- projet à très long terme, donc difficile à financer et suicidaire pour des chercheurs. ]

Examinons d'abord les conséquences de la deuxième exigence (preuve de S→P): il est hors de question de tenter une preuve a posteriori sans savoir quels raisonnements et choix sont intervenus dans la construction du programme. Autrement dit, on peut concevoir la programmation comme l'établissement d'une suite de transformations légales partant d'une spécification initiale et aboutissant à un programme.




Ce principe de construction et preuve simultanées de programmes par transformations de spécifications a fait l'objet d'un rapport technique [10] qui ne sera pas repris ici sauf pour ce qui concerne son incidence sur le langage de spécifications. On pourra se référer à l'EXEMPLE développé en ANNEXE IV.

## 3.1. Spécification de base S

Elle constitue le texte de la garantie et comporte typiquement les éléments suivants:

(1) les *Hypothèses*

a) Environnement = ensemble des "cas" que le programme est susceptible de rencontrer:
- limites sur les valeurs, fréquences et variations des variables,
- éventuellement, limites sur les pannes matérielles (types et nombres).

b) Moyens disponibles (machines: rapidité, taille mémoire, taille des mots... ; langages: compilateur, sémantique... ; domaine d'application: lois, théorèmes...).

(2) les *Exigences*

a) Spécification positive: fonction à réaliser au minimum, avec contraintes de délais, coûts, etc...

b) Spécification négative: accidents à éviter.

(2a) et (2b) sont nécessaires, car on ne peut jamais spécifier complètement un problème (au sens où il existe de nombreuses solutions non équivalentes) et il faut donc le spécifier "par encadrement".

*Plus formellement*, on peut traduire cette façon de spécifier un *"système"* comme suit:

- le système P à réaliser sera dans un *"univers"* décrit par des variables X, Y, t, où:
  - . X est un ensemble de variables "d'entrée" qui évoluent indépendamment du système,
  - . Y est un ensemble de variables susceptibles de dépendre du système,
  - . t est le temps.
- on peut considérer que le système P établit une relation (réduite à une fonction s'il est déterministe): P(x,y) où :
  - . $x : t \to X$
  - . $y : t \to Y$

  sont deux fonctions du temps décrivant l'histoire de l'univers
- les éléments de la spécification sont alors les suivants:

hypothèses

(1a) l'*environnement* (ensemble des "cas"possibles) correspond à un prédicat $E[x,y(t_0)]$ indiquant quels sont les états initiaux $y(t_0)$ et les histoires des variables d'entrée (fonctions x(t)) que le système est susceptible d'avoir à traiter.

(1b) les *moyens* délimitent l'ensemble PP des systèmes possibles.

exigences
- (2a) la *spécification positive* correspond à une relation F(x,y),
- (2b) la *spécification négative* correspond à une relation A(x,y) souvent de la forme $\exists t\ \alpha[x(t),y(t)]$ décrivant les états accidentels à éviter.

Certifier le programme, c'est prouver que:

si l'environnement (1) est respecté,
alors la fonction (2a) est respectée et les accidents (2b) sont évités,
ce qui se traduit par:

$$E[x,y(t_0)] \supset \{P(x,y) \supset [F(x,y) \wedge \neg A(x,y)]\}$$

les spécifications peuvent être incompatibles, auquel cas il n'existe pas de système $P \in PP$ ayant cette propriété. Dans ce cas, il faut *affaiblir* la spécification par l'un ou plusieurs des moyens suivants:

(1a). restreindre l'ensemble des cas possibles (ex.: diminution de la fréquence maximum des entrées si (2a) exige le temps réel).

(1b). élargir les moyens disponibles (ex.: taille mémoire, compilateur optimiseur, analyse d'algorithmes donnant des bornes supérieures plus précises...).

(2a). assouplir la fonction à réaliser (ex.: allonger les temps de réponse exigés, diminuer la précision des résultats, supprimer des options coûteuses).

(2b). ne plus considérer certains évènements comme accidents (ex.: perte d'un message, atterrissage plus dur d'1 g....).

## Remarque

Même si le programme est ainsi certifié, il reste nécessaire de le *tester*, car c'est le seul moyen de se convaincre que la spécification est conforme à l'intention. Dans le cas où le test révèle un comportement indésirable, la cause de l'anomalie n'a pas à être cherchée ailleurs que dans la spécification et sera toujours l'une des suivantes:

1 - violation d'une hypothèse:
- (1.a) données excédant les limites supposées,
- (1.b) utilisation d'un moyen non conforme à sa spécification ;

2 - *oubli* d'une exigence:
- (2.a) oubli d'une contrainte à respecter,
- (2.b) oubli d'un accident à interdire.

## 3.2. Spécifications intermédiaires $S_i$ et finale P

Chaque spécification $S_i$ doit être *conforme* à la précédente $S_{i-1}$. Ceci veut dire que $S_i$ est *équivalente à* ou *plus précise* que $S_{i-1}$

donc que $H \supset [S_i \supset S_{i-1}]$

où H désigne les hypothèses (1a) et (1b) de la spécification.

Ceci confirme que le langage de spécification doit se prêter à des démonstrations.
On voit aussi que ce langage doit pouvoir inclure les langages de programmation, puisque la spécification finale P comporte le programme.

## 3.3. Transformations

Puisque chaque transformation $S_i \rightarrow S_{i+1}$ doit être validée par la preuve du théorème:

$$H \vdash S_{i+1} \supset S_i$$

la *base d'hypothèses* H doit donc contenir des *axiomes* qui justifient toutes les transformations effectuées. En particulier, à chaque fois qu'une transformation ne pourra pas être justifiée au moyen des informations contenues dans H, le programmeur devra fournir des justificatifs qui iront s'ajouter à H dans la partie "moyens" (1b). Par exemple, si une transformation consiste à compiler un programme, et si on ne peut pas prouver que le compilateur utilisé est correct, il faudra rajouter l'axiome que ce compilateur est correct, et cette supposition apparaîtra donc explicitement dans le texte de la certification.

Au pire, si on ne sait rien prouver, la certification comportera l'énoncé complet de toutes les transformations effectuées: ce sera un document complet mais illisible et peu convaincant.
La certification ne sera crédible que si elle est réduite à un petit nombre d'axiomes "convaincants": il faut donc, là encore, disposer de moyens de preuves.

Remarquons aussi que certaines transformations sont suffisamment fréquentes dans la pratique (ex.: passage de spécification parallèle à spécification itérative) pour qu'on établisse, en s'inspirant par exemple des travaux d'ABRIAL [2], une bibliothèque de *lemmes de transformations*, prouvés une fois pour toutes, et rajoutée à H pour faciliter les preuves (et synthèses) de transformations spécifiques.

De la même manière, on peut établir pour tout domaine d'application particulier une bibliothèque de théorèmes traduisant des connaissances sur ce domaine et pouvant même contenir à la limite des sous-programmes déjà prouvés. Le seul axiome qu'il serait nécessaire de faire apparaître dans la certification serait le nom de la bibliothèque utilisée.

### 3.4. Comparaison avec d'autres méthodes de programmation

α/ Transformations de programmes: elles présentent par rapport aux transformations de spécifications deux inconvénients:
- il faut déjà disposer au départ d'un programme correct,
- les gains en temps d'exécution sont très limités, et ce n'est pas étonnant puisque dans la plupart des cas, le programme initial a été obtenu à la suite de choix critiques (sur le plan de l'efficacité maximum) dont il ne reste aucune trace.

A ce propos , on peut dire qu' *"on ne peut rien faire d'intelligent sur un programme (preuve, maintenance, transport, transformations...) si on ne sait pas comment il a été obtenu"*.

β/ Programmation en Logique du Premier Ordre [12,50]
Dans cette technique astucieuse, spécification et programme ne font qu'un (S = P), et par définition P est donc correct.
Mais l'exécution de P consiste à prouver un théorème, ce qui tend à limiter le champ d'application de cette méthode. De plus, la spécification d'un problème complexe peut difficilement être donnée de façon compréhensible en Logique du premier ordre.

γ/ Situation par rapport aux autres techniques de preuves de programmes
A la lumière de ce qui précède, on peut classer les méthodes de preuves de programmes selon l'un des cinq types ci-dessous:

- Deux méthodes non récurrentes:

1/ le *test exhaustif* pour des problèmes très simples,

2/ l'*exécution symbolique* consiste à exécuter le programme formellement (tous les cas sont exécutés en parallèle), simplifier les formules obtenues pour qu'elles restent finies et qu'on puisse en prouver des propriétés.

- Trois types de méthodes récurrentes:

3/ *récurrences sur le domaine* (ex: "Structural Induction")
On teste un sous-domaine limité et on montre par récurrence sur le domaine que le résultat est valide sur tout le domaine.

4/ *récurrences sur le texte du programme*
ex: Récurrences de Floyd (pour les boucles) et Scott (pour la récursion).

5/ *récurrence sur le procédé d'obtention du programme*
C'est le cas de la méthode de transformation de spécifications proposée ici.

Parmi ces types de méthodes, aucun n'est suffisant ; un véritable système de certification de programmes doit faire appel à un mélange de ces types de méthodes, sous le contrôle de la dernière.

## II - "INTELLIGENCE" ARTIFICIELLE

Puisque le langage de spécifications est prévu pour exprimer tous les problèmes et raisonnements pouvant intervenir dans la programmation, il peut servir de base non seulement au système de Programmation Assistée par Ordinateur esquissé précédemment, mais aussi bien à tout système pour l' "Assistance à la résolution de problèmes" (conception de produits satisfaisant à des spécifications données). Un tel langage, accompagné d'un système de déduction interactif, correspondrait donc parfaitement à ce que l'on cherche à définir comme Langages de très haut niveau pour ce que l'on appelle (d'un vocable malencontreux) l'Intelligence Artificielle.

Pour comprendre la suite de ce document, il faut s'éloigner de l'aspect "preuves de programmes" qui n'est qu'une application particulière du langage proposé.

Ce langage doit être un support permettant à ses utilisateurs d'*introduire les théories de leur choix et d'exprimer des raisonnements à l'intérieur de ces théories.*

# I

# TYPES

## RESUME DU CHAPITRE I - TYPES

*Dans la communication de théories et raisonnements, une des principales difficultés est de s'entendre, pour chaque identifieur, sur son type (~ domaine) qui n'est presque jamais explicitement déclaré, mais progressivement inféré à partir des divers usages de cet identifieur.*

*A partir d'observations de multiples textes scientifiques, j'essaye ici de décrire l'ensemble des types qui interviennent dans le discours mathématique, comment ils sont construits, et quelle relation (d'inclusion) existe entre eux.*

*Cette relation détermine normalement un treillis, et l'ambiguité sur le type de chaque identifieur ou expression est représentée par un sous-treillis qu'on cherche à réduire à un seul point.*

*Avant toute implémentation, cette structure de types devrait être largement discutée, car c'est d'elle que dépend en premier lieu toute la généralité d'applications du langage.*

# CHAPITRE I

# TYPES

## 0 - INTRODUCTION

Le langage de spécifications dont nous avons besoin doit être suffisamment général pour permettre d'échafauder des théories destinées à représenter des domaines arbitraires.

La variété imprévisible des applications interdit que ce langage soit figé et nous oblige à étudier des moyens efficaces d'étendre le langage en cours d'utilisation.

L'expérience des langages extensibles [98] nous intéresse donc au premier chef.

A partir d'un noyau composé de règles de grammaire "standard", l'extension consiste à introduire de nouvelles règles de grammaire comportant de nouveaux opérateurs: la portée de ces règles de grammaire est délimitée par les types exacts des arguments qui interviennent. La souplesse de ces langages dépend donc en grande partie de la façon dont ces types sont définis et organisés.

En général, les langages extensibles reposent sur un certain ensemble de types défini à partir d'une collection finie de types de base disjoints (tels que entiers, réels, booléens etc...) et de quelques règles de construction de types (telles que produit cartésien, application, union , etc...). Mais le langage de spécifications n'est pas un simple langage de programmation et nécessite une structure de types plus élaborée, comme on va le voir. D'ailleurs certains aspects de la notion de type décrite ici commencent à apparaître dans les études sur les langages de très haut niveau destinés justement à l'Intelligence artificielle [9,51,100].

# I - NOTION DE TYPE

## 1. Types et domaines

Notre langage se propose d'exprimer des raisonnements mathématiques. Ces raisonnements portent sur des objets mathématiques représentés par des *identifieurs* ou des *expressions*.

On dira que *"le type" associé à un domaine* est l'ensemble de toutes les expressions à valeur dans ce domaine.

La notion de *domaine* est importante: on pourra par exemple considérer comme domaines les nombres entiers, les nombres premiers, les applications, les groupes, les langages context-free etc... et associer à chacun d'eux le type correspondant.

Mais on pourrait aussi considérer que toute *propriété* définit le sous-ensemble sur lequel elle est vraie comme un domaine auquel peut être associé un type. Par exemple, tous les sous-ensembles énumérables de $\mathbb{N}$ définiraient des types.

Il ne faut donc associer de types qu'à certains domaines "remarquables". Cette notion de domaine remarquable est évidemment arbitraire et subjective, mais peut être caractérisée en première approximation par le fait qu'un tel domaine *a reçu un nom* et que ce nom est normalement *sous-entendu*, par exemple:

Pour exprimer que *"tout entier pair supérieur à 2 est composite"* (non-premier), diverses phrases sont possibles selon les types disponibles:

(1) si l'on a le type "ENTIER PAIR" avec comme variable p, on pourra écrire:
"$p > 2 \supset$ p *composite*"

(2) si l'on a le type "ENTIER" avec comme variable n, on pourra écrire:
"n *pair et* $n > 2 \supset$ n *composite*"

(3) si l'on n'a que le type "REEL" avec comme variable x, on pourra écrire:
"x *entier et* x *pair et* $x > 2 \supset$ x *composite*"

En généralisant cet exemple, on voit qu'il faut trouver un compromis: s'il n'y a pas assez de domaines remarquables donnant lieu à des types, l'expression sera alourdie et maladroite, mais s'il y en a trop, l'analyse syntaxique sera délicate à cause de la difficulté d'identifier les types des expressions.

Remarquons en effet qu'il serait malencontreux d'obliger un utilisateur à déclarer a priori les types de tous ses identifieurs (soit explicitement, soit par des restrictions sur les choix de ces identifieurs): dans la plupart des cas, les divers contextes dans lesquels apparaît un identifieur donnent suffisamment d'information sur son type. On indiquera (chapitre II) comment l'analyseur syntaxique peut *inférer* ces types automatiquement ou, dans les cas difficiles, à l'aide de questions auxquelles l'utilisateur peut répondre sans connaître la notion de type.

## 2. Types et classes d'expressions

Jusqu'ici, on n'a associé à un domaine que *le* type comportant *toutes* les expressions à valeur dans ce domaine.

Or il est des situations où l'on ne peut pas rencontrer n'importe quelles expressions mais seulement une certaine classe d'expressions: par exemple, on ne peut pas écrire "∃(x+y)Q", seul un identifieur de variable est autorisé à la place de "(x+y)".

Donc si, comme dans le langage ici proposé, on fait jouer aux types un rôle de "filtre syntaxique", il faut définir les types d'une façon plus précise:

*Un type est une classe d'expressions à valeur dans un domaine donné*
(Il peut donc y avoir plusieurs types pour un même domaine)
On peut par exemple distinguer, pour le domaine "T", les types suivants:
EXPR.T (noté en général T): classe de toutes les expressions,
EXPR.1 VAR.T: classe de toutes les expressions dépendant d'une seule variable,
EXPR. CONST.T: classe de toutes les expressions à valeur constante,
ID.T: classe des expressions réduites à un identifieur de variable,
CONST.T: classe des "littéraux" désignant les valeurs du domaine T
(comporte tous les mots du langage désignant les éléments du domaine T)
...

## 3. Relations entre types

Les types sont traditionnellement disjoints ( une expression ne pouvant apparte à deux types différents). Cette particularité correspond d'une part au rôle de protection que jouent les types, et d'autre part à l'organisation même du matériel: par exemple les entiers n'y constituent pas un sous-ensemble des réels, ni en ce qui concerne leur représentation, ni en ce qui concerne les opérations.

Cette règle d'indépendance des types, qui est peut être justifiée pour les langages de programmation est certainement inacceptable pour un langage destiné à représenter des raisonnements:
Puisque tout nombre entier est aussi réel, il faut considérer que les deux types correspondants sont liés par une relation d'ordre

$$\mathbb{N} < \mathbb{R}$$

La relation T<U signifie que toute expression de type T est (a fortiori) de typ et correspond donc à l'inclusion des domaines.

Exemple (voir paragraphe précédent) $ID.T < T$

Autre exemple:
Un domaine très remarquable est celui des relations et on peut donc lui associer le type RELATION .
Donc la relation ">" sur les réels en fait partie, mais on peut considérer aussi qu'elle est, plus précisément, du type [REEL x REEL → BOOLEEN]
On a [REEL x REEL → BOOLEEN]<[RELATION]
et, plus généralement, pour tous types T,U on a

$$[T \times U \rightarrow \text{BOOLEEN}] < [\text{RELATION}]$$

Ce type RELATION sera d'ailleurs plus loin noté

$$[\lambda TU : T \times U \rightarrow \text{BOOLEEN}]$$

avec la convention que λ est un opérateur de plus petite borne supérieure (cf. p. 28).

## 4. Types et opérateurs

Le principal inconvénient de la logique du premier ordre est la partition des symboles entre opérateurs (prédicats) et opérandes (arguments).

Naturellement, les objets qu'on utilise dans le raisonnement peuvent être appelés à jouer les deux rôles. Par exemple, l'objet "+" peut jouer un rôle d'opérateur dans "a+b" aussi bien qu'un rôle d'opérande dans "+ est commutatif".

On rencontrera, dans l'analyse syntaxique, la difficulté de décider si les symboles sont des séparateurs(exemple: parenthèses,virgules,etc...) ou des identifieurs destinés à recevoir un type. On peut dire que tant qu'un symbole n'a pas été rencontré dans un rôle d'opérande, on peut, sans risque, le considérer comme un simple caractère séparateur servant, par exemple, à reconnaître une règle de grammaire.

Au contraire, si de nombreuses règles ne diffèrent que par un symbole, on peut considérer ces symboles différents, non plus comme des séparateurs identifiant les diverses règles mais comme des éléments d'un même type, ce qui permet de fusionner toutes ces règles en une seule.

Exemple: les règles $N ::= N+N$

$N ::= N-N$

$N ::= N \times N$

$N ::= N/N$

$N ::= N \uparrow N$

distinguées par les séparateurs + - x / ↑ peuvent être remplacées par la règle unique

$$N ::= N[N^2 \to N]N$$

où les symboles sont maintenant considérés comme désignant des éléments du type $[N^2 \to N]$.

Ceci confirme donc encore le caractère inévitablement arbitraire de la notion de types.

## II - DEFINITIONS DES TYPES

Les types vont être définis comme d'habitude à partir de types de base et d'un certain nombre de règles de construction, le tout définissant un "langage des types".

## 1. Types de base

Aux domaines remarquables "simples", c'est-à-dire non composés à partir d'autres domaines, on fait correspondre les types de base, ordonnés conformément à l'inclusion des domaines. Ces types de base, en nombre non limité, comportent les types standard (Booléen, Entier < Réel, etc...) et ceux que l'utilisateur aura introduits, ordre compris (cf. p. 72-74).

Il n'y a aucun inconvénient à supposer l'existence d'un type vide Ø inférieur à tous les autres.

## 2. Constructeurs acceptables

On trouvera ci-dessous un ensemble de règles de construction de types présentées à titre d'exemple. Avant toute implémentation du langage de spécifications,il conviendra de fixer définitivement les règles de construction.

En même temps qu'on définit les constructeurs du langage des types, il faut définir la relation d'ordre entre ces types. Il faudra donc obéir à la discipline suivante:

Chaque construction sera définie par sa syntaxe *et par l'ordre qu'elle induit sur les types ainsi définis.*

Ceci entraîne une double précaution:

- non ambiguïté du langage des types (comme d'habitude),
- non bouclage de la relation.

On ne devra donc définir que des "constructeurs acceptables" respectant ces deux impératifs.

## 3. Exemples de constructeurs

Les règles de construction ci-dessous sont toutes acceptables.

Notation: Dans tous ces exemples on supposera par convention

$$T_1 < T_2 < T_3 \quad \ldots$$
$$U_1 < U_2 < U_3 \quad \ldots$$
$$V_1 < V_2 < V_3 \quad \ldots$$

### a/ Produit cartésien

Notation Si T,U sont deux types non vides, [TxU] est un type.

Abréviations admises

$[TxUxV]$ signifie $[[TxU]xV]$

$[T^2]$ signifie $[TxT]$

$[T^i]$ signifie $[TxTxT...]$ (i fois)

Signification Désignation de couples ordonnés d'expressions

Ordre induit (Rappelons une dernière fois que $T_1 < T_2 \ldots U_1 < U_2 \ldots$)

$[T_1 xU] < [T_2 xU]$

$[TxU_1] < [TxU_2]$

Cette construction n'induit ni ambiguité de notation, ni contradiction sur la relation d'ordre.

## b/ Application

Notation Si T,U sont deux types non vides, $[T \to U]$ est un type.

Abréviations admises

$[TxU \to V]$ pour $[[TxU] \to V]$

$[T \to UxV]$ pour $[T \to [UxV]]$

$[T \to U \to V]$ pour $[T \to [U \to V]]$

Signification Désignation des applications (fonctions *partielles* ou totales), avec argument de type T et valeur de type U.

Ordre induit $[T_1 \to U] < [T_2 \to U]$

$[T \to U_1] < [T \to U_2]$

Cette construction d'induit ni ambiguité de notation, ni contradiction sur la relation d'ordre.

## c/ Itération

Notation $[T^*]$

Signification Désignation des suites (éventuellement de longueur nulle ou infinie) d'expressions de type T.

Ordre induit $[T_1{}^*] < [T_2^*]$

pour tout i $[T^i] < [T^*]$

Cette construction n'induit, avec les précédentes, ni ambiguité de notation, ni contradiction sur la relation d'ordre. Il n'est pas du tout certain qu'il soit justifié de la conserver.

Exemple:

L'expression "suite $2^i$, $i=0...\infty$" est du type $\mathbb{N}^*$

### d/ Partie

Notation $[\mathcal{P}(T)]$

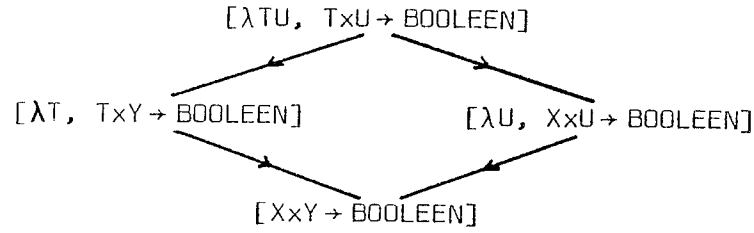
Signification Type des expressions dont la valeur est un ensemble (vide, fini ou infini) de valeurs du domaine associé à T.

Ordre induit $[\mathcal{P}(T_1)] < [\mathcal{P}(T_2)]$

Exemple: l'expression "$\{x \mid ax^2+bx+c=0\}$" est du type $\mathcal{P}(\mathbb{R})$

### e/ Polymorphisme

Introduction On a proposé plus haut, de nommer le type [RELATION] par la notation [λTU, TxU→BOOLEEN], avec, en particulier, les relations d'ordre suivantes (où X et Y sont deux types arbitraires).




En généralisant:

Notation [λT U V ..., E(T,U,V,...)]
où E(T,U,V...) est n'importe quel type construit à partir des types de base et des types symboliques T, U, V...

Les "paramètres" T,U,V doivent être choisis de façon à ne pas coïncider avec des types de base.

Abréviation , équivalence, forme canonique (règles plus simples que celles du λ-calcul):

[λT,[λ U, E(T,U)]] ≡ [λTU, E(T,U)] ≡ [λUT, E(T,U)]

[λT, E(T)] ≡ [λU, E(U)]

E([λT,T],U,...)≡ [λT, E(T,U,...)]

Il y a plusieurs façons de définir une forme canonique: il faut en choisir une et s'y tenir.

Signification voir [28,40,52]
[λT,E(T)]contient toutes les expressions ayant un type de la forme E(X), où X est un type arbitraire (autre que [λT,T] puisqu'on convient que E(λT.T) = λT,E(T) )

Ordre induit

1 - si $E_1(T) < E_2(T)$, alors $[\lambda T, E_1(T)] < [\lambda T, E_2(T)]$

2 - pour tout type$^X$ autre que $[\lambda T, T]$, $[\lambda T\ E(T)] > [E(X)]$

Plus précisément, $[\lambda T, E(T)]$ est la borne supérieure (unique) de l'ensemble des types: $\{E(X) \mid$ pour tout type $X\}$.

L'ensemble de tous les types a une borne supérieure unique $[\lambda T, T]$ qui désigne l'ensemble de toutes les expressions quelque soit leur type.

Avec ces précautions, cette construction n'introduit, avec les autres, aucune ambiguïté de notation et aucune contradiction sur l'ordre.

### e'/ Polymorphisme dual

De façon duale on peut désigner par $[\mu TUV\ E(T,U,V)]$ la borne inférieure de tous les types de la forme $E(T,U,V)$.

### f/ Restriction

Cette construction ne s'avérera probablement pas nécessaire, mais on peut la laisser à titre d'exemple.

Notation $[\lambda TUV\ \ R(T,U,V) \Rightarrow E(T,U,V)]$

où $R(T,U,V)$ désigne une restriction, c'est-à-dire une relation entre T,U,V.

Signification Désignation du type de toutes les expressions ayant un type de la forme $E(X,Y,Z)$ à condition que les types X,Y,Z soient liés par la relation R.

exemple: $[\lambda TU\ \ U>T \Rightarrow T \to U]$ désigne toutes les applications d'un type dans un type strictement plus large(telles que $N \to R$ ou $T^i \to T^*$).

Noter les équivalences $[\lambda TUV\ E(TUV)] \equiv [\lambda TUV\ \ VRAI \Rightarrow E(TUV)]$

$[\emptyset] \equiv [\lambda TUV\ \ FAUX \Rightarrow E(TUV)]$

Ordre induit

Si $R(T,U,V) \not\equiv VRAI$::

$[\lambda TUV\ R(T,U,V) \Rightarrow E(T,U,V)] < [\lambda TUV,\ E(T,U,V)]$

Si $R_1(T,U,V) \supset R_2(T,U,V)$, $R_1 \not\equiv R_2$ :

$[\lambda TUV\ R_1(T,U,V) \Rightarrow E(T,U,V)] < [\lambda TUV\ R_2(T,U,V) \Rightarrow E(TUV)]$

Si $R(X,Y,Z)$ est vrai et $R(T,U,V) \not\equiv \{(T,U,V) = (X,Y,Z)\}$:

$[E(X,Y,Z)] < [\lambda TUV\ R(T,U,V) \Rightarrow E(T,U,V)]$

Cette construction n'introduit avec les autres ni ambiguité de notation, ni contradiction sur l'ordre.

g/ Union

h/ Intersection

i/ Différence

L'utilisation conjointe de ces trois constructions n'est pas recommandée: elle permettrait de désigner trop de domaines et rendrait trop difficile l'identification du type minimal d'une expression.

Notation (pour l'union) $[T \cup U]$

abréviations $[T \cup U \cup V]$ pour $[[T \cup U] \cup V]$

$T$ pour $T \cup T$

équivalences $[T \cup U] \equiv [U \cup T]$

et toutes les propriétés usuelles de $\cup$, $\cap$, $-$

Ordre induit Correspond aux propriétés des opérations $\cup$, $\cap$, $-$, vis-à-vis de l'inclusion.

j/ Nomination (récursive ou non)

Cette construction permet de définir (éventuellement de façon récursive) de nouveaux types ou même de nouvelles règles de construction de types.

Notation

1ère notation: nouveau type de base DEF : T

2ème notation: nouveau type $Z \stackrel{def}{=} E(T,U,V,\ldots Z)$

où Z est un nouvel identifieur de type

et E un type construit à partir des types T,U,V,... et éventuellement Z lui-même.

exemple: $[LISTENTIERS] \stackrel{def}{=} [Nu[N \times LISTENTIERS]]$

3ème notation: nouvelle construction

$Z(T,U,V\ldots) \stackrel{def}{=} E(T,U,V,\ldots,Z(T,U,V))$

où Z est un nouveau nom,

T,U,V... sont des variables désignant des types arbitraires

E est un type construit à partir de ces types arbitraires, des types de base et éventuellement de Z lui-même.

exemple: $[LISTE(T)] \stackrel{def}{=} [Tu[T \times LISTE(T)]]$

Signification

1er cas: "déclaration" d'un type de base

2ème cas: "déclaration" d'un type particulier qu'on ne sait pas définir à l'aide des autres constructeurs.

3ème cas: "déclaration" d'une construction.

La nécessité des 2ème et 3ème cas n'est pas évidente pour les constructions récursives, particulièrement si l'on maintient la construction c) (itération).

Ordre induit

1er cas: des relations entre le nouveau type de base et d'autres types doivent être données explicitement et sans contradiction.

2ème et 3ème cas: il peut exister des relations d'ordre induites qui doivent être prouvées par récurrence.

Ainsi, dans l'exemple du 3ème cas, $[LISTE(T_1)] < [LISTE(T_2)]$.

En effet, en appelant $\Omega$ le type indéfini, on a

(cas de base) $T_1 \cup (T_1 \times \Omega) < T_2 \cup (T_2 \times \Omega)$ , et

(étape de récurrence: la propriété est point fixe de la définition):

$$LISTE(T_1) < LISTE(T_2) \supset \left| T_1 \cup [T_1 \times LISTE(T_1)] \supset T_2 \cup [T_2 \times LISTE(T_2)] \right|$$

## k/ Qualification

Notation [QUALIF.T]

(On pourrait tout aussi bien adopter une notation par suffixe)

Signification

On a vu (§2) qu'il pouvait être nécessaire de partitionner un type en sous-types permettant de distinguer constantes, variables, expressions, etc...

D'une manière plus générale, on peut considérer comme constructeur de type, n'importe quel qualificatif délimitant des expressions selon un critère qui ne dépend pas du type T lui-même (le qualificatif doit pouvoir s'appliquer à n'importe quel type).

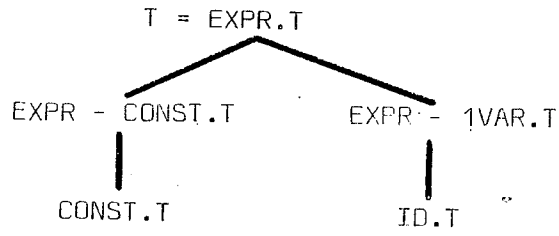
Ordre induit

Les divers qualificatifs peuvent être munis d'un ordre partiel.

On a toujours $[QUALIF.T] \leq [T]$

et $[QUALIF.T_1] \leq [QUALIF.T_2]$

*exemple* (cf. §2) :




## 4. Redondances des constructions, égalités de types

Nous avons négligé jusqu'ici l'un des problèmes traditionnels qui se rencontrent dans la définition des types: il s'agit de savoir si deux types apparemment distincts (c'est-à-dire obtenus par des suites de constructions distinctes) désignent ou non les mêmes ensembles d'expressions.

Exemples d'égalités traditionnelles de types

Exemple 1:

Une relation peut être indifféremment considérée comme

- une application [TxU → Booléen]
- un ensemble de couples liés par cette relation, donc du type [P(TxU)]
- une application indiquant pour chaque objet de T l'ensemble des objets de U qui lui sont liés par la relation, donc du type [T → P(U)]

Exemple 2:

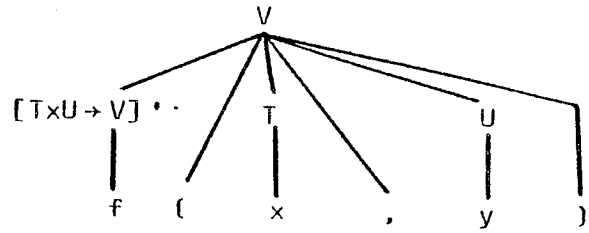
[TxU] et [UxT] peuvent être facilement identifiés.

Exemple 3:

Soit une fonction f de deux variables x,y de types respectifs T et U, et à valeur dans un type V.

L'expression "f(x,y)" a pour analyse syntaxique l'arbre:

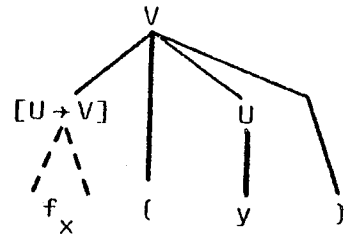
(1)




où f a pour type [TxU→V]

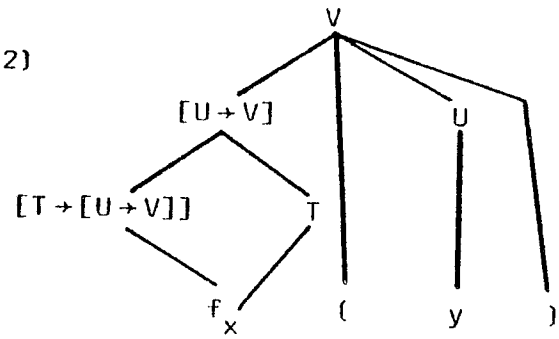
Mais on peut considérer aussi que le même objet f(x,y) peut être décrit par l'expression "$f_x(y)$"

où $f_x$ est donc une fonction de U dans V qui dépend de x




et où, par conséquent f est du type: [T→[U→V] :

(2)




On a donc l'habitude d'identifier les types:

[TxU→V] et [T→[U→V]]

Nous allons pourtant considérer que ces deux types sont distincts et que l'identifieur f dans la phrase 1 et l'identifieur f dans la phrase 2 désignent deux objets différents qui ont reçu le même nom par abus de langage parce qu'il

sont liés par une application canonique entre $T \times U \to V$ et $T \to [U \to V]$.
Cet artifice, qui repousse le problème de l'égalité des types vers la phase d'analyse syntaxique s'avèrera très utile.

Conflits de types

A quelques exceptions près recensées ci-dessous, on considérera donc que deux types qui s'écrivent différemment sont par définition différents.
Ainsi,

(Ex1) $[T \times U \to BOOLEEN]$, $[P(T \times U)]$, $[T \to P(U)]$
sont des types différents,

(Ex2) $[T \times U]$ et $[U \times T]$ sont différents

(Ex3) $[T \times U \to V]$ et $[T \to [U \to V]]$ sont différents.

Dans ces trois cas et dans de nombreux autres, il existe des applications canoniques tellement évidentes entre les types que les objets correspondants reçoivent en général le même nom. Il s'agit d'*abus de langage* dont l'utilisateur n'aura en général pas conscience, mais qui poseront lors de l'analyse syntaxique un problème de conflits de types.
On verra plus loin qu'en présence de tels conflits de types, l'analyseur syntaxique devra acquérir les règles de grammaire supplémentaires traduisant les relations canoniques entre types et permettant de passer de l'un à l'autre

Cas d'égalité entre types

Les seuls cas qui subsistent correspondent à des simplifications de notations.

. Factorisation à gauche:

$T \times U \times V$ signifie $[T \times U] \times V$

même règle pour les constructeurs $\to$, $\cup$, $\cap$.

. Précédences:

$\times$ précède $\to$ donc $T \times U \to V$ signifie $[T \times U] \to V$
$T \to U \times V$ signifie $T \to [U \times V]$

$*$ précède $\times$ donc $T \times U^*$ signifie $T \times [U^*]$

On peut décider d'autres règles de précédence concernant $\cap$, $\cup$, $\lambda$...

. formes canoniques:

Pour les " $\lambda$ types": on définit l'égalité par réduction à une forme canonique, par exemple: un seul $\lambda$ en tête de type et les variables s'appellent $T_1$ $T_2$ $T_3$... dans leur ordre d'apparition dans l'expression.
Ainsi $Ru[\lambda U,[U \to \lambda V[R \to V]]]$
se réécrira $\lambda T_1 T_2[R \cup T_1 \to [R \to T_2]]$
pour les "$\lambda$ R⇒"types: il faut en plus disposer d'une forme canonique pour les restrictions afin de détecter l'équivalence de deux restrictions.

. pour les types récursifs (constructeur j)

L'égalité est plus difficile à déterminer. Même si les membres de droite des équations de définition sont mis sous forme canonique, on peut définir de façons différentes des types identiques, par exemple:

(1) $LIST(T) \stackrel{def}{=} Tu[TxLIST(T)]$

est équivalente à

(2) $LIST(T) \stackrel{def}{=} LIST0(T) \cup LIST1(T)$

avec $LIST0(T) \stackrel{def}{=} T^2 \cup [T^2 x LIST0(T)]$

$LIST1(T) \stackrel{def}{=} T \cup [T^2 x LIST1(T)]$

Les études portant sur certaines constructions pouvant intervenir dans des définitions récursives [28,52] montrent que l'équivalence des types récursifs peut être difficile ou impossible à établir.

Ce constructeur doit donc être manipulé avec précautions et remplacé si possible par un constructeur itératif tel que (c). En effet, l'existence insoupçonnée de types équivalents peut conduire à des ambiguités insolubles d'analyse syntaxique (chapitre II) pouvant faire échouer des démonstrations (chapitre III).

## 5. Tableau récapitulatif

Le tableau ci-joint est un résumé des diverses constructions de types proposées.
Ces constructions sont celles qui ont paru intervenir pour prendre en compte l'expression de propriétés de programmes ou la transcription de textes mathématiques. En cas d'implémentation, il est vraisemblable qu'on pourra se passer de certaines de ces constructions.

Il est très recommandé de choisir un sous-ensemble ou un sur-ensemble de ces constructions pour lequel la relation d'ordre assure une structure de *treillis*, ceci afin de faciliter l'analyse syntaxique décrite au prochain chapitre.

| Constructeur | Désignation | Signification | Ordre avec $T_1 < T_2$ ... $U_1 < U_2$ ... |
|---|---|---|---|
| × | $T \times U$, $T^2$ | Produit cartésien | $T_1 \times U < T_2 \times U$ ; $T \times U_1 < T \times U_2$ |
| → | $T \to U$ | Application | $T_1 \to U < T_2 \to U$ ; $T \to U_1 < T \to U_2$ |
| * | $T^*$ | Suites(finies ou non) d'expressions de type T | $T_1^* < T_2^*$ |
| $\mathcal{P}$ | $\mathcal{P}(T)$ | Ensembles(finis,infinis,vides) d'expressions de type T | $\mathcal{P}(T_1) < \mathcal{P}(T_2)$ |
| λ | λT,U,V... E(T,U,V...) | Borne sup. de tous les types de la forme E | $E(T_1,U_1,V_1...) < \lambda T,U,V...E(T,U,V)$ |
| μ | μT,U,V... E(T,U,V) | Borne inf de tous les types de la forme E | $\mu T,U,V...\ E(T,U,V) < E(T_1,U_1,V_1...)$ |
| λR⇒ | λT,U,V... R(T,U,V...) ⇒E(T,U,V...) | Borne sup de tous les types de la forme E avec la restriction R | si $R(T_1,U_1,V_1...)$: $E(T_1,U_1,V_1...) \leq [\lambda TUV...R(T,U,V...) \Rightarrow E(T,U,V...)]$ $[\lambda TUV\ VRAI \Rightarrow E(T,U,V)] \equiv [\lambda TUV\ E(T,U,V)]$ si $R \neq VRAI$ $\lambda T\ R(T) \Rightarrow E(T) < \lambda T\ E(T)$ |
| ∪ | T ∪ U | Union | $T_1 \cup U \leq T_2 \cup U$ ; $T \cup U_1 \leq T \cup U_2$ |
| ∩ | T ∩ U | Intersection | $T_1 \cap U \leq T_2 \cap U$ ; $T \cap U_1 \leq T \cap U_2$ |
| - | T - U | Différence | $T_1 - U \leq T_2 - U$ ; $T - U_1 \geq T - U_2$ |
| DEF | | voir texte | |
| QUALIF | QUALIF.T | Expressions du type T conformes à QUALIF | $QUALIF.T_1 < QUALIF.T_2$ si QUALIF 1 ⊃ QUALIF 2 : QUALIF1.T ≤ QUALIF2.T |

## III - CONNAISSANCE DES TYPES D'UNE EXPRESSION

L'importance de la connaissance des types d'une expression a été soulignée plus haut par l'affirmation:

*"Toute incompréhension, ou toute ambiguité d'interprétation viennent d'une connaissance insuffisante des types de certaines expressions"*

Ainsi, après avoir donné un exemple de langage de types, on peut maintenant voir avec plus de précision comment déterminer les types d'une expression.

L'ensemble des types d'une expression est - par définition - stable par la relation $<$ . Cela signifie que si une expression appartient au type T et si $T<U$, alors cette expression appartient (a fortiori) au type U.

L'ensemble des types d'une expression est entièrement défini par son minimum qui est toujours unique si l'on a le constructeur $\cap$ (intersection de types). Par abus de langage, ce type minimum pourra être appelé *le* type de l'expression.

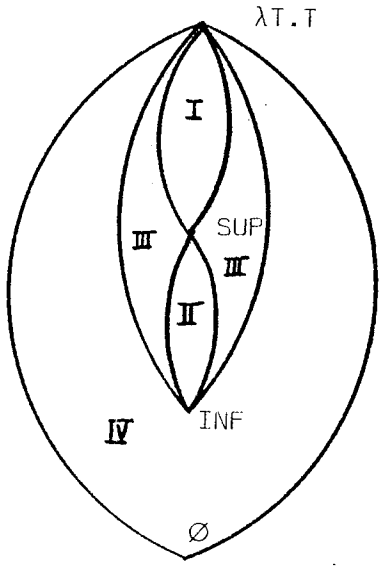
### Détermination par contexte du type (minimum) d'une expression

Exemple: Supposons les types de base $\mathbb{R}$ (réel et $\mathbb{N}$ (entier) avec l'ordre $R>N$ (puisque tout entier est réel),
l'expression "f(x)=y" implique que l'objet f est une fonction, mais on n'en connait ni le domaine ni le but.
Son type (minimum) est donc *borné supérieurement* par $[\lambda TU, T \to U]$
la nouvelle expression "f(2)=3" implique maintenant que le domaine et le but sont au moins de type N.
Donc le type (minimum) de f est *borné inférieurement* par $[N \to N]$.
Si enfin, on lit "f(3)=3.14", on peut en déduire que la borne inférieure s'élève à $[N \to R]$ (qui est supérieure à $[N \to N]$).

### Cas général:

Le type (minimum) d'une expression est toujours approximé par une borne INF et une borne SUP (éventuellement $\emptyset$ et $\lambda T.T$ si on ne sait *rien* sur ce type).

Par rapport à cette expression, on peut répartir l'ensemble de tous les types quatre régions:




Région I : tous les types T ≥ SUP
Région II: tous les types INF ≤ T < SUP
Régions III+II+I : tous les types T ≥ INF
Région IV: tous les autres types.

Pour l'expression concernée le type (minimum) est certainement dans la région II+SUP, l'expression appartient certainement à tous les types de la région I et n'appartient certainement à aucun des types de la région IV.

Les régions II+III représentent *l'indétermination* sur les types: on ne sait pas si l'expression leur appartient.

L'indétermination est nulle si borne INF = borne SUP
L'indétermination est totale si borne INF = Ø, borne SUP = λT.T

- Il est intuitif, et il se confirmera avec précision (chapitre III, sémantique déductive) que l'utilisation des spécifications peut nécessiter une réduction de l'indétermination des types des identifieurs.

- La seule façon de réduire l'indétermination sera d'utiliser les identifieurs dans de nouveaux contextes qui permettront *d'inférer* des bornes de plus en plus précises sur leur type (voir l'exemple ci-dessus et l'analyse syntaxique, chapitre II).

## CONCLUSION

On voit qu'une grande partie de la tâche de l'analyseur syntaxique sera de *deviner* les types des expressions en présence, en effet, le langage de spécification serait inutilisable si l'on obligeait l'utilisateur à déclarer les

identifieurs en leur affectant un type. D'ailleurs l'utilisateur est sensé tout ignorer de la notion de type sous-jacente au langage.

On verra, dans le prochain chapitre, que l'analyseur syntaxique peut approximer le type minimum de chaque identifieur par des techniques d'*apprentissage* dont on vient de montrer un exemple), l'autorisant éventuellement à poser des questions destinées à mieux localiser ce type.

II

# SYNTAXE

## RESUME DU CHAPITRE II - SYNTAXE

*On examine dans ce chapitre la représentation des expressions et, tout particulièrement, des expressions du type "Spécification ". On part du principe que toute expression est soit un identifieur, soit le résultat d'une opération sur des expressions.*

*Le langage est donc constitué d'un ensemble d'opérateurs et d'identifieurs.*

*Chaque opérateur a ses arguments et son résultat délimités par des types éventuellement liés par une relation. Chaque opérateur est aussi décrit par une règle de production - style context-free - où les types jouent le rôle de symboles non-terminaux.*

*Il faut établir sur chaque phrase un arbre syntaxique et l'interpréter, c'est-à-dire affecter à chaque noeud un type, conformément à l'opérateur correspondant.* *

*L'ambiguité est représentée par la multiplicité de ces interprétations. L'absence ou la pluralité des arbres interprétables indiquent la nécessité d'une extension du langage:*
*L'extension est le processus permanent et interactif par lequel le système infère les nouveaux éléments du langage (opérateurs, types, identifieurs) que l'utilisateur emploie sans déclaration explicite, y compris les indispensables figures de rhétorique, considérées comme transformateurs d'opérateurs.*

*Cet apprentissage, par le système, du langage de l'utilisateur, nécessite un noyau de départ suffisant, et un minimum de discipline de l'utilisateur. La rapidité de cet apprentissage dépend certainement beaucoup de la qualité du mécanisme d'extension.*

* *Certains auteurs considèrent la détermination des types comme relevant de la "sémantique".*

## 0 - INTRODUCTION

La forme du langage que je propose vient de l'observation expérimentale que le discours mathématique est décomposable en *opérateurs*.

On constate en effet que toute expression, y compris les phrases entières, est soit réduite à un identifieur, soit le résultat d'une opération sur des expressions, comme pour les langages context-free.

Mais dans un langage context-free, l'alphabet des symboles non-terminaux est fini, et toute expression doit pouvoir être dérivée de l'un de ces symboles ; or le chapitre précédent indique qu'il nous faut une infinité de types pour classer les expressions.

Le langage ici proposé se distingue des langages context-free par les points suivants:

- L'alphabet non terminal est remplacé par un ensemble infini qui est le langage des types (comparer avec ALGOL 68 [62]).
- Le langage comporte une infinité de règles de production décrites par un ensemble fini de "règles opératoires"(cf. page suivante).
- Une forme d'ambiguité (incertitude sur les types d'expressions) est bien tolérée grâce à l'ordre partiel sur les types qui permet de trouver une forme close de toutes les analyses possibles de chaque phrase (p. 51-58).
- Le langage est extensible, non pas seulement en début de session, mais par évolution permanente de la grammaire (apprentissage interactif du langage de l'utilisateur) (p. 65-83).

## I - ETAT DU LANGAGE

Le langage, qui est extensible, est à tout instant défini par trois ensembles,

- un ensemble de types partiellement ordonnés,
- en ensemble de "règles opératoires",
- un ensemble d'identifieurs typés.

Les contenus de ces trois ensembles constituent l'état du langage et rassemblent toute l'information à partir de laquelle peut s'effectuer l'analyse syntaxique décrite dans ce chapitre.

Le contenu et le format de ces trois ensembles sont les suivants:

## 1/ l'ensemble des types

L'ensemble partiellement ordonné des types est entièrement défini par:

- un ensemble partiellement ordonné de types de base,
- un ensemble de constructeurs de types tels que ceux qui sont présentés dans le chapitre précédent.

A chaque construction sont associées:

. sa syntaxe (ambiguités exclues par l'usage de symboles exclusifs de parenthésages et de règles de priorité),
. éventuellement les règles d'égalité correspondantes,
. toujours les règles concernant la relation d'ordre.

En principe, on doit pouvoir à tout instant trouver automatiquement et pour tout couple de types l'éventuelle relation d'identité ou d'ordre qui les lie.

## 2/ ensemble des"règles opératoires"

Chaque opérateur a sa syntaxe propre décrite par une *règle opératoire*. Pour un opérateur d'arité fixe, cette règle ressemblera aux règles de production des langages context-free, la partie gauche désignant le type du résultat, la partie droite une expression comportant les types des arguments et du "sucre syntaxique" permettant de reconnaître l'opérateur. Si l'on admet des opérateurs d'arité variable, la partie droite sera une expression un peu plus compliquée (description d'un langage régulier).

Si les types des arguments et du résultat, au lieu d'être déterminés, sont seulement liés par une contrainte, celle-ci apparaît dans la règle opératoire qui représente donc autant de règles de production que la contrainte n'a de solutions (souvent une infinité).

Définition :

Une règle opératoire est un triplet <C,P,E> qui se présente sous la forme:

*si* $C(T_1...T_n)$ : $P(T_1...T_n)$ ::= $E(V_1(T_1...T_n)...,V_m(T_1...T_n))$

où

$T_1...T_n$ sont des identifieurs distincts des identifieurs de types de base et désignant des types "libres" (substituables);

$C(T_1...T_n)$ est une relation entre les types désignés par ces variables ;

$P(T_1...T_n)$ } sont des types construits à l'aide de tout ou partie des
$V_j(T_1...T_n)$ } types $T_1...T_n$ et des types de base ;

$E(V_1...V_m)$ est une expression (chaîne de caractères) comportant
- les types $V_1(T_1...T_n)$ ..., $V_m(T_1...T_n)$
- des symboles terminaux.

- Signification

Pour tous les types $T_1...T_n$ liés par la relation C, on obtient une expression de type $P(T_1...T_n)$ en écrivant l'expression E.

- Exemple: règle de la notation fonctionnelle

*si* $\underline{W} = \underline{V \to Z}$ ; $\underline{Z}$ ::= $\underline{W}\ (\underline{V})$

$C(V,W,Z)$ $P(V,W,Z)$ $E(V,W,Z)$

- Note

Pour ne pas confondre des symboles de construction de types (parenthèses, ×, →, ...) avec d'éventuels symboles terminaux, chaque type est individuellement souligné.

- Réécriture, équivalence, forme canonique

Si C implique une égalité de deux types, on la substituera à chaque fois que c'est possible dans P et E de façon à minimiser la taille de C et le nombre de types variables.

Par exemple, la règle de notation fonctionnelle ci-dessus se réécrira beaucoup plus simplement:

*pour tous* V,Z : $\underline{Z}$ ::= $\underline{V \to Z(V)}$

Nous verrons plus loin (p 75-79) qu'on peut aussi avoir intérêt parfois:

. soit à scinder une règle en deux:

$<C_1 \vee C_2, P, E>$

étant remplacée par deux règles équivalentes à $\begin{cases} <C_1, P, E> \\ <C_2, P, E> \end{cases}$

. soit à fusionner deux règles analogues:

$\begin{cases} <C_1, P, E> \\ <C_2, P, E> \end{cases}$

étant remplacées par la seule règle $<C_1 \vee C_2, P, E>$

. soit à singulariser des cas importants d'une règle (emphase):

$<C, P, E>$

donnant lieu à l'addition (redondante) d'une règle

$<C', P, E>$ pour des cas $C' \supset C$.

- Autres informations

Au triplet qui constitue une règle opératoire, peuvent être associés d'autres renseignements utiles à l'analyseur syntaxique tels que

- nom de la règle,
- précédence par rapport à d'autres règles,
- caractère local ou global des variables,
- statistiques d'utilisation,

etc...

## 3/ ensemble des identifieurs typés

Il s'agit de l'ensemble des identifieurs de variables globales (utilisables dans le contexte présent). On indique ce qu'on sait sur leur type, c'est-à-dire sa borne inférieure I et sa borne supérieure S (cf. p. 37-38). Si le type est parfaitement déterminé, ces bornes sont égales, s'il est totalement inconnu, elles valent respectivement $\emptyset$ et $\lambda T.T$.

L'ensemble des identifieurs typés se présente donc comme une suite de triplets <NOM, Borne Inf I, Borne Sup S>. Il faut évidemment toujours que $I \leq S$.

Enfin, rappelons qu'*aucun de ces trois ensembles qui constituent l'état du langage de spécifications n'est directement accessible à l'utilisateur qui peut même en ignorer l'existence.* Il n'y a pas de déclaration explicite de types, de règles ou d'identifieurs ; tout ce que l'utilisateur écrit est considéré par l'analyseur comme faisant partie du langage de spécifications et non d'un quelconque métalangage. C'est donc l'analyseur qui devra déduire les modifications à apporter à l'état du langage selon le processus d'extension et d'inférence qui sera décrit plus loin.

## II - ANALYSE SYNTAXIQUE

### 1. La fonction de l'analyseur syntaxique est double:

a/ *Reconnaître et interpréter les phrases de l'utilisateur,*

b/ *Même si une phrase n'est pas conforme à l'état du langage de spécifications, déterminer à l'aide de questions-réponses l'éventuelle erreur ou les extensions à apporter au langage pour inclure cette phrase.*

Cette deuxième fonction d'"apprentissage" qui remplace la traditionnelle phase de récupération d'erreurs sera plus particulièrement explorée en section III: extensions.

La première fonction (reconnaissance et interprétation) n'a que quelques ressemblances avec l'analyse des langages "context free" et a pour but d'établir un *"Arbre Interprété"* ayant les propriétés suivantes:

### Arbre Interprété:

1 - chaque noeud est étiqueté par un type et le nom de la règle opératoire appliquée.

2 - le noeud origine est étiqueté par le type "SPECIF"

3 - le sous-arbre de chaque noeud correspond à l'application de la règle nommée par ce noeud.

4 - les terminaux ("feuilles") lus de gauche à droite constituent le texte soumis à l'analyse.

5 - tout terminal désignant un identifieur est surmonté d'un noeud dont le type est compris entre les bornes désignées pour cet identifieur.

Le reste de cette section a pour but de montrer comment obtenir un tel arbre interprété. La procédure proposée opère en deux phases:

- Phase A : Analyse Structurelle
- Phase B : Interprétation.

Pour faciliter la description de ces deux phases et éviter l'excès de formalisation, nous allons utiliser l'exemple illustratif suivant:

Exemple:

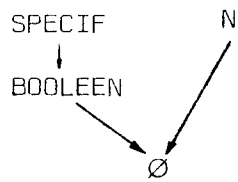
Soit la phrase à analyser sous sa forme proposée par l'utilisateur:

| "α est transitif" signifie"xαy et yαz ⊃ xαz" |
|---|

et nous supposerons que l'*état du langage* (à l'instant donné) est le suivant:

Etat du langage

a/ *types de base:*




b/ *règles opératoires:*

| N° | C | P | E | commentaires |
|---|---|---|---|---|
| (1) | VRAI(T) | SPECIF | ::= " T " signifie " T " | (définition) |
| (2) | VRAI(T) | SPECIF | ::= T est T→BOOLEEN | 2ème argument (attribut):global |
| (3) | VRAI(T,U,V) | V | ::= T TxU→V U | (notation infix) |
| (4) | VRAI(T) | T | ::= ( T ) | |
| (5) | VRAI | SPECIF | ::= SPECIF et SPECIF | |
| (6) | VRAI | SPECIF | ::= SPECIF ⊃ .SPECIF | |
| (7) | T ⩾ borne inf | T | ::= *identifieur* | n'importe quel identifieur |
| ... | ... | ... | | |

c/ *identifieurs typés*

| nom | type minimum | |
|---|---|---|
| | borne inf | bonne sup |
| transitif | (NxN→BOOLEEN)→BOOLEEN | - |

(Cette borne inf pourrait résulter d'une phrase antérieure telle que "> *est transitif*" )

## 2. Phase A: analyse structurelle

Il s'agit dans ce premier temps d'établir l'arbre syntaxique sans aucune vérification de types, c'est-à-dire de constituer *l'arbre non interprété* défini comme suit:

### Arbre non interprété

1'- chaque noeud est étiqueté par le nom de la règle opératoire appliquée,

3 - le sous-arbre de chaque noeud correspond à l'application de la règle nommée par ce noeud,

4 - les terminaux ("feuilles") lus de gauche à droite constituent le texte soumis à l'analyse.

Définition du langage context free adjoint:
C'est un langage qui n'a qu'un seul non-terminal noté: □
Il est obtenu à partir de l'*état* du langage de spécifications en remplaçant toute règle opératoire:

*si* $C(T_1 \ldots T_n) : P(T_1 \ldots T_n) ::= E(V_1 \ldots V_m)$
par la règle de production: $\square ::= E(\square \ldots \square)$
(les types $V_j$ apparaissant dans E sont tous remplacés par: □)

Exemple: le langage adjoint correspondant à notre exemple est:

(1) □ ::= "□" signifie "□"
(2) □ ::= □ est □
(3) □ ::= □ □ □
(4) □ ::= (□)
(5) □ ::= □ et □
(6) □ ::= □ ⊃ □
(7) □ ::= *identifieur*

Remarque

S'il existe un arbre interprété pour une phrase, l'arbre non interprété correspondant est un arbre syntaxique de cette phrase dans le langage adjoint.

Conséquence
Le langage adjoint inclut le langage de spécifications.

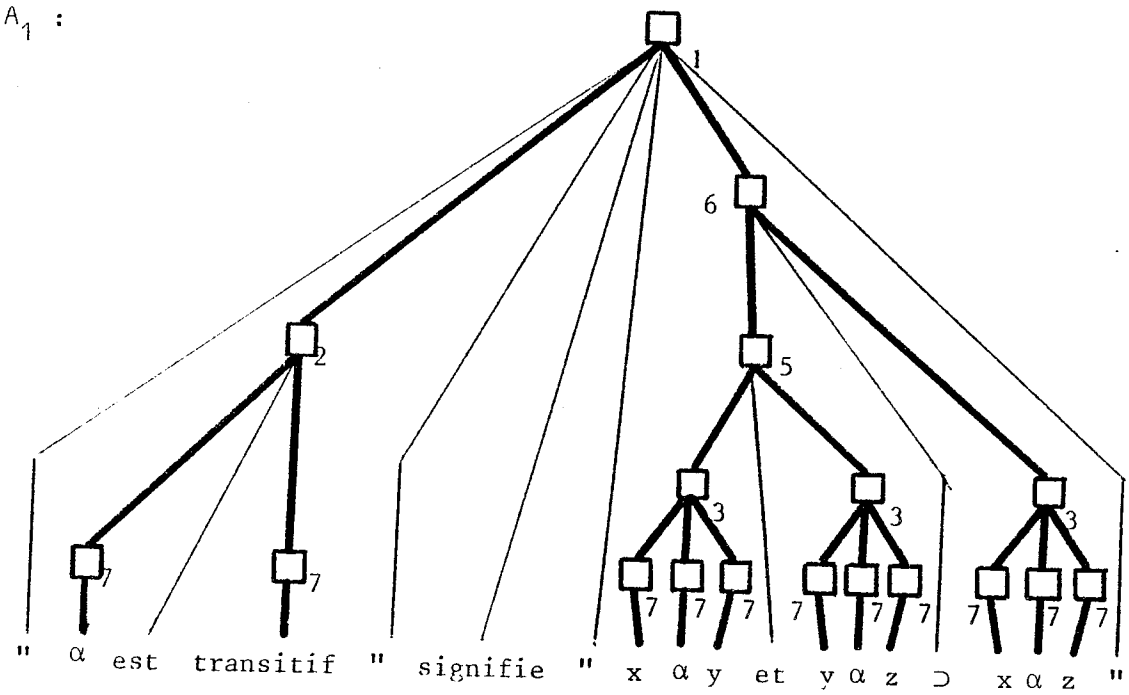
L'inverse est faux: nous verrons que dans la Phase B (Interprétation) un arbre syntaxique correct pour le langage adjoint peut n'avoir aucune interprétation (affectation de types aux noeuds) conforme à l'état du langage de spécifications.

Analyse structurelle

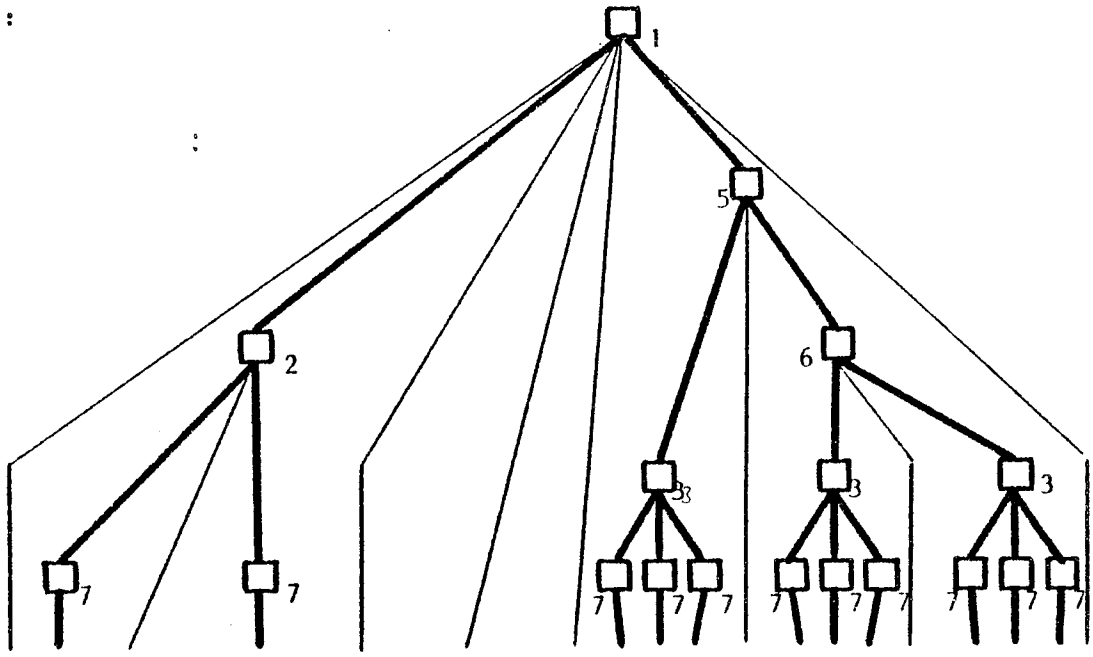
Elle consiste simplement à analyser la phrase au sens du langage adjoint, ce qui aboutit à 0, 1 ou plusieurs arbres syntaxiques.
Si cette analyse ne reconnait aucun arbre, le corollaire ci-dessus nous autorise à conclure qu'a fortiori la phrase n'est pas conforme à l'état du langage de spécifications ;
Si elle reconnait plus d'un arbre, le langage adjoint est ambigu: c'est le cas pour l'exemple qui est proposé ici et qui donne lieu à deux arbres syntaxiques $A_1$ et $A_2$.

Nous montrons les deux arbres non interprétés obtenus avec, à coté de chaque noeud, le numéro de la règle appliquée:

$A_1$ :



$A_2$ :




Identification de l'ambiguité

L'ambiguité peut être localisée avec plus ou moins de précision, par exemple:

- localisation grossière: très facile à déterminer automatiquement, elle consiste à identifier la plus courte sous-chaîne donnant lieu à l'ambiguité, ici: xαy et yαz ⊃ xαz

- localisation précise: plus coûteuse à établir, elle consiste à isoler la différence minimale entre les arbres. Ici, on passe d'un arbre à l'autre, en interchangeant:


distingués par les sous-graphes respectifs:

Selon la qualité de la localisation, l'analyseur pourra poser des questions plus ou moins précises à l'utilisateur pour résoudre l'ambiguité.(cf.p.65-71)

Par exemple, la localisation précise se réduit simplement à un choix entre


c'est-à-dire à décider laquelle des deux règles (5) et (6) a la priorité sur l'autre ;
l'analyseur pourra donc simplement demander:

lequel de "et" et "⊃" a priorité sur l'autre?

la réponse ⊃ permet de choisir l'arbre $A_1$
et l'analyseur ajoute cette information (priorité de Règle 6 sur Règle 5) à l'état du langage.

Dans le cas d'une localisation grossière, l'analyseur pourra seulement répondre par une phrase du genre:

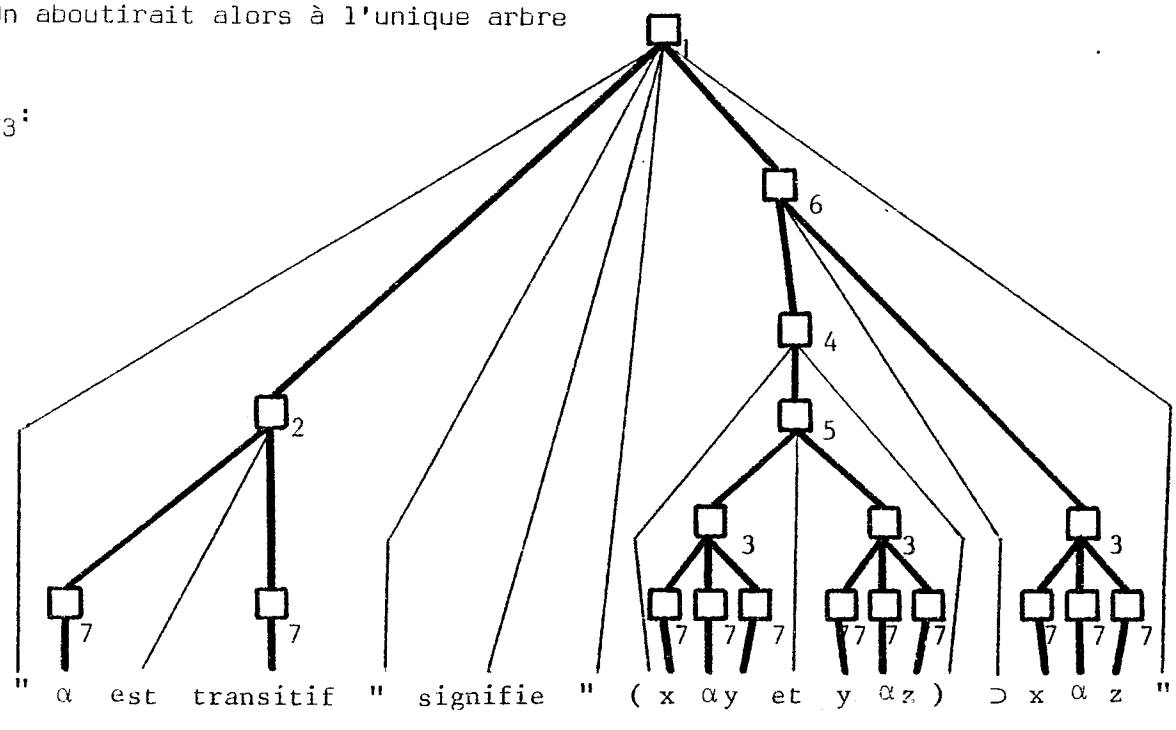
ambiguité, préciser "xαy et yαz ⊃ xαz"

qui devrait aboutir (éventuellement après d'autres question-réponses) à une réponse:

(xαy et yαz) ⊃ xαz

On aboutirait alors à l'unique arbre

$A_3$:



## Résultats de l'analyse structurelle

Cette première phase aboutit à quatre résultats possibles:

1 - un seul arbre

2 - aucun arbre.

3 - un petit nombre d'arbres } ambiguité non résolue.

4 - un grand nombre d'arbres }

Cas 1: un seul arbre:

Cas idéal: il ne reste qu'à passer en phase II : l'interprétation.

Cas 2: aucun arbre:

Il y a eu soit erreur, soit introduction implicite d'une ou plusieurs nouvelles règles.

La première tâche de l'analyseur est de localiser avec autant de précision que possible la ou les sous-chaînes qui ont causé l'échec de l'analyse.

Comme il s'agit essentiellement d'inférer une extension du langage, ce point sera traité en détail dans la section III (extensions).

Cas 3: un petit nombre d'arbres:

Le langage adjoint est ambigu.

Il se peut qu'en réalité un seul de ces arbres soit interprétable, ce que révèlera la phase B.

Si ce n'est pas le cas, il y a *"ambiguité fondamentale"*, comme pour l'exemple qui a été traité ici.

Il faut alors localiser la source d'ambiguité en isolantune partie du texte, ou mieux, une partie de l'arbre, pour identifier les règles responsables de l'ambiguité et les faire modifier par l'intermédiaire de l'utilisateur (voir section III: extensions).

Cas 4 : un grand nombre d'arbres:

On ne peut pas se permettre de les interpréter tous, il faut donc d'emblée réduire l'ambiguité comme au cas 3.

Exemple:

Avec les seules règles [] ::= [][] et [] ::= [][][]

on peut analyser:

une suite de 4 identifieurs de 10 façons différentes,

" " 5 " " 38 " " , etc...

Dans ces cas, il faut obtenir de l'utilisateur

- soit qu'il fasse un large usage de parenthèses pour limiter les ambiguités
- soit qu'il fasse usage de règles plus reconnaissables en variant les symboles terminaux (séparateurs) ou en en introduisant là où il n'y en a pas.

## 3. Phase B: interprétation

Si l'analyse structurelle (Phase A) produit au moins un arbre non interprété, il s'agit d'en effectuer *l'interprétation* c'est-à-dire d'affecter à chaque noeud un type, de façon que toutes les règles opératoire soient légales.

Il sera rare qu'il existe exactement une interprétation: il pourra fréquemment n'y en avoir aucune ou au contraire plusieurs ou une infinité.

### a/ Caractérisation de l'ensemble des interprétations valides

Supposons que l'arbre comporte un noeud à interpréter et notons le: $A(V_1 \ldots V_m)$.

On peut considérer qu'au départ l'arbre non interprété produit par l'analyse structurelle est caractérisé par: $\forall_j : V_j = []$

Une interprétation est donc l'affectation à $V_1 \ldots V_m$ de m types construits à partir des types de base et telle que l'arbre obtenu $A(V_1 \ldots V_m)$ satisfasse aux cinq conditions qui définissent un arbre interprété conformément à l'état du langage (page 44 ).

Notre but est de déterminer *l'ensemble I de toutes ces interprétations.*

Comme I peut contenir un grand nombre ou une infinité d'interprétations, il faut en donner une description compacte,par exemple de la forme:

$$\underbrace{R(T_1 \ldots T_n)}_{\text{préfixe}} \; : \; \underbrace{A(V_1(T_1 \ldots T_n), \ldots, V_m(T_1 \ldots T_n))}_{\text{arbre}}$$

où

$T_1 \ldots T_n$ sont des noms de types "libres" (donc distincts des noms des types de base),

$R(T_1 \ldots T_n)$ est une relation sur ces types,

les $V_j(T_1 \ldots T_n)$ sont des types construits à partir des types libres $T_1 \ldots T_n$ et des types de base,

$A(V_1(T_1 \ldots T_n), \ldots, V_m(T_1 \ldots T_n))$ est l'arbre complètement étiqueté par les m types

Cette description compacte (Préfixe + arbre) de toutes les interprétations possibles sera appelée dans la suite "INTERPRETATION COMPLETE".

Comme pour les règles opératoires, il existe diverses expressions de l'interprétation complète ; il faut préférer celles qui minimisent le nombre de variables $T_1 \ldots T_n$ et la taille de la condition restrictive R: notamment, les égalités qui résultent de R devraient être directement traduites dans le choix des expressions $V_j$.

Remarque sur la terminologie

Nombreux sont ceux qui appelleraient *"sémantique"* ce que j'appelle *"interprétation"* J'ai réservé le terme "sémantique" au prochain chapitre qui traite de la *signification* des phrases du langage. Dans la mesure où les types sont considérés comme symboles non terminaux du langage, l'interprétation, c'est-à-dire la recherche des types des expressions, fait bien partie de l'analyse syntaxique.

### b/ Algorithme d'interprétation complète

On trouvera ici un algorithme très simple, écrit en Franco Algol et sans souci d'optimisation, calqué sur l'habituel algorithme récursif de parcours d'arbres: le *premier passage* attribue des types libres $T_i$ et collectionne les contraintes imposées par les règles appliquées ; le *deuxième passage* résoud les contraintes d'égalité entre types par substitutions dans l'arbre.

En résumé:

*Effectuer l'interprétation complète consiste à résoudre (c'est-à-dire simplifier) le système des contraintes de types.*

Il en résulte une expression assez concise de l'interprétation complète. De nombreuses variantes plus efficaces de cet algorithme peuvent être mises au point, notamment celles où les égalités sont prises en compte sur-le-champ et non dans un deuxième passage.

Enfin, l'algorithme ne connait que l'arbre d'opérateurs (reconnaissable par les traits épais sur les figures) et ignore tous les arcs (en traits minces) correspondant au "sucre syntaxique" qui intervient dans la reconnaissance des opérateurs (phase A), mais pas dans l'attribution des types (phase B).

Après cet algorithme, on trouvera son illustration sur le même *exemple* que précédemment.

# ALGORITHME INTERPRETER (ARBRE)

paramètre:

ARBRE ::= <RACINE = <TYPE,REGLE>, suite de SOUS-ARBREs>
| IDENTIFIEUR

global

ETAT ::=
- ENSEMBLE de TYPES
- ENSEMBLE de REGLES: {<C,P,E>}
- ENSEMBLE d'IDENTIFIEURS: {<nom;type = <borne inf,borne sup> >}

local

R ::= ensemble de conditions (chaînes de caractères)

ENSEMBLE.NOMS ::= {<nom,type>} pour recenser tous les identifieurs de l'arbr

i ::= entier (utilisé pour générer des noms de types libres $T_i$)

procédure:

PROCEDURE RECURSIVE ETIQUETER (arbre A)

DEBUT

*si* A est un identifieur ∉ ENSEMBLE.NOMS

*alors* 1° ENSEMBLE.NOMS ← ENSEMBLE.NOMS ∪ <A,$T_i$> ;
2° *si* A ∈ ENSEMBLE d'IDENTIFIEURS
*alors* R ← R ∪ "borne inf(A) ≤ $T_i$" ;
3° i ← i+1

*sinon* 1° ETIQUETER (chaque sous-arbre de A)
2° (à détailler) TYPE(RACINE) ← P(REGLE(RACINE) )
3° (à détailler) R ← R ∪ contrainte C(REGLE(RACINE) )
note: C exprime la contrainte entre les types affectés à la racine A et aux racines des sous-arbres de A

FIN

programme:

DEBUT

1° [Initialisations] R ← ∅ ; ENSEMBLE.NOMS ← ∅ ; i ← 1

2° [Premier Passage] ETIQUETER(ARBRE)

3° [Deuxième Passage, à détailler]
Simplifier R : utiliser les égalités pour minimiser le nombre de types libres

RESULTAT: *si* "FAUX" ∈ R *alors* "ARBRE NON-INTERPRETABLE"
*sinon*

| R : ARBRE |
|---|
| (préfixerestrictif) (étiqueté) |

FIN

Remarque:

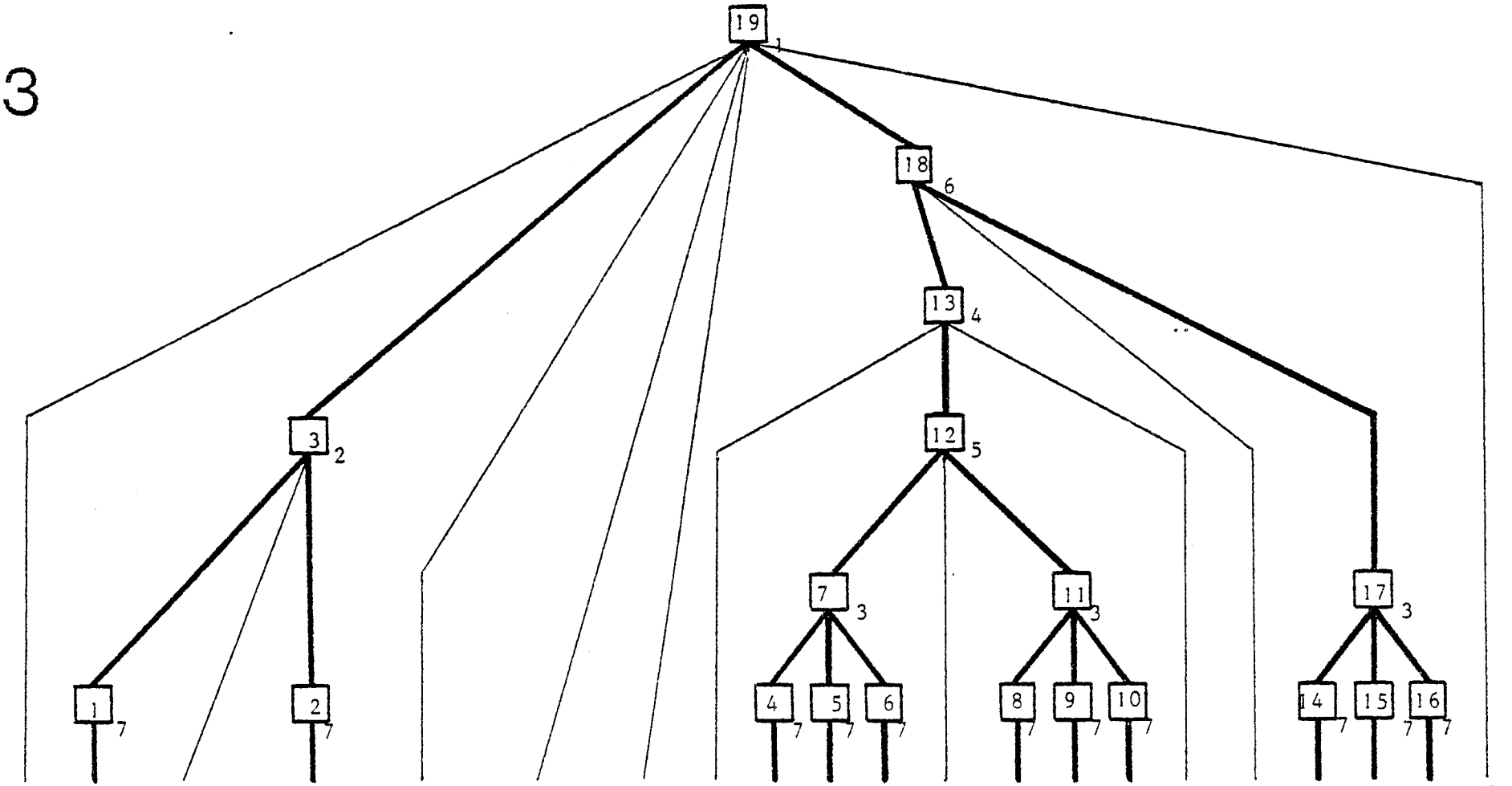
Cet algorithme affecte les types "de bas en haut", une variante est possible, affectant les types de haut en bas.

c/ Exemple

Toujours en partant du même texte, page 45 , l'arbre $A_3$ ci-joint, produit par l'analyse structurelle, et soumis à l'algorithme d'interprétation complè donne lieu à la suite des évènements ci-dessous(premier passage):

| noeud N° d'ordre | noeud N° de règle | type inséré | R contraintes liées à la règle | liste des identifieurs nom : type |
|---|---|---|---|---|
| 1 | 1° règle 7 | $T_1$ (nouveau) | | $\alpha$ : $T_1$ |
| 2 | 2° règle 7 | $T_2$ (nouveau) | $T_2 \geq (N \times N \rightarrow BOOLEEN) \rightarrow BOOLEEN$ | $\alpha$ : $T_1$<br>transitif : $T_2$ |
| 3 | règle 2 | SPECIF | $T_2 = T_1 \rightarrow BOOLEEN$ | idem |
| 4 | 3° règle 7 | $T_3$ (nouveau) | | $\alpha$ : $T_1$<br>transitif : $T_2$<br>x : $T_3$ |
| 5 | 4° règle 7 | $T_1$ (type de $\alpha$) | | idem |
| 6 | 5° règle 7 | $T_4$ (nouveau) | | $\alpha$ : $T_1$<br>transitif : $T_2$<br>x : $T_3$<br>y : $T_4$ |
| 7 | 1° règle 3 | $T_5$ (nouveau) | $T_1 = T_3 \times T_4 \rightarrow T_5$ | idem |
| 8 | 6° règle 7 | $T_4$ (type de y) | | idem |
| 9 | 7° règle 7 | $T_1$ (type de $\alpha$) | | idem |
| 10 | 8° règle 7 | $T_6$ (nouveau) | | $\alpha$ : $T_1$<br>transitif : $T_2$<br>x : $T_3$<br>y : $T_4$<br>z : $T_6$ |
| 11 | 2° règle 3 | $T_7$ (nouveau) | $T_1 = T_4 \times T_6 \rightarrow T_7$ | idem |
| 12 | règle 5 | SPECIF | $T_5 = T_7$ = SPECIF | idem |
| 13 | règle 4 | SPECIF | | idem |
| 14 | 9° règle 7 | $T_3$ (type de x) | | idem |
| 15 | 10° règle 7 | $T_1$ (type de $\alpha$) | | idem |
| 16 | 11° règle 7 | $T_6$ (type de z) | | idem |
| 17 | 3° règle 3 | $T_8$ nouveau | $T_1 = T_3 \times T_6 \rightarrow T_8$ | idem |
| 18 | règle 6 | SPECIF | SPECIF = $T_8$ | idem |
| 19 | règle 1 | SPECIF | [SPECIF = SPECIF] | idem |

# A3




numero d'ordre

numero de règle

branches principales

branches secondaires

Le deuxième passage (résolution des contraintes) consiste à *simplifier* l'ensemble R des contraintes de types collectionnées au cours du premier passage, à savoir:

$$T_2 \geq (N \times N \rightarrow BOOLEEN) \rightarrow BOOLEEN$$
$$T_2 = T_1 \rightarrow BOOLEEN$$
$$T_1 = T_3 \times T_4 \rightarrow T_5$$
$$T_1 = T_4 \times T_6 \rightarrow T_7$$
$$T_5 = T_7 = SPECIF$$
$$T_1 = T_3 \times T_6 \rightarrow T_8$$
$$T_8 = SPECIF$$
$$[SPECIF = SPECIF]$$

Les égalités sont compatibles et peuvent toutes s'exprimer en fonction de $T_3$ (un seul degré de liberté):

$$T_1 = T_3 \times T_3 \rightarrow SPECIF$$
$$T_2 = (T_3 \times T_3 \rightarrow SPECIF) \rightarrow BOOLEEN$$
$$T_3 = T_3$$
$$T_4 = T_3$$
$$T_5 = SPECIF$$
$$T_6 = T_3$$
$$T_7 = SPECIF$$
$$T_8 = SPECIF$$

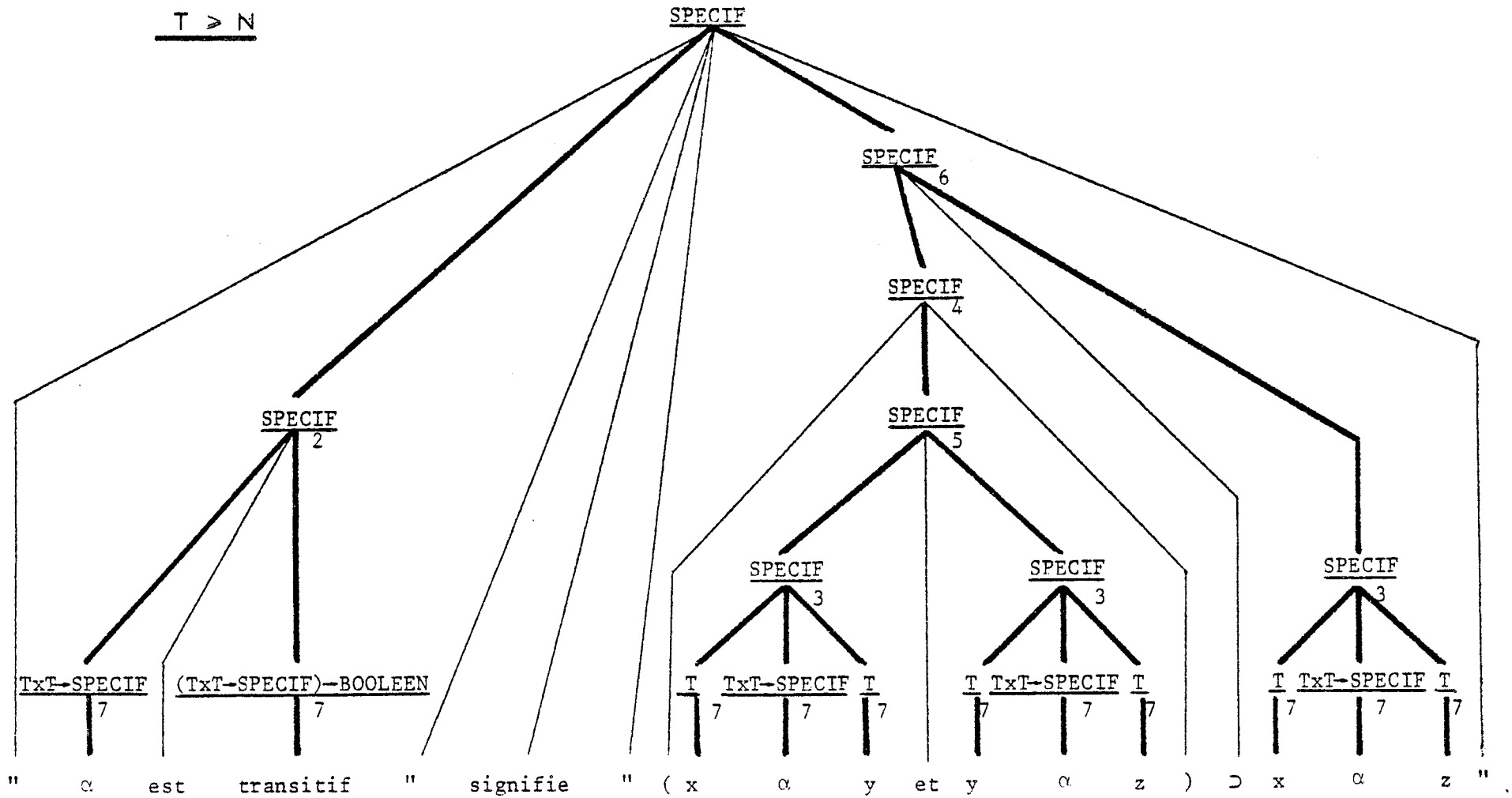
L'inégalité se réécrit:

$$(T_3 \times T_3 \rightarrow SPECIF) \rightarrow BOOLEEN \geq (N \times N \rightarrow BOOLEEN) \rightarrow BOOLEEN$$

et comme SPECIF > BOOLEEN (voir "Etat du Langage") cette inégalité se simplifie en:

$$T_3 \geqslant N$$

Il en résulte que l'ensemble (infini) des interprétations conformes à l'état du langage est décrit par l'arbre ci-joint (figure 3 - interprétation complète) où $T_3$ a été remplacé par T.




Chaque noeud comporte le numero de la règle appliquée et le type produit.

Figure 3 - INTERPRETATION COMPLETE

d) Gestion des identifieurs

Même avec l'interprétation complète fournie par les phases A et B de l'analyse syntaxique, on ne peut pas savoir ce que "représente" exactement une phrase si on ne connait pas la *portée* exacte des identifieurs qui y apparaissent. Il faut d'abord savoir si un identifieur désigne une variable locale ou une variable globale, c'est-à-dire pouvant être utilisée dans d'autres phrases.

La distinction local-global n'est pas facile à faire et peut nécessiter des questions réponses:

- toute constante est globale,
- toute variable présente dans la liste des identifieurs-typés (Etat du Langage) est globale,
- toute variable quantifiée est locale,
- toute variable qui *sera* utilisée dans une autre phrase est globale, notamment les données qui apparaissent dans la spécification initiale du problème,
- une règle peut spécifier si un objet doit être global (ex.: règle 2 de l'exemple),
- dans les autres cas, une variable est locale s'il y a eu *élision* de quantifieur.

α) Règles d'élision de quantifieurs

L'élision du quantifieur ∀ est une pratique très courante, mal définie et source de nombreux malentendus.
Dans l'exemple donné, il est "clair" que α, x, y, z sont des variables locales et que les préfixes ∀ ont été oubliés car ils"vont sans dire".

L'observation de textes mathématiques permet de voir que l'élision du quantifieur obéit presque toujours à la règle suivante qui sera donc adoptée en première approximation comme unique *règle d'élision.*

> *Si la phrase est sous forme d'implication ou d'équivalence (P⊃Q, P≡Q, P signifie Q) toute variable nouvelle est locale*

Ainsi, si x, y, z sont nouveaux, H(x,y)⊃C(x,z) signifie:
∀x y z [H(x,y) ⊃ C(x,z)]

β) Variables globales: inférence de types

Toute nouvelle variable globale donne lieu à l'adjonction de l'identifieur correspondant à l'ensemble des identifieurs typés.

De plus, pour chaque variable globale de la phrase, il faut mettre à jour son type,(Borne inf, Borne sup) dans l'ensemble des identifieurs typés. En effet, l'analyse syntaxique a apporté un complément d'information sur les types de ces variables.

Pour reprendre le même exemple, l'objet "transitif" était global et on savait seulement au départ que son type était *borné inférieurement* par (NxN→BOOLEEN)→BOOLEN, comme l'indiquait l'Etat du Langage.
On sait maintenant que l'ensemble des types de "transitif" compatibles avec l'interprétation complète est

$$\{(T \times T \to SPECIF) \to BOOLEN \mid T \geqslant N\}$$

Donc le type minimum de "transitif" est certainement:

- borné inférieurement par (NxN→SPECIF)→BOOLEEN
- borné supérieurement par λT(TxT→SPECIF)→BOOLEEN

A l'issue de l'interprétation, il convient donc d'effectuer une mise à jour de l'état du langage en améliorant les bornes du type minimum de chaque variable globale.

Comme il s'agit à proprement parler, d'une *extension* du langage, ceci sera rappelé dans la Section III, extensions, pages 80-81.

Enfin, le système doit pouvoir gérer des contextes bien imbriqués (variables définies sur un ensemble de phrases constituant un bloc) en regroupant les phrases en blocs auxquels sont attachés leurs propres identifieurs typés, inaccessibles de l'extérieur (cf. ALGOL).

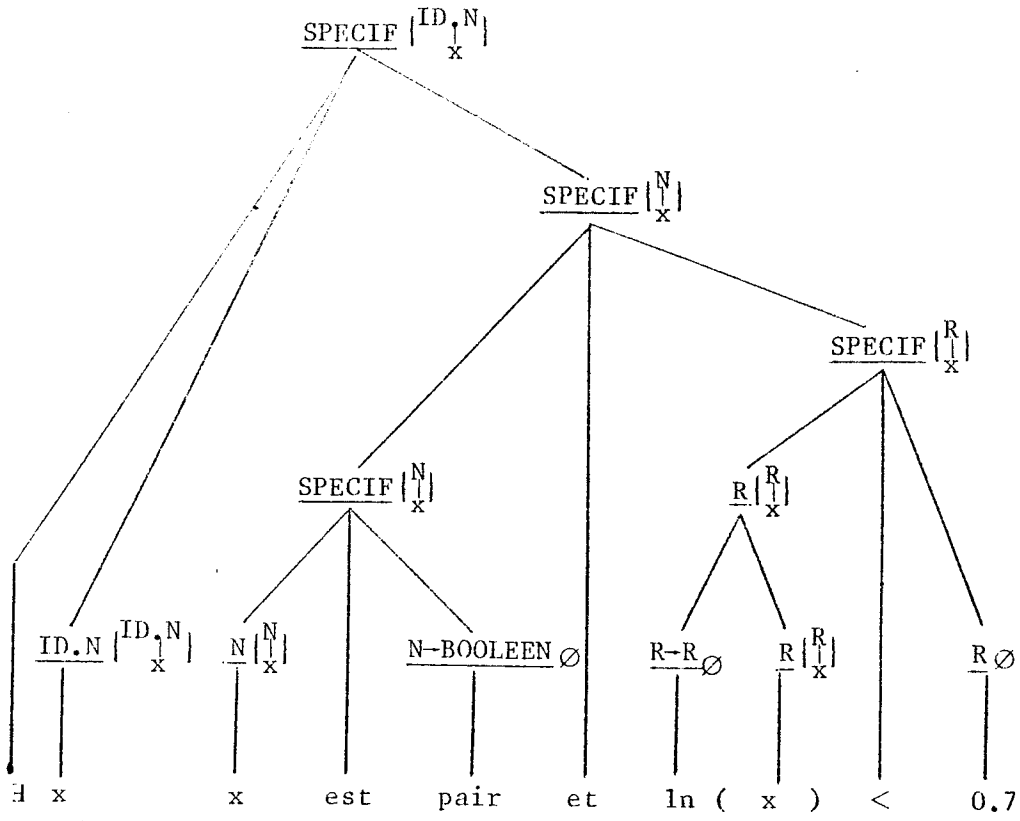
γ/ Variables locales, marquage des arbres

Pour des raisons qui apparaitront clairement dans la partie "Sémantique déductive" (chapitre III), il est avantageux d'associer à chaque noeud de l'arbre l'ensemble des variables locales de la phrase qui apparaissent dans le sous-arbre correspondant.

(Ce marquage indique exactement "de quoi dépend" chaque sous-arbre et sera indispensable à la définition des substitutions, chapitre III).

On indique de plus, pour chaque variable marquant le noeud, le type minimum qui lui est affecté dans tout le sous-arbre correspondant.

Exemple:



## 4. Résultats de l'analyse syntaxique (Phases A et B)

Huit cas peuvent se présenter.

| | | Nombre total d'interprétations | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | >1 toutes sur le même arbre | >1 réparties sur plus d'un arbre |
| nombre d'arbres (phase A) | 0 | (1) | | | |
| | 1 | (2) | (3) | (4) | |
| | >1 | (5) | (6) | (7) | (8) |

Cas 1 - Echec complet dès la phase A

Cette situation déjà mentionnée précédemment, implique l'identification d'une erreur de syntaxe ou de règle(s) nouvelle(s) employée(s) par l'utilisateur (extension de l'état du langage; ensemble des règles, cf. section III).

Cas 2 - Echec de la phase B: aucune interprétation

Ceci indique que l'ensemble des contraintes de types n'a pas de solution. Donc, s'il n'y a pas erreur de syntaxe, cette situation implique *l'élargissement* de règle(s), c'est-à-dire remplacement de règle(s) <C,P,E> par <C',P,E> avec condition(s) moins restrictive(s): $C \supset C'$.

(Extension de l'état du langage: ensemble des règles, cf. section III).

Cas 3 - Aucune ambiguité: analyse définitive.

Cas 4 - Ambiguité d'interprétation

C'est le résultat normal spécifié par l'interprétation complète (qui est une forme close de l'ensemble des interprétations valides) déterminée par la phase B.

Cette ambiguité résulte de l'indétermination sur les types des identifieurs. Lorsqu'un identifieur voit son type précisé (notamment par le processus d'inférence automatique de type déjà mentionné), les phrases qui le contiennent deviennent en général moins ambigües (l'ensemble des interprétations diminue).

Au cours d'une "session" l'ambiguïté a donc tendance à diminuer automatiquement, par le simple fait que de nouvelles phrases (impliquant les identifieurs dans de nouveaux contextes) diminuent l'ambiguïté des anciennes.

L'ambiguïté d'interprétation n'est pas a priori un défaut (elle économise même beaucoup de dialogue inutile ) mais on verra dans le Chapitre III (Sémantique) qu'elle peut perturber le processus de déduction. Ce cas fréquent nécessitera une réduction de l'ambiguïté par l'une au moins des deux techniques suivantes:

- amélioration des bornes du type d'un identifieur,
- réduction du champ d'application (condition C) d'une règle opératoire.

(Extension de l'état du langage: ensemble des identifieurs, ensemble des règles, cf. Section III).

Cas 5 Il y a à la fois *ambiguité fondamentale* déjà mentionnée, et défaut d'interprétation comme au cas (2).

(Extension de l'état du langage: ensemble des règles, cf. Section III).

Cas 6 Analogue au cas (3).

Cas 7 Analogue au cas (4).

Cas 8

a/ ou bien les arbres sont superposables (noeuds et arcs) et ne diffèrent que par des numéros de règles attachés aux noeuds.

Les règles associées à un même noeud doivent normalement être fusionnées, leurs syntaxes étant identiques sauf pour ce qui concerne les conditions sur les types.
(Extension de l'état du langage: fusion de règles, cf. Section III).

b/ ou bien les arbres ne sont pas superposables: il y a *ambiguité fondamentale* à résoudre.

(Extension de l'état du langage: ensemble des règles, cf.Section III).

On voit que les cas 1, 2, 5, 8 sont anormaux et ne peuvent être résolus que par recours au processus d'extension, qui est donc essentiel à la bonne marche de l'analyse syntaxique.

Ce sont maintenant ces mécanismes d'extension qui vont être étudiés.

# III - EXTENSIONS - INFERENCE - APPRENTISSAGE

Nous avons vu à plusieurs reprises que l'analyseur syntaxique doit *"apprendre"* le langage de l'utilisateur sans que celui-ci ait à connaître la structure interne de son propre langage.

C'est à la suite de difficultés d'analyse syntaxique (ambiguïté fondamentale, absence d'interprétations, etc...) que l'analyseur doit poser les questions appropriées pour provoquer des réponses, qui - par *inférence* - permettent la mise à jour de l'état du langage.

Nous verrons aussi(Chapitre III) que les nécessités de la déduction peuvent obliger à préciser les interprétations donc à restreindre les types des identifieurs et/ou les conditions des règles opératoires.

## 1. Localisation des difficultés syntaxiques

En présence d'une difficulté syntaxique, et pour choisir au mieux les questions qui permettent de la résoudre, l'analyseur doit tout d'abord trouver une explication aussi précise que possible de cette difficulté, que nous appelons *"localisation"*.

Cette localisation peut consister en

- une partie de la phrase correspondant à un sous-arbre,
- plus précisément, des parties d'arbres ("fenêtres").

Les principaux types de difficultés se décomposent en:

a/ absence d'arbre syntaxique

C'est l'échec complet de la phase A (correspondant au premier des

huit résultats possibles de l'analyse syntaxique).

Le problème de la localisation est alors identique à celui de la récupération d'erreur (finesse de diagnostic) dans l'analyse syntaxique des langages context-free ; la seule différence étant qu'ici on considère que l'erreur peut se trouver aussi bien dans la grammaire (nécessitant une extension) que dans la phrase.

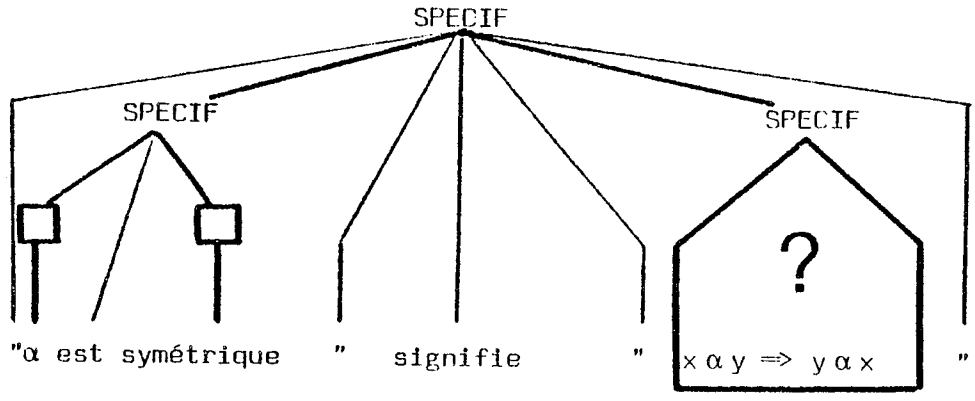
Une double analyse syntaxique (de gauche à droite et droite à gauche) permet d'isoler une *partie de phrase* pour laquelle l'analyse a échoué ; l' "erreur" est sensée s'y trouver.
Si on tente d'analyser cette sous-phrase en mode *"Bottom-up"*, ceci échouera nécessairement, mais permettra de délimiter dans l'arbre syntaxique une *"fenêtre d'incertitude"* qui fournit un diagnostic plus précis que la seule sous-phrase.

Exemple:

La phrase ci-dessous est incorrecte, le symbole "⊃" ayant été remplacé par "⇒":

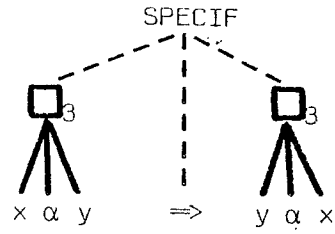
"α est symétrique" signifie "x α y ⇒ y α x"

Le résultat de l'analyse Top-down sera:

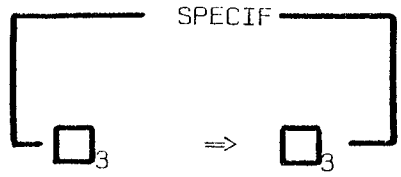


qui localise l'erreur dans $\boxed{x\,\alpha\,y \Rightarrow y\,\alpha\,x}$

Mais,en tentant sur cette sous-phrase l'analyse Bottom-up on trouvera:




qui localise l'erreur dans la fenêtre:




b/ ambiguïté fondamentale

On a au moins deux arbres interprétables (Cas 8).
Là encore, on peut localiser l'ambiguité, soit par une sous-phrase, soit par des fenêtres.
Il suffit en effet de définir le désaccord entre deux arbres:

α/ *désaccord terminal*

Soient A1 et A2 deux arbres syntaxiques pour la même phrase
Le désaccord terminal est la plus petite sous-phrase w telle que:

. w dérive d'un sous-arbre B1 de A1
. w dérive d'un sous-arbre B2 de A2
. A1 amputé de B1 ≡ A2 amputé de B2

On généralise facilement cette définition à plusieurs arbres.

Un algorithme très simple (parcours récursif des arbres en parallèle) permet de le calculer.
Ce désaccord terminal localise l'ambiguïté sous forme de sous-phrase.

*ß/ désaccord précis*

Soit w le désaccord terminal dérivant de B1 dans A1 et de B2 dans A2 (voir définition α);
le désaccord précis de A1 et A2 est un couple d'arbres C1, C2 tels que:

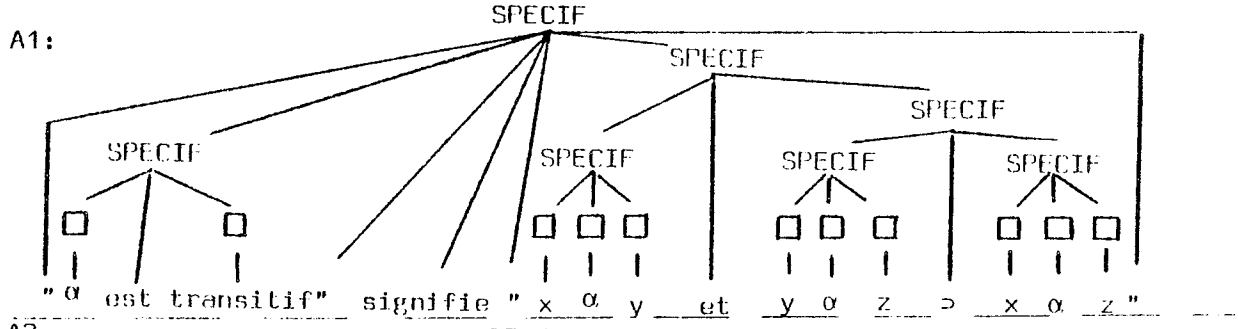
. C1 est un radical de B1
. C2 est un radical de B2
. en remplaçant C1 par C2 dans B1 on obtient B2 (ce qui implique la réciproque)
. C1 et C2 sont les arbres minimaux ayant ces propriétés.

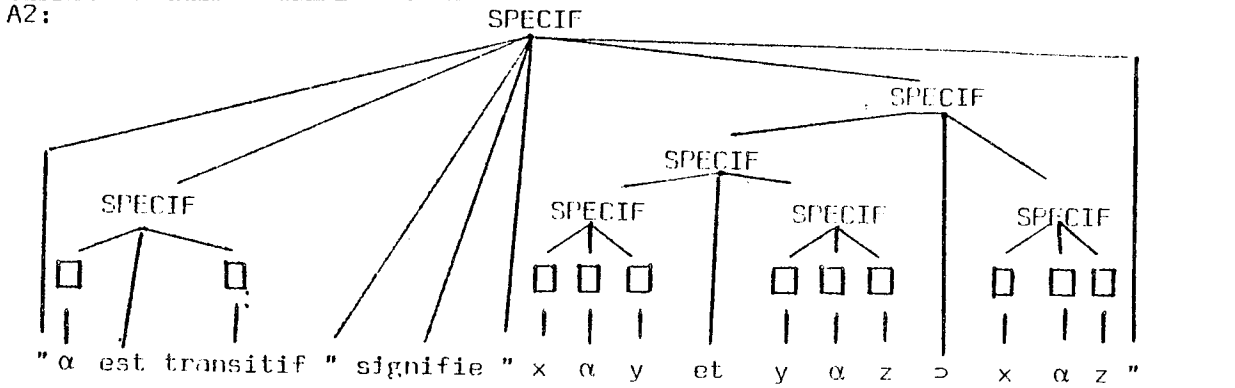
On généralise facilement cette définition à plusieurs arbres.

Un algorithme assez simple (parcours BOTTOM-UP de B1, B2...) permet de déterminer le désaccord précis C1, C2...

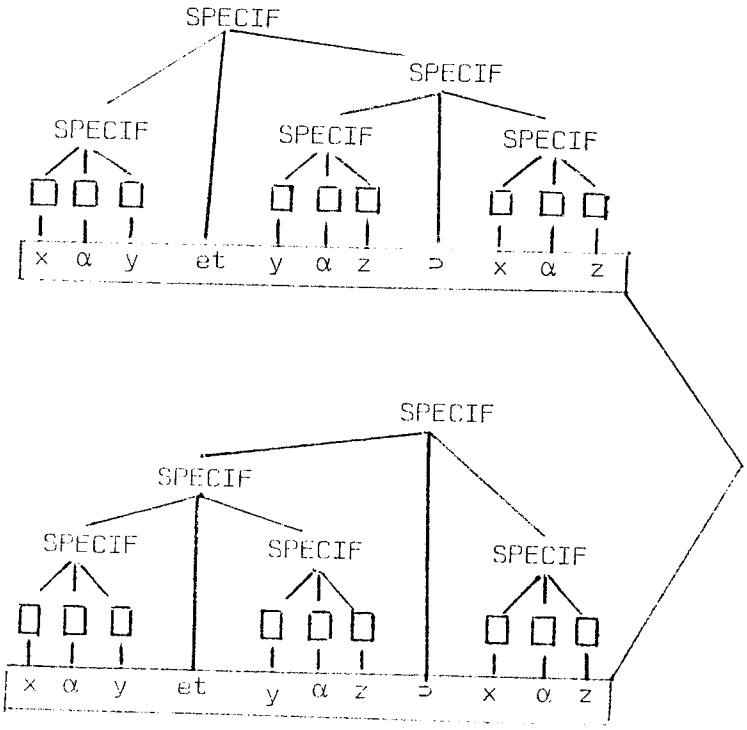
Ce désaccord précis localise l'ambiguïté sous forme de fenêtres interchangeables.

Exemple:
Il suffit de reprendre l'exemple illustratif de la Phase A de l'analyse où les désaccords (terminal et précis) étaient indiqués sous les noms respectifs de localisations grossière et précise:
On avait les deux arbres syntaxiques:



Ces arbres diffèrent par les sous-arbres respectifs:

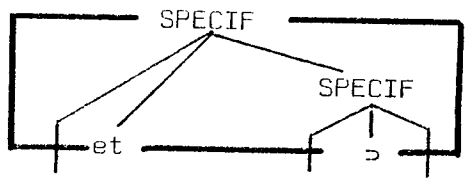



w = x α y et y α z ⊃ x α z

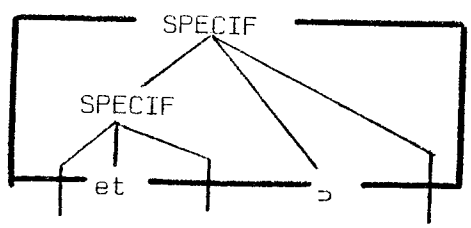
(localisation par sous-phrase)

Ces sous-arbres diffèrent par les racines respectives:

C1:




C2:




localisation par couple de fenêtres interchangeables

c/ absence d'interprétation

Le ou les arbres issus de la phase A n'ont aucune interprétation valide (cas 2 et 5). On peut en voir un exemple, page 78.

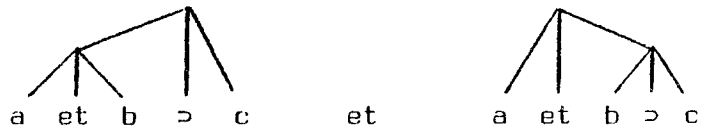
L'ensemble des contraintes de types n'a pas de solution. On peut donc tenter une *localisation heuristique* en cherchant une contrainte dont la suppression permettrait de trouver au moins une interprétation. La règle associée et le(s) noeud(s) correspondant(s) de l'arbre seraient les premiers suspects de la difficulté d'analyse.

## 2. Questions-réponses: Principe de discrimination

Ayant obtenu une localisation d'une difficulté de syntaxe, l'analyseur doit poser des questions dont les réponses lui permettront de déterminer parmi toutes les explications possibles (y compris la véritable erreur de syntaxe) quelle est la bonne.

L'utilisateur remarquera son éventuelle erreur de syntaxe d'autant plus facilement que la localisation aura été plus précise.
Hormis ce cas, une extension du langage est nécessaire pour accepter la phrase.

a/ Si la difficulté est une ambiguité fondamentale:
la question consiste à soumettre un *exemple discriminateur*.
Par exemple, dans le choix entre:




plutôt que de poser une question comportant le terme technique de "précédence" que l'utilisateur ne connait peut-être pas, il faudrait lui demander d'évaluer l'expression

? FAUX et FAUX ⊃ VRAI ?

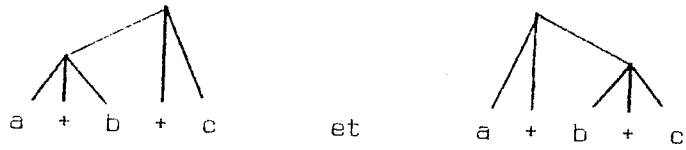
S'il répond "OUI", le premier arbre est le bon, sinon c'est l'autre.
L'analyseur peut en déduire la loi de précédence entre et et ⊃.
Ceci suppose:

1° Que l'analyseur sache évaluer les expressions du langage donc qu'on puisse associer aux règles opératoires une exécution de l'opérateur correspondant (intersection avec la sémantique).

2° Que l'analyseur sache trouver un exemple discriminateur. Ce sera en général facile, par tirage au hasard en un petit nombre d'essais.

3° Qu'un exemple discriminateur existe, c'est-à-dire que les arbres ne soient pas équivalents.

Par exemple:




n'ont pas d'exemple discriminateur.

(Faute de trouver rapidement un tel exemple, l'analyseur demandera explicitement si (a+b)+c et a+(b+c) sont équivalents, et dans l'affirmative, ou bien il fera un choix définitif entre les deux arbres (information de précédence ajoutée à la règle opératoire), ou bien il transformera la règle binaire $\underline{R} ::= \underline{R}+\underline{R}$ en une règle d'arité variable $\underline{R} ::= \underline{R}+\underline{R}(+\underline{R})^*$ traduisant l'associativité de "+".

4° Que l'ambiguité soit assez précisément localisée (ce que l'on sait faire sinon la découverte d'un exemple discriminateur et surtout son calcul par l'utilisateur seront plus longs.

b/ Autres difficultés (pas d'arbre ou pas d'interprétation):

La localisation concerne un petit nombre de règles et d'identifieurs. La question normale est de demander à l'utilisateur de définir (par une autre phrase) la sous-phrase concernée ou (si la localisation est très précise) les règles et identifieurs en question (cf. §3 ci-dessous).

## 3. Mécanismes d'extension

Puisqu'une extension consiste à faire évoluer l'état du langage défini par trois ensembles (types, règles, identifieurs) nous allons étudier séparément les mises à jour de ces trois ensembles par l'analyseur syntaxique.

### 3.1. extension des types

On a vu (chapitre I) que l'ensemble partiellement ordonné des types est défini par

- un ensemble partiellement ordonné de types de base,
- un ensemble de constructeurs de types.

Introduction d'un type de base

L'utilisateur sera parfois amené, soit spontanément, soit sur demande du système à indiquer le type de tel ou tel identifieur.

Rappelons que, pour nous, un type est lié à un domaine (=propriété) remarquable. Il suffit donc de savoir quel est le dispositif linguistique qui permet de distinguer une propriété remarquable.

En première approximation on peut remarquer qu'un type est normalement désigné par un substantif et une propriété par un adjectif:

"x est *un* nombre", "α est *une* relation" désignent des types ;
"x est pair", "α est transitif" désignent des propriétés ;
"x est un nombre premier" désigne à la fois le type nombre pour x et la propriété d'être premier.

Les mathématiciens font d'ailleurs constamment usage (sans le savoir) d'une figure de rhétorique qu'on peut appeler *"substantification"* pour exprimer cette distinction, ainsi:

"x est entier" signifie en général que x est un nombre réel qui se trouve avoir à ce moment là une valeur entière, tandisque:
"x est un entier" indique clairement que x ne peut prendre que des valeurs entières et est du type entier.

En utilisant ce critère, il suffit d'avoir une règle opératoire standard (donc faisant partie du *"NOYAU"* du langage)

SPECIF ::= NOM est un[e] NOM

et d'y associer le théorème standard (voir chapitre III: Sémantique) qui pour toute phrase "x est un[e] y":

- insère,dans l'ensemble des types, le type y, s'il n'y est pas déjà,
- ajoute à l'ensemble des identifieurs typés l'entrée:

<x, borne inf = y, borne sup = y>

Introduction d'un ordre sur les types de base

Ce sera, le plus souvent, par suite de l'absence d'interprétation valide que l'utilisateur sera amené à spécifier un ordre.

Par exemple, s'il apparait un ENTIER là où il faut un REEL, il n'y aura pas d'interprétation valide tant qu'il ne sera pas indiqué dans l'état du langage que ENTIER < REEL.

Sur les types de base, l'ordre coïncide avec l'inclusion des domaines. Il faut donc, là aussi, inclure dans le NOYAU du langage la ou les phrases par lesquelles un utilisateur exprime normalement l'inclusion de domaines.

Par exemple on pourra mettre dans le NOYAU:

- la règle SPECIF ::= tout[e] NOM est un[e] NOM
- et le théorème associé qui, pour une phrase telle que:

"tout entier est un réel"

créera, s'ils n'existent déjà, les types ENTIER et REEL, et la relation d'ordre ENTIER < REEL, en vérifiant qu'elle n'introduit pas de contradiction sur l'ordre.

Constructeurs "interactifs"

Si le langage des types comporte des constructeurs interactifs (tels que la Nomination Récursive: Chapitre I, § II, 2.j), il faut,là encore, associer à chacun d'eux, dans le NOYAU du langage, un couple <règle opératoire, théorème associé> permettant à l'utilisateur d'introduire dans l'état du langage de nouveaux types composés.

Remarques

1. On peut arguer du fait que ce processus n'est qu'une forme déguisée de déclaration explicite.

   C'est essentiellement inexact pour les raisons suivantes:
   - pour la plupart des identifieurs, le type n'est pas mentionné par l'utilisateur, il est automatiquement déduit ou approximé par l'analyseur syntaxique en fonction des contextes où apparaissent ces identifieurs.
   - pour les nouveaux types, ceux-ci sont introduits le plus souvent à l'initiative de l'analyseur qui, dans l'impossibilité de trouver une interprétation valide, demande "qu'est ce que x?".
   - en répondant, l'utilisateur ne sait pas nécessairement qu'il "déclare" un type.

2. On a vu pour les extensions de types, et cela se reproduira pour les autres extensions, qu'elles sont spécifiées à l'intérieur du langage lui-même et non dans un quelconque *métalangage*.
   Ceci implique que la partie sémantique et la partie syntaxe ne peuvent pas être disjointes, puisque la sémantique d'une phrase peut inclure la mise à jour de la syntaxe du langage.

## 3.2. extension des règles opératoires

Les règles se composent de quatre parties:

- C: condition d'application (contrainte sur des types)
- P: partie gauche: type obtenu
- E: partie droite: expression
- autres informations: précédence etc...

On considérera donc les huit sortes d'extensions suivantes:

a/ ajouter une nouvelle règle

b/ fusionner deux règles <C,P,E>, <C',P,E> → <C ou C',P,E>

c/ scinder une règle <C,P,E> → <C',P,E>, <C-C',P,E>

d/ élargir une règle <C,P,E> → <C',P,E> avec C ⊃ C'

e/ restreindre une règle <C,P,E> → <C',P,E> avec C'⊃ C

f/ dériver une règle <C,P,E> → ρ(C,P,E)

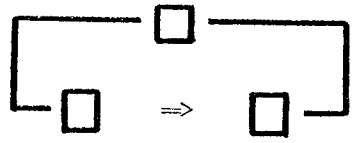
g/ diversifier la syntaxe <C,P,E> → <C,P,E'>

h/ mettre à jour des informations annexes (précédence...)

Ces modifications seront principalement dues à cinq causes:

1 - Résolution de l'absence d'arbre syntaxique

Il faut *ajouter une règle* (extension a) qui permette d'analyser la partie de l'arbre où a été localisée la difficulté.
En s'inspirant de l'exemple page 66/67, si la difficulté est localisée dans une fenêtre:




l'analyseur demandera simplement:

*que signifie u ⇒ v ?*

Si l'utilisateur répond: $u \supset v$

Il sera généré une phrase:

*"u ⇒ v" signifie "u ⊃ v"*

dont l'analyse complète indiquera que u et v sont du type SPECIF ainsi que u ⇒ v ce qui permettra l'addition de la règle:

SPECIF ::= SPECIF ⇒ SPECIF

(ainsi que d'un théorème: u ⇒ v ≡ u ⊃ v)

Il s'agit là d'une technique très efficace permettant de trouver un triplet <C,P,E> suffisant en une seule question (à condition bien sûr que la réponse de l'utilisateur soit analysable).

2 - Résolution de l'ambiguité fondamentale

La résolution de l'ambiguité impliquera l'une des extensions b,g,h (fusion, diversification, information de sélection).

a/ Si la localisation précise de l'ambiguité se limite à un seul noeud ou à deux règles qui ne diffèrent que par les types et qui sont identiques par leurs symboles terminaux (séparateurs):

- Si les deux arbres sont corrects(ce qui se détermine par questions-réponses) il faut *fusionner les deux règles*, après les avoir réécrites pour faire coïncider les parties P et E.

*Exemple:*

| | C | P | E |
|---|---|---|---|
| Règle 1 | VRAI(T) | T | ::= T→T (T) |
| Règle 2 | VRAI(T,U) | U | ::= T→U(T) |

se réécrivent:

| | | | |
|---|---|---|---|
| Règle 1 | X=Z<br>Y=X→X | X | ::= Y (Z) |
| Règle 2 | Y=Z→X | X | ::= Y (Z) |

et se fusionnent en:

| | | |
|---|---|---|
| Y=Z→X | X | ::= Y (Z) |

qui se simplifie en:

| | | |
|---|---|---|
| VRAI(X,Z) | X | ::= Z→X (Z) |

(Dans ce cas la fusion s'est terminée en absorption de la règle 1 par la règle 2).

- Si un seul des deux arbres est correct,il faut *diversifier la syntaxe* de la règle associée à l'autre arbre, en demandant à l'utilisateur de varier les symboles terminaux.

b/ Si la localisation précise de l'ambiguité entre deux arbres ne concerne que deux noeuds, comme dans les exemples page 48 , les questions-réponses permettent de déterminer une *précédence* entre les deux règles correspondantes qui doit être ajoutée à l'information associée aux règles.

c/ Si la localisation de l'ambiguité est plus étendue, normalement un seul des deux arbres est correct (à déterminer par questions-réponses) et au moins l'une des règles impliquées dans la localisation doit faire l'objet d'une *diversification de syntaxe* (en demandant à l'utilisateur de varier le choix, la position ou le nombre de symboles terminaux).

3/ Résolution de l'absence d'interprétation

En ce qui concerne l'extension de règles, lorsque les contraintes de la phase B sont incompatibles (aucune interprétation) on peut tenter de résoudre cette difficulté par l'une des extensions (d,f) (règle élargie, règle dérivée) ou à défaut a) (addition de règle).

a/ abus de langage et figures de rhétorique

La principale cause d'absence d'interprétation est l'usage d'un *abus de langage*. Dans tout document scientifique les abus de langage sont indispensables dans l'énoncé des définitions, théorèmes, preuves et raisonnements, faute de quoi ceux-ci seraient tout à fait illisibles. Ces abus consistent en l'usage de *figures de rhétorique* syntaxiques (ellipse, syllepse, inversion, hypallage, anaphore, synecdoque, métonymie). Toutes consistent à remplacer certaines règles opératoires en *règles dérivées* (extension f). Les quatre premières, assez peu employées, peuvent rendre l'analyse syntaxique plus complexe (introduction d'ambiguités), la dernière (métonymie) est d'un emploi très fréquent et consiste à dériver une règle identique sauf pour la partie C (Contrainte sur les types).

*Exemple de métonymie:*

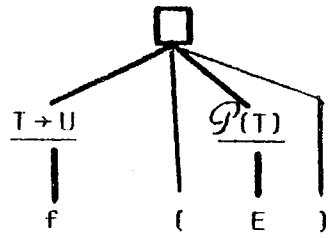
Soient f une fonction de type $R \to N$

x une variable de type R

E un ensemble de type $\mathcal{P}(R)$

l'expression f(x) est valide et de type N,

mais si l'expression f(E) apparait dans une phrase, celle-ci n'aura pas d'interprétation, et la localisation indiquera le sous-arbre:




comme "fautif"

Si, questionné sur la signification de f(E), l'utilisateur répond: $\{f(x) \mid x \in E\}$

(en supposant l'existence des règles:

$$\begin{cases} - \text{ VT } \underline{\text{SPECIF}} ::= \underline{T} \in \underline{\mathcal{P}(T)} \\ - \text{ VT } \underline{\mathcal{P}(T)} ::= \{\underline{T} \mid \underline{\text{SPECIF}}\} \end{cases} )$$

on en déduira la règle appliquée:

$$\underline{\mathcal{P}(N)} ::= \underline{R \to N} \ (\underline{\mathcal{P}(R)} )$$

et on pourra procéder:

- soit à *l'addition* pure et simple de cette règle (extension a) )
- soit à *l'élargissement* de la règle initiale (extension d) par fusion des règles $\underline{N} = \underline{R \to N(R)}$ et $\underline{\mathcal{P}(N)} ::= \underline{R \to N} \ (\underline{\mathcal{P}(R)} )$.
- soit à une *"extension métonymique"* qui, à toute règle de la forme C(T,U) $\underline{U} ::= \underline{T \to U(T)}$ associe la *règle dérivée:*

$$C(T,U) \underline{\mathcal{P}(U)} ::= \underline{T \to U} \ (\underline{\mathcal{P}(T)} )$$

Cette dernière solution est préférable, car elle formalise, en le généralisant un type d'abus de langage qui se répétera éventuellement dans d'autres phrases.

b/ règle manquante

Il se peut aussi que l'absence d'interprétation soit due à un sous-arbre incorrect. Ce cas se traite comme l'absence d'arbre syntaxique, avec la même localisation et donne lieu à une addition de règle (extension a).

4/ Résolution de difficultés de déduction

On verra (Chapitre III: Sémantique) que les difficultés de déduction (échec de déduction, ou déduction abusive) sont dues à un ensemble d'interprétations trop vaste pour une phrase. Il faut donc réduire l'interprétation complète de cette phrase. Si la localisation est réduite à un identifieur, on doit préciser le type de l'identifieur (voir page 80), sinon il faut *restreindre* ou *scinder* une règle (extensions c,e).

5/ Conversion de séparateur en identifieur de constante

(Voir page 80).

Le séparateur est considéré comme le symbole d'un opérateur exécutant la règle opératoire où il se trouve.

On remplace cette règle <C,P,E> par une règle <C,P,E'> où E' est obtenu en remplaçant dans E le séparateur par le type de l'opérateur qu'il représente

*Exemple:*

$$\underline{R} \ ::= \ \text{MAX} \ (\underline{R}, \ \underline{R}, \ \underline{R}) \ :$$

si le séparateur MAX devient un identifieur, la règle devient:

$$\underline{R} \ ::= \ \underline{R^3 \rightarrow R} \ (\underline{R}, \ \underline{R}, \ \underline{R})$$

Cette nouvelle règle peut donner lieu à fusion avec (ou absorption par) une autre.

## 3.3. extension des identifieurs typés

L'ensemble des identifieurs typés est constitué de triplets

<nom, borne inf, borne sup>

associés à chaque variable globale. L'extension consiste soit à introduire une nouvelle variable globale, soit à améliorer les bornes sur le type d'une variable pour diminuer l'ambiguité de type(réduction de l'incertitude sur le type)

a/ introduction d'identifieur

Toute constante ou variable globale apparaissant dans une phrase est introduite.

Le cas particulier est celui d'un caractère considéré comme séparateur qui devient un identifieur.
*Exemple:* + dans la règle R ::= R + R est un séparateur qui, dans la phrase: "+ est commutatif"devient un identifieur. Dans ce cas on le considère automatiquement comme un opérateur de type *au moins* RxR→R, et la règle est modifiée en R ::= R RxR→R R (voir § 3.2.5. précédent).

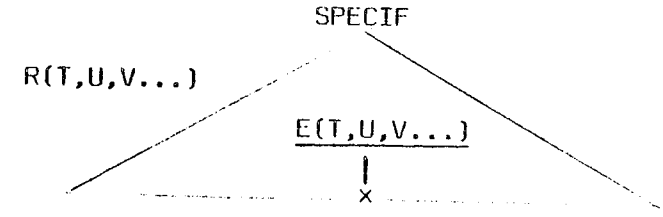
Si rien n'est connu sur le type de l'identifieur, on introduit par défaut <nom, Ø, λT.T>
Les modifications de ces bornes se font de trois façons:

b/ modification par déclaration

On a déjà vu que l'effet de la phrase: "x est un entier" est d'introduire <x, ENTIER, ENTIER>

c/ modification par inférence syntaxique

Si on a l'identifieur <x, borne inf = $I_x$, borne sup = $S_x$>
et si l'interprétation complète d'une phrase est de la forme:



α/ *amélioration de la borne inférieure* $I_x$

Une interprétation de cette phrase n'est syntaxiquement correcte que si elle attribue à x un type pris dans l'ensemble:

$$\mathcal{E} = \{E(T,U,V) \mid R(T,U,V)\}$$

Soit I la borne inférieure de cet ensemble.

Tout type de x doit donc être borné inférieurement par $I_{\mathcal{E}}$
La borne inférieure s'élève donc à: $I_x \sup I_{\mathcal{E}}$ .

Mais $I_x$ est un minorant de $\mathcal{E}$, puisque toute interprétation de l'arbre doit respecter la contrainte $I_x \leq \underbrace{E(T,U,V...)}_{\text{type de x}}$

donc $I_x \leq I_{\mathcal{E}}$ et $I_x \sup I_{\mathcal{E}} = I_{\mathcal{E}}$

Par conséquent, la mise à jour de la borne inf. des types de x consiste à effectuer: $I_x \leftarrow I_{\mathcal{E}}$

β/ *amélioration de la borne supérieure* $S_x$

Soit S la borne supérieure de $\mathcal{E}$

si $S_{\mathcal{E}} \not\leq$ type de x, aucun type de $\mathcal{E}$ ne donne à la phrase une interprétation syntaxiquement correcte.

On peut donc effectuer la mise à jour de la borne sup du type minimum de x en la diminuant:

$$S_x \leftarrow S_x \inf S_{\mathcal{E}}$$

d/ modification par inférence déductive

C'est le cas, déjà mentionné (§3.2.4) où le type d'un identifieur doit être précisé pour réduire l'interprétation complète d'une phrase afin de résoudre une difficulté de déduction (chapitre III).

## 4. Conclusion sur l'extension

a/ Le but de l'extension est l'*apprentissage progressif*, par le système, du langage de l'utilisateur avec lequel il communique.

La difficulté de cet apprentissage est que l'utilisateur doit pouvoir "enseigner" le langage qu'il se crée sans en connaître la structure syntaxique.

Les quelques techniques d'extension proposées ici constituent un mécanisme d'apprentissage encore fruste mais qui ressemble déjà beaucoup à la façon dont, habituellement, on apprend, en même temps qu'une théorie, le langage qui sert à la décrire. En quelque sorte, le système se comporte comme un interlocuteur compétent (et patient).

b/ Le succès de ce procédé nécessite au moins:

- que les deux interlocuteurs disposent au départ d'un langage commun déjà suffisamment élaboré ; nous reviendrons plus loin sur ce langage *"NOYAU"*.

- que l'utilisateur fasse preuve d'une bonne consistance dans son expression (éviter de dire la même chose de trop de façons différentes), ce qui n'empêche pas les *figures de rhétorique*.
- qu'il accepte de réviser des éléments de son langage qui créent la confusion chez l'analyseur syntaxique.

c/ On remarquera aussi que la séparation traditionnelle totale entre syntaxe et sémantique n'a pas été respectée. Ceci est dû à l'absence de *métalangage* pour décrire la syntaxe: l'extension de la syntaxe est un résultat sémantique de phrases du langage lui-même.

Je pense d'ailleurs que tout langage dont l'implémentation respecterait une frontière entre syntaxe et sémantique est a priori impropre pour les applications avancées de l'"Intelligence" Artificielle.

d/ Enfin, on peut se demander si la représentation du langage telle que la constitue le système est un *modèle* fidèle du langage de l'utilisateur.

La réponse est négative: le système peut inférer une règle ou un type qui ne sont pas conformes à l'intention de l'utilisateur, et cela se traduira dans la phase Sémantique (chapitre III) par un désaccord entre les déductions du système et celles de l'utilisateur.

Le seul moyen définitif d'éviter une différence entre les langages des deux interlocuteurs serait d'exiger de l'utilisateur qu'il donne lui-même une définition formelle complète de son langage.

Cette solution fastidieuse est en contradiction avec la souplesse requise pour le langage de spécifications, mais surtout elle n'élimine pas le danger:

En effet, supposons le texte d'une certification, et soient

- $I_1$ son interprétation dans le langage du concepteur
- $I_2$ son interprétation dans le langage du système
- $I_3$ son interprétation dans le langage du "client".

La certification a été prouvée (interactivement) par le système donc elle est correcte sous l'interprétation $I_2$. Même si (à grands frais) on a assuré l'équivalence de $I_1$ et $I_2$ , rien ne prouve qu'elle soit correcte sous l'interprétation $I_3$ du client. Comme cela a été indiqué en introduction (page 11) la *crédibilité* d'une certification ne peut pas être égale à 1 et ne peut être améliorée que par une meilleure "clarté" du langage.

# IV - PERFORMANCES DE L'ANALYSE SYNTAXIQUE

Par rapport aux langages de programmation, ce langage de spécifications aura un coût d'analyse syntaxique inévitablement élevé en raison de deux particularités essentielles:

1° son *extensibilité* permanente (grammaire évolutive),
2° sa *tolérance* aux ambiguïtés (multiplicité des interprétations valides).

## 1. Conséquences de l'extensibilité

La Phase A (analyse syntaxique du langage context-free adjoint) est nécessairement coûteuse, car on ne peut pas utiliser les algorithmes LR usuels qui sont très efficaces (coût O(d) proportionnel à la longueur d de la phrase) ; en effet:

1ère raison: L'analyse LR (voir[ 4 ]) suppose que la grammaire ait été au préalable convertie de sa forme BNF en un automate à pile (que l'analyseur n'a plus qu'à parcourir). L'utilisation de cet algorithme LR nécessiterait donc qu'après chaque extension de règle, cet automate à pile soit mis à jour pour rester conforme à la grammaire ainsi étendue. Ne sachant actuellement pas effectuer cette mise à jour rapidement, on est obligé d'abandonner l'algorithme LR et d'adopter l'un des algorithmes à "backtrack" qui opèrent directement sur la représentation BNF de la grammaire (coût O(d) à $O(d^3)$ selon la grammaire).

2ème raison: On a vu que les difficultés syntaxiques devaient être *localisées* précisément pour permettre les extensions. Cette localisation, qui s'apparente à la phase de diagnostic d'erreur est très délicate si l'on utilise l'algorithme LR (voir [36,44,48]) alors qu'elle est assez simple avec les algorithmes *"Top-down"* et *"Bottom-up"* combinés (cf. chapitre III: extensions §1).

L'extensibilité a une autre conséquence défavorable sur le coût de l'analyse syntaxique: chaque phrase peut entraîner une extension du langage, et chaque extension peut impliquer que certaines phrases déjà analysées et conservées doivent être à nouveau analysées conformément à la grammaire étendue.

Donc, *au pire*, l'analyse d'une suite de n phrases peut entraîner un facteur de coût $O(n^2)$.

## 2. Conséquences de la tolérance aux ambiguités

La Phase B (interprétation) calcule l'ensemble des interprétations valides, et on a vu que ceci implique la résolution d'un système de contraintes (égalités et ordres) sur les types attachés aux noeuds. Pour une phrase de longueur d, il peut y avoir O(d) contraintes. L'algorithme qui détermine l'existence d'une solution et en donne une forme compacte prendra un temps O(d), avec un facteur constant qui dépend de la structure des types (cf. chapitre I) et peut être très élevé.

## 3. Améliorations possibles

L'analyse syntaxique d'une session de n phrases de longueur moyenne d pourra donc coûter dans les *cas extrêmes* un temps:

$kn^2d^3 + k'n^2d$ de la classe de complexité $O(n^2d^3)$

Il est difficile d'évaluer les coûts asymptotiques *moyens* et d'estimer des bornes supérieures des constantes k et k', qui dépendent d'un trop grand nombre de facteurs peu mesurables.

Néanmoins il faut tout faire pour diminuer le facteur k (primordial pour les longues sessions) et le facteur k' (primordial pour les courtes sessions).

1ère amélioration

Pour simplifier la présentation de l'analyse syntaxique, on a défini un *langage adjoint* extrêmement pauvre (un seul symbole non terminal) qui envoie en Phase B un grand nombre d'arbres non interprétables. Pour améliorer le rôle de *"filtre"* de la phase A (donc diminuer le facteur k') il faudrait enrichir le langage adjoint, par exemple en conservant tels quels dans les règles tous les types constants et en ne remplaçant par le symbole [] que les types variables (ceci diminuerait aussi le facteur k lié à la phase A, qui produirait moins d'arbres).

2ème amélioration

Dans la Phase B, au lieu de résoudre l'ensemble des contraintes à la fin, les résoudre au fur et à mesure de leur apparition.
Pour des arbres qui n'ont pas d'interprétation, on a des chances de s'en apercevoir plus tôt (amélioration du facteur k').

3ème amélioration

Fusionner la phase A et la phase B (suppression du langage adjoint).
Ceci permet d'abandonner un arbre en cours de construction dès que l'ensemble des contraintes qui s'y trouvent n'a pas de solution.
(Amélioration de k et k', algorithme plus compliqué.

4ème amélioration

Imposer que les constructeurs de types (cf. chapitre I) assurent non plus seulement un ordre partiel mais une structure de *treillis*.
Ceci facilitera nettement (si on sait calculer Sup et Inf rapidement)

- les résolutions de contraintes (facteur k' amélioré),
- l'inférence de types d'identifieurs (cf. Extensions, §3.3.c).

5ème amélioration

Le mécanisme d'extension sera plus efficace (moins de questions-réponses, donc n plus faible) si le NOYAU du langage est

- suffisant: au moins une phrase (règle) standard pour chacune des principales classes d'extensions ;
- judicieux: que ces phrases soient faciles à retenir et, à la limite, n'aient pas besoin d'être apprises par l'utilisateur.

# III

# SEMANTIQUE DEDUCTIVE

## RESUME DU CHAPITRE III - SEMANTIQUE DEDUCTIVE

*La sémantique du langage est définie par rapport au langage lui-même par quelques règles mécaniques de déduction qui permettent de dériver les conséquences d'un ensemble quelconque de phrases.*

*Pour justifier, réfuter ou corriger les transformations appliquées aux spécifications, il faut donc un démonstrateur de théorèmes fondé sur ces règles de déduction.*

*Après avoir passé en revue les critères de qualité des démonstrateurs de théorèmes, on montre l'importance prédominante du critère "autonomie" (profondeur moyenne des théorèmes prouvés avant un temps donné), et on propose des moyens originaux de l'améliorer.*

*En particulier, on énonce une conjecture (dite principe d'incertitude), selon laquelle on ne peut obtenir d'autonomie élevée qu'au prix d'un risque d'erreur. Ainsi, de même qu'on ne peut rapidement résoudre un problème de géométrie sans prendre le risque de raisonner sur quelques figures particulières, de même on propose de représenter chaque phrase par un "spectre" imparfait qui, par "résonance", permet de remarquer rapidement les théorèmes qui s'y appliquent le mieux.*

*On propose d'améliorer, puis d'implémenter ce genre de techniques qui semblent permettre de contourner au moins partiellement l'un des plus tenaces obstacles à l'Intelligence Artificielle.*

# CHAPITRE III

# SEMANTIQUE DEDUCTIVE

## I - NOTION DE SEMANTIQUE

### 1. Rappel

L'expression: *"sémantique du langage L"* désigne en général un procédé* qui, pour toute phrase du langage L, permet de répondre *oui* ou *non* à certaines questions exprimées dans un langage S.
La *"sémantique de L par rapport à S"* est donc une *relation* (éventuellement partielle) entre les ensembles de phrases L et S.

Par définition, ce langage S sert à exprimer la *signification* des phrases du langage L. Donc, très souvent, il existera:

- sur le langage S: une relation d'ordre partiel $\Rightarrow$ (implication)
- dans le langage S: une *négation* $\neg$ telle que, pour tout $s \in S$:
  - $\neg s \in S$
  - $\neg\neg s \Rightarrow s \Rightarrow \neg\neg s$ (équivalence)

Dans ce cas, pour que la sémantique $\sigma : L \times S \to \{VRAI, FAUX, \Omega\}$ soit *consistante* il faut que $\forall l \in L$:

- si $s_1 \Rightarrow s_2$ alors $\sigma(l,s_1) \supset \sigma(l,s_2)$**
- $\forall s \ ( \sigma(l,1) \supset \neg\sigma(l,\neg s) )$**

et, pour qu'elle soit *totale* (ou *complète*) il faut que:

$\forall l,s$ on ait $\sigma(l,s)$ ou $\sigma(l,\neg s)$, donc jamais $\Omega$.

---

* *"procédé"* est ici synonyme de *"calcul"*

** $\supset$ étant entièrement défini par $F \supset F \supset \Omega \supset \Omega \supset V \supset V$

$\neg$ étant entièrement défini par $\neg F \equiv V, \neg\Omega \equiv \Omega, \neg V \equiv F$

## 2. Hétéro-sémantique

Normalement, L et S sont des langages distincts:

a/ Pour des langages (L) de programmation (ou, plus généralement des langages décrivant des transformations, actions, mouvements, etc...), on choisit le plus souvent d'exprimer les significations (ou *"effets"*) dans un langage SxS dont les phrases sont des couples:

<prédicat avant, prédicat après>

Quand la sémantique $\sigma(l,<\phi,\psi>)$ est totale, elle est le plus souvent définie

- soit à l'aide d'une *fonction* $\sigma_+$ (calcul de *"postcondition"*) par la valeur VRAI/FAUX de : " $\sigma_+(l,\phi) \supset \psi$"
- soit à l'aide d'une *fonction* $\sigma_-$ (calcul de *"précondition"*) par la valeur VRAI/FAUX de "$\phi \supset \sigma_-(l,\psi)$"

Dans ces cas là, la sémantique est donc ramenée à une fonction qu'on peut considérer indifféremment comme étant du type $L \times S \to S$ ou $L \to (S \to S)$, faisant jouer au programme un *rôle de transformateur de prédicats*.

b/ Pour la *traduction* d'un langage $L_1$ dans un langage $L_2$ (dans le cas de langages de programmation, on parle - Dieu sait pourquoi - de *compilation*), les sémantiques doivent être compatibles, c'est-à-dire exprimées par rapport au même langage S.

Le traducteur t est correct si:

$$\forall l \in L_1 \ , \ \forall s \in S \qquad \sigma_1(l,s) \equiv \sigma_2(t(l),s)$$

Remarque:

Pour souligner le caractère arbitraire du choix d'une sémantique, remarquons qu'on peut "prouver" n'importe quel compilateur $L_1 \xrightarrow{t} L_2$ à peu de frais en définissant:

. la sémantique de $L_2$ par rapport à $L_2$ par la relation identité

. la sémantique de $L_1$ par rapport à $L_2$ par la relation:

$$\sigma_1(l_1, l_2) \iff l_2 = t(l_1)$$

. si $L_2$ n'exprime pas des prédicats ou des propriétés, il n'y a pas de relation d'implication $\Rightarrow$, ni de négation $\neg$ ; donc la question de la consistance des sémantiques ne se pose même pas.

## 3. Auto-sémantique

Malgré cette dernière remarque, il n'est pas anormal de définir la sémantique d'un langage L par rapport à lui-même.

a/ Pour les langages de programmation, cela revient à considérer qu'un programme est sa propre spécification (sémantique identité) ce qui est parfaitement acceptable s'il est *lisible* : voir par exemple PROLOG [12,50] et les langages de très haut niveau. Il reste que si ces programmes sont traduits de $L_1$ dans $L_2$, la preuve du traducteur reste à faire: avec une sémantique $\sigma$ de $L_2$ par rapport à $L_1$ , il faut vérifier que $\sigma_0 t$ = identité.

b/ Pour les langages de spécifications servant à décrire des théories et raisonnements, le meilleur critère de compréhension d'un ensemble de phrases est, comme pour les humains, *l'aptitude à en déduire d'autres phrases: un résumé, une conséquence logique, une contradiction, un contreexemple , une question etc...*

Dans ce cas, la sémantique correspondante consiste en un *procédé de déduction* qui, à tout *ensemble de phrases* du langage de spécification S, fait correspondre l'ensemble de ses conséquences exprimées aussi dans S.

C'est donc une relation $\sigma$: $S^* \times S \rightarrow \{VRAI, FAUX, \Omega\}$
(Dès qu'on spécifie une théorie incomplète, ex. l'arithmétique, la possibilité de $\Omega$ est indispensable).

## II - PRINCIPE DE LA SEMANTIQUE DEDUCTIVE

Si l'on appelle S le langage de spécifications, la sémantique de S est une relation entre

- $S^*$ : ensemble arbitraire de spécifications,
- S : le langage lui-même.

Avant de définir précisément cette relation entre $S^*$ et S, nous allons donner les principales indications concernant la représentation de S qui est donc une *"Base de spécifications"*.

### 1. Organisation de la base de spécifications

L'ensemble des connaissances du système est, à tout instant, constitué d'un certain nombre de spécifications qui sont regroupées en une base de données.

Nous allons voir successivement le contenu de la base et les opérations dont elle peut faire l'objet.

a/ Eléments de la base

Chaque élément est une phrase analysée par un arbre interprété et qui contient donc en général:

- une partie: contrainte de types, que nous symboliserons R
- un arbre syntaxique étiqueté par des types, que nous symboliserons A ;

le tout étant dans la suite désigné par $\boxed{R : A}$ .

b/ Opérations sur la base

Les principales opérations dont la base peut faire l'objet sont:

α/ Insertion d'élément par l'utilisateur

C'est une spécification d'axiome ou d'hypothèse.
La phrase analysée R : A est introduite dans la base.

β/ Insertion d'élément par déduction

Il s'agit d'un *lemme* ou *théorème*.
La règle de déduction décrite plus loin permet d'insérer un nouvel élément déduit d'un ou plusieurs anciens éléments. Il faut alors, non seulement insérer le nouvel élément, mais en plus indiquer de quels éléments il dépend *(Base de données avec relation de dépendance)*.

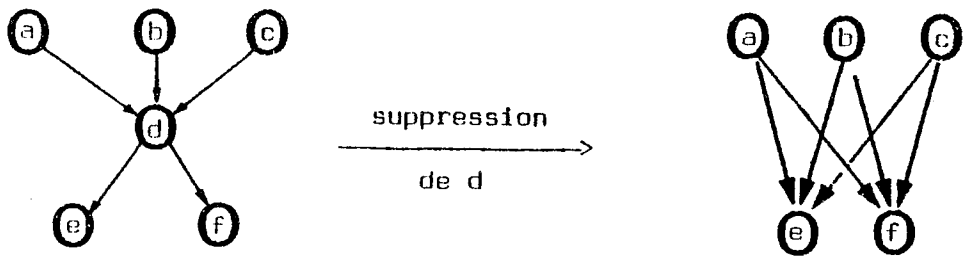
γ/ Suppression d'élément

Il y a deux cas bien distincts:

1er cas: l'élément à supprimer a été introduit par déduction ou, de façon équivalente, l'élément *dépend* d'au moins un autre élément.

Il faut le supprimer en mettant à jour la relation de dépendance par *transitivité*.

*exemple:*



2ème cas: l'élément à supprimer a été introduit directement ou, de façon équivalente , l'élément *ne dépend* d'aucun autre élément. C'est donc un axiome ou une hypothèse, et tout ce qui en dépend doit être supprimé.

δ/ Introduction d'une sous-base

Aucun problème, la sous-base est ajoutée telle quelle, avec ses dépendances internes, à la base.

ε/ Suppression d'une sous-base

Il faut distinguer deux types de suppressions.

1er cas: Négation du contenu
Les axiomes et hypothèses de la sous-base ne sont plus considérés valides. On supprime donc tout ce qui en dépend: toute la sous-base est considérée comme dépendant d'un axiome unique qui est supprimé.

2ème cas: Restitution de la sous-base
La sous-base était utilisée comme outil pour prouver des théorèmes, on n'en a plus besoin mais ses conséquences restent.
On supprime donc seulement le contenu de la base, et on fait dépendre ses conséquences d'un axiome unique attaché à la sous-base et qui, lui, reste.

Remarquons que dans les deux cas, toute sous-base peut être considérée comme dépendant d'un axiome unique qui exprime donc que cette sous-base est correcte.

c/ Représentation de la base

Le problème est de choisir une bonne représentation de la relation de dépendance. Celle-ci est une relation d'ordre partiel, donc transitive, donc susceptible de nombreuses représentations distinctes. Celle qui consiste à lier tous les éléments uniquement à leurs axiomes d'origine est la plus économique si l'on se limite aux opérations mentionnées, mais elle

détruit une partie de la relation de dépendance (par exemple entre lemmes et théorèmes) dont il ne reste plus trace. Si l'on introduit d'autres opérations (telles que reconstituer la trace de la preuve d'un élément) cette solution ne sera plus valable.

## 2. Règle de déduction

Dans la suite de ce chapitre, pour pouvoir exprimer les règles "mécaniques" de déduction qui constituent la sémantique du langage, on désignera les phrases au moyen de la *notation* ci-dessous.

### a/ Convention de notation et signification

Soit un élément (arbre interprété) R:A.
A est du type SPECIF et (voir page 61) marqué de la *liste* $\overline{x}$, éventuellement vide, de ses variables locales. Lorsque ce sera nécessaire pour la compréhension, cet arbre sera noté explicitement $A(\overline{x})$ et l'élément sera noté $R:A(\overline{x})$.

De la même manière, lorsqu'on voudra préciser qu'un *sous-arbre* B a sa racine marquée d'une liste de variables $\overline{y}$, on écrira $B(\overline{y})$.

$R:A(\overline{x})$ a précisément la *signification* suivante:

Si $R(T,U,V \ldots)$ est une contrainte sur les types libres T,U,V ...

et si $\overline{x}$ est la liste:

$$\begin{cases} x_1 \text{ de type } E_1(T,U,V \ldots) \\ \ldots \\ x_n \text{ de type } E_n(T,U,V \ldots) \end{cases}$$

alors $R:A(\overline{x})$ signifie:

$\forall T,U,V \ldots$ tels que $R(T,U,V \ldots)$,

$$\forall \begin{cases} e_1 & \text{expression de type } E_1(T,U,V,\ldots) \\ \cdot & \\ \cdot & \\ \cdot & \\ e_n & \text{expression de type } E_n(T,U,V,\ldots) \end{cases} \quad A(e_1,\ldots,e_n) \text{ est VRAI}$$

En particulier, si A est une *implication*, il est de la forme:

$$H(\overline{x}_1,\overline{x}_2) \supset C(\overline{x}_1,\overline{x}_3)$$

(où $\overline{x}_1$ est la liste des variables locales présentes à la fois dans H et C
$\overline{x}_2$ est la liste des variables locales présentes seulement dans H
$\overline{x}_3$ est la liste des variables locales présentes seulement dans C.)

Cette phrase signifie: $\forall\ \overline{e}_1\ \overline{e}_2\ \overline{e}_3 \quad H(\overline{e}_1,\overline{e}_2) \supset C(\overline{e}_1,\overline{e}_3)$

(où $\overline{e}_1,\overline{e}_2,\overline{e}_3$ sont des listes d'expressions *substituées* aux variables des listes $\overline{x}_1,\overline{x}_2,\overline{x}_3$).

Il faut remarquer, pour bien comprendre la suite, que cette formule équivaut à:

$$\forall e_1\ (\ \exists \overline{x}_2\ H(\overline{e}_1,\overline{x}_2) \supset \forall \overline{e}_3\ C(\overline{e}_1,\overline{e}_3)\ )$$

(Pour chaque liste $\overline{x}_1$, $\overline{x}_2$ ou $\overline{x}_3$ vide, le quantifieur correspondant est supprimé).

b/ Définition de la substitution

Comme dans tout système de déduction qui inclut la Logique du Premier Ordre, les substitutions jouent un rôle essentiel.

L'*objet* de la substitution est un [sous]-arbre $R:B(\overline{x})$
où, comme convenu, $\overline{x}$ est la liste des variables attachées à la racine de l'arbre B.

L'*opération* de substitution, notée ici $\sigma$, est le produit de l'une ou plusieurs des *quatre opérations* ci-dessous:

α/ changement de noms des types libres

(aussi bien dans R que dans B).
Il en résulte un élément équivalent.

*exemple:*

β/ renforcement de la contrainte R

C'est le remplacement de R:B par R':B tel que R' ⊃ R.
Ceci restreint l'ensemble des interprétations.
(Note: rien n'interdit de confier la preuve de R' ⊃ R au système lui-même).

γ/ substitution de variables par des sous-arbres

(voir l'*exemple* figure 4)

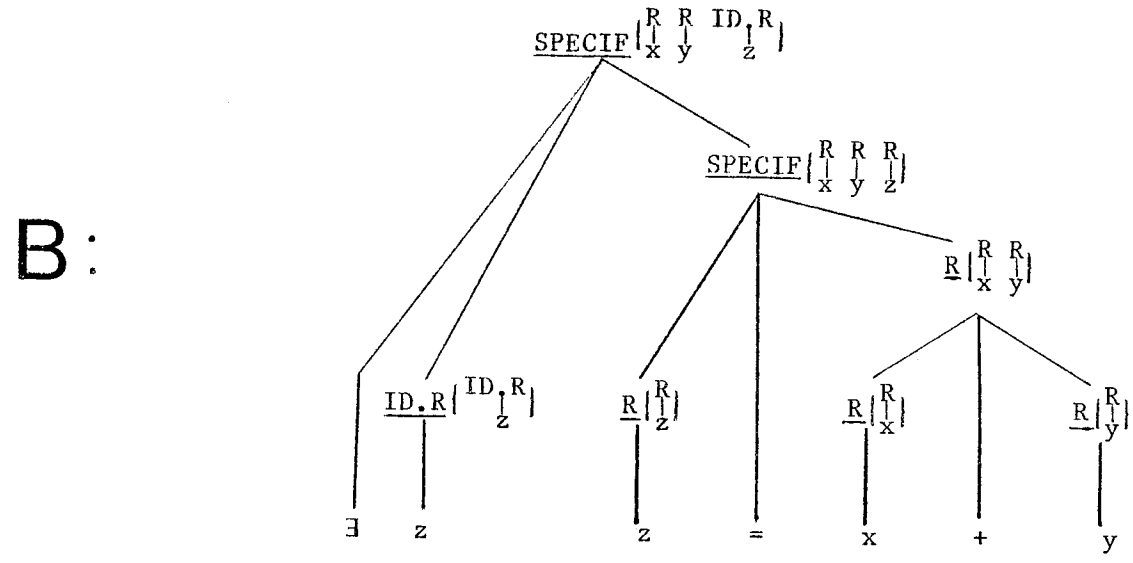
- parmi ces variables, seules sont remplacées celles qui figurent dans la liste $\bar{x}$ des variables attachées à B,
- les variables sont remplacées en parallèle, et non en séquence,
- une variable ne peut être remplacée que par un sous-arbre de même type (rappelons que la liste $\bar{x}$ indique pour chaque variable son type),
- on met à jour la liste des variables attachées à B comme suit:
  1 . chaque variable substituée est remplacée par la liste des variables locales (c'est-à-dire attachées à la racine) du sous-arbre substituant,
  2 . aucune variable n'apparaît plus d'une fois dans la liste.

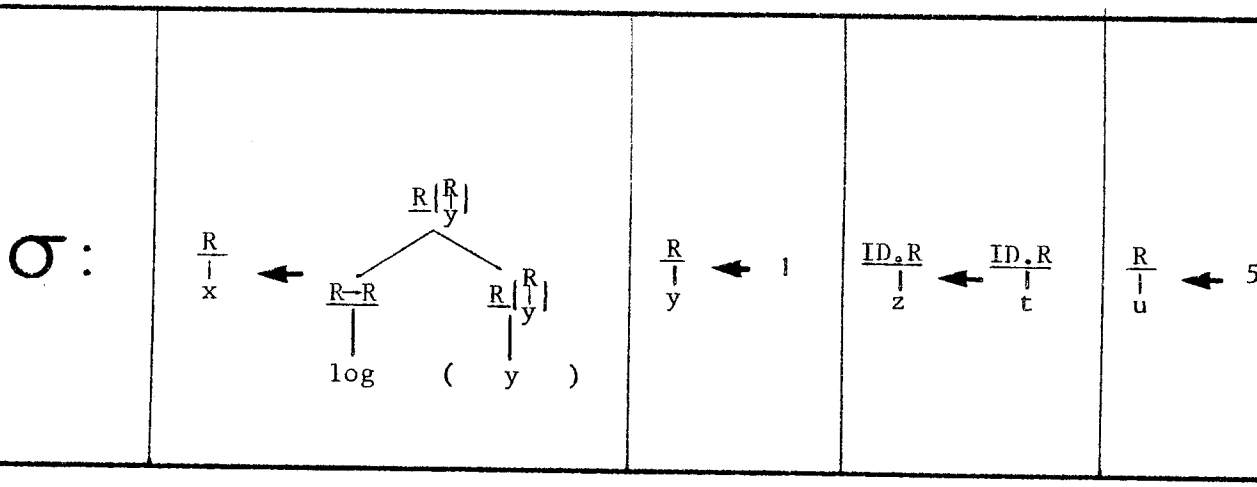
δ/ application d'une égalité

Soit X un sous-arbre de B.
Si l'on peut prouver R:X=Y, alors le remplacement de X par Y dans B est valide.

(Note: comme pour l'opération β, rien n'interdit de confier la preuve de R:X=Y au système lui-même).

Cette dernière règle de substitution est puissante et sera largement utilisée dans la suite.



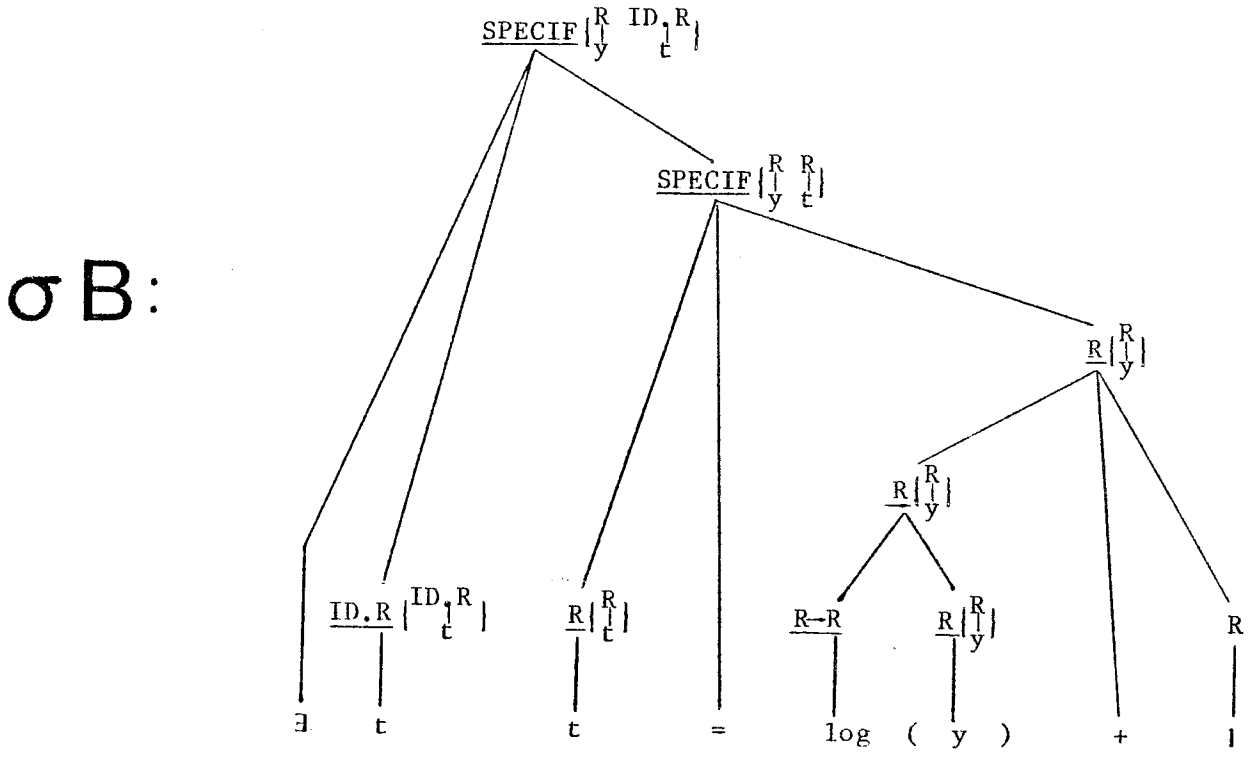




Figure 4 : EXEMPLE DE SUBSTITUTION DE VARIABLES

La *notation* d'une substitution σ sera indifféremment:

$$\boxed{\sigma(R : A)} \qquad \text{ou} \qquad \boxed{\sigma R : \sigma A}$$

La notation $R : \sigma A$ indiquera en plus que la substitution ne porte pas sur les types (opérations α et β exclues).

Enfin $\sigma_{\overline{X}}$ indiquera que la substitution est restreinte aux variables de $\overline{X}$.

c/ Théorème de substitution

- En se référant à la signification donnée page 95:

$$\forall\sigma \quad \boxed{R : A \quad \text{implique} \quad \sigma(R : A)}$$

- Autrement dit, *toute substitution préserve la validité.*
- On le montre facilement en traitant les quatre opérations séparément.
- La réciproque n'est pas vraie: "R:A implique R':A'" n' entraîne pas l'existence d'une substitution σ telle que (R':A') = σ(R,A), même si R=R'.

Par exemple: $\underbrace{x > 1}_{A}$ implique $\underbrace{x > 0}_{A'}$

et on ne peut passer de A à A' par une substitution.

d/ Règles d'inférence

On peut vérifier que les quatre règles d'inférence suivantes sont valides compte tenu de la définition des substitutions et de la significatio des phrases (page 95).

α/ emphase $\sigma$

$$\frac{R : A}{\sigma R : \sigma A}$$

C'est l'énoncé du théorème de substitution ci-dessus

β/ modus ponens(∿) $\mu$

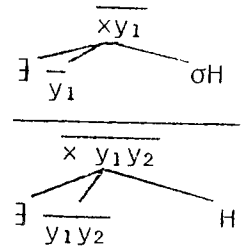
$$\frac{R : A \supset B \qquad \sigma R : \sigma A}{\sigma R : \sigma B}$$

Ceci se vérifie, pour chaque interprétation, par réduction à:*

$$\frac{\begin{array}{ll}\forall\ x\ y\ z & H(x,y) \supset C(x,z) \\ \forall\ t & H\,[u(t)\ ,\ v(t)]\end{array}}{\forall\ t \qquad C\,[u(t)\ ,\ w(t)]}$$

γ/ instantiation $\exists_{y_2}$

$$\frac{R : \exists\ \overline{y_1}\ \sigma\!\!-_{y_2} H}{R : \exists\ \frac{}{y_1 y_2}\ H}$$



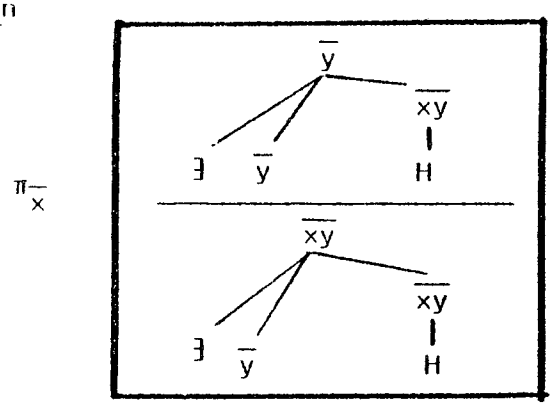

Ceci se vérifie, pour chaque interprétation, par réduction à:*

$$\frac{\forall x\ \exists\ y_1\ \forall t\ \ H\ \ [x, y_1\ ,\ u(t,x,y_1)]}{\forall x\ \exists\ y_1 y_2 \qquad H\ \ [x, y_1,\ y_2]}$$

*Note:* ceci est valable aussi pour $\overline{y}_1$ vide, à condition d'avoir l'axiome $\exists\ \emptyset\ H = H$

*Les expressions u, v, w ne dépendent pas nécessairement de t.

δ/ projection




Ceci peut être indifféremment considéré comme une règle d'inférence ou comme une cinquième opération de substitution (§b).

Sa validité se ramène à $\dfrac{\exists y\ \forall x\ H(x,y)}{\forall x\ \exists y\ H(x,y)}$

Avant de discuter de l'utilisation de ces règles par un démonstrateur de théorèmes, il convient de rappeler que la validité des déductions connait quelques limites inévitables:

c/ Effet de l'ambiguité d'interprétation

Rappelons que, pour chaque phrase écrite par l'utilisateur, l'analyseur syntaxique effectue son *interprétation complète* sous forme d'une contrainte R (sur les types libres) qui préfixe l'arbre syntaxique. R traduit l'ensemble des interprétations compatibles avec l'état du langage, mais peut en inclure qui ne correspondent pas à l'*intention* de l'utilisateur. Dans ce cas, il y aura *excès d'ambiguité*, l'interprétation de la phrase étant *trop vaste*, et la contrainte R *trop faible* ; ceci pouvant avoir deux effets sur la déduction:

α/ Déductions abusives , par propagation de l'excès d'ambiguité:

*Interprétation trop vaste d'une hypothèse*

Soit une phrase $A_0 \supset A_1$ dont l'interprétation par l'analyseur syntaxique est $R : A_0 \supset A_1$ , alors que, dans l'esprit de l'utilisateur, elle n'est valable que pour $R' : A_0 \supset A_1$ (avec $R' \supset R$).

Supposons qu'à partir de cette première phrase, la séquence de preuve suivante soit obtenue:

$$
\begin{array}{rl}
R & : \quad A_0 \supset A_1 \\
(\sigma_1\ R & : \quad \sigma_1\ A_1 \supset A_2) \\
\sigma_1\ R & : \quad \sigma_1\ A_0 \supset A_2 \\
(\sigma_2\ \sigma_1\ R & : \quad \sigma_2\ A_2 \supset A_3) \\
\sigma_2\ \sigma_1\ R & : \sigma_2\ \sigma_1\ A_0 \supset A_3 \\
 & \ldots \\
\sigma_n \ldots \sigma_1\ R & : \sigma_n \ldots \sigma_1\ A_0 \supset A_{n+1}
\end{array}
$$

et que cette séquence soit arrêtée là par l'utilisateur qui *refuse* le dernier théorème, ou plus précisément ne l'accepte que pour des interprétations conformes à une restriction R" *plus forte* que $\sigma_n \ldots \sigma_1\ R$ (Très souvent dans la pratique R" sera identiquement "FAUX", l'utilisateur refusant toute interprétation du théorème).

Le système"saura" alors qu'il doit remonter la preuve jusqu'à ce qu'il trouve (de façon interactive) la phrase trop ambigüe (ici $R : A_0 \supset A_1$). Il devra alors trouver R' tel que $\sigma_n \ldots \sigma_1\ R' = R$".

La *différence* entre R' et R permet de *localiser* dans "$A_0 \supset A_1$":

- soit un identifieur dont il faut rendre le type plus précis,
- soit une règle opératoire dont il faut renforcer la contrainte ou qu'il faut scinder. (voir Chapitre II: Extensions)

β/ Déductions impossibles

*Interprétation trop vaste d'une conjecture*

C'est le cas inverse où, avec

$$\frac{R : A \supset B \quad \sigma R : \sigma A}{\sigma R : \sigma B}$$

on ne peut prouver que.....

alors que l'utilisateur veut prouver un théorème $R' : \sigma B$ (dont il est "certain") mais pour lequel R' est plus faible que $\sigma R$.

Dans ce cas, il faut réduire l'interprétation complète de $\sigma B$ en remplaçant sa contrainte R' par $\sigma R$ au moyen des extensions appropriées, comme en α).

γ/ Apprentissage

Ceci montre que l'un des moyens dont dispose le système pour apprendre la *syntaxe* du langage est la *résolution de ses désaccords sémantiques avec l'utilisateur.*

C'est en effet grâce à ces désaccords que le système peut ajuster l' "Etat du Langage" considéré comme son *modèle* du langage de l'utilisateur.

## 3. Utilisation par un démonstrateur de théorèmes (exemple page 110)

Un démonstrateur de théorèmes fondé sur les règles d'inférence (§ 2.d) peut opérer de deux façons.

a/ mode exploratoire

Il s'agit, pour le démonstrateur, d'appliquer les règles *à l'endroit*, c'est-à-dire de combiner des éléments de la base pour en produire de nouveaux.
Ceci correspondra en général à l'application de l'une des deux règles dérivées suivantes:

1ère règle dérivée

$$\frac{R : A \supset B \qquad \sigma R : \sigma B \supset C}{\sigma R : \sigma A \supset C}$$

["A ⊃" est facultatif]

Celle-ci s'applique à tout couple d'éléments tel que la partie hypothèse de l'un soit une substitution de la partie conclusion de l'autre.

2ème règle dérivée

$$\frac{\sigma R : A \supset \sigma B \qquad R : B \supset C}{\sigma R : A \supset \sigma C}$$

["A ⊃" est facultatif]

Elle s'applique à tout couple d'éléments tel que la partie conclusion de l'un soit une substitution de la partie hypothèse de l'autre.

Dans les deux cas, la déduction se réduit à la recherche d'une substitution entre couples d'arbres , opération qui sera évoquée plus loin.

b/ mode inverse

Il s'agit, partant d'une conjecture; d'appliquer les règles *à l'envers*, pour obtenir des sous-buts.

Ceci correspondra en général à l'application de la règle dérivée:

3ème règle dérivée

$$R : H(\bar{x},\bar{y}) \supset C(\bar{x},\bar{z})$$

*sous-but:* $\sigma R : \exists\bar{y}\ \sigma H$ [si $\bar{y}$ est vide, $\exists\bar{y}$ disparait]

$$\sigma R : \sigma C$$

On rappelle que $H(\bar{x},\bar{y}) \supset C(\bar{x},\bar{z})$ signifie $\forall\bar{x}\ (\ \exists\bar{y}\ H(\bar{x},\bar{y}) \supset \forall\bar{z}\ C(\bar{x},\bar{z})\ )$

Là encore, la déduction se réduit à la recherche d'une substitution entre deux arbres (la conjecture et la partie conclusion d'un élément de la base).

Le sous-but "$\exists\bar{y}\ \sigma H$" indique que la substitution $\sigma$ , qui porte sur les variables de C (c'est-à-dire $\bar{x}$ et $\bar{z}$) pourrait porter aussi sur $\bar{y}$.

Il faut noter que,pour une conjecture donnée C', il peut y avoir plusieurs éléments $H_i \supset C_i$ tels que $C' = \sigma_i\ C_i$. Il faut alors procéder comme indiqué ci-dessous.

c/ réduction du "domaine d'incertitude"

On attachera à chaque conjecture son *"domaine d'incertitude":* D.

D représente à chaque instant le domaine hors duquel la conjecture est déjà prouvée et à l'intérieur duquel elle n'est pas encore prouvée.

Ce domaine D est caractérisé par une propriété exprimée dans le langage de spécifications. Au départ, la conjecture n'est prouvée nulle part et son domaine d'incertitude est représenté par D = "VRAI".

On tente alors de prouver la conjecture comme suit:

Preuve par parties de C':

Pour chaque élément $H_i \supset C_i$ de la base, tel que:

$C' = \sigma C_i$ , on obtient un sous-but $S_i$ conformément à la 3ème règle dérivée.

Il faut alors prouver $D \supset S_i$ , et en cas d'échec, réduire le domaine d'incertitude D :

$$D \leftarrow D \wedge \neg S_i$$

(Remarquons évidemment que si $D \supset S_i$ , $D \wedge \neg S_i \equiv$ "FAUX")

On peut dire que le domaine d'incertitude caractérise *l'hypothèse qui manque pour qu'on soit capable de prouver la conjecture.*

On peut aussi considérer que c'est la *différence entre les axiomes et la conjecture.*

On verra plus loin que la connaissance de D est essentielle pour un démonstrateur interactif, car elle permet d'indiquer exactement à l'utilisateur la *cause d'échec* de la preuve.

d/ recherche de substitutions

On a vu que la déduction se ramène à de simples manipulations d'arbres provoquées par la *détection de substitutions* entre certains couples d'arbres interprétés.

Il faut d'abord examiner comment on décide si deux arbres sont liés ou non par une substitution (au sens de la définition pages 96-98).

Etant donnés deux arbres syntaxiques munis d'interprétations complètes:

$$R : A \qquad \text{et} \qquad R' : A' ,$$

la recherche d'une substitution $\sigma$ telle que $R':A' = \sigma(R:A)$ peut s'effectuer en trois temps.

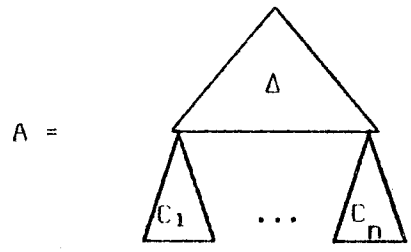
premier temps choix d'un *"radical" non interprété commun à A et A'*

Tout arbre interprété A peut être représenté de plusieurs manières par

- un *"radical"* $\Delta_n$: arbre interprété dont on singularise n feuilles non interprétées $\alpha_1 \ldots \alpha_n$
- une suite d' *"appendices"*: n arbres interprétés $C_1 \ldots C_n$

de façon que A s'obtienne en remplaçant ces feuilles $\alpha_1 \ldots \alpha_n$ de $\Delta_n$ par les appendices $C_1 \ldots C_n$ respectivement.

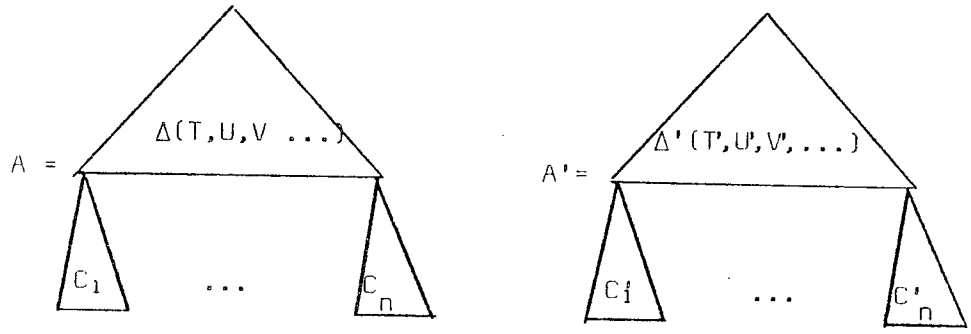
On peut illustrer graphiquement cette décomposition par un schéma:




Il faut remarquer que toute substitution (au sens de la définition page 96 ) préserve un radical,sauf en ce qui concerne les types qui sont marqués aux noeuds.

On appellera "radical non interprété" un radical dont on a supprimé tous les types attachés aux noeuds (les noeuds ne représentent donc *que des opérateurs*).
On démarrera la recherche d'une substitution $\sigma(R:A)=R':A'$ en cherchant les radicaux non interprétés communs à A et A'.
On montre facilement que l'ensemble de ces radicaux est constitué de tous les radicaux d'un *radical commun maximum.*
On verra (page 109) qu'une substitution ne préserve pas nécessairement ce radical maximum et qu'on peut donc être amené à essayer plusieurs radicaux non interprétés communs.

Ayant choisi un radical non interprété $\Delta$ commun à A et A', on a donc




2ème temps - *recherche d'une substitution de types*

Il convient maintenant de voir si une substitution de types (au sens des définitions $\alpha$, $\beta$ page 96) permet de passer du radical $\Delta(T,U,V,...)$ au radical $\Delta(T',U',V' ...)$.

Il faut pour cela trouver un changement de variables $\delta$ sur les types T,U,V .. qui fasse coïncider les radicaux interprétés, puis prouver $R' \supset \delta R$.

3ème temps - *existence d'une substitution*

Il reste à voir s'il existe une substitution préservant $\Delta$ (au sens des définitions $\gamma$, $\delta$ page 97).

Appelons $\overline{x}$ les variables locales à A (indiquées avec la racine de A, cf. page 61) et $\overline{x}'$ celles de A'.

(il faut renommer les variables de $\overline{x}'$ qui apparaitraient dans $\overline{x}$)

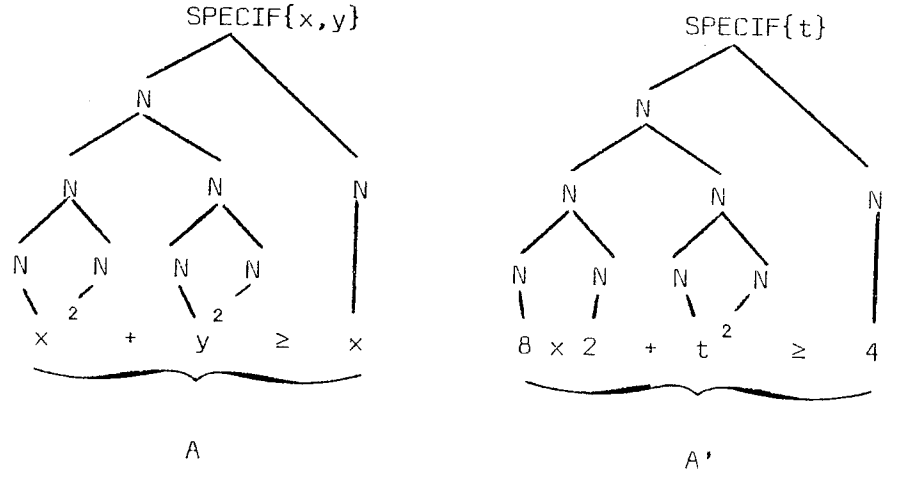
Il y a substitution $\sigma A=A'$ si:

$$\forall \overline{x}' \; \exists \overline{x} \; \forall i \; ( \; C_i(\overline{x}) = C'_i(\overline{x}') \; )$$

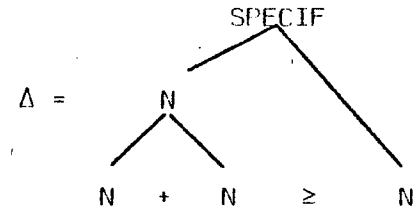
(si $\overline{x}'$ est vide, on supprime "$\forall \overline{x}'$")

Les solutions $\sigma$ sont les fonctions de Skolem correspondant au quantifieur $\exists$ .

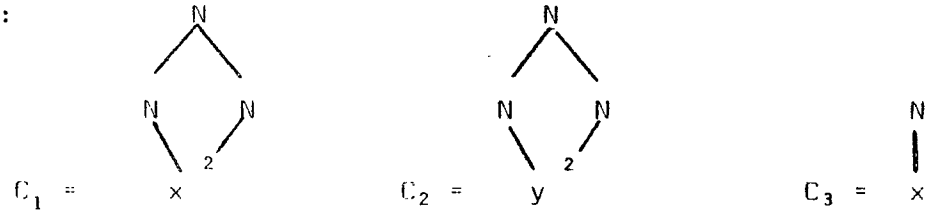
*exemple:*

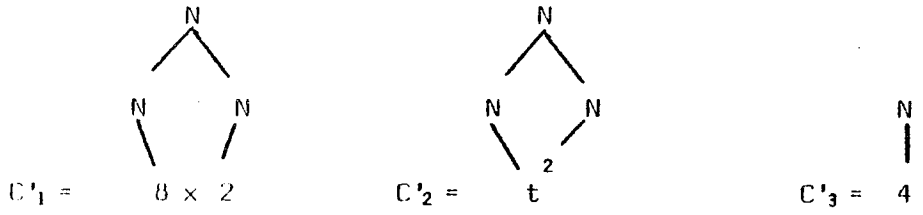


en prenant comme radical commun




on a :




et la condition de substitution est:

$$\forall t \; \exists xy \; ( \; x^2 = 8x2 \quad \text{et} \quad y^2 = t^2 \quad \text{et} \quad x=4 \; )$$

qui a pour solution $\sigma$ : $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} 4 \\ \pm t \end{pmatrix}$

*Efficacité*

La recherche systématique de toutes les substitutions est coûteuse:

- Dans une base de n éléments il existe $O(n^2)$ couples, d'où un facteur $n^2$ dans la recherche des substitutions à l'intérieur de la base.
- Il peut exister beaucoup de radicaux communs à A et A': si l'on en prend un trop petit, la condition d'existence peut être trop difficile à résoudre ; si l'on en prend un trop grand, il peut ne pas exister de substitution qui le préserve (c'est le cas de l'exemple ci-dessus si on prend le radical commun maximum).

En revanche, la définition des substitutions est "riche" et permet d'éviter des étapes de démonstration.

De plus, on verra dans la prochaine section (Démonstrateur) que des techniques de "pattern matching" permettent de trouver très rapidement la plupart des substitutions "intéressantes".

## 4. Exemple de déduction

Pour faciliter la compréhension de l'exemple qui suit, chaque phrase est désignée par un numéro de la forme:

($N_i$) pour les phrases du noyau ;

($S_i$) pour les spécifications du problème et leurs conséquences ;

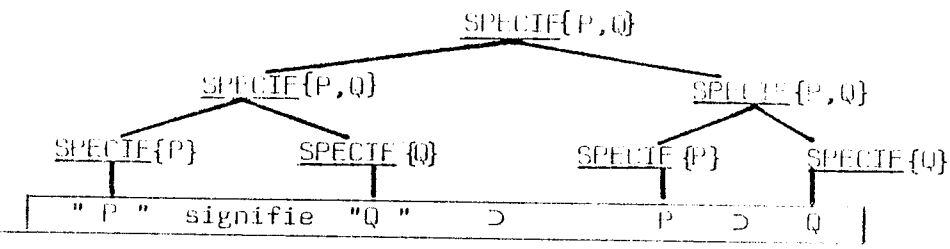
($D_i$) pour les phrases caractérisant le domaine d'incertitude ;

($C_i$) pour les phrases représentant conjectures et sous-buts.

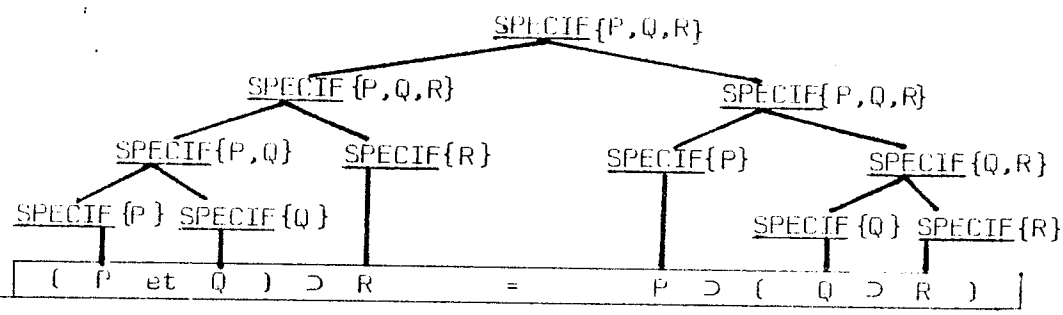
D'autre part, chaque phrase est présentée avec son arbre syntaxique muni d'une interprétation complète. En particulier, chaque noeud est étiqueté par le type du sous-arbre correspondant, et, entre accolades {}, l'ensemble des variables locales de la phrase qui apparaissent dans ce sous-arbre. Lorsqu'il y a doute, on indique pour ces variables leur type dans le sous-arbre.

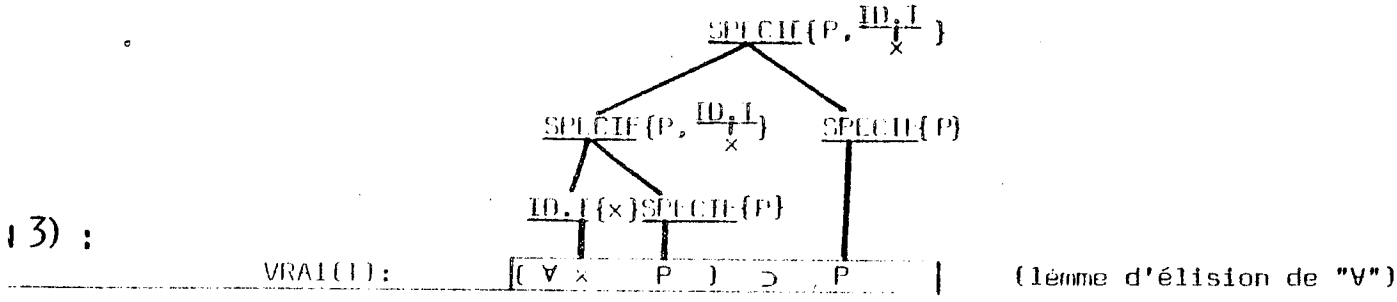
Supposons que la base contient - entre autres - les dix phrases ci-dessous:

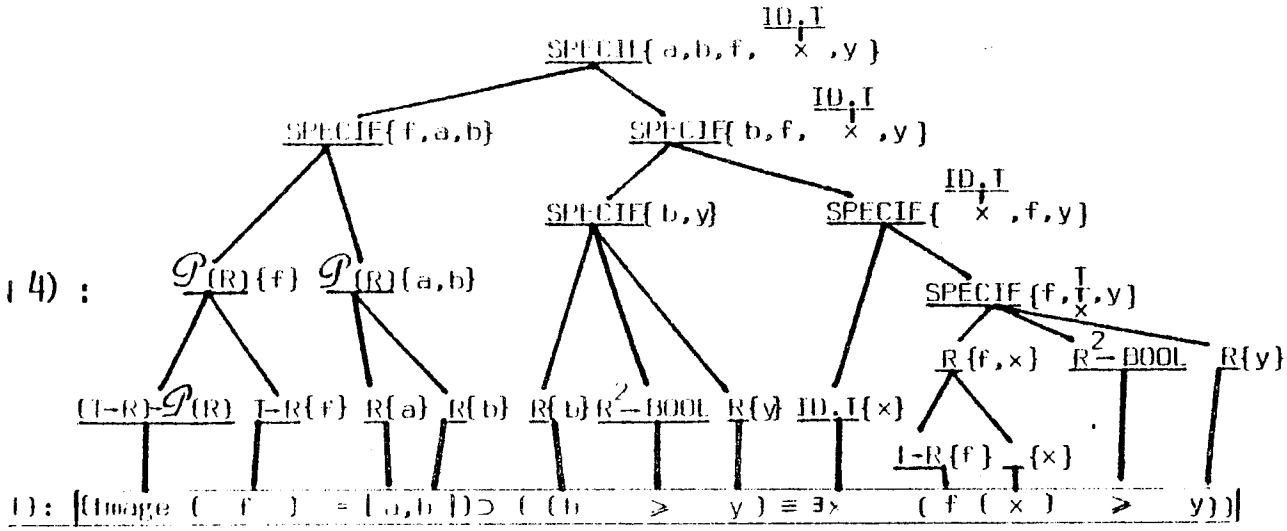
(N 1) :

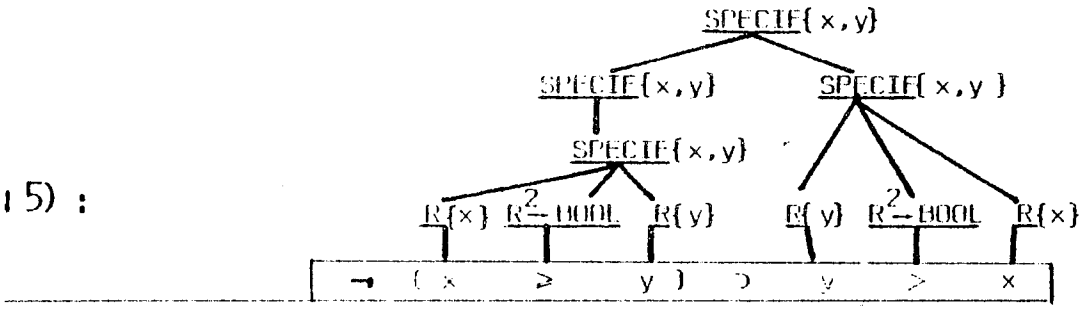


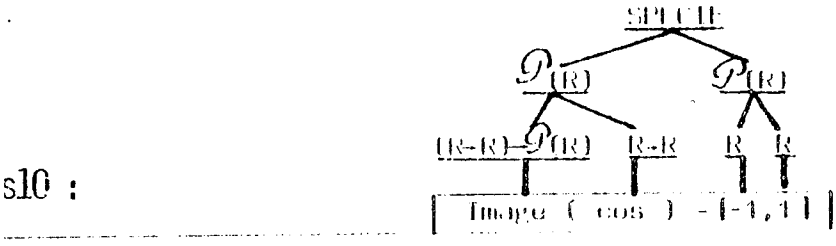

(N 2) :

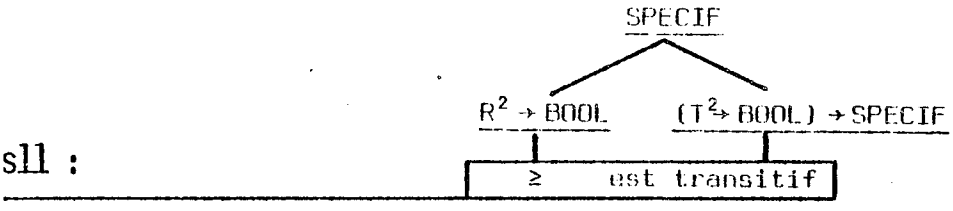



SPECIF(P, ID.I/x)
SPECIF(P, ID.I/x)
SPECIF(P)
ID.I(x)
SPECIF(P)
VRAI(I):
(∀x P) ⊃ P
(lemme d'élision de "∀")

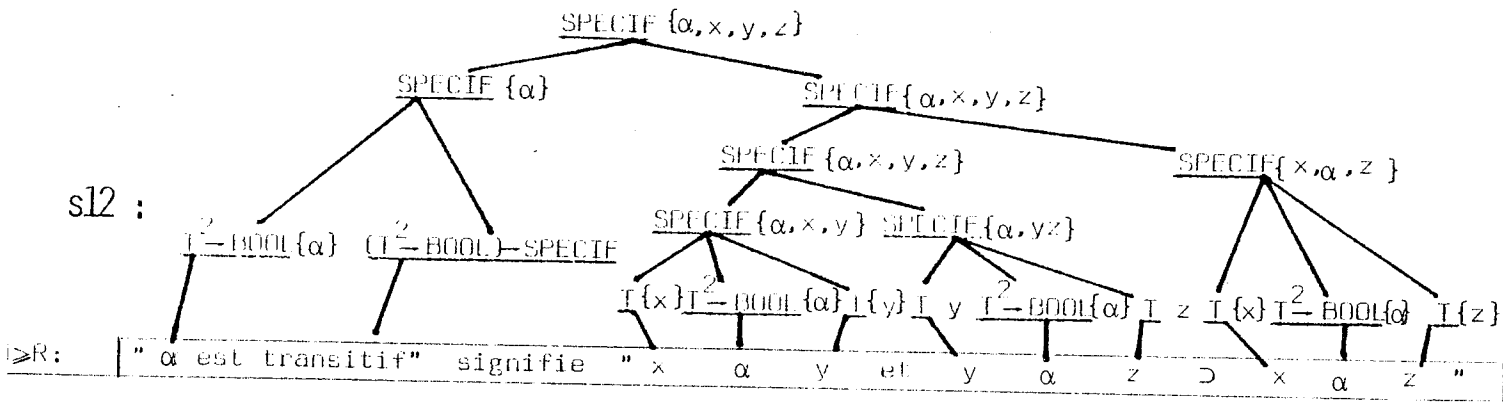

SPECIF(a,b,f, ID.I/x ,y)
SPECIF(f,a,b)
SPECIF(b,f, ID.I/x ,y)
SPECIF(b,y)
SPECIF( ID.I/x ,f,y)
SPECIF(f, I/x ,y)
(I-R)-P(R)
I-R(f)
R(a)
R(b)
R(b)
R²-BOOL
R(y)
ID.I(x)
R(f,x)
R²-BOOL
R(y)
I-R(f)
(x)
I): (Image(f) = [a,b]) ⊃ ((b ≥ y) ≡ ∃x (f(x) ≥ y))


SPECIF(x,y)
SPECIF(x,y)
SPECIF(x,y)
SPECIF(x,y)
R(x)
R²-BOOL
R(y)
R(y)
R²-BOOL
R(x)
¬(x ≥ y) ⊃ y > x

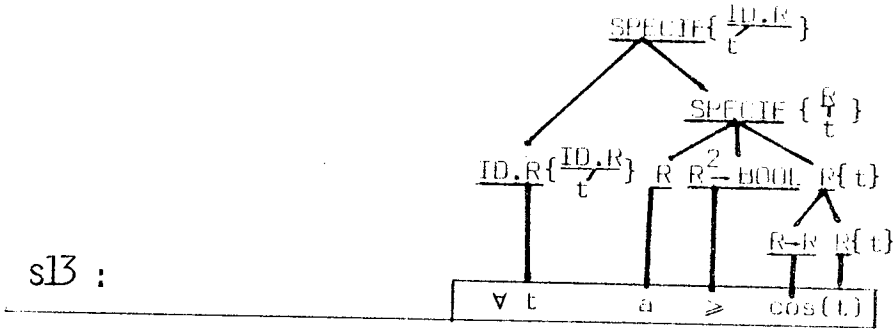

SPECIF
P(R)
P(R)
(R-R)-P(R)
R-R
R
R
Image(cos) = [-1,1]


SPECIF
R² → BOOL
(T² → BOOL) → SPECIF
≥ est transitif

s12 :




s13 :

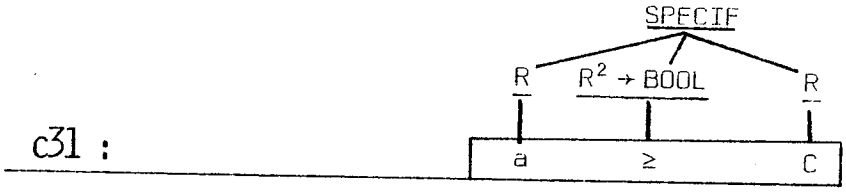



s14 :


---

Et soit la *conjecture:*

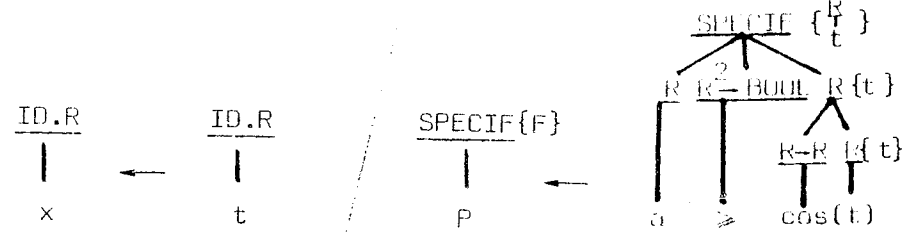
c31 :



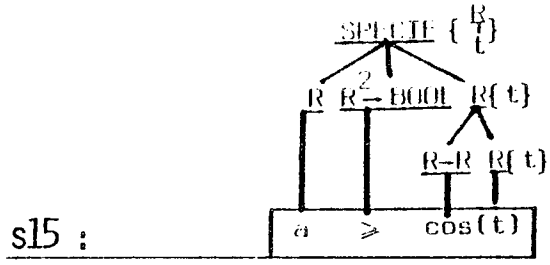

Le système de déduction *pourrait* se comporter de la façon suivante:

a/ Détection d'une substitution entre N3 (partie hypothèse) et S13:

- substitution de types (cas β,page 96): T=R ⊃ VRAI(T)
- substitution de variables (γ, page 97):

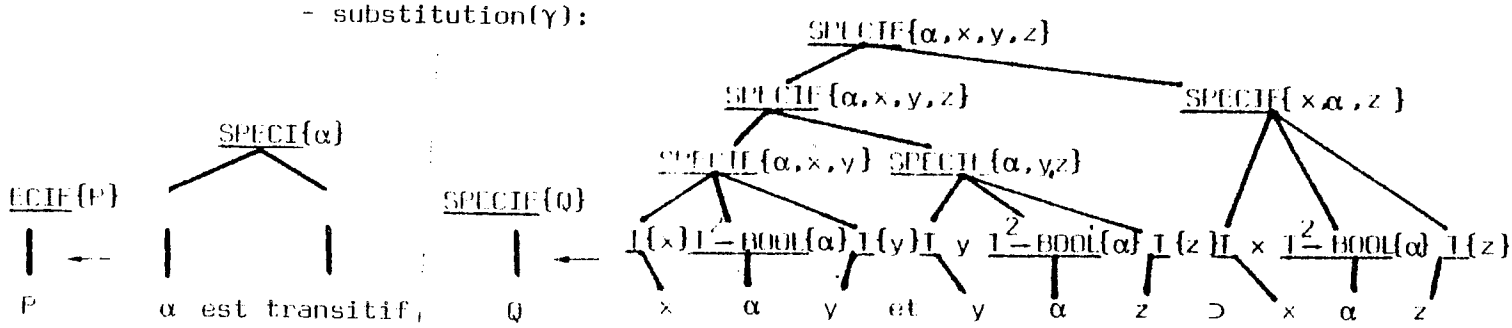


Cette substitution provoque l'application de la *(2ème règle dérivée (mode exploratoire,* page104*)* et produit la conséquence S15:

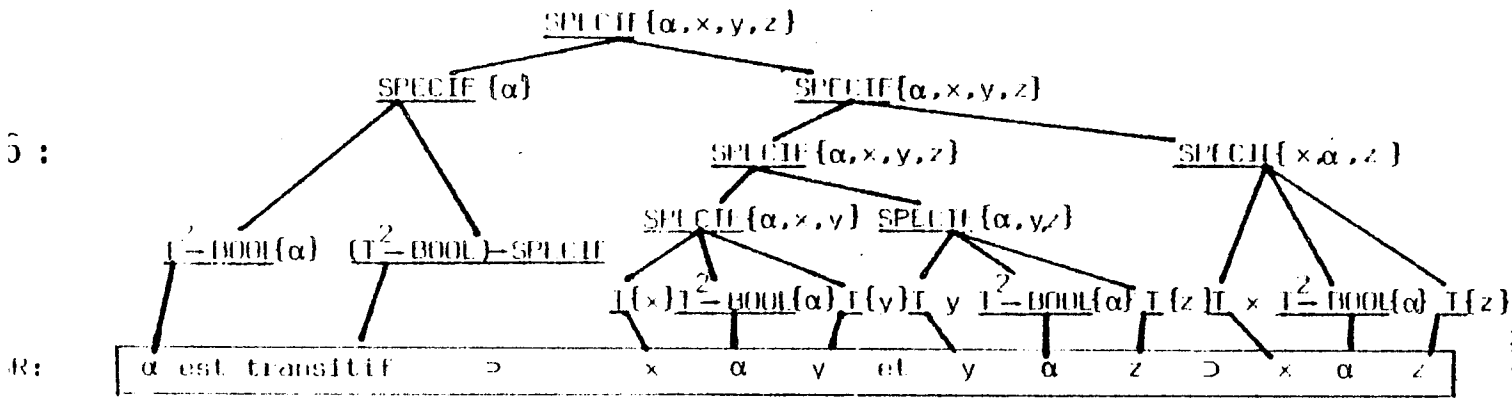



b/ Détection d'une substitution entre N1 (partie hypothèse) et S12:

- substitution(β): T ≥ R ⊃ VRAI
- substitution(γ):



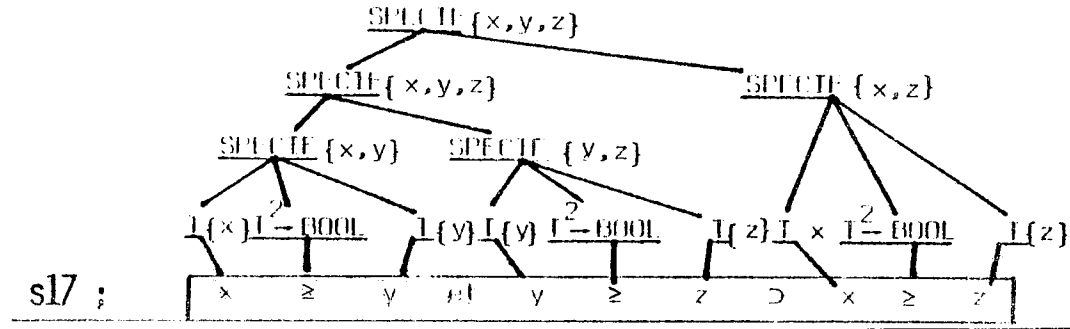

La deuxième règle dérivée produit alors la conséquence S16:




c/ Détection d'une substitution entre S11 et S16 (partie hypothèse):

- substitution(β): T=R ⊃ T ≥ R
- substitution(γ): $\frac{R^2 \to BOOL\{\alpha\}}{\beta \atop \alpha} \leftarrow \frac{R^2 \to BOOL}{\geq}$

et la deuxième règle dérivée produit la conséquence S17:



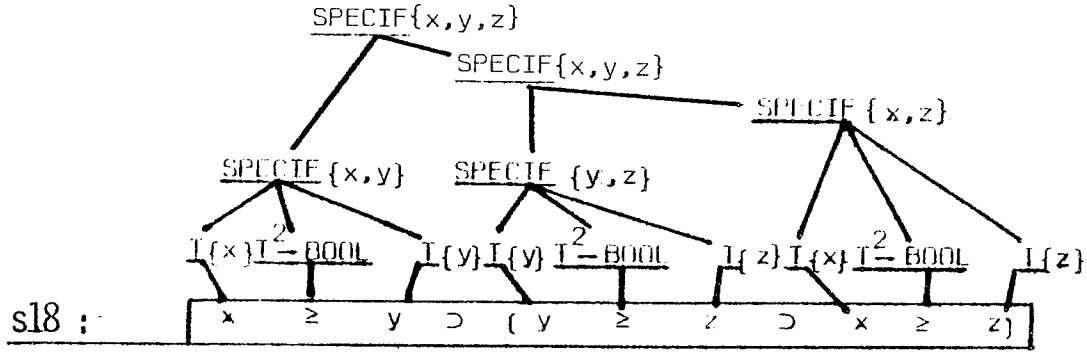
d/ Détection d'une substitution (γ) entre N2 (partie gauche) et S17 (partie hypothèse):

$$P \leftarrow x \geq y \quad | \quad \alpha \leftarrow y \geq z \quad | \quad R \leftarrow x \geq z$$

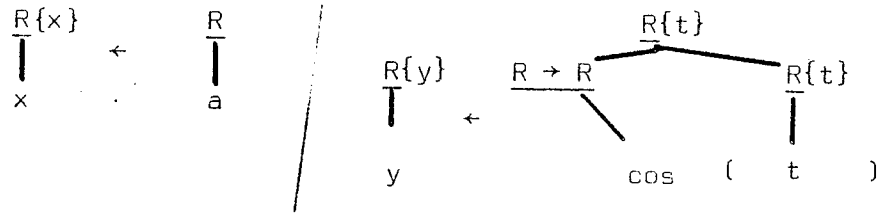
en appliquant cette substitution à N2 (règle d'emphase page 100) on obtient*:

$$( (x \geq y \text{ et } y \geq z) \supset x \geq z) = (x \geq y \supset (y \geq z \supset x \geq z) )$$

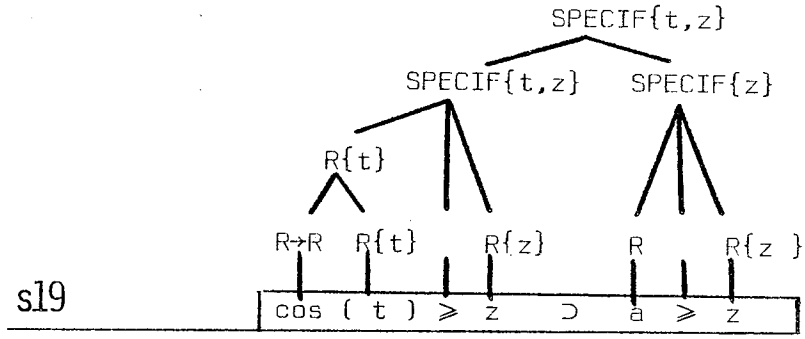
Ceci permet d'appliquer à S17 une substitution (δ) d'égalité et de produire, encore par la règle d'emphase, S18:




e/ Détection d'une substitution (γ) entre S15 et S18 (partie hypothèse):




l'application de la 2ème règle dérivée produit S19:

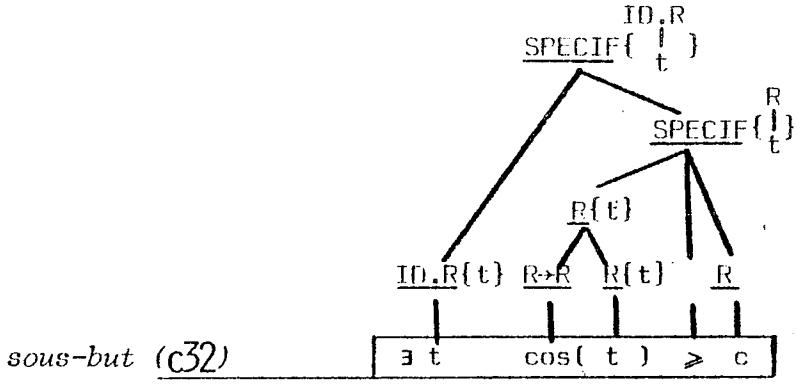



f/ Détection d'une substitution entre la conjecture C31 et S19 (partie conclusion):

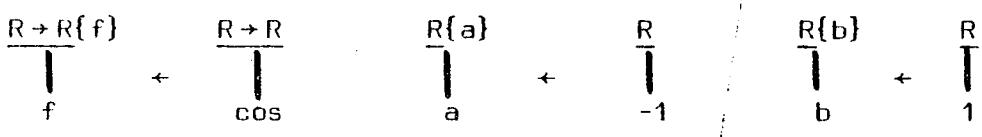

---

* les parenthèses sont introduites pour le lecteur, on verra plus loin (p.135 que l'introduction ou la suppression de parenthèses ne perturbe pas la reconnaissance des arbres.

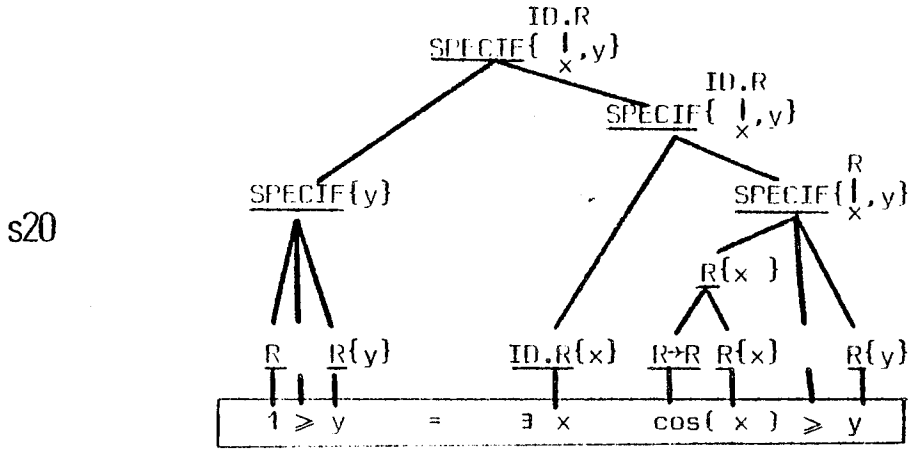
Ceci provoque l'application de la *3ème règle dérivée(mode inverse)* et produit le *sous-but* C32.



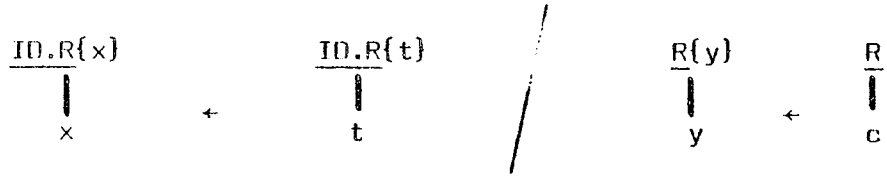

g/ Détection d'une substitution entre N4 (partie hypothèse) et S10:



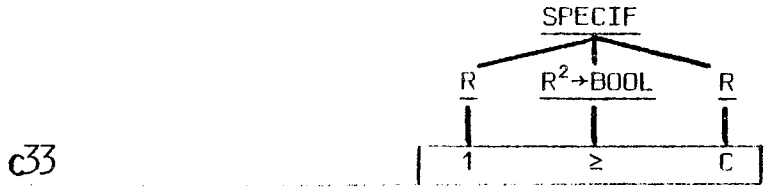

La 2ème règle dérivée produit la conséquence S20:




h/ Détection d'une substitution entre C32 et S20 (partie droite):




Ceci produit le sous-but C33:



i/ Ne pouvant prouver ce sous-but, le système met son complément dans le *domaine d'incertitude* D25 :

D25 ¬(1 ≥ c)

j/ Détection d'une substitution entre N5 et D25:


qui permet de *remplacer* D25 par D26 :

D26 c > 1

k/ Détection d'une substitution entre S14 et S18:

x ← a / y ← 2c - 2

produisant la conséquence S21:

S21 2c-2 ≥ z ⊃ a ≥ z

l/ Détection d'une substitution entre la conjecture C31 et S21:

z ← c , avec production du sous-but C34:

C34 2c - 2 ≥ c

m/ Si le noyau est suffisamment fourni, le système saura résoudre cette inégalité linéaire en la ramenant au sous-but C35:

C35 c ≥ 2

n/ Les éléments de la base (Noyau + S10 à S21 + D26) ne permettant pas de prouver le sous-but C35, le système rajoute son complément au domaine d'incertitude:

D27 c < 2

Le domaine d'incertitude (D26 et D27) exprime ce qui manque pour que la conjecture puisse être prouvée:

"Conjecture non prouvée si 1 < c < 2 "

# III - SYSTEME DE DEDUCTION

Ayant défini le principe de la sémantique déductive, il s'agit maintenant de pouvoir l'implémenter, c'est-à-dire de mettre au point un *démonstrateur de théorèmes* aussi efficace que possible. Cette section commencera donc par un énoncé de critères permettant d'évaluer la qualité d'un démonstrateur, et on trouvera ensuite des solutions simples mais originales permettant d'atteindre un bon compromis.

## 1. Démonstrateurs de théorèmes: critères de qualité

Si évidents que soient les critères c-dessous, il m'a paru utile de les énoncer car ils sont le plus souvent ignorés dans la pratique:

### a/ critères qualitatifs

#### α/ *facilité d'accès et d'utilisation*

Ceci concerne les langages de commande très souvent ésotériques, et les langages d'expression de théorèmes allant du langage naturel (idéal irréalisable) au langage des prédicats en forme normale conjonctive skolemisée (très peu commode).

Le langage dépend lui-même de l'équipement: terminaux graphiques, chaînes de caractères étendues (lettres gecques, indices, signes mathématiques, etc...), bon support système ... peuvent faire la différence entre un excellent démonstrateur et un outil inutilisable.

*β/ précision de la réponse*

Chargé de résoudre une conjecture, un démonstrateur peut offrir plusieurs types de réponses selon sa qualité:

Réponse peu précise:
- VRAI
- FAUX
- INCERTAIN

Réponse plus précise:
- VRAI
- FAUX: contre exemple
- INCERTAIN

Réponse très précise:
- VRAI
- FAUX
- INCERTAIN

+ ensemble de tous les contre-exemples, hypothèses manquantes.

Lorsqu'un démonstrateur est utilisé pour la construction d'objets (programmes ou autres) conformes à des spécifications données, une réponse précise est essentielle car elle fournit un diagnostic permettant de corriger les erreurs de conception.

*γ/ résistance au "bruit"*

Presque toujours ignoré, ce critère est l'un des plus importants. On appelle "bruit" l'ensemble des hypothèses inutiles à la preuve d'une conjecture donnée. Dans la réalité, on dispose en général d'un vaste excès d'information, et un démonstrateur très efficace avec exactement les hypothèses nécessaires peut se révéler lamentable lorsqu'on lui en fournit trop.

δ/ *portée (degré de complétude)*

La portée d'un démonstrateur est définie ici comme l'ensemble des conjectures qu'il peut résoudre (réponse VRAI ou FAUX) en un temps non limité.

b/ critères quantitatifs

α/ *occupation mémoire*

β/ *rapidité*

Le temps nécessaire à un démonstrateur pour résoudre une conjecture augmente très rapidement avec la *"difficulté"* de celle-ci:

*Définition:* difficulté d'une conjecture.

> On appellera difficulté d'une conjecture la *longueur* de la plus courte preuve de cette conjecture ou de son contraire.

(La longueur d'une preuve étant le nombre d'applications de règles d'inférence qu'elle comporte).

En particulier, une conjecture indécidable est de difficulté infinie.

Il est sommairement établi en ANNEXE II que, pour prouver un théorème arbitraire de difficulté d, tout démonstrateur a besoin d'un temps moyen:

$$\sim \alpha e^{\beta d}$$

où α et β sont des caractéristiques du démonstrateur.

*Remarque:*

La difficulté d'une conjecture étant très délicate, voire impossible à évaluer, on peut être tenté de choisir une mesure directement accessible, telle que la longueur de l'énoncé de la conjecture. Malheureusement, il n'y a en général pas de relation entre la longueur d'une conjecture et le temps qu'il faut pour la résoudre.

γ/ *autonomie*

Un démonstrateur de rapidité moyenne $\alpha e^{\beta d}$ permet de prouver en un "temps acceptable" Tmax des théorèmes de difficulté d telle que $\alpha e^{\beta d} < T_{max}$, c'est-à-dire:

$$d < \frac{1}{\beta} (\log T_{max} - \log \alpha)$$

Cette quantité sera appelée la $[T_{max}]$-autonomie du démonstrateur et désigne la difficulté maximale des théorèmes qu'on peut raisonnablement soumettre à un démonstrateur.

On constate que cette autonomie est inversement proportionnelle à β. Tous les efforts pour diminuer β, bien qu'ils conservent le caractère exponentiel du démonstrateur, sont donc très utiles pour améliorer l'autonomie. Cette remarque sera largement appliquée par la suite.

δ/ *fiabilité*

En principe un démonstrateur ne tolère pas d'erreur.
Pourtant, on pourrait imaginer des *"ε-démonstrateurs"* dans lesquels la probabilité d'exactitude (fiabilité) de la réponse VRAI ou FAUX est supérieure à 1-ε.

c/ compromis entre ces critères

Ces différents critères de qualité ne sont pas indépendants (exemple: l'autonomie est directement liée à la rapidité), et certains sont contradictoires, ce qui implique la recherche de compromis. Les principales contradictions sont:

α/ *portée ou rapidité*

Parmi toutes les conjectures possibles, une infime minorité sont "intéressantes". Si l'on savait caractériser les conjectures intéressantes et limiter la portée d'un démonstrateur à cette classe de conjectures, on obtiendrait des performances bien meilleures (mémoire et temps).

D'une manière plus nuancée, les but des "heuristiques" est de faire en sorte que, pour les conjectures intéressantes, la durée de la preuve soit très inférieure à $\alpha e^{\beta d}$, et que pour les autres elle soit donc supérieure à $\alpha e^{\beta d}$.

On verra plus loin des exemples de telles heuristiques.

β/ *rapidité ou fiabilité*

A priori on demande à un démonstrateur une fiabilité 1, mais si une fiabilité $1-\varepsilon$ permet des performances considérablement supérieures, on peut envisager l'opportunité d'un compromis, comme on le verra dans la suite.

Nous pouvons maintenant examiner les diverses techniques permettant d'obtenir un démonstrateur de théorèmes efficace.

## 2. Interaction

Quelle que soit l'autonomie du démonstrateur, il est certain qu'elle sera très inférieure à la difficulté des théorèmes qu'on devra prouver. C'est pour cette raison qu'il faut abandonner tout espoir de démonstrateurs entièrement automatiques.
On s'orientera donc vers des démonstrateurs interactifs:
il s'agit, en présence d'une conjecture, de donner suffisamment d'indications au démonstrateur pour que cette conjecture soit divisée en sous-théorèmes qui soient de difficultés inférieures à l'autonomie.

Une façon de procéder consiste à donner la preuve dans les grandes lignes et à laisser le démonstrateur automatique "remplir les vides" ; c'est exactement ce dont on a besoin dans la pratique: tous les détails simples, répétitifs et ennuyeux doivent être automatisés, mais l'essentiel de l'argumentation peut être laissé à l'initiative de l'utilisateur.

Plus précisément, soit un démonstrateur d'autonomie $\underline{a}$ ($\alpha e^{\beta a} = T_{max}$), et une conjecture de difficulté $\underline{d} > a$.
La preuve automatique de cette conjecture prendrait un temps probable de $\alpha e^{\beta d} = \alpha \left(\frac{T_{max}}{\alpha}\right)^{\frac{d}{a}}$ croissant très rapidement avec d.

Si l'utilisateur trouve $\sim \frac{d}{a}$ étapes intermédiaires bien placées, la conjecture est décomposée en $\sim \frac{d}{a}$ théorèmes de difficultés $\sim a$ qui seront prouvés en un temps total probable $\sim \frac{d}{a} T_{max}$ .

> Autrement dit, si l'utilisateur donne un "squelette de preuve" *proportionnel* à la difficulté de la conjecture, le démonstrateur la prouvera en général en un temps *proportionnel* à cette difficulté.

- Encore faut-il que l'utilisateur sache décomposer la conjecture en sous-théorèmes de difficultés voisines de a ; pour cela, il peut procéder de façon dichotomique: tout sous-théorème qui n'a pas été prouvé en un temps $T_{max}$ doit être décomposé par l'utilisateur en deux sous-théorèmes.

- Enfin, un démonstrateur interactif ne sera réellement utilisable que si les deux contraintes suivantes sont respectées:

. qu'en cas d'échec dans la preuve d'un théorème (théorème faux ou temps maximum écoulé) le démonstrateur puisse indiquer les causes possibles de cet échec: c'est ce qui a été prévu avec la mise à jour du "domaine d'incertitude" décrite précédemment.

. que ces indications du démonstrateur soient compréhensibles (lisibles) et, inversement, que l'utilisateur ait effectivement des moyens linguistiques simples de fournir une preuve dégrossie: ceci suppose que le langage d'expression des théorèmes soit suffisamment souple, et justifie l'attention particulière qui a été portée à la forme du Langage de Spécifications.

## 3. Amélioration de l'autonomie (I)

Nous venons de voir l'importance de l'autonomie du démonstrateur: en l'accroissant, on diminue *proportionnellement* aussi bien le temps machine nécessaire que le squelette de preuve exigible de l'utilisateur.

Nous allons examiner une technique qui améliore à la fois l'autonomie et la résistance au bruit, au prix d'une diminution de la portée.

### a/ rappel sur les substitutions (coût)

On a vu (§ 3d) que la recherche systématique de toutes les substitutions était très coûteuse pour les raisons suivantes:

1 - pour chaque couple de phrases il faut essayer tous les radicaux communs et non pas seulement le radical commun maximum ;

2 - tout couple de phrases doit être inspecté, puisque toutes les phrases ont un radical commun ;

3 - pour chaque couple de phrases et chaque radical commun à ce couple, la recherche d'une substitution peut impliquer la preuve d'un théorème éventuellement complexe.

La solution proposée maintenant a l'intérêt de diminuer considérablement ces facteurs de coût en se limitant à la recherche des substitutions "intéressantes".

### b/ classe réduite de substitutions

Soient deux phrases A et B, et supposons que l'on recherche une substitution $B = \sigma A$. Au lieu de considérer tous les radicaux communs à A et B, on considérera seulement un radical bien particulier de A, appelé *radical principal* de A. On ne cherchera de substitution que si le radical principal de A est un radical de B.
Il s'agit (cf. page 109) de définir un radical principal suffisamment petit pour n'éliminer aucune substitution "intéressante", et suffisamment grand pour faciliter la résolution des conditions de substitution:

*Radical Principal d'une phrase*

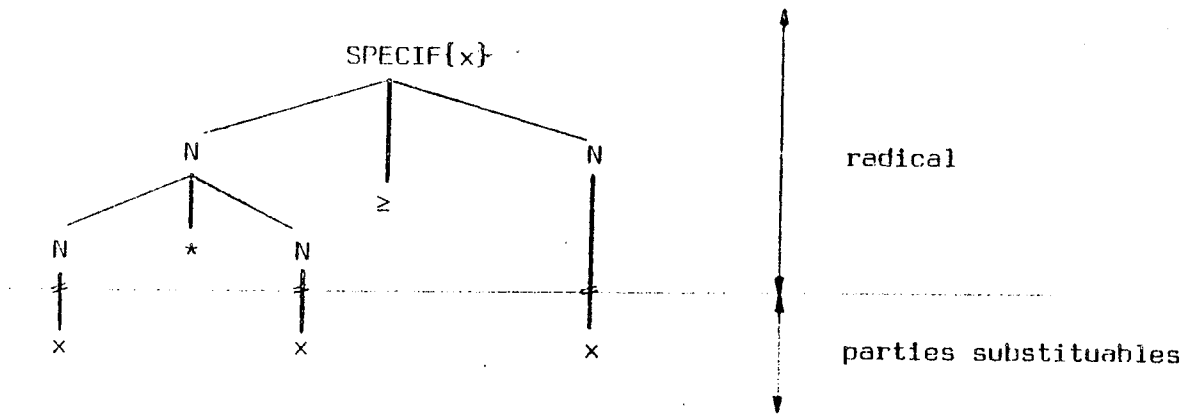
Tout arbre syntaxique sera désormais décomposé de *façon unique* en un radical principal et des appendices considérés comme seuls substituables.

La frontière entre radical et partie substituable sera indiquée sur les arbres syntaxiques par des "coupures".

Pour définir ce radical principal, la première idée est de placer les coupures immédiatement au-dessus des variables locales, puisque celles-ci - implicitement quantifiées - sont substituables.

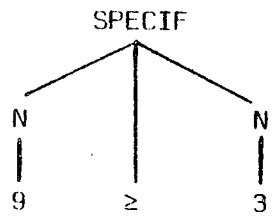
L'arbre ci-dessous serait alors décomposé comme suit:

A:




Malheureusement l'arbre:

B:




ne serait pas reconnu comme une substitution de A puisqu'il ne contient pas le radical spécifié pour A.

La raison de cet échec est que "9" peut s'écrire d'une infinité de façons (3x3, 5+4,...), ainsi d'ailleurs que "x * x" ($x^2$, $\sum_{1}^{x}(2i-1)$,...).

Il faut donc tenir compte, dans la détermination du radical principal, du *polymorphisme* des expressions, en plaçant les coupures suffisamment haut, selon l'algorithme suivant:

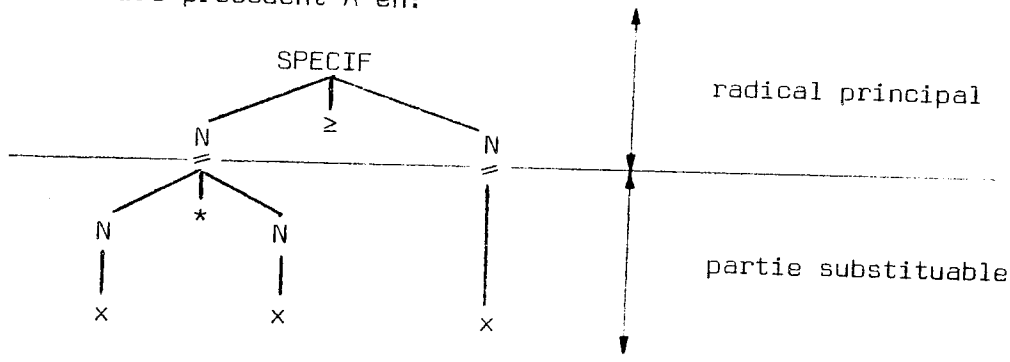
*Algorithme de placement des coupures:*

1° placer une coupure au-dessus de chaque variable locale,

2° (itérer:) si le type associé à une coupure est immédiatement surmonté du même type, remonter la coupure au noeud correspondant, et supprimer les coupures plus basses.

Cet algorithme peut être exécuté en même temps que l'analyse syntaxique et décomposera l'arbre précédent A en:

A:




(D'après cette heuristique, les opérateurs qui "font remonter les coupures" sont les *lois de compositions internes:* l'expérience indiquera s'il faut améliorer ce critère).

Comme l'arbre B admet 

pour radical, il pourra être considéré comme une substitution de A *sous réserve du sous-but:* $\exists x\ [x * x = 9 \text{ et } x = 3]$, qui est trivialement vrai.

Pour chaque couple de phrases (A,B), la recherche d'une substitution se réduit donc à:

1er temps: voir si le radical principal de A est un radical de B ;

2ème temps: substitution de types (souvent triviale) ;

3ème temps: [seulement si la substitution doit être utilisée dans une preuve]: preuve d'un sous-but (d'autant plus simple que le radical est grand) ou adjonction de son complément au domaine d'incertitude.

Bien que cette technique réduise considérablement le coût de recherche des substitutions, il y a néanmoins n(n-1) couples ordonnés de phrases à examiner (pour une base de spécifications comportant n éléments) où n peut être très élevé. Nous allons décrire ci-dessous un moyen de réduire ce facteur $O(n^2)$ à $O(n \log n)$, par application d'une technique efficace de "pattern matching" analogue au *"filtrage anti-bruit"*.

c/ reconnaissance d'arbres

Supposons que l'on sache représenter chaque arbre syntaxique A par un *"spectre"* $\langle s_A ; S_A \rangle$ tel que:

- $s_A$ est un ensemble de nombres caractérisant le radical principal de A,
- $S_A$ est un ensemble de nombres caractérisant la totalité de A ;
- $s_A \subset S_B$ si et seulement si le radical principal de A est un radical de B.

La recherche de substitutions (1er temps) se fait alors comme suit:

- On maintient avec la base de spécifications $\{A_1 \ldots A_n\}$ les données suivantes:

. les deux ensembles $\bigcup_i s_{Ai}$ , $\bigcup_i S_{Ai}$ ;

. à chaque élément "e" de $\bigcup s_{Ai}$ est associé l'ensemble:

$$f(e) = \{Ai \mid e \in s_{Ai}\}$$

. à chaque élément "E" de $\bigcup S_{Ai}$ est associé l'ensemble:

$$F(E) = \{Ai \mid E \in S_{Ai}\}$$

- Soit une phrase B dont le spectre est $\langle s_B ; S_B \rangle$ , on cherche les $A_i$ tels que $A_i = \sigma_i B$ ou $B = \sigma'_i A_i$ :

- Recherche des Ai tels que $Ai = \sigma_i B$ , c'est-à-dire $s_B \subset S_{A_i}$ :

  . si $s_B \not\subset \bigcup_i S_{Ai}$ : aucune solution,

  . sinon ($s_B \subset \bigcup_i S_{Ai}$), les solutions sont les $A_i$ de l'ensemble: $\bigcap_{x \in s_B} F(x)$

  (c'est l'ensemble des Ai tels que $s_B \subset S_{Ai}$)

- Recherche des Ai tels que $B = \sigma'_i Ai$ , c'est-à-dire $s_{A_i} \subset S_B$ :

Les solutions sont restreintes à l'ensemble:

$$\bigcup_{x \in S_B \cap \bigcup_i S_{Ai}} f(x)$$

(c'est l'ensemble des Ai tels que $s_{Ai} \cap S_B \neq \emptyset$)

*Efficacité:*

Si le domaine du spectre est étendu (densités de $\bigcup_i s_{Ai}$ et $\bigcup_i S_{Ai}$ voisines de 0) et la répartition des nombres uniforme (pas de points d'accumulation), ce procédé élimine d'emblée la quasi-totalité des couples non liés par une substitution*, et constitue donc un moyen d'éliminer le *bruit* au sens où nous l'avons défini précédemment (le démonstrateur n'est pas sensible à la quantité d'hypothèses "irrelevantes").

L'élément dominant du coût n'est pas dans la recherche d'une substitution proprement dite, mais dans la mise à jour des ensembles ( $\bigcup s_{Ai}$, $\bigcup S_{Ai}$), et des fonctions (f,F), qui coûte 0( log n) pour l'introduction d'une nouvelle phrase, donc 0(n log n) pour la constitution d'une base de n phrases.

---

* parce que en général, pour chaque e:
card ( f(e) ) << n (n = nombre d'arbres dans la base)
de même pour chaque E et F(E).

Pour pouvoir appliquer ce procédé, il reste donc à inventer un *"codage spectral"* des arbres ayant les propriétés requises, ce qui est fait ci-dessous.

d/ Spectre exact d'un arbre

Supposons qu'à chaque branche principale de chaque règle opératoire (cf. Syntaxe) on attribue en propre une fonction

$$f_i : \text{ENTIER} \rightarrow \text{ENTIER}$$

et que l'ensemble des fonctions ainsi attribuées ait la *propriété d'exclusivité* suivante:

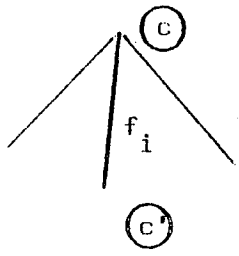
$$\forall\ i\ j\ x\ x' \qquad f_i(x) \neq f_j(x')$$

> C'est une propriété d'inversibilité très forte puisque, connaissant $c = f_i(x)$ , on peut en déduire à la fois $f_i$ et $x$

On peut alors coder tous les noeuds d'un arbre syntaxique de la façon suivante:

. le noeud origine reçoit un code standard.

. tout autre noeud hérite d'un code dépendant du code du noeud amont et de la branche de la règle appliquée.




$$c' = f_i(c)$$

*Lemme:*

Le code d'un noeud représente de façon unique la totalité du chemin de la racine à ce noeud et toutes les règles appliquées en chaque point de ce chemin.

Preuve: en raison de la propriété d'exclusivité, on peut déduire du code c':

- la fonction $f_i$, c'est-à-dire la règle appliquée,
- le code c du noeud amont, et on peut donc itérer la reconstitution du chemin jusqu'à la racine.

*Théorème:*

> L'ensemble des codes aux noeuds terminaux (feuilles) d'un radical représente (en général de façon redondante) la totalité de ce radical non interprété.

(Ceci vaut en particulier si on prend pour radical l'arbre entier).

Ceci résulte immédiatement du lemme.

*Spectre de l'arbre A*

Le spectre $\langle s_A ; S_A \rangle$ de l'arbre A est défini par

- $s_A$ = ensemble des codes aux noeuds terminaux du radical principal de A ;
- $S_A$ = ensemble de tous les codes aux noeuds de A.

On voit que $s_A \subset S_A$ et que le spectre est une caractérisation très redondante de A .

(Noter bien sûr que deux arbres non identiques ont nécessairement des spectres différents).

Enfin et surtout, les spectres ainsi définis ont toutes les propriétés requises pour l'application du procédé de "pattern matching" d'arbres décrit au §c précédent.

Malheureusement, il n'est pas facile de trouver un ensemble de fonctions ayant la propriété d'exclusivité annoncée (c'est en particulier impossible si le domaine des codes est borné). Ceci nous oblige donc à corrompre cette définition trop rigoureuse des spectres:

e/ Spectre approché d'un arbre

Le spectre approché sera obtenu exactement de la même façon que le spectre exact, mais la restriction sur les fonctions sera affaiblie ; les fonctions $f_i$ devront être:

. à valeur dans un *ensemble fini*: les entiers de l'intervalle [0,N-1]

. à *distribution plate*, ce qui veut dire que:

$\forall i$ { si la distribution de c dans [0,N-1] est plate, alors la distribution de $f_i(c)$ dans [0,N-1] est plate,

ou, de façon équivalente:

$\forall i$ la fonction $f_i(c)$ est *surjective*,

ou encore:

$\forall i$ la fonction $f_i$ est une *permutation* de [0,N-1].

. *indépendantes*, ce qui veut dire que:

en prenant au hasard i, i' et c, la probabilité pour que $f_i(c) = f_{i'}(c)$ est la même que pour deux permutations arbitraires, c'est-à-dire 1/N. En effet, on peut montrer que le nombre de points communs à deux permutations arbitraires, ou - de façon équivalente - le nombre de *points fixes* d'une permutation arbitraire est en moyenne 1 (avec une déviation standard 1).

On voit que le lemme et le théorème précédents ne sont plus valables en raison de la possibilité de *collision:*
Il y a collision lorsque $f_i(c) = f_{i'}(c)$ avec $(i,c) \neq (i',c)$.

*Evaluation du risque de collision:*

Avec les hypothèses indiquées sur les fonctions $f_i$, l'utilisation des résultats établis en Annexe III indique que, si le domaine des $f_i$ a N valeurs et si un total de k codes ont été calculés, le risque d'avoir au moins une collision est

$$\boxed{\varepsilon \simeq \frac{k^2}{2N}} \qquad \text{pour } N >> k^2$$

*Conséquence d'une collision*

Les arbres n'étant reconnus que par leur spectre (donc par des codes) une collision *peut* entraîner la détection d'une fausse substitution, mais ce sera rarement le cas pour trois raisons:

- pour qu'il y ait substitution, il faut aussi que le sous-but associé aux "appendices" soit vérifié.
- comme indiqué au § d, le spectre est très redondant et une seule collision ne suffira généralement pas à provoquer une erreur.
- le code incriminé étant commun à un $s_A$ et un $S_B$, l'erreur ne peut se produire que si tout le reste de $s_A$ est inclus dans $S_B$ ce qui serait étonnant.

Le risque d'une fausse preuve peut donc être considéré comme certainement *très inférieur à* $\frac{k^2}{2N}$.

*Contrôle du risque*

Si l'on veut limiter à $\varepsilon$ le risque au cours d'une session où interviendront k noeuds d'arbres au maximum, il faut choisir $N > \frac{k^2}{2\varepsilon}$.

Par exemple, si l'on estime que le nombre total de codes sera inférieur à 100 000 et qu'on désire un risque global d'erreur limité à $10^{-4}$, il suffit que l'amplitude du domaine soit $N = 5 \times 10^{13}$.

Si N est trop grand pour la taille du mot machine, on peut utiliser des codes vectoriels, le nombre de composantes croissant proportionnellement à $\log \frac{1}{\varepsilon}$.

f/ ε-démonstrateur

Il est donc possible de limiter le risque d'erreur au cours d'une session à un taux ε *arbitrairement faible*, et pour un facteur de coût de seulement $\log \frac{1}{\varepsilon}$ .

Deux options sont alors possibles:

. Refus de tout risque, si minime soit-il: chaque substitution détectée au moyen de spectres doit être vérifiée directement sur les arbres. Dans ce cas, il est inutile (et même coûteux) de chercher à limiter ε à des valeurs très faibles ;

. Accepter un risque (par exemple comparable au risque d'erreur due à une panne matérielle).

*Remarque: "Principe d'Incertitude"*

Il me semble que la preuve de théorèmes est soumise à un principe d'incertitude, qu'il faudrait préciser, selon lequel on ne peut dépasser un seuil de rapidité qu'au prix d'une diminution de fiabilité.

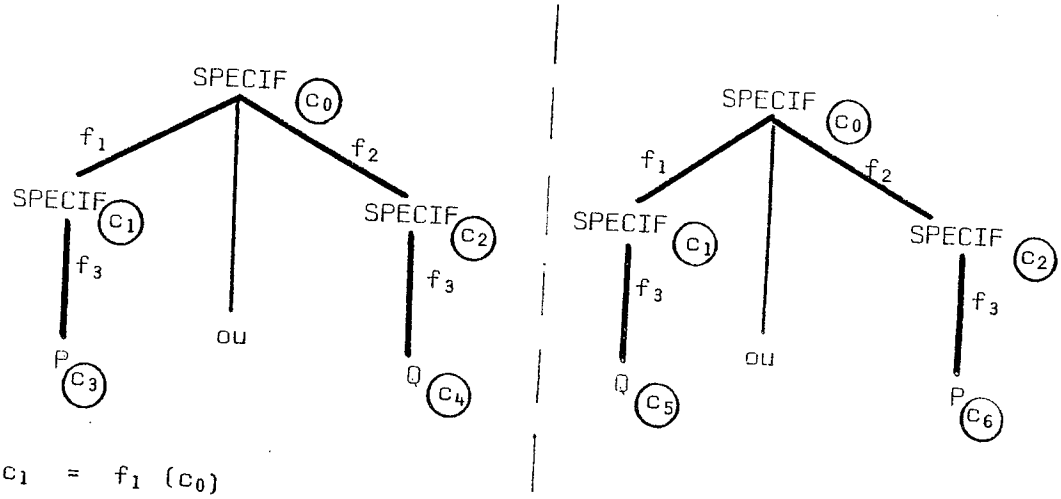
*exemple classique:* On trouve plus facilement la preuve d'un théorème de géométrie avec une figure que par des moyens formels, mais une figure peut aussi induire en erreur.

Ici, le spectre approché d'une phrase joue le même rôle que les figures en géométrie: il facilite la découverte d'une preuve tout en introduisant un risque d'erreur.

g/ "pattern-matching" amélioré par les codes homomorphes

On a exigé jusqu'ici que toutes les fonctions de codage $f_i$ associées aux branches des règles soient différentes, et même indépendantes.

Dans ces conditions, les arbres suivants seront codés comme indiqué (dans les ronds):




avec $c_1 = f_1(c_0)$

$c_2 = f_2(c_0)$

$c_3 = f_3(c_1)$

$c_4 = f_3(c_2)$

$c_5 = f_3(c_1)$

$c_6 = f_3(c_2)$

Ces arbres ne seront donc pas reconnus comme identiques puisque

$$\{c_3, c_4\} \neq \{c_6, c_5\}$$

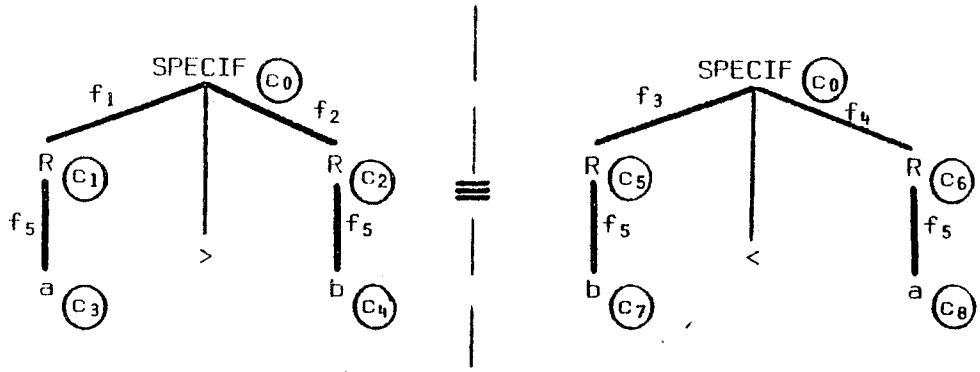
Or il aurait suffi de choisir $f_1$ identique à $f_2$ pour que les deux arbres aient le même spectre.

La *commutativité* d'un opérateur peut donc être directement exprimée par l'attribution de fonctions de codage identiques aux deux branches de la règle opératoire associée.

De la même manière, d'autres propriétés algébriques des opérateurs correspondant aux règles opératoires peuvent se traduire directement par le choix des fonctions de codage.

*exemples:*

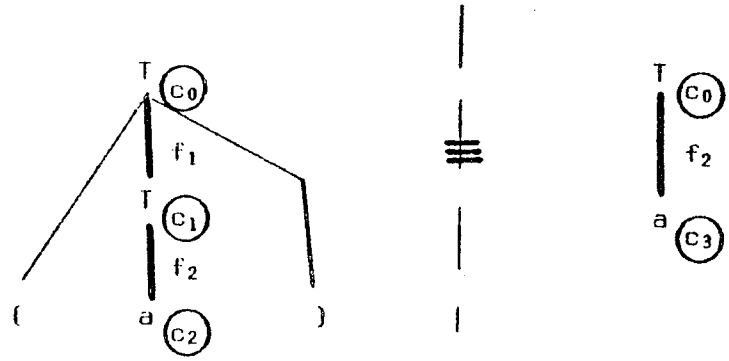
- la *réciprocité* de deux opérateurs:




se traduit par le choix $\begin{cases} f_1 \equiv f_4 \\ f_2 \equiv f_3 \end{cases}$

qui assure $c_3 = c_8$ et $c_4 = c_7$

- la *nullité* d'un opérateur:




se traduit par le choix $f_1 \equiv$ identité

qui assure $c_2 = c_3$

*Vocabulaire:*

L'attribution de fonctions de codage traduisant des propriétés d'opérateurs constitue ce que l'on appellera ici un procédé de *codage homomorphe*.

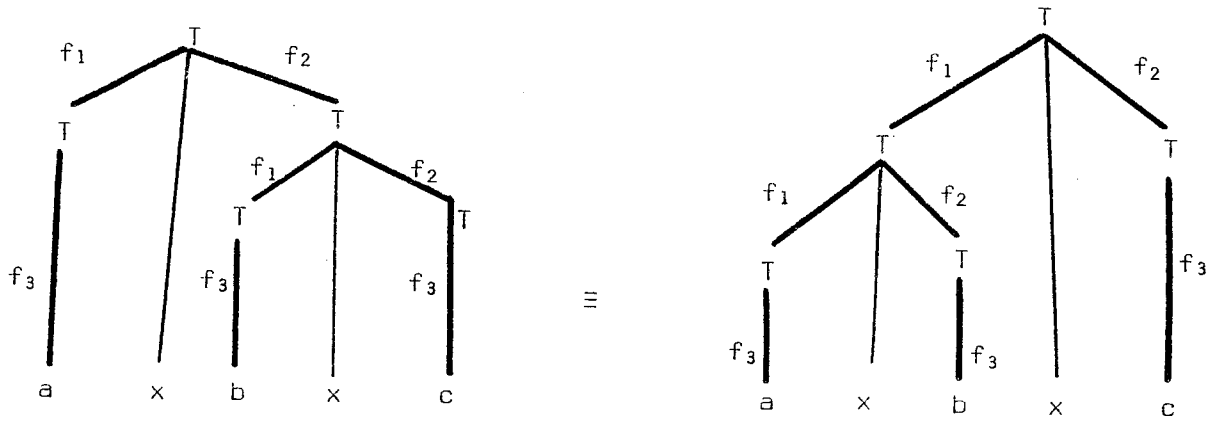
*Intérêt:*

Ce procédé permet, sans aucun coût supplémentaire, d'identifier des arbres (ou radicaux) dont l'équivalence est souvent très coûteuse à établir. On évite ainsi la nécessité de mise sous forme canonique ou le recours à des étapes de preuves supplémentaires (augmentation de l'autonomie).

*Difficultés:*

- Certaines propriétés algébriques sont plus difficiles à traduire en codes homomorphes, par exemple:

- l'*associativité* d'un opérateur:




devrait se traduire par un choix de $f_1, f_2$ tels que:

$$f_1 \circ \begin{cases} f_1 \equiv f_1 \circ f_1 \\ f_2 \equiv f_2 \circ f_1 \\ f_2 \equiv f_2 \circ f_2 \end{cases}$$

Un tel système est facile à résoudre: prendre deux projections orthogonales $p_1$ $p_2$ et une fonction inversible g, on a alors les solutions:

$$f_1 = g \circ p_1 \circ g^{-1}$$
$$f_2 = g \circ p_2 \circ g^{-1}$$

Malheureusement, les contraintes sur $f_1, f_2$ sont trop fortes et traduisent *plus* que l'associativité puisqu'elles feraient reconnaître comme identiques les expressions:

ax(bxc)xd et ax(cxb)xd

Une meilleure solution semble donc être de traduire l'associativité par la transformation de l'opérateur binaire en opérateur n-aire (cf. p.71). Dans ce cas, les substitutions seront légèrement plus difficiles à effectuer (interdiction des occurrences consécutives de l'opérateur).

- La principale difficulté consiste à:

1° traduire *automatiquement* certaines propriétés des opérateurs en systèmes d'équations fonctionnelles entre des $f_i$ ;

2° et surtout, générer *automatiquement* des jeux de fonctions satisfaisant ces équations: voir la solution proposée ci-dessous(§ h).

- Une autre difficulté est que certaines contraintes sur les fonctions (exemple: $f_1 \equiv f_1 \circ f_1$) interdisent qu'elles soient surjectives. Il faut en tenir compte dans l'évaluation du risque de collision.

- Une difficulté moins sérieuse est que, lorsque des arbres sont ainsi déclarés équivalents, on ne peut plus le vérifier directement (par parcours des arbres en parallèle), donc le mieux est d'accepter le très faible risque de fausse identification due à une collision (cf. § e): une *perte minime de fiabilité permet un gain important d'autonomie* (comparer avec Rabin[46]).

h/ Fonctions de codage dans un corps $\mathbb{Z}/N$

Etant données toutes les contraintes auxquelles sont soumises les diverses fonctions du codage $f_i(c)$, on peut se demander s'il est effectivement possible de trouver de telles fonctions.

Rappelons ces contraintes:

- fonctions le moins coûteuses possibles à calculer,
- fonctions surjectives: $\forall i \quad f_i([0,N-1]) = [0,N-1]$
- fonctions indépendantes:
  pour $i \neq i'$, l'équation $f_i(x) = f_{i'}(x)$ n'a en moyenne qu'une solution.
- possibilité de choisir des fonctions satisfaisant entre elles certains systèmes d'équations fonctionnelles,
- possibilité d'introduire dynamiquement de nouvelles fonctions.

Heureusement, pour N premier, Z/N est un corps dont les propriétés permettent de générer automatiquement de telles fonctions de codage:

α/ *Fonctions linéaires non dégénérées*

Dans le corps Z/N, les fonctions linéaires non dégénérées sont les fonctions $f_i(c)$ de la forme $p_i c + q_i \pmod N$, avec $p_i \neq 0$.

Toutes ces fonctions satisfont bien les trois premières contraintes ci-dessus:

- elles se calculent très rapidement,
- elles sont *surjectives*,
- si $p_i \neq p_j$, les fonctions $p_i c+q_i$ et $p_j c+q_j$ sont *indépendantes* car l'équation $p_i x+q_i = p_j x+q_j$ n'a qu'une solution.

De plus, on trouve facilement des fonctions satisfaisant aux contraintes intervenant pour les codages homomorphes:

. fonction *identité:*

On peut l'appeler fo, et elle est caractérisée par $p_0=1$, $q_0=0$.

. fonctions $f_i$ *(symétries)* telles que $f_i \circ f_i$ = identité.
Elles sont caractérisées par l'identité $p_i(p_i c+q_i)+q_i \equiv c$

donc par $\begin{cases} p_i = 1 \\ p_i q_i+q_i = 0 \end{cases}$ qui est équivalent à $\begin{cases} p_i = -1 \\ q_i \text{ quelconque} \end{cases}$

. fonctions *inverses* $f_i \circ f_j$ = identité

Elles sont carcatérisées par l'identité $p_i(p_j c+q_j)+q_i \equiv c$

donc par $\begin{cases} p_i p_j = 1 \\ p_i q_j+q_i = 0 \end{cases}$ qui se résoud en prenant $\begin{cases} p_j = 1/p_i \\ q_j = -q_i/p_i \end{cases}$

*Remarque:*

La division par $p_i$ est équivalente à la multiplication par $p_i^{N-2}$ qui se fait au pire en $2 \log_2 N$ multiplications.
On trouve donc l'inverse d'une fonction $f_i$ en O(logN) opérations.

. fonctions *commutatives* $f_i \circ f_j = f_j \circ f_i$

Elles sont caractérisées par l'identité $p_i(p_j c+q_j)+q_i \equiv p_j(p_i c+q_i)+q_j$

donc par $p_i q_j+q_i = p_i q_i+q_j$ ,

qui se résoud (pour $p_i \neq 1$) en prenant: $\begin{cases} p_j \text{ quelconque} \\ q_j = (p_j-1)q_i/(p_i-1) \end{cases}$

On trouve donc une fonction $f_j$ commutant avec une fonction $f_i$ en O(logN) opérations.

. r fonctions liées par un *système* de s *"équations fonctionnelles"* (s < r)

Une équation fonctionnelle de degré d est l'égalité de deux produits de composition dont le plus long comporte d fonctions.

Exemple: $f_i \circ f_j = f_i \circ f_k \circ f_i$ est de degré 3.

Chaque équation fonctionnelle de degré d donne lieu à deux équations polynomiales de degré d sur les coefficients $p_i, q_i$ des fonctions $f_i$.

Dans un corps Z/N, on sait résoudre de telles équations, même de degré élevé.

s équations fonctionnelles indépendantes entre r fonctions donnent donc 2s équations polynomiales entre 2r variables $p_i, q_j$.

Si $s < r$, on trouve donc en général une *famille de solutions* non triviales.

Mais si $r \geq s$ on ne trouve en général pas de solutions (et il n'existe souvent même pas de fonctions surjectives non triviales satisfaisant à ces équations).

Exemple: le système (r=1, s=1) : $\{f_i \circ f_i = f_i$

n'a que la solution triviale $f_0$ (identité).

*β/ Fonctions linéaires dépendant d'un seul paramètre*

Au lieu de définir les fonctions par leurs deux paramètres p et q, il serait intéressant de définir deux fonctions p(i), q(i) de façon que les équations fonctionnelles sur les fonctions $f_i(c) = p(i)c+q(i)$ se ramènent à des équations simples sur les seuls indices i.

C'est ce qui est fait ci-dessous:

Une famille de fonctions très remarquable (exemple §δ)

Tout nombre premier N est de la forme N=rs+1

Soit e une *racine primitive* de Z/N:

les éléments non nuls de Z/N sont $\{e, e^2, \ldots, e^{N-1}=1\}$

et l'équation $x^r=1$ a r racines: $\{e^0=1, e^s, \ldots, e^{(r-1)s}\}$

Considérons la famille de fonctions linéaires:

$$\begin{cases} \boxed{f_i(c) = (1+e^{(r+1)i})c + e^i} \\ f_0(c) = c \end{cases} \quad i = 1,2,\ldots,N-1$$

Posons $d = PGCD(r+1,rs) = PGCD(r+1,s)$ ,

On a donc $r+1=\alpha d$, $rs=\beta d$ et $PGCD(\alpha,\beta)=1$.

Ces N fonctions linéaires sont différentes et ont les propriétés suivantes:

- existence de *fonctions dégénérées*

On montre facilement que $f_i$ est dégénérée ssi $i = (2k+1)\frac{\beta}{2\alpha}$

> Donc, *si β est impair* (c'est-à-dire r+1 a au moins autant de facteurs 2 que s),
>
> *il n'y a aucune fonction dégénérée.*

Si, au contraire, β est pair, il y a d fonctions dégénérées, correspondant

à $\boxed{i = (2k+1)\frac{rs}{2d}}$ $k = 0,1,\ldots,d-1$.

- recherche des fonctions (symétries) $f_i$: $f_i \circ f_i = f_0$

On montre facilement qu'elles correspondent aux solutions de l'équation:

$$e^{(r+1)i} = -2$$

Soit μ tel que $e^{\mu} = -2$

l'équation est équivalente à $(r+1)i = krs+\mu$

donc à $\alpha d i = k\beta d+\mu$

> qui n'a de solution que si μ est un multiple de d ;
>
> ce qui est équivalent à: $\boxed{(-2)^{rs/d} = 1}$

Il y a alors d solutions en progression arithmétique:

$$\boxed{i = i_0+k\frac{rs}{d}} \quad k = 0,1,\ldots,d-1$$

une équation $f_i \circ f_i = 0$ se résoud donc en:

1 opération (choix de k),

0 opération pour le calcul de p(i) qui vaut -1,

0 (logk) opérations pour le calcul de $q(i)=e^{i\theta}(e^{rs/d})^k$

(Il y a donc intérêt à choisir k petit).

- Recherche d'une fonction $f_j$ : $f_j \circ f_i = f_i \circ f_j$ —

Elle correspond à $q(j) = (p(j)-1)\, q(i)/(p(i)-1)$

qui se réduit à $e^{r(j-i)}=1$

et a r solutions en progression arithmétique:

$$\boxed{j = i+ks} \qquad k = 0,1,\ldots,\ r-1 \pmod{N-1}$$

Une équation $f_i \circ f_j = f_j \circ f_i$ se résoud donc en:

1 opération (choix de k),

0 (logk) opérations pour le calcul de $p(j) = 1+(p_i-1)(e^{(r+i)s})^k$

(d'où intérêt de choisir k petit)

4 opérations pour le calcul de $q(j) = (p_j-1)q(i)/(p_i-1)$

C'est donc *beaucoup moins coûteux* que O(logN) trouvé pour la classe de toutes les fonctions linéaires.

Remarque:

On ne peut spécifier que s fonctions qui *ne commutent pas* deux à deux, il faut donc que s *soit suffisamment grand.*

- Recherche d'une fonction inversible: $f_j \circ f_i = f_0$

On peut montrer, par des calculs franchement fastidieux, que $f_i$ a un inverse $f_j$, si et seulement si:

$1 + e^{(1+r)i} = -e^{ks}$, et dans ce cas: $j = i-ks$

pour k=0 on retrouve les $\underline{d}$ fonctions $f_i$ telles que $f_i \circ f_i = f_0$ ;
pour chaque autre $\underline{k} \in \{1,2,\ldots r-1\}$, le nombre de solutions est soit 0, soit $\underline{d}$ si et seulement si $(-1-e^{-ks})^{rs/d} = 1$ ; cette égalité n'ayant qu'une probabilité $\frac{1}{d}$ d'être vraie.

Il faut donc s'attendre à *environ* $\frac{r}{d} \times d = r$ fonctions inversibles.

Coût moyen:

Il faut *en moyenne* essayer $\underline{d}$ valeurs de k pour en trouver une "qui marche", et on a alors $\underline{d}$ solutions à la fois.
Une solution coûte donc *en moyenne* un essai.
Son coût O(log N) est comparable au coût du calcul de l'inverse d'une fonction linéaire arbitraire.

- Avant de montrer un exemple §δ, nous allons voir quels sont les nombres premiers N qui ont les propriétés requises pour l'application de cette technique très puissante:

*γ/ Les "bons" nombres premiers*

Cette dernière technique de codage homomorphe n'est intéressante qu'à condition qu'on sache trouver des couples r,s tels que (en posant d = PGCD(r+1,s) ):

- rs+1 soit premier
- s/d soit impair (pour éviter les fonctions dégénérées)
- $(-2)^{rs/d} = 1 \bmod (rs+1)$ (pour trouver des $f_i \circ f_i = f_0$)

Les nombres premiers rs+1 ayant cette propriété sont ici appelés *"bons"* nombres premiers.

Remarque: *Tout nombre premier est bon.*

Il suffit en effet, pour le nombre premier N, de prendre:

$r = N-1$, $s=1$ (donc $d=1$)

et les trois conditions sont immédiatement remplies.

Malheureusement, ce cas est trop particulier car:

- on ne trouve que d=1 fonction $f_i \circ f_i = f_0$
- comme s=1, tous les couples de fonctions commutent (ce qui est fâcheux).

Ceci n'est pas étonnant, puisque la famille de fonctions est la famille des *translations*. $f_i(c) = c+e^i$.

Si, dans l'application considérée, on envisage d'avoir besoin de:

- $d_0$ fonctions $f_i$ telles que $f_i \circ f_i = f_0$,
- $s_0$ fonctions (qui en général ne commuteront pas),
- $r_0$ fonctions inversibles,

Alors il faudra que r et s aient en plus, les propriétés suivantes:

- $d \geq d_0$
- $s \geq s_0$
- $r \geq r_0$

Dans ce cas, le nombre rs+1 sera appelé un
*"très bon" $(d_0,s_0,r_0)$-nombre premier*

Conjecture:

*Pour tous $d_0,s_0,r_0$ : il existe une infinité de très bons $(d_0,s_0,r_0)$-nombres premiers.*

Seuls des arguments probabilistes fondés sur la répartition des nombres premiers me permettent de penser que cette conjecture est vraie.

(Plus précisément, on peut trouver un tel nombre N au voisinage de $X >> r_0 s_0$, en un temps *moyen* $O(d_0 \log^2 X)$ ).

*δ/ exemple de codage utilisant un "très bon" nombre premier*

19 est un très bon (2,6,3)-nombre premier.

Il suffit en effet de prendre r=3, s=6 (donc d=2).

2 est racine primitive de Z/19 dont les éléments non nuls sont:

| $2^i$ : | 2 | 4 | 8 | 16 | 13 | 7 | 14 | 9 | 18 | 17 | 15 | 11 | 3 | 6 | 12 | 5 | 10 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 1 |

et $x^3=1$ a les racines $\{2^0=1 , 2^6=7 , 2^{12}=11\}$

La famille de fonctions est:

$$f_0(c) = c \quad \text{et} \quad \boxed{f_i(c) = (1+2^{4i})c + 2^i} \qquad i=1,\ldots,18$$

- fonctions $f_i \circ f_i = f_0$: il y en a d=2 : $f_7$ et $f_{7+9}$

en effet, $f_7(c) = (1+2^{28})c+2^7 = 18c+14$

$f_{16}(c) = (1+2^{64})c+2^{16} = 18c+5$

et $f_7 \circ f_7(c) = 18(18c+14)+14 = -1(-1c+14)+14 = c$

$f_{16} \circ f_{16}(c) = 18(18c+5)+5 = c$

- fonctions commutant avec $f_7$: il y en a r=3 : $f_7$, $f_{7+6}$, $f_{7+12}$

en effet, $f_7 \circ f_7 = f_7 \circ f_7$ évidemment

$f_{13}(c) = (1+2^{52})c+2^{13} = 6c + 3$

et $\begin{cases} f_{13} \circ f_7(c) = 6(18c+14)+3 = 13c+11 \\ f_7 \circ f_{13}(c) = 18(6c+3)+14 = 13c+11 \end{cases}$

enfin $f_{19}$, c'est-à-dire $f_1(c) = (1+2^4)c+2 = 17c+2$ commute également avec $f_7$.

- fonctions ne commutant pas:

On peut vérifier qu'aucun des 15 couples de $\{f_1\ f_2\ f_3\ f_4\ f_5\ f_6\}$ ne commute.

- fonctions inversibles:

Il y en a d=2 pour chaque $k \in \{0,1,2\}$ tel que $(-1-2^{-6k})^9 = 1$.

Nous avons la chance que 0, 1, 2 soient tous trois des solutions, on trouve donc 6 fonctions inversibles:

- fonctions inversibles:

pour k=0 : $f_7$ $f_{16}$

pour k=1 : $f_3$ $f_{12}$

pour k=2 : $f_6$ $f_{15}$

*ε/ Autres fonctions dans Z/N*

La classe de fonctions $f_i(c) = q_i\ c^{p_i}$ $(p_i q_i \neq 0)$ a été également étudiée, et a des propriétés analogues à la classe des fonctions linéaires non dégénérées. Certaines contraintes sont plus faciles à résoudre, mais le calcul des fonctions est plus long: O(log N) opérations, au lieu de deux opérations.

*ζ/ [Fonctions dans d'autres corps finis]*

Les autres corps finis sont d'ordre $N^k$, N premier.
Il serait intéressant de voir si leurs propriétés permettent de trouver de bonnes fonctions de codage, notamment pour des équations fonctionnelles n'ayant pas de solutions surjectives.

## 4. Amélioration de l'autonomie (II)

On a vu qu'un démonstrateur peut fonctionner selon deux modes:

- mode exploratoire, de coût $\alpha_1 e^{\beta_1 d}$ ;
- mode inverse (recherche de sous-buts) , de coût $\alpha_2 e^{\beta_2 d}$ .

Nous allons voir ici qu'un mélange judicieux de ces deux modes de fonctionnement améliore nettement l'autonomie du démonstrateur.*

Pour une conjecture de difficulté d, si l'on décide de parcourir une partie $\lambda d$ du chemin en mode exploratoire, et le reste $(1-\lambda)d$ en mode inverse, le temps nécessaire à la preuve sera en moyenne:

$$\alpha_1 e^{\beta_1 \lambda d} + \alpha_2 e^{\beta_2 (1-\lambda) d}$$

On peut aisément vérifier que:

a/ le temps minimum est obtenu pour le choix:

$$\lambda(d) = \frac{\beta_2}{\beta_1 + \beta_2} + \frac{1}{d} \frac{\log(\alpha_2 \beta_2) - \log(\alpha_1 \beta_1)}{\beta_1 + \beta_2}$$

(qui tend vers $\frac{\beta_2}{\beta_1 + \beta_2}$ pour $d \to \infty$)

b/ que ce $\lambda(d)$ optimal est obtenu en donnant:

- à la méthode exploratoire un temps proportionnel à $\beta_2$ ,
- à la méthode inverse un temps proportionnel à $\beta_1$ ,

(la méthode la plus rapide dispose de plus de temps).

* L'argument utilisé pourrait servir aussi à la résolution de problèmes de labyrinthes.

c/ ce temps minimum est de la forme:

$$L\, e^{Md} \quad \text{avec} \quad M = \frac{\beta_1 \beta_2}{\beta_1 + \beta_2}$$

d/ l'autonomie est donc portée à:

$$\frac{\log T_{max} - \log L}{M} \sim \frac{\log T_{max}}{M}$$

c'est-à-dire, approximativement à la *somme des autonomies* des deux méthodes $\frac{\log T_{max}}{\beta_1} + \frac{\log T_{max}}{\beta_2}$.

(puisque $\frac{1}{M} = \frac{1}{\beta_1} + \frac{1}{\beta_2}$ )

*Application:*

Ce calcul indique qu'on peut *doubler l'autonomie* moyenne des deux modes (exploratoire et inverse) en les faisant alterner avec un rapport de durée $\frac{\beta_2}{\beta_1}$.
Ceci soulève deux questions:

a - comment évaluer $\beta_1$ et $\beta_2$ afin d'ajuster au mieux le temps imparti à chaque méthode?

. une évaluation analytique semble difficile car elle suppose qu'on connaisse bien les caractéristiques géométriques du latticiel de preuve (cf. annexe II).

. une évaluation expérimentale semble plus indiquée.

. en l'absence d'évaluation, on peut partir de l'hypothèse $\beta_1 = \beta_2$ qui est certainement assez proche de la réalité.

b - quelle doit être la *fréquence de l'alternance ?*

L'expérimentation "sur le papier" semble indiquer qu'elle doit être rapide: lancement du mode exploratoire après chaque sous-but.

## 5. Amélioration de l'autonomie (III): heuristiques

- Chaque déduction (application d'une règle d'inférence) implique l'introduction d'un nouveau théorème ; le nombre n d'éléments de la base de spécifications a donc tendance à croître proportionnellement au temps écoulé, avec les conséquences suivantes:

a/ tôt ou tard la mémoire disponible est saturée ;

b/ le coût de recherche des nouvelles substitutions (avec la méthode rapide proposée §3) augmente (comme log n) ;

c/ le nombre de déductions possibles augmente (accroissement du facteur $\beta$ et diminution de l'autonomie).

- Il est donc nécessaire d'*éliminer des théorèmes* de la base de spécifications, ce qui appelle deux remarques:

a/ en éliminant des théorèmes on élimine les conséquences qu'ils pourraient avoir ultérieurement, donc on *réduit la portée* du démonstrateur: c'est le prix qu'il faut payer pour améliorer l'autonomie (cf. § 1.C.$\alpha$) ;

b/ il ne faut pas supprimer n'importe quels théorèmes: certains (les lemmes "passe-partout") sont très souvent utiles et leur suppression compliquerait gravement la preuve de nombreuses conjectures. On ne veut supprimer que les théorèmes peu "utiles" (ceux qui ont le moins de chances de servir à prouver des conjectures "intéressantes"). Naturellement, seuls des critères heuristiques permettent de comparer l'utilité des théorèmes. Ce qui suit est une tentative de mesurer cette "utilité".

- *Utilité d'un théorème:*

A chaque théorème, on peut attacher quatre quantités:

. d : sa *difficulté* par rapport au reste de la base:
comme celle-ci est difficile à déterminer, on peut y substituer le nombre d'étapes qui ont conduit à sa preuve initiale ;

. g : son *degré de liberté*, ou nombre de variables et de types libres dont il dépend ;

. l : sa *longueur:* longueur du texte de son énoncé ou nombre de noeuds de l'arbre syntaxique de ce texte ;

. f : sa *fréquence* d'utilisation.

L'utilité d'un théorème devra être définie par une fonction *croissante* de (d, g, -l, f):

. *critère d élevé:* si on supprime un théorème de difficulté élevée, il sera difficile de le retrouver en cas de nécessité ;

. *critère g élevé:* toutes choses égales d'ailleurs, un lemme ayant plus de variables et de types libres est plus général (s'appliquant à un domaine plus étendu) ;

. *critère l faible:* un théorème court occupe moins de place, il est donc moins urgent de le supprimer ;

. *critère f élevé:* un lemme qui intervient souvent dans de nombreuses preuves est vraisemblablement plus utile.

La fonction croissante d'utilité U(d, g, -l, f) est sensée traduire l'importance relative de ces quatre critères.

Je ne me hasarderai pas ici à en proposer une: un bon compromis entre ces quatre critères ne peut être déterminé que de façon expérimentale.

## 6. Conclusion

Malgré le coût exponentiel des démonstrations automatiques, les techniques présentées ici indiquent qu'on peut obtenir des démonstrateurs de théorèmes efficaces à condition:

- d'accepter qu'ils soient interactifs ;
- que les théorèmes puissent être exprimés dans un langage très évolué ;
- d'accepter un "principe d'incertitude" exprimant qu'un gain en rapidité s'accompagne nécessairement d'une perte (comparativement minime) de fiabilité.
- de ne pas chercher à tout prix une trop grande complétude (qui coûte cher).

# IV

# UTILISATION - PROLONGEMENTS

# CHAPITRE IV

# UTILISATION - PROLONGEMENTS

## I - MISE EN OEUVRE DU LANGAGE

### 1. Problèmes de performance

L'implémentation de tout ou partie du langage ne saurait être entreprise sans une bonne compréhension préalable des performances possibles:

a/ effet de maquette

J'attire tout particulièrement l'attention sur un phénomène simple dont la méconnaissance trop fréquente a transformé certains grands projets informatiques en désastreux abîmes financiers (ex. Robotique jusqu'en 1972).

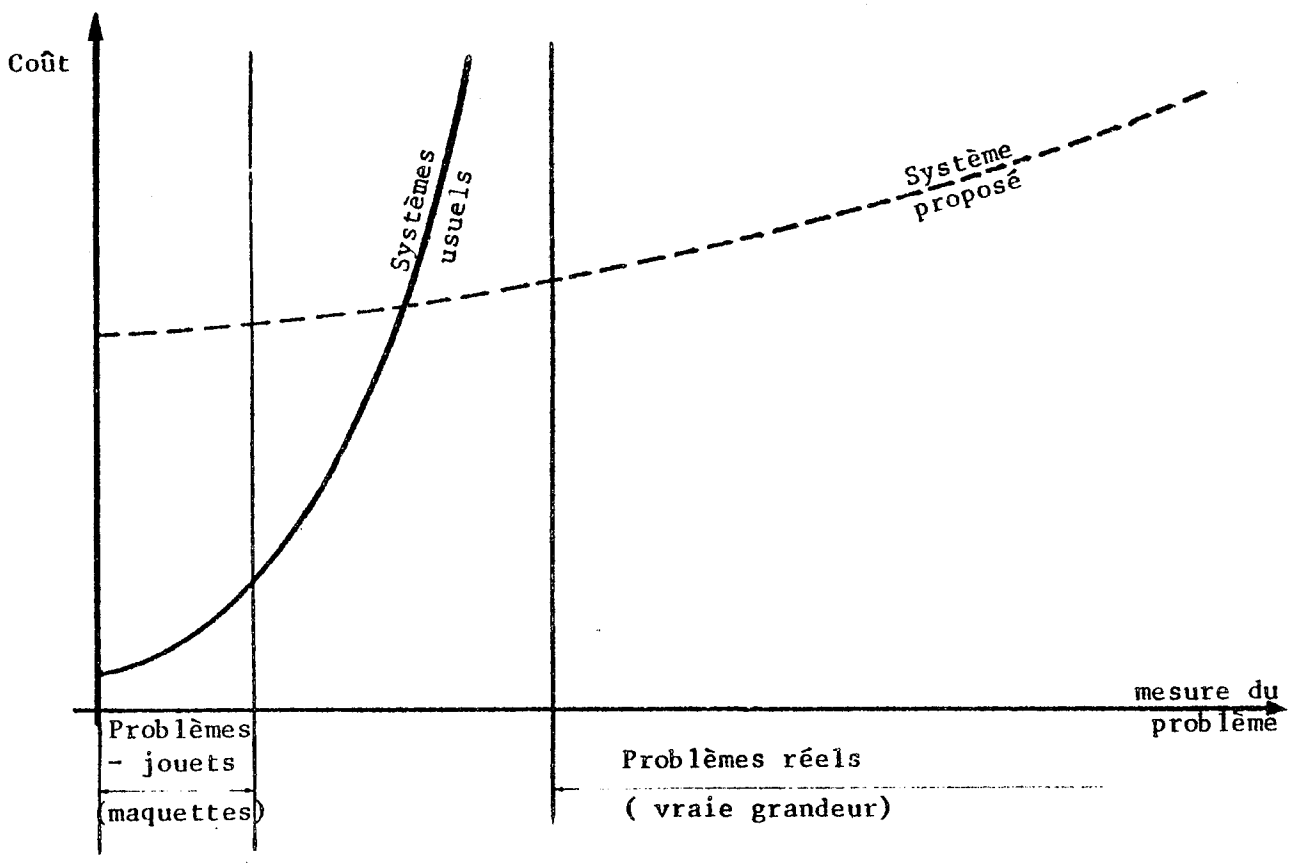



Figure 5 Effet de maquette

La figure 5 ci-jointe illustre cet *effet de maquette* que l'on peut très approximativement énoncer comme suit:

[Pour presque toutes les classes de problèmes], *on ne peut réduire le coût des cas difficiles sans accepter d'augmenter le coût des cas simples.*

Ceci implique les deux paradoxes (?) suivants:

Corollaire 1: (Avis aux distributeurs de contrats)

*Si l'on sélectionne les solutions selon leurs performances sur une maquette, on est sûr de ne pas choisir les meilleures.*

Corollaire 2:

*Le langage ici proposé ne doit en aucun cas être évalué sur une maquette.*

Il y apparaîtrait en effet moins performant que des systèmes pourtant moins efficaces.

b/ état initial et divergence

Jusqu'ici, il n'a été qu'occasionnellement fait référence au *NOYAU* du langage, c'est-à-dire à l'état initial:

- de sa syntaxe:
  - types et constructeurs de types,
  - opérateurs,
  - identifieurs,
- de sa sémantique:
  - axiomes standard de la base de spécifications.

Il existe un *"noyau théorique"* sans lequel le système ne peut même pas fonctionner du tout.

En particulier, ce noyau doit absolument contenir:

- les moyens d'extension du langage, c'est-à-dire au moins une phrase typique pour chaque sorte d'extension (cf. p. 72/81) faute de quoi l'analyse syntaxique ne convergera pas.

- les moyens d'initialiser la déduction, c'est-à-dire les règles opératoires relatives aux opérateurs logiques et les axiomes associés.

Il est clair qu'un tel noyau minimum serait très insuffisant, car un utilisateur serait très rapidement découragé d'utiliser le système, si celui-ci ne connait pas, dès le départ, les notations les plus courantes et leur signification, ainsi que les théorèmes usuels que tout le monde applique (information *"pragmatique"*).

L'*état initial* du langage doit donc contenir une vaste quantité d'information que seule une longue expérience permettra de rassembler. C'est là la principale difficulté d'implémentation du langage.

Pour ce langage, comme pour tout autre se voulant utile à l'intelligence artificielle, il faudra:

- recenser sur une variété d'applications les opérateurs, types et lemmes les plus fréquemment utilisés dans la pratique. Cette tâche promet d'être assez longue, car on utilise en permanence des lemmes essentiels sans s'en rendre compte (exemple: principe du pigeonnier: s'il y a plus de n pigeons dans n cases, une case au moins contient deux pigeons au moins).

- après un certain temps, utiliser le système lui-même, pour que, par ses questions, il montre ce qu'on a oublié d'inclure dans son état initial ;

- corriger les insuffisances techniques du langage (notamment dans le mécanisme d'extension) qui ne manqueront pas d'apparaître.

c/ autonomie du démonstrateur

On a vu (chapitre III) qu'elle est directement liée à l'efficacité du système. Il serait donc utile d'en avoir une estimation a priori.
La technique de reconnaissance de phrases par codage spectral des arbres a été expérimentée avec succès, mais je reste persuadé qu'on peut trouver un procédé de codage encore plus efficace surtout dans ses caractéristiques

homomorphes. En remplaçant le procédé de codage dans un corps Z/N, tel qu'il est décrit ici, par un procédé analogue dans des corps d'ordre $p^k$ (p premier), j'espère rendre possible la reconnaissance instantanée d'une classe très étendue de théorèmes d'équivalence.

### 2. Coût d'implémentation

Dans sa version décrite ici, le coût d'implémentation du langage nu devrait se situer entre 7 et 15 programmeurs-années.Il s'y ajoute le coût de détermination du noyau qui dépend comme on l'a vu de la masse critique nécessaire à la divergence du système. Il peut aller de 5 à 10 hommes-années selon la difficulté de ce problème expérimental.

L'effort est donc important et seule l'étendue de ses applications peut le justifier.

## II - APPLICATIONS

### 1. Programmation assistée par ordinateur

Rappelons que l'objectif initial de ce langage était de servir de support à un système de programmation assistée par ordinateur, permettant de certifier les programmes, y compris en cas de maintenance, transport et modifications.

Le langage permet d'exprimer, non seulement une spécification de tout problème (limites de l'environnement y compris les hypothèses de pannes, moyens disponibles, fonctions et performances à réaliser, accidents à éviter), mais aussi les raisonnements que le concepteur utilise pour justifier la suite de transformations qu'il a imaginées pour passer de cette spécification à un programme.

Le démonstrateur interactif de théorèmes associé au langage permet de compléter et de contrôler la validité de ces raisonnements.

La syntaxe du langage est suffisamment souple pour qu'un utilisateur n'ait pas de difficulté à en respecter les limites.

Pour faciliter l'utilisation du système, il ne faut pas se contenter de partir de l'état initial, mais lui adjoindre une bibliothèque de théorèmes exprimant les techniques et méthodes les plus efficaces développées jusqu'ici pour les preuves de programmes [2, 6, 14, 20, 26, 30, 38, 56, 58, etc...].

En attendant le système de Programmation Assistée par Ordinateur, on peut en imiter le fonctionnement "à la main" pour obtenir méthodiquement des programmes, sinon certifiés, du moins bien documentés.

## 2. Conception assistée par ordinateur

En remplaçant la bibliothèque de transformations de programmes par une bibliothèque des principales propriétés d'un autre domaine d'application , on peut utiliser ce même langage pour spécifier et contrôler la conception de produits divers.
Exemple: on peut concevoir un barrage résistant à une enveloppe donnée de vibrations en fournissant les théorèmes de mécanique, résistance des matériaux, etc...

Plus généralement, tous les produits dont il faut certifier la très haute sécurité et qui ne peuvent pas être testés exhaustivement (nombre de cas élevé ou infini) sont d'excellents candidats à l'utilisation d'un tel système.

## 3. Relations avec l'Intelligence artificielle

Un tel langage paraît pouvoir compléter avantageusement les systèmes de déduction usuels:

. C'est le seul démonstrateur, à ma connaissance, qui utilise le *Principe d'Incertitude* mentionné page 133, d'après lequel il faut accepter un risque d'erreur pour atteindre des performances non négligeables.
Ce principe est peut-être l'une des raisons pour lesquelles le raisonnement humain est très supérieur au "raisonnement" machine: le premier laisse en effet une place indiscutable à l'erreur.

. C'est aussi apparemment, le seul démonstrateur qui laisse une place aussi importante au "pattern-matching" et aussi faible à des opérations de combinaison et de réécriture (simplifications, formes canoniques).
Ce pourrait être une autre raison de la supériorité du raisonnement humain qui ne serait pas très différent du processus de perception par reconnaissance (approximative) des formes.

. Enfin ce démonstrateur tient compte de l'effet de maquette mentionné précédemment: il est sûrement moins rapide que les autres sur les conjectures très simples (mais c'est aussi le cas du raisonnement humain...).

En résumé, si ce langage est loin d'être naturel, il l'est certainement plus que la plupart de ceux qui prétendent l'être.

Pour terminer enfin ces considérations peut être exagérées, disons qu'il serait intéressant pour des études *pédagogiques* de comparer les difficultés du système à acquérir le langage de l'utilisateur, avec les difficultés classiques des élèves et étudiants.

## 4 - Calcul par résonance

Remarquons qu'ici les phrases sont reconnues à travers leur spectre par simulation digitale d'une technique de *résonance*.

Il est donc naturel d'aller plus loin et d'étudier l'intérêt de "machines à résonateurs" capables de trouver *directement* un "pattern" parmi un vaste ensemble (les applications seraient multiples: preuve de théorèmes, reconnaissance d'images et de sons, accès aux bases de données par leur contenu, etc...).

L'avantage de telles machines serait l'accès instantané, alors que sur machines digitales il faut un temps de parcours O(n).
(Dans des cas très particuliers, ce temps peut être réduit par définition d'un ordre total ou par des techniques de hash code).

L'inconvénient est que si la résonance n'est pas assez aigüe, des objets voisins peuvent être confondus (encore une fois, il y a compromis entre rapidité et risque d'erreur).

De telles machines existent sous des formes rudimentaires (tuner radio, convertisseurs optiques [ ]).

Il serait intéressant d'étudier maintenant le gain de performance qu'apporterait l'adjonction aux habituels circuits digitaux de tels résonateurs, et si ce gain est réel (cest-à-dire si on ne peut pas simuler ces résonateurs en temps réel), les technologies permettant de les implémenter.

## III - CONCLUSION

La définition de ce Langage de Spécifications est maintenant suffisamment détaillée pour qu'on puisse entreprendre son implémentation.

Le coût prévu de sa mise en oeuvre complète (12 à 25 hommes-années) peut paraître impressionnant, mais il est minime si on le compare à son utilité pratique:

- La *certification de programmes* (indispensable dans certaines applications où la sécurité est critique) devient possible grâce au langage proposé qui permet de formuler des spécifications très claires, puis de construire - par transformations successives - des programmes certifiés corrects, maintenables et transportables.

Pour les étapes finales de la programmation (exemple: passage d'une formulation assertionnelle à un programme), le système permet de transcrire et d'utiliser les outils qui existent et continuent d'être développés ailleurs (systèmes de Luckham [30], Milner [38], Good [20] et autres).

En quelque sorte, ce langage est proposé comme le chaînon manquant entre l'objectif "certification" et les moyens techniques et théoriques déjà développés.

- La *théorie des langages* est enrichie:

Après les langages context-free dont l'alphabet non-terminal fini limitait sévèrement la puissance d'expression, puis les langages plus récents [62] où les symboles non-terminaux appartiennent à un langage infini, le langage ici proposé utilise un ordre partiel sur les non-terminaux pour accroître considérablement la puissance d'expression (plus de déclarations, tolérance aux ambiguïtés, langage adaptatif) au prix d'une analyse syntaxique légèrement plus complexe.

- L'efficacité des démonstrateurs de théorèmes (donc l'*Intelligence Artificielle* et la *Robotique*) trouve ici des moyens de s'améliorer, grâce à des techniques de preuve par "pattern matching" nouvelles et efficaces.

# ANNEXES

# ANNEXE I

## ESTIMATION DE FIABILITE:
## SYSTEME SANS VIEILLISSEMENT N'AYANT JAMAIS SUBI DE PANNE

### RAPPEL

La fiabilité d'un système sans vieillissement est caractérisée par une constante de temps T telle que:

Probabilité (durée de vie $\geq$ t) = $e^{-\frac{t}{T}}$ et

Espérance de vie = $\int_0^\infty e^{-\frac{t}{T}} t\, dt = T$ (=MTBF = temps moyen entre pannes)

### PROBLEME

Un système sans vieillissement vit depuis un temps t , que sait-on de sa fiabilité? Que sait-on du temps qui lui reste à vivre?

Intuitivement : plus t est grand, plus on s'attend à une fiabilité (T) élevée et plus on s'attend à ce qu'il vive encore longtemps.

C'est cette intuition qu'il's'agit de préciser.

Définition "crédibilité du MTBF $\geq$ T"
Si la constante de temps valait T, la probabilité que le système soit encore en vie au bout du temps t serait $e^{-\frac{t}{T}}$

$e^{-\frac{t}{T}}$ est la"vraisemblance" du MTBF: T : plus T est court, plus il est invraisemblable.

Si l'on garantit que la constante de temps vaut au moins T, la "crédibilité" de cette garantie est $1 - e^{-\frac{t}{T}}$

Exemple : Si le système vit depuis 10 ans et que tout ce qu'on sait sur lui est qu'il ne vieillit pas, le MTBF est très probablement supérieur à 1 an, ce qui se traduit par:

"vraisemblance de MTBF = 1 an" = $e^{-10}$ = 0.0000454

"crédibilité de la garantie: MTBF ≥1 an" = 1 - $e^{-10}$ = 0.9999546

Le MTBF n'étant qu'une durée de vie moyenne, il est plus important d'estimer la crédibilité d'une garantie exprimant que "le système doit vivre encore au moins un temps t' ":

Définition : "Crédibilité d'une garantie de durée de vie t' "
Si l'on connaissait le MTBF≥T, la probabilité d'une telle durée de vie serait $e^{-\frac{t'}{T}}$ .
Puisqu'on définit la crédibilité d'un MTBF ≥T comme étant $1 - e^{-\frac{t}{T}}$ ,
la "densité" de crédibilité d'un MTBF=T est la dérivée $\frac{t}{T^2} e^{-\frac{t}{T}}$ .

On peut donc exprimer la crédibilité d'une durée de vie t' comme étant:

$$\int_0^\infty \frac{t}{T^2} e^{-\frac{t}{T}} e^{-\frac{t'}{T}} dt = \frac{t}{t+t'} \left[ e^{-\frac{t+t'}{T}} \right]_{T=0}^{T=\infty}$$

soit $\boxed{\frac{t}{t+t'}}$

Exemple: Pour le même système vivant depuis 10 ans, la garantie qu'il vivra encore au moins 1 an a une crédibilité $\frac{10}{11} \simeq 90.9\%$.

# ANNEXE II

## COMPLEXITE DES DEMONSTRATEURS DE THEOREMES

### 1. Latticiels de preuve

a/ preuve exploratoire

Etant donnés un ensemble d'hypothèses et de règles d'inférence, on peut représenter tous les théorèmes qui en résultent par un latticiel (graphe orienté sans boucle), de la façon suivante:

- chaque noeud représente un ensemble de théorèmes ;
- deux noeuds différents représentent deux ensembles différents ;
- le noeud initial représente l'ensemble des hypothèses ;
- arcs: si d'un ensemble de théorèmes E1, on peut déduire par une règle d'inférence un théorème t, un arc orienté joint le noeud associé à E1 au noeud associé à E1 ∪ {t}.

b/ preuve par sous-buts

Etant donnés un ensemble d'hypothèses et une conjecture, on peut représenter tous les sous-buts (qui permettraient de prouver cette conjecture) par un latticiel, de la façon suivante:

- chaque noeud représente une disjonction de sous-buts ;
- deux noeuds différents représentent deux ensembles différents ;
- le noeud initial représente la seule conjecture ;
- arcs: si une règle d'inférence appliquée à l'ensemble des hypothèses H et à un ensemble de buts B en déduit un ensemble de sous-buts B', un arc orienté lie tout ensemble E ∪ B à l'ensemble E ∪ B'.

c/ preuve par parcours de latticiel

La preuve d'une conjecture c consiste à partir du noeud initial (qui, dans le cas d'une preuve par sous-but représente {c}) et, en suivant les arcs, à aboutir à un noeud qui:

- dans le cas de la preuve exploratoire, contient c,
- dans le cas de la preuve par sous-buts représente l'ensemble {VRAI}.

C'est le coût moyen de cette recherche qui est étudié ci-dessous.

## 2. Complexité de la preuve

a/ mesure du problème

Comme indiqué dans le texte, on adopte comme mesure d'une conjecture sa "difficulté" qui est le minimum du nombre d'applications de règles d'inférence permettant d'achever la preuve. C'est donc ici la *profondeur* minimum des noeuds recherchés dans le latticiel.

b/ coût de la recherche

Pour chaque noeud on peut définir les caractéristiques géométriques suivantes:

. fan-out : nombre d'arcs issus d'un noeud ;
. fan-in : nombre d'arcs aboutissant à un noeud.

Une caractéristique importante du latticiel est la moyenne, prise sur tous les noeuds, du quotient $\frac{\text{fan-out}}{\text{fan-in}}$ .

Si cette moyenne est décrite par son logarithme $\beta = \log \left\langle \frac{\text{fan-out}}{\text{fan-in}} \right\rangle$ le nombre approximatif de noeuds à la profondeur d est $e^{\beta d}$.

Quelque soit le mode de parcours du latticiel, le *nombre moyen* d'arcs qu'il faut parcourir avant de rencontrer un noeud donné à la profondeur d est donc au moins $\sum_{1}^{d-1} e^{\beta i} + \frac{1}{2} e^{\beta d}$

(optimum obtenu par le parcours horizontal *"breadth first"*).

Cette moyenne est de l'ordre de $\left( \frac{1}{2} + \frac{1}{e^{\beta}-1} \right) e^{\beta d}$.

Si le temps moyen de parcours d'un arc (exécution d'une règle d'inférence) est $\theta$, le temps moyen que met un démonstrateur à prouver une conjecture quelconque de difficulté d est $\sim \theta \left( \frac{1}{2} + \frac{1}{2^{\beta}-1} \right) e^{\beta d}$,

de la forme $\boxed{\alpha e^{\beta d}}$

où $\alpha$ et $\beta$ sont des caractéristiques du démonstrateur.

# ANNEXE III

## PROBABILITES DE COLLISION

Soit un ensemble de N objets "équiprobables".

Si on prend k objets choisis chacun au hasard parmi les N, la probabilité pour qu'il y en ait deux identiques (*"collision"*)

est $$1 - \frac{N-1}{N} \frac{N-2}{N} \cdots \frac{N-k+1}{N}$$

qui croît très rapidement avec k (comme l'indique le célèbre "birthday paradox" : parmi 20 personnes au hasard, il est probable que 2 au moins ont la même date anniversaire)

Pour $k << \sqrt{N}$, cette probabilité de collision est bien approximée par $\boxed{\frac{k^2}{2N}}$ .

# ANNEXE IV

## EXEMPLE D'APPLICATION A LA PROGRAMMATION

Calcul rapide $x^m \bmod n$ en nombres entiers très grands.

Le calcul de $x^m \bmod n$ pour des entiers x, m, n *très grands* intervient dans diverses applications, notamment dans le procédé cryptographique dit de Rivest, et dans les fonctions de reconnaissance d'arbres proposées ici même (page 137).

On voit ci-dessous l'une des multiples façons dont pourrait se spécifier et se résoudre ce problème avec l'aide du système.

### 1. Spécification initiale (cahier des charges)

*Hypothèses*

*a/ données*

$(H_1)$ $\quad 0 \leq x, m < 10^{50}, \quad 1 \leq n < 10^{50}$

*b/ moyens*

$(H_2)$ Algol IBM 360-67

*Exigences*

$(E_1)$ $G(x,m,n) = x^m \bmod n$

$(E_2)$ Coût de $G(x,m,n) < 2$ sec

### 2. Chargement de bibliothèques d'applications dans la base

Le système va adjoindre à la partie hypothèses (moyens) toutes les bibliothèques d'axiomes et lemmes concernant le domaine d'application, c'est-à-dire ici:

$(H_3)$ Bibliothèque d'algorithmique

...

$H_{3.1}$ E (si p alors u sinon v) = si p alors E(u) sinon E(v)

$H_{3.2}$ lemme de récursivité simple:

$$\left\{\begin{array}{l} \text{- } F(x) \leftarrow \text{si } x=g(x) \text{ alors } h(x) \\ \qquad\qquad\qquad\quad \text{sinon } E(F(x)) \\ \text{- } x=g(x) \supset h(x) = E(h(x)) \\ \text{- coût de } F(x) < \infty \end{array}\right\} \supset F(x) = E(F(g(x)))$$

$H_{3.3}$ (factorisation)

si p alors E(g(x)) sinon F(g(x)) = $Z \leftarrow g(x)$ ;
si p alors E(z) sinon F(z)

...

$(H_4)$ Bibliothèque d'arithmétique entière (puisque les entiers apparaissent dans $H_1$)

...

$H_{4.1}$ simplificateur d'expressions arithmétiques

$H_{4.2}$ < est transitif

$H_{4.3}$ $\log_b$ est monotone croissante

...

$(H_5)$ Bibliothèque concernant Algol, IBM mentionné en $H_2$

...

$H_{5.1}$ NMax = $2^{31} - 1$

$H_{5.2}$ cycle $< 3.10^{-6}$ sec (durée d'une instruction)

...

$(H_6)$ Bibliothèque des propriétés de *l'exponentielle* (mentionnée en $E_1$)

...

$H_{6.1}$ $x^0 = 1$

$H_{6.2}$ $x^1 = x$

$H_{6.3}$ $x^{a.b+c} = (x^a)^b.x^c$ (lemme très utile d'après les critères heuristiques page 148)

$(H_7)$ Bibliothèque des propriétés de *modulo* (mentionné en $E_1$)

...

$H_{7.1}$ $x = \lfloor x/b \rfloor . b + x \bmod b$

$H_{7.2}$ $0 \leq x \bmod b < b$

$H_{7.3}$ $(x+y) \bmod b = (x \bmod b + y \bmod b) \bmod b$

$H_{7.4}$ $(x.y) \bmod b = (x \bmod b \ . \ y \bmod b) \bmod b$

$H_{7.5}$ $(x \bmod b) \bmod b = x \bmod b$

$H_{7.6}$ $x \bmod 2 =$ si x impair alors 1 sinon 0

...

$(H_8)$ Bibliothèque d'évaluation des *coûts* (mentionné en $E_2$)

...

$H_{8.1}$ programme d'évaluation symbolique des coûts

...

## 3. Transformations de spécifications

Une bonne partie des transformations ci-dessous, marquées * , peuvent être proposées automatiquement par un système modérément perfectionné.

Toutes ces transformations sont résumées par le graphe documentaire (§4) qui contient l'information suffisante pour justifier, comprendre, maintenir, transporter et modifier le produit final.

| | |
|---|---|
| $E_1 \overset{*}{\to} (E_3)$ | avec $H_{6.3}$ :<br>$m = sq+r \supset G(x,m,n) = ((x^s)^q.x^r) \bmod n$<br>(noter que $H_{6.3} \supset (E_3 \supset E_1)$ |
| $E_3 \overset{*}{\to} (E_4)$ | avec $H_{7.4}$ :<br>$m = sq+r \supset G(x,m,n) = ((x^s)^q \bmod n . x^r \bmod n) \bmod n$ |
| utilisateur: $(H_9)$ | $(x^s)^q \bmod n = (x^s \bmod n)^q \bmod n$ |
| utilisateur: $(H_{10})$ | Système: pourquoi?<br>$y^q \bmod n = (y \bmod n)^q \bmod n$<br>(noter que $H_{10} \supset H_9$) |
| | Système: pourquoi? Utilisateur: on verra plus tard |
| $E_4 \overset{*}{\to} (E_5)$ | avec $H_9$ :<br>$m = sq+r \supset G(x,m,n) = ((x^s \bmod n)^q \bmod n . x^r \bmod n) \bmod n$ |
| utilisateur: $(H_{11})$ | avec $H_{7.1}$:<br>$m = \lfloor m/2 \rfloor .2 + m \bmod 2$ (note: indication décisive) |
| $E_5 \overset{*}{\to} (E_6)$ | avec $H_{11}$:<br>$G(x,m,n) = ((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n . x^{m \bmod 2} \bmod n) \bmod n$ |
| $E_6 \overset{*}{\to} (E_7)$ | avec $H_{7.6}$:<br>$G(x,m,n) =$<br>$= ((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n, x$ si m impair alors 1 sinon $0 \bmod n) \bmod n$ |
| $E_7 \overset{*}{\to} (E_8)$ | avec $H_{3.1}$ :<br>$G(x,m,n)$=si m impair alors $((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n.x^1 \bmod n) \bmod n$<br>sinon $((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n.x^0 \bmod n) \bmod n$ |
| $E_8 \overset{*}{\to} (E_9)$ | avec $H_{6.1}$ et $H_{6.2}$ :<br>$G(x,m,n)$ = si m impair alors $((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n.x \bmod n) \bmod n$<br>sinon $((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n.1 \bmod n) \bmod n$ |
| utilisateur: $(H_{12})$ | $1 \bmod n = 1$<br>Système: pourquoi? utilisateur: plus tard |

avec $H_{12}$ , $H_{4\cdot1}$ et $H_{7\cdot5}$ :

$E_9 \xrightarrow{*} E_{10}$ : $G(x,m,n) = \text{si } m \text{ impair alors } ((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n . x \bmod n) \bmod n$ sinon $((x^{\lfloor m/2 \rfloor} \bmod n) \bmod n$

---

$E_{10} \xrightarrow{*} E_{11}$ & $E_{12}$ : $x' = x \bmod n$

$G(x,m,n) = \text{si } m \text{ impair alors } ((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n . x') \bmod n$ sinon $((x^{\lfloor m/2 \rfloor} \bmod n)^2 \bmod n$

---

$E_1$ & $E_{12} \xrightarrow{*} E_{13}$ : $G(x,m,n) = \text{si } m \text{ impair alors } (G(x,\lfloor m/2 \rfloor,n)^2 \bmod n . x') \bmod n$ sinon $G(x,\lfloor m/2 \rfloor,n)^2 \bmod n$

---

avec $H_{3\cdot2}$ : (noter que $E_{13}$ est une substitution de la partie droite de $H_{3\cdot2}$)

$E_{13} \xrightarrow{*} E_{14}$ & $E_{15}$ & $E_{16}$ :

$G(x,m,n) \leftarrow$ si $m = \lfloor m/2 \rfloor$ alors $h(m)$
sinon si $m$ impair
alors $(G(x,\lfloor m/2 \rfloor,n)^2 \bmod n . x') \bmod n$
sinon $G(x,\lfloor m/2 \rfloor,n)^2 \bmod n$

---

$m = \lfloor m/2 \rfloor \supset h(m) = \text{si } m \text{ impair alors } (G(x,m,n)^2 \bmod n . x') \bmod n$ sinon $G(x,m,n)^2 \bmod n$

---

Coût de $G(x,m,n) < \infty$

---

$E_2$ & $E_{16} \xrightarrow{*} E_2$ :

avec $H_{4\cdot2}$ :

(note: suppression de $E_{16}$ puisque $E_{16} \supset E_2$)

---

Remarque: les exigences sont actuellement $E_2$, $E_{11}$, $E_{14}$, $E_{15}$

---

Utilisateur: $H_{13}$ $\quad m = \lfloor m/2 \rfloor \equiv m = 0$

Système: Pourquoi? Utilisateur: plus tard

---

$E_{15} \xrightarrow{*} E_{17}$ : avec $H_{13}$ et $H_{4\cdot1}$ :

$h(0) = G(x,0,n)^2 \bmod n$

---

$E_1$ & $E_{17} \xrightarrow{*} E_{18}$ : $h(0) = (x^0 \bmod n)^2 \bmod n$

---

$E_{18} \overset{*}{\to} (E_{19})$

avec $H_{6.1}$ et $H_{12}$

$h(0) = 1$

---

$E_{14}$ & $E_{19}$ $\overset{*}{\to} (E_{20})$

avec $H_{13}$ :

$G(x,m,n) \leftarrow$ si $m = 0$ alors 1

sinon si m impair alors $(G(x,\lfloor m/2 \rfloor,n)^2 \bmod n.x')\bmod n$

sinon $G(x,\lfloor m/2 \rfloor,n)^2 \bmod n$

---

$E_{20} \overset{*}{\to} (E_{21})$

avec $H_{3.3}$:

$G(x,m,n) \leftarrow$ si $m = 0$ alors 1

sinon $Z \leftarrow G(x,\lfloor m/2 \rfloor,n)^2 \bmod n$ ;

si m impair alors $(Z.x')\bmod n$

sinon Z

---

Remarque: l'algorithme $E_{21}$ est en Algol sur des *nombres illimités* alors qu'il faut un algorithme en Algol IBM sur des nombres ≤ Nmax

---

Chargement de la bibliothèque: *arithmétique en base b*

note: si cette bibliothèque n'existe pas, elle peut être définie ici par l'utilisateur.

---

$H_{14} \overset{*}{\to}$

$(H_{14})$ $(b-1)^2 \leq$ Nmax $\supset H_{15}$ ($H_{15}$ est une bibliothèque "conditionnelle")

...

$(H_{15})$

$H_{15.1}$ NUL(x) = x=0 ; coût de NUL(x) = 1. cycle

$H_{15.2}$ PRODUIT(x,y) = x.y ; coût de PRODUIT(x,y) ≤ 5.cycle.$\log_b x.\log_b y$

$H_{15.3}$ CARRE(x) = $x^2$ ; coût de CARRE(x) ≤ 5.cycle.$\log_b^2 x$

$H_{15.4}$ REM(x,n) = x modn ; coût de REM(x,n) ≤ 5.cycle.$\log_b n \log_b (x/n)$

$H_{15.5}$ MOITIE(x) = $\lfloor x/2 \rfloor$ ; coût de MOITIE(x) ≤ 2.cycle.$\log_b x$

$H_{15.6}$ IMPAIR(x) = x impair ; coût de IMPAIR(x) ≤ 2.cycle

---

$(E_{22})$ $(b-1)^2 \leq$ Nmax

---

$E_{21} \overset{*}{\to} (E_{23})$

avec $H_{15.1 \text{ à } 6}$ :

$G(x,m,n) \leftarrow$ si NUL(m) alors 1

sinon $Z \leftarrow$ REM(CARRE(G(x,MOITIE(m),n),n) ;

si IMPAIR(m) alors REM(PRODUIT(Z,x'),n)

sinon Z

avec $H_8$, $H_{15.1 \text{ à } 6}$, $E_{23}$ :

$H_{16}$ Coût de $G(x,m,n) \leq$ 2.cycle
+ 1.cycle
+ 2.cycle.$\log_b m$
+ coût de $G(n,\lfloor m/2 \rfloor,n)$
+ 5.cycle.$\log_b^2$ $G(x,\lfloor m/2 \rfloor,n)$
+ 5.cycle.$\log_b$ n $\log_b \frac{G(x,\lfloor m/2 \rfloor,n)^2}{n}$
+ 2.cycle
+ 2.cycle
+ 5.cycle.$\log_b Z \log_b x'$
+ 5.cycle.$\log_b$ n $\log_b \frac{Z.x}{n}$

---

Utilisateur $H_{17}$ avec $E_1$, $E_{7.2}$, $E_{12}$ :

$G(x,\lfloor m/2 \rfloor,n) < n$, $Z < n$ , $x' < n$

---

avec $H_1$ $H_{16}$ $H_{17}$ $H_{4.1}$ et $H_{4.2}$ :

$H_{18}$ Coût $G(x,m,n) \leq (7+2\log_b 10^{50} + 20 \log_b^2 10^{50})$.cycle + coût de $G(x,\lfloor m/2 \rfloor,n)$

---

Utilisateur $H_{19}$ $f(m) = k+f(\lfloor m/2 \rfloor) \supset f(m) \leq k.\log_2 m \,.\, f(0)$

Système: pourquoi? Utilisateur: plus tard

---

avec $H_{19}$ , $H_{18}$ :

$H_{20}$ Coût $G(x,m,n) \leq (7 + 2 \log_b 10^{50} + 20 \log_b^2 10^{50})$.cycle.$\log_2 10^{50}$

---

$_{22} \rightarrow E_{24}$ avec $H_{5.1}$ :

b = 10000

---

avec $E_{24}$, $H_{20}$ et $H_{4.1}$ :

$H_{21}$ coût de $G(x,m,n) < 524\,367$.cycle

---

avec $H_{21}$ :

$_2 \rightarrow E_{25}$ cycle $< 3.81\ 10^{-6}$ sec

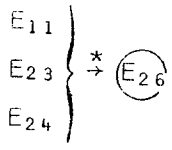
---

avec $H_{5.2}$ :

$_{25} \rightarrow$ VRAI

---

Il reste comme exigences $E_{11}$, $E_{23}$, $E_{24}$
qui peuvent être fusionnées en un programme Algol IBM 360-67




Programme

---

## 4 - Graphe documentaire

Toutes les étapes précédentes de la construction sont organisées selon le graphe de dépendance ci-joint qui fait apparaître:

- le cahier des charges,
- le texte de la garantie (cahier des charges + tout ce qui a été utilisé *sans être prouvé*),
- la preuve (chaque élément résulte de ses ascendants immédiats).

La maintenance, le transport et toutes mises à jour obéissent à la règle ci-dessous:

*Modifications: règle fondamentale:*

a/ Le produit n'est accessible qu'à travers ce graphe qui sert de clé d'accès ; aucune modification directe n'est possible.

b/ Seuls les éléments du texte de la garantie peuvent être arbitrairement modifiés (addition, suppression, changement).

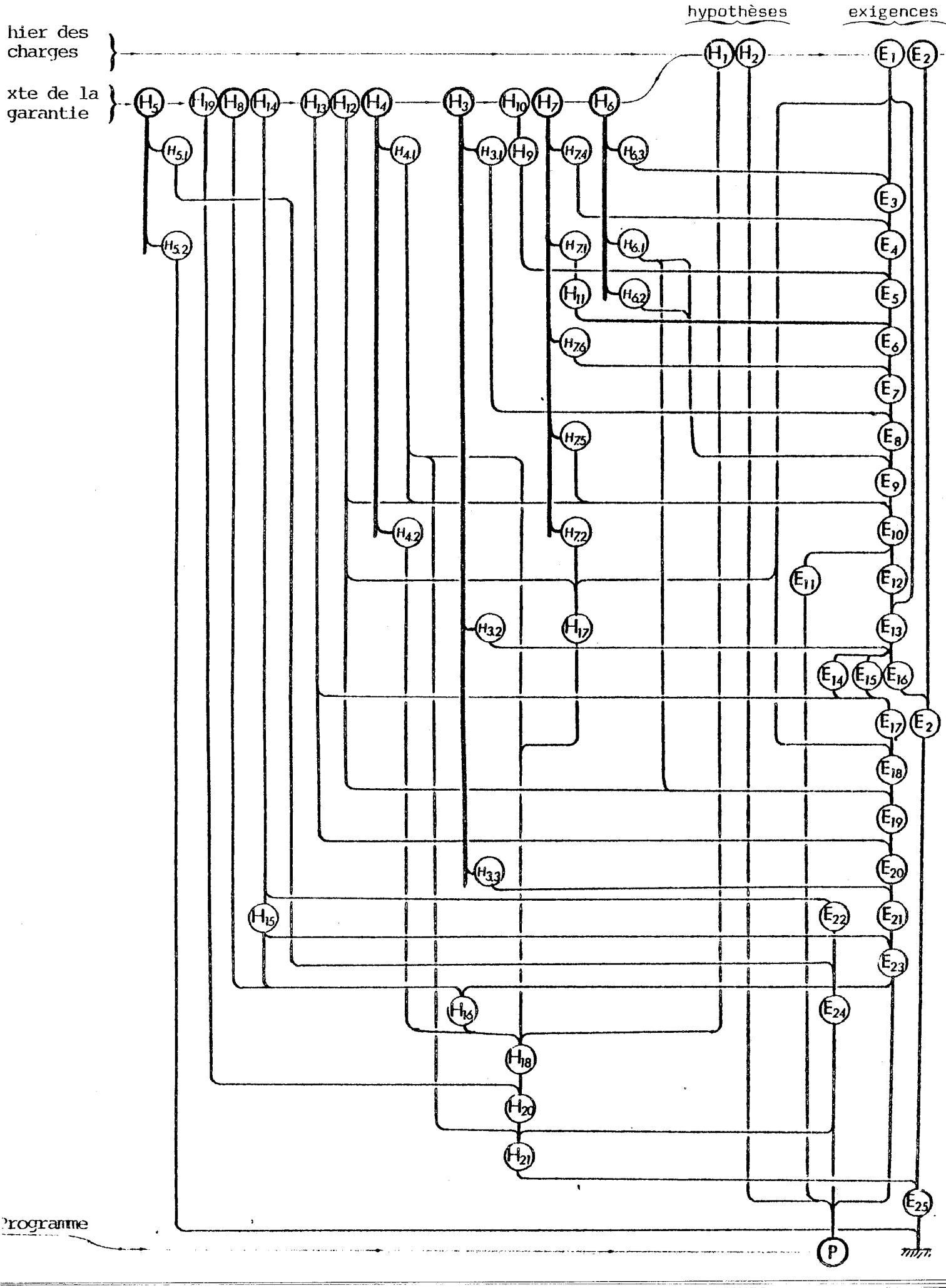
c/ Tout changement ou suppression d'un élément *détruit* le sous-graphe de ses descendants.

d/ La conception doit être reprise à partir de ce qui reste du graphe.

## 5 - Conclusion

On voit que, même pour un exemple assez simple, le système doit posséder une base de connaissances "pragmatiques" très bien renseignée, faute de quoi la moindre transformation de spécifications peut nécessiter une justificatio très laborieuse: quelle que soit la qualité du moteur, son rendement dépend de la qualité du carburant.

# - GRAPHE DOCUMENTAIRE -



# REFERENCES

La liste ci-dessous ne doit être considérée ni comme une bibliographie ni comme un palmarès privé, mais simplement comme un ensemble de publications citées dans le texte ou susceptibles de l'éclairer.

## Abréviations:

CACM : Communications de l'ACM (Association for Computing Machinery)

ICRS : ACM-IEEE-NSF - International Conference on Reliable Software, Los Angeles, Mars 1975.

IJCAI : International Joint Conference on Artificial Intelligence.

IFIP : International Federation for Information Processing.

JACM : Journal de l'ACM.

POPL : ACM Symposia on Principles of Programming Languages.

[2] J.R.ABRIAL
*Z: a specification language*
Document non publié(?) 1978

[4] A.V.AHO, S.C.JOHNSON
*LR parsing*
Computing Surveys 6-2, 99/124, Juin 1974.

[6] B.AMY, M.C.CAPLAIN
*Invariances algorithmiques et récurrences*
3° Conférence Internationale sur la Programmation Paris, Mars 1978.

[8] W.W.BLEDSOE, P.BRUELL
*A man-machine theorem proving system*
IJCAI 3, Stanford, Août 1973

[9] D.G.BOBROW, B.RAPHAEL
*New programming languages for artificial intelligence research*
ACM Computing surveys 6-3, September 1974

[10] M.C.CAPLAIN
*Construction et preuve de logiciel par transformations de spécifications*
Rapport DRME 76-114, 1977.

[12] A.COLMERAUER & al.
*Un système de communication homme-machine en français*
Université Aix-Marseille, Octobre 1972

[14] P.COUSOT
*Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*
Thèse d'Etat, USMG, Mars 1978.

[16] B.ELSPAS
*The semi-automatic generation of inductive assertions for proving program correctness*
SRI report, Menlo Park, California, Juillet 1974.

[18] J.A.GOGUEN, J.W.THATCHER, E.G.WAGNER, J.B.WRIGHT
*Initial algebra semantics and continuous algebras*
JACM 24-1, 68/95, January 1977

[20] A.I.GOOD, R.L.LONDON, W.W.BLEDSOE
*An interactive program verification system*
ICRS

[22] J.R.HOBBS
*What the nature of natural language tells us about how to make natural-language like programming more natural*
Symposium Artificial Intelligence and Programming Languages
SIGPLAN-SIGACT(ACM), 85/93, Rochester, August 1977.

[24] N.KAPLAN, J.ULLMAN
*A general scheme for the automatic inference of variable types*
POPL 5, Tucson Arizona, January 1978.

[26] S.KATZ, Z.MANNA
*Logical analysis of programs*
CACM 19-4, 188/206, April 1976.

[28] C.H.LEWIS, B.K.ROSEN
*Recursively defined data types*
*PART I:* POPL, Boston, Mass., 1973
*PART II:* RC 4713, IBM Yorktown Heights, N.Y., 1974.

[30] D.LUCKHAM, F. von HENKE, N.SUZUKI
*Automatic Program Verification I, ..., V*
Stanford AIM reports

[32] Z.MANNA, R.WALDINGER
*Synthesis: intention → programs*
Stanford AIM report, November 1977

[34] W.A.MARTIN
*Determining the equivalence of algebraic expressions by hash coding*
JACM 18-4, 549/558, October 1971

[36] M.D.MICKUNAS, J.A.MODRY
*Automatic error recovery for LR parsers*
CACM 21-6, 459/465, June 1978

[38] R.MILNER
*LCF: a methodology for performing rigorous proofs about programs*
Proc. 1st IBM Symposium Mathematical foundations of Computer Science
AMAGI, Japon 1976

[40] R.MILNER
*A theory of type polymorphism in programming*
Dept Computer Science CSR 9-77, University of Edinburgh, 1977

[42] F.OUABDESSELAM
*Proving program correctness by pattern difference*
2nd International Conference on Pattern Recognition
Copenhague, Août 1974

[44] T.J.PENNELLO; F. de REMER
*A forward move algorithm for LR error recovery*
POPL 5, Tucson, Arizona, January 1978

[46] M.O.RABIN
*Theoretical impediments to artificial intelligence*
IFIP 74, 615/619.

[48] S.P.RHODES
*Practical syntactic error recovery for programming languages*
PhD thesis TR 15, University of California, Berkeley, June 1973

[50] P.ROUSSEL
*PROLOG: manuel de référence et d'utilisation*
Groupe d'Intelligence Artificielle, Marseille Luminy, Septembre 1975

[51] B.A.SHAPIRO
*A survey of problem solving languages and systems*
University of Maryland, TR 235, Mars 1973

[52] M.SOLOMON
*Type definitions with parameters*
POPL 5, Tucson, Arizona, January 1978

[54] A.TENENBAUM
*Type determination for very high level languages*
Rep. NSO-3, Courant Inst. of Mathematical sciences
New York, 1974

[56] J.THIELE
*Semantique partielle interactive*
Thèse 3ème cycle, INPG, mars 1978

[58] R.TOPOR
*Interactive program verification using virtual programs*
PhD thesis, Edinburgh, 1975

[60] B.WEGBREIT
*The synthesis of loop predicates*
CACM 17-2, 102/112, February 1974.

[62] A. van WIJNGAARDEN
*Report on the algorithmic language ALGOL 68*
Numer. Math. 14, 79/218, 1969

[96] voir textes de AMY & CAPLAIN, KAHN, PAIR:
*Journées Logique et Programmation*
IRIA, le Bischenberg, novembre 1975

[98] *Proceedings of a symposium on extensible languages*
(S.A. SCHUMAN ed.)
SIGPLAN Notices 6-12, December 1971

[100] *Proceedings of a symposium on very high level languages*
(B.LEAVENWORTH ed.)
SIGPLAN Notices 9-4, March 1974.