# Synthèse de programmes : connaissances et déduction dans les domaines d'application

Ramon Brena

**THESE**

présentée par

BRENA Ramón

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministeriel du 5 juillet 1984)

(Spécialité : Informatique)

==========

# SYNTHESE DE PROGRAMMES : CONNAISSANCES ET DÉDUCTION DANS LES DOMAINES D'APPLICATION

==========

Date de soutenance : 20 juin 1989

Composition du jury :

| | |
|---|---|
| J. MOSSIERE | président |
| C. GRESSE | rapporteur |
| L. TRILLING | rapporteur |
| P. JACQUET | |
| P. JORRAND | |

Thèse préparée au sein du Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle

# RÉSUMÉ

Parmi les connaissances intervenant dans la construction de programmes, celles relatives au domaine du problème ont été peu étudiées. Dans le cadre d'un projet de recherche en synthèse déductive de programmes à partir de spécifications logiques, cette thèse étudie les représentation, structuration et utilisation de connaissances. Celles-ci sont considérées comme des sous-ensembles finis d'une théorie du premier ordre. Nous prêtons une attention particulière au problème du guidage de l'utilisation des connaissances en synthèse de programmes. En effet, dans la plupart des approches pour la synthèse de programmes, des connaissances du domaine sont simplement introduites sans que leur choix soit justifié. Dans une approche guidée par le but, nous essayons de caractériser les connaissances nécessaires pour mettre en rapport une idée algorithmique (un schéma de programme), choisie par l'utilisateur, avec le problème de synthèse à résoudre. Il devient alors possible de chercher, parmi les connaissances disponibles, celles qui correspondent à la caractérisation ainsi engendrée. Les connaissances disponibles sont stockées dans une Base de Connaissances où elles sont structurées de façon modulaire. Nous définissons un langage d'interrogation de cette Base permettant d'exprimer les caractérisation de connaissances, et nous proposons des mécanismes d'exécution des requêtes de ce langage. Nous en avons réalisé une maquette en Prolog.
Dans cette thèse nous proposons également un système parallèle de déduction naturelle, pour le module de déduction utilisé dans notre approche.

# ABSTRACT

Among the kinds of knowledge participating in program construction, domain-dependent knowledge has not been much studied. In the framework of a research project on Deductive Program Synthesis from logic specifications, this Thesis studies the representation, structuring and use of domain-dependent knowledge. This one is considered as a finite subset of a first-order theory. Particular attention is given to the problem of knowledge use guidance in Program Synthesis. In most approaches, spontaneous knowledge items are introduced without a justification of their relevance. In our goal-directed approach, we try to characterise the domain knowledge needed to apply an algorithmic idea (a program scheme) chosen by the user, to the Program Synthesis problem at hand. It becomes then possible to look, among the available knowledge, for the items fitting a characterisation. Available knowledge is stored in a Knowledge Base, where it is structured in a modular way. We define a query language allowing to express the knowledge characterisations, and we give mechanisms for their execution in the Knowledge Base. A prototype of the latter has been implemented in Prolog.
We propose also in this Thesis a parallel Natural Deduction System, for the deductive module used in our approach.

# Remerciements

Je remercie les membres du jury:

• C. Gresse, professeur à l'Université d'Orléans, qui a accepté d'être rapporteur de cette thèse; je le remercie pour le temps qu'il a dédié a ce travail.

• L. Trilling, professeur à l'Université J. Fourier, rapporteur de cette thèse; je le remercie vivement pour son intérêt et ses très utiles remarques

• J. Mossière, professeur à l'ENSIMAG, qui m'a fait l'honneur de présider le jury de cette thèse.

• P. Jorrand, Directeur de recherche CNRS. Je le remercie de sa participation au jury et surtout de son accueil dans son laboratoire, accueil qui a commencé au Mexique même.

• P. Jacquet, Maître de conférences à l'ENSIMAG, et responsable de cette thèse. Je tiens à le remercier très particulièrement pour tout le temps qu'il a dédié a mon travail et pour sa diligence à matérialiser la terminaison et la soutenance de cette thèse.

Je remercie chaleureusement Marie-Laure Potet, co-équipière de thèse, qui, avec sa grande gentillesse et ses qualités scientifiques et humaines, vous fait accepter de bon gré les critiques les plus impitoyables.

Un grand merci à Valérie L., pour avoir corrigé le texte français en pleine période d'examens.

Je tiens à remercier Ricardo Caferra pour sa participation dans les débuts de cette thèse.

Je fais un remerciement générique à mes compagons de bureau, de laboratoire, de R.U., et tous mes amis de Grenoble, pour le plaisir de leur compagnie quotidienne; chacun d'entre eux saura faire l'instanciation qu'il convient.

Mon dernier et plus vif remerciement est pour mes parents, qui de très loin n'ont jamais cessé de me soutenir.

Grenoble, juin 1989

# TABLE DES MATIERES

# PRÉFACE

Le présent travail s'est déroulé dans le cadre d'un projet de recherche en Synthèse assistée de Programmes, placé sous la direction de P. Jacquet, au sein du Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle (LIFIA).

Le LIFIA accueille des projets de recherche en IA (robotique, mécanisation du raisonnement) ainsi qu'en Informatique Fondamentale (langages, en particulier). La Synthèse de Programmes se trouve, si l'on peut dire, à cheval entre les deux, relevant de la recherche en théorie la programmation, ainsi que des méthodes de résolution de problèmes basées sur des stratégies - voire heuristiques- propres à l'IA.

## *Deux têtes en font plus qu'une.*

Cette thèse est complémentaire de celle élaborée par M-L Potet, "Preuves et stratégies pour la synthèse déductive de programmes", dirigée également par P.Jacquet; chacune de ces deux thèses est relative à un aspect de notre approche pour la synthèse de programmes: celle de M-L Potet étudie la définition formelle du système de synthèse, ainsi que les stratégies permettant de diriger la recherche d'une solution, tandis que la présente thèse se propose d'étudier la prise en compte de la sémantique de l'application considérée.

Ce partage des tâches revient à distinguer deux types de connaissances nécessaires à l'activité de programmation:

1) Connaissances concernant la programmation elle-même (langages source et cible, paradigmes de programmation, stratégies);

2) Connaissances sur les domaines d'application (définition de ses objets, leurs propriétés).

Cette même division se traduit, au niveau du système de synthèse, en une architecture comprennant deux modules principaux, appelés par la suite "Système de Preuves" (SP) pour la partie programmation, et "Base de Connaissances" (BC) relative aux domaines d'application. Nous espérons ainsi isoler les difficultés inhérentes à chacune des deux parties, et par là-même, les maîtriser davantage.

## *Notre partie de cette recherche*

La définition du rôle, du contenu et des méthodes de cette BC constitue donc le sujet de cette thèse. C'est un sujet qui fait intervenir plusieurs disciplines, notamment:

- Les méthodologies de programmation;
- Les systèmes formels et la déduction automatique;
- Les Bases de Données (organisation et recherche des informations);
- Les systèmes à connaissances de l'IA.

D'où la richesse du sujet, mais aussi sa difficulté...

## *Plan général*

Dans le premier chapitre, nous placerons notre sujet dans le contexte de la synthèse de programmes, nous discuterons le rôle des connaissances sur un domaine d'application donné, l'état de l'art sur cette question, et la problématique qui s'en dégage.

Le second chapitre sera consacré à la présentation de notre approche pour la synthèse de programmes et à celle de ses implications vis-à-vis de l'utilisation des connaissances.

Dans le troisième chapitre nous étudierons la BC elle-même. Nous discutons d'abord les fonctions à remplir par la BC, en se basant sur une analyse de quelques bases de connaissances utilisées dans des systèmes de synthèse de programmes, et nous en proposons une architecture. L'un des composants de la BC, la "Base de Théorèmes", fera l'objet du reste du chapitre. Nous y décrivons le formalisme utilisé pour exprimer les connaissances, ainsi que la structuration de ces dernières.

Dans le quatrième chapitre nous étudierons le problème de la recherche des informations.

Le cinquième chapitre sera relatif au module de déduction de la BC. Il est chargé de résoudre des problèmes posés par le SP en utilisant les informations contenues dans la Base de Théorèmes. Nous proposons une version "parallélisée" d'un système de déduction naturelle.

Dans une conclusion nous présenterons une vue synthétique de notre méthode pour l'utilisation des connaissances, autour d'un concept appelé "espace de caractérisation des connaissances"; nous faisons également un bilan de nos propositions et des perspectives pour le futur.

*Automatique: Qui s'exécute sans la participation de la volonté // Qui intervient d'une manière régulière, qui doit forcément se produire // Qui opère par des moyens mécaniques. [Larousse 1984]*

# 1.- Introduction

L'objet de ce chapitre est de placer notre recherche dans un contexte général; caractériser la problématique actuelle dans le domaine, et en dégager nos orientations. Nous plaçons d'abord, au §1.1, notre travail dans un contexte général. Au §1.2 nous discutons l'utilisation de connaissances dans les différentes approches pour la synthèse de programmes.

## 1.1.- Contexte de notre recherche

"Synthèse de programmes", "Programmation automatique", "Construction assistée de programmes", sont tous des termes qui évoquent la participation de l'ordinateur dans le processus de construction de programmes.

L'idée de faire participer l'ordinateur à l'élaboration de programmes est presque aussi ancienne que la programmation elle-même. Elle fût d'abord proposée dans le domaine de l'IA et reprise après, avec des différences importantes de motivations, de point de vue et de méthodes, par la recherche en : Génie Logiciel, Algorithmique, Langages, Bases de Données, Systèmes Formels; cette énumération ne saurait être exhaustive.

Aujourd'hui, la Synthèse de Programmes est donc un point de convergence de plusieurs disciplines de l'informatique, ce qui apporte, d'une part, une grande richesse d'idées et de méthodes, et d'autre part, une grande confusion en ce qui concerne la portée et le sens de termes tels que "Programmation Automatique"[1].

Dans les années 50 on a appelé "Programmation Automatique" l'utilisation de langages de haut niveau[2] et de compilateurs produisant le "vrai" programme, selon le point de vue de l'époque. L'utilisateur était alors libéré de la gestion des registres de la machine, des instructions spécifiques de chaque processeur, et du partage de la mémoire.

Néanmoins, à nos jours les tâches à programmer sont devenues tellement complexes que la programmation conventionnelle, en utilisant les langages de haut niveau, s'avère insuffisante pour les résoudre. En effet, le logiciel est cher, peu fiable, difficile à modifier et à adapter, et incompréhensible -sauf pour son créateur. Il est nécessaire de passer à une programmation d'un

---

[1] G.Guiho en présente au moins trois sens différents dans [Guiho 83]

[2] FORTRAN, en particulier

niveau encore plus haut, qui permettrait de résoudre des problèmes complexes tout en restant dans des marges de coût et de fiabilité acceptables. Une caractéristique essentielle des langages de très haut niveau serait de permettre au programmeur de décrire des solutions de façon déclarative, sans se soucier des détails du séquencement des opérations dans la machine.

Le problème est alors que la compilation d'un langage de très haut niveau soulève des difficultés d'un tout autre ordre par rapport à celles de la compilation traditionnelle. En effet, dans ce dernier cas il est possible d'associer directement aux nœuds de l'arbre abstrait, issu de l'analyse syntaxique, les instructions du langage cible. Il existe donc une concordance parfaite entre la syntaxe et la sémantique. Ceci n'est plus valable pour des langages "déclaratifs", car il est nécessaire de prendre en compte des informations sur la sémantique de l'application *qui ne se trouvent pas contenues* dans le "programme déclaratif". Les techniques de compilation basées sur la syntaxe sont arrivées à leurs extrèmes limites; la synthèse de programmes pourrait constituer une voie pour continuer à avancer, vers l'utilisation des langages déclaratifs de spécification de programmes.

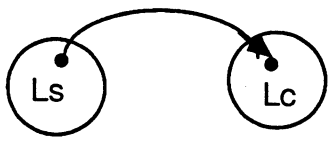
## *Notre projet*

Notre projet de synthèse de programmes se situe dans le cadre de la synthèse à partir de spécifications déclaratives et complètes, comme elles ont été définies dans le paragraphe précédent. C'est dans ce cadre que la plupart des contributions en synthèse de programmes ont été faites [Biermann 85], [Potet 88]. Les directions de recherche dans lesquelles nous pourrions éventuellement faire une (petite) contribution sont discutées dans le paragraphe suivant. Les caractéristiques de notre approche sont présentées dans le chapitre 2.

On appelle *spécification* une expression dans un *langage de spécification* Ls qui sert à décrire le comportement attendu d'un programme écrit dans un *langage cible* Lc exécutable par l'ordinateur[1].

Les expressions de Ls définissent des relations Q permettant de discriminer les couples entrée-sortie (e, s) qui font partie du comportement attendu du programme, c'est-à-dire, (e, s) $\in$ Q. On dit qu'un programme p $\in$ Lc satisfait une spécification S associée à une relation Q ssi pour toute entrée e donnant une sortie s on a: (e, s)$\in$ Q.[2]

On appelle "Synthèse de programmes" le processus qui permet de passer d'une spécification S $\in$ Ls à un programme p $\in$ Lc tel que p satisfait s:

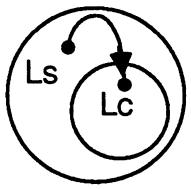



Le langage Lc peut être inclus proprement dans Ls, ce qui veut dire que les programmes sont eux-mêmes des spécifications; tel est le cas de LPG [Bert 83]:

---

[1] c'est-à-dire, il existe des compilateurs pour Lc.

[2] C'est la correction partielle




Une spécification donnée peut éventuellement être satisfaite par plusieurs programmes différents: la relation de satisfaisabilité n'est pas injective.

Cette façon de définir la synthèse de programmes est limitative puisqu'elle ne permet de décrire que les programmes calculant une fonction f(e)=s, ce qui n'est pas le cas, par exemple, dans les systèmes d'exploitation.

## *Langages de spécification*

La façon de définir les relations Q discriminant les couples E/S valides détermine en grande mesure la nature du processus de synthèse. Parmi les styles de langages de spécification les plus utilisés nous trouvons:

1) Spécifications déclaratives complètes
2) Descriptions procédurales de très haut niveau;
3) Exemples d'entrée / sortie.

Dans les spécifications déclaratives complètes, l'énoncé est une formule logique (dans le sens large proposé par [Goguen, Meseguer 87]) décrivant:

- Les caractéristiques des entrées valides;
- Les caractéristiques des sorties;
- Le lien entre les données et le résultat.

Ces spécifications sont dites *complètes* dans le sens où elles déterminent complètement la fonction à synthétiser.

Nous assimilons à cette catégorie les spécifications *algébriques* [Bert 79], [?? ].

Par exemple, une spécification d'un programme pour ordonner une liste d'entiers est:

***opérateur*** $\sigma$ *: liste[entier]* $\rightarrow$ *liste[entier]*
***équations***
*bag(l) = bag(*$\sigma$*(l))*
*ord(*$\sigma$*(l)) =* <u>*vrai*</u>
***fin***

où *bag* construit le multi-ensemble des éléments d'une liste (elle fait abstraction de l'ordre) et *ord* teste si son argument est une liste ordonnée.

Dans le cadre de la logique du premier ordre les spécifications sont de la forme :

$$P(x) \Rightarrow Q(x, f(x))$$

où $x$ est l'entrée, $f$ est la fonction que doit calculer le programme, et $P$ et $Q$ sont des prédicats qui caractérisent respectivement les entrées valides (*précondition*) et le rapport entrée-sortie (*postcondition*). Par exemple, une spécification pour un programme de tri d'une liste d'entiers est:

$$\textit{liste-entier}(x) \Rightarrow (\textit{perm}(x, f(x)) \wedge \textit{ord}(f(x)))$$

où "liste-entier" teste si l'argument est une liste d'entiers, et "perm" indique si ses deux arguments sont une permutation l'un de l'autre.

Dans [Potet 88] les spécifications logiques sont de la forme:

$$P(x) \rightarrow Q(x f(x)),$$

où le symbole '$\rightarrow$' est une forme plus restreinte de l'implication qui tient compte des fonctions partielles.

Les descriptions procédurales de très haut niveau sont des formes intermédiaires entre spécification déclarative et programme. Typiquement [Barstow 79] une telle description procédurale consiste en un schéma de programme où les actions à exécuter sont décrites sans détail ou par leurs effets (déclarativement). Exemple:

***algorithme*** *QUICK (l):*
*e := choisir-élément (l); {e ∈ l}*
*segmenter(l, e, pp, pg);*
*{perm(l, (e) + pp + pg),*
*x ∈ pp ⇒ x ≤ e, y ∈ pg ⇒ x > e}*
*QUICK := QUICK(pp) + (e) + QUICK(pg).*

Les formules logiques entre crochets sont des *assertions* sur l'effet produit par l'instruction qui les précède.

Dans les descriptions procédurales, "l'idée algorithmique" est donnée au départ par l'utilisateur. Cette forme de spécification n'a guère d'intérêt pour la découverte d'algorithmes. Par contre, du point de vue pratique, il peut parfois s'avérer plus aisé de donner une description algorithmique d'une tâche à accomplir que d'en formuler une spécification déclarative consistente et complète.

Les spécification par exemples d'entrée / sortie sont utilisées de deux façons différentes dans la synthèse de programmes:

a) La spécification est elle-même constituée d'un ensemble fini de couples d'entrée / sortie (e,s) ∈ Q. Ceux-ci sont la seule source d'information disponible pour la synthèse.

b) Les exemples sont un complément d'une autre méthodologie de synthèse.

Dans le cas (a), puisque le domaine de l'entrée est en général infini, un ensemble fini d'exemples est une spécification incomplète. La synthèse de programmes à partir d'exemples utilise des méthodes inductives présentées dans [Biermann 78, 85], [Jouannaud, Kodratoff 78, 79], [Brena 84]. Dans cette dernière étude, nous sommes arrivés à la conclusion que, malgré la puissance des outils mathématiques employés, la synthèse de programmes en utilisant seulement des exemples s'avère trop limitative, en raison de la pauvreté de l'information fournie par ces derniers.

Dans le cas (b), des nombreuses possibilités sont envisageables. Ainsi, dans le système PSI [Green 76], les exemples sont utilisés comme une aide pour former une spécification à partir d'un dialogue interactif. Dans le système LOPS [Bibel, Hörning 84] l'usage des exemples est plus lié au processus de synthèse lui-même. On peut aussi utiliser les exemples comme contraintes d'intégrité ou même comme documentation pour l'utilisateur d'un système interactif.

## *Langages cibles*

Normalement, le langage cible Lc des systèmes de synthèse de programmes n'est pas le code binaire d'une machine, car on a intérêt à réduire autant que possible la "distance" Ls/Lc à parcourir durant le processus de synthèse. Les langages cibles seront donc des langages de haut niveau pour lesquels on pourrait fournir un compilateur ou un interprète adéquat.

A notre avis, il est très important de choisir un langage cible sans effets de bord et d'une sémantique simple et claire. Un tel langage aura des propriétés mathématiques de *"régularité"* qui pourront faciliter la manipulation des programmes. Ainsi, dans [Potet 88] le langage cible utilisé est purement fonctionnel. Il comporte comme constructions la composition fonctionnelle, la conditionnelle 'si alors sinon' et la récursion, ainsi qu'un ensemble de fonctions de base.

## *La problématique actuelle et notre recherche*

A présent, plusieurs méthodes de synthèse de programmes à partir de spécifications déclaratives ont été développées et prouvées logiquement correctes [Darlington 75], [Manna, Waldinger 80], [Bibel 80], [Smith 85], [Dershowitz 83], etc. Pour une discussion générale de ces approches, consulter [Potet 88], [Gresse 84], [Biermann 85]. Toutefois, nous voudrions ici faire une constatation, qui est à l'origine de notre façon d'aborder la synthèse de programmes: face à une abondance de mécanismes formels permettant, en théorie, d'entreprendre la synthèse de toute fonction calculable, les réalisations et même les exemples présentés dans les différents travaux restent toujours dans le cadre restreint du laboratoire (combien de fois a-t-on vu traiter l'exemple du tri d'une liste ?).

La possibilité de passer à un niveau de complexité plus en accord avec la production professionnelle du logiciel se heurte souvent à la rigidité des formalismes utilisés. Une solution possible est de se restreindre aux aspects bien maitrisés de l'automatisation de la programmation, donc à la "construction assistée de programmes" [Gresse 84], comme un moyen de construire des systèmes utilisables en pratique. Ceci n'élimine pas, bien entendu, la nécessité d'effectuer un travail de recherche visant à cerner les difficultés affrontés par la synthèse de programmes.

A notre avis, deux questions sont essentielles pour permettre de progresser sensiblement en synthèse de programmes :

a) Le guidage du processus de synthèse;
b) L'utilisation des connaissances sur le problème considéré.

Le point (a) concerne les techniques et stratégies nécessaires pour guider la recherche d'un programme satisfaisant la spécification dans un espace en général infini. Il faut disposer d'un contrôle assez puissant pour permettre une augmentation de la taille des problèmes entrepris sans pour autant produire une explosion combinatoire. Cette question est discutée en détail dans [Potet 88]. Nous présentons brièvement au chapitre 2 les contributions de notre approche à ce sujet.

Le point (b) constitue le sujet même de cette thèse. Il sera présenté dans la section suivante.

## 1.2. Le rôle des connaissances dans la Synthèse de Programmes

Quand on parle de *connaissances* on fait référence au méthodes et points de vue de l'IA. Les connaissances sont le composant *modélisation du monde* des système de l'IA.

La synthèse de programmes se propose de simuler partiellement l'activité des programmeurs. Par conséquent, les connaissances en synthèse de programmes sont, d'un point de vue très général, ce bagage aquis par les programmeurs au cours des années d'expérience dans l'exercice de la programmation.

Les connaissances nécessaires à la programmation sont très larges et variées. Nous pouvons, comme [Barstow 79], classifier ces connaissances de la façon suivante:

- Connaissances générales de programmation
  - langages de programmation
  - paradigmes de programmation;
  - stratégies de raffinement

- Connaissances sur les données
  - structures et types de données;
  - propriétés des données.

Il y a certainement un recouvrement entre ces formes de connaissance: on sait, par exemple, quelles sont les structures de données qui se prêtent le mieux aux différents paradigmes de programmation.

Des effort importants ont été faits pour recenser de telles connaissances; le plus important est le recueil de [Barstow 79], fait dans le double but de l'enseignement de la programmation et de son automatisation.

## *Le rôle de chaque type de connaissance*

Les différentes approches en synthèse de programmes n'utilisent pas de la même façon chaque type de connaissances [Barstow 84].

Dans les approches "basées sur la connaissance" [Bartow 79], une même représentation est utilisée pour toutes les connaissances, à savoir, les règles de production. Les diverses connaissances sont donc utilisées de manière uniforme et exploitées par un même mécanisme d'inférence.

D'autres approches [Darlington 75], [Manna, Waldinger 80], [Dershowitz 83], [Smith 85] assimilent les connaissances générales de programmation à une *méthodologie de synthèse*, laissant le terme *connaissance* restreint aux connaissances sur le domaine d'application. Ceci revient à fournir des mécanismes *ad hoc* pour l'application des connaissances générales de programmation. C'est la synthèse classique de programmes à partir de leur spécifications. C'est dans ce cadre que nous allons placer la discussion sur les connaissances en synthèse de programmes.

La partie *méthodologie de synthèse* de notre recherche, relative aux connaissances générales de programmation, est abordée par [Potet 88].

## *Les connaissances sur des domaines d'application définis mathématiquement*

Nous allons discuter la question de l'utilisation des connaissances concernant le domaine d'application du problème considéré.

Il est nécessaire d'abord de préciser le sens du terme *connaissance* dans ce contexte.

Dans la synthèse classique de programmes à partir de spécifications déclaratives (voir § précédent) on considère des problèmes dans des domaines définis mathématiquement, de façon à pouvoir mener à des preuves et garantir ainsi la correction des programmes synthétisés. Nous proposons donc la suivante

*Définition:*
*Les connaissances sont des formules valides dans les modèles sous-jacents aux énoncés faisant partie du langage source..*

Par *modèles sous-jacents aux énoncés* nous voulons dire l'ensemble de modèles des définitions des symboles intervenant dans un énoncé donné. Ces définitions sont supposées connues du système de synthèse. Par exemple, étant donné l'énoncé suivant[1]:

***profil:*** *allperm=f(l:liste)* $\rightarrow$ *liste[liste]*
***précondition:*** *vrai*
***postcondition:*** $\forall x\ [x \in allperm \Rightarrow perm(x,l)]$,

nous supposons que le type de données *'liste'* est connu, ainsi que les symboles *perm*, $\in$, $\Rightarrow$.

---

[1] Nous nous conformons au langage des énoncés de [Potet 88]

Soit on dispose de définitions explicites pour les types de données et les symboles, soit on fait référence à une *interprétation voulue* implicite à l'énoncé. En tout cas, cet énoncé fait référence à un modèle où les formules suivantes sont valides:

$perm(nil,a) \Rightarrow a=nil$
$perm(a,b) \wedge perm(b,c) \Rightarrow perm(a,c)$
$perm(l,l)$
... ...

*Théories et Connaissances*

L'ensemble de formules valides dans un modèle d'intérêt sera appelé *théorie* [Burstall, Goguen 77]. En général les théories qui nous intéressent sont infinies. La connaissance, malheureusement, elle est toujours finie. Ceci nous empêche d'identifier sommairement théories et connaissance.

Il est souvent possible de décrire les théories en intention, en distinguant un ensemble d'*axiomes*, lequel, par application de *règles d'inférence* suffit pour générer toutes les formules de la théorie. Dans cette vision des choses, la connaissance est réduite aux axiomes.

Néanmoins, nous voudrions adopter un point de vue pragmatique. Il ne nous suffit pas de savoir qu'une certaine formule pourrait être déduite des axiomes; encore faut-il l'obtenir effectivement. Or, ceci est quasiment impossible si l'on ne sait même pas à quelle formule il faut arriver !

Par conséquent, nous reformulons notre définition de connaissances dans les termes suivants:

*Les connaissances sont les formules, faisant partie des théories référencées par les énoncés, qui sont effectivement accessibles .*

Cette formulation est en accord avec la notion de connaissance comme *représentation du monde*, car *le monde* est représenté par la partie disponible de la théorie. Elle l'est aussi avec la définition de D. Lenat de connaissance en tant que *"compiled search"*, car les théorèmes, étant déductibles des axiomes, sont des points dans un espace de recherche arborescent où les arcs représentent l'application d'une règle d'inférence.

## 1.3.- L'état de l'art de l'utilisation de connaissances en synthèse de programmes

Tous les travaux en synthèse de programmes font, tôt ou tard, un usage parfois astucieux de quelques propriétés intéressantes des données, permettant de poursuivre l'application de la méthode en question et d'aboutir à un programme efficace. Pourtant, dans la plupart des articles sur la synthèse de programmes le problème de l'utilisation des connaissances est peu abordé, et même dans les papiers qui présentent explicitement les connaissances sur le problème traité,[1] la façon dont elles sont invoquées reste mystérieuse: elles sont tout simplement introduites au "bon moment" (l'eurêka de [Darlington 75]).

[1] Par exemple, dans [Smith 85]

Dans ce paragraphe nous allons exposer la façon dont les principales méthodes de synthèse de programmes font usage des connaissances. Nous ne nous occuperons pas de la description détaillée de ces méthodes, ce qui a été fait ailleurs [Potet 88], [Biermann 85]. Par contre, nous allons concentrer notre attention sur la question de l'utilisation des connaissances; c'est donc une analyse (intentionnellement) très partielle.

Pour comparer les différentes méthodes, nous allons considérer les axes suivants:

A) Forme d'expression des connaissances;
B) Mécanisme "d'injection" des connaissances;
C) Guidage de l'utilisation des connaissances.

Les points A et B concernent des aspects formels de la méthode considérée: par A on compare les formats dans lesquels les connaissances sont exprimées (règles de transformation, équations, ...); B concerne la façon d'insérer les connaissances au cours de la synthèse, c'est à dire, les mécanismes formels de l'incorporation des connaissances. Le point C est lié aux questions suivantes:

- Pourquoi une certaine connaissance a été sélectionnée en laissant de côté les autres?
- A quel moment les connaissances doivent être appliquées?

Nous essayons donc d'expliciter les critères et mécanismes qui permettent d'introduire la "bonne connaissance" au "bon moment". C'est la question du contrôle de l'utilisation des connaissances; plus que les mécanismes formels, il concerne les stratégies.

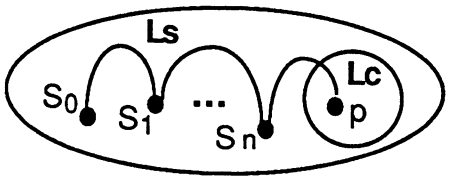
Nous allons discuter l'utilisation des connaissances dans les méthodes suivantes:

- L'approche transformationnelle [Darlington 75], ...
- La méthode des tableaux de [Manna, Waldinger 80]
- La méthode descendante de [Smith 85]
- L'approche stratégique de [Bibel 80]
- L'approche réécriture [Dershowitz 83], [Perdrix 84]

Pour s'aider dans cet exposé nous reprenons l'exemple présenté dans [Biermann 85], à savoir, synthétiser un programme qui élimine d'une liste ses éléments non-atomiques. Par exemple: *f[ ((a b) c d) ] = (c d).*

## L'approche transformationnelle

Par approche transformationnelle nous entendons la méthode développée par [Darlington 75], [Burstall, Darlington 77], [Clark, Darlington 78]. L'idée de cette approche est, à partir d'une spécification donnée, d'appliquer des transformations qui rapprochent graduellement d'un programme , en essayant d'introduire des primitives du langage cible:




avec **S0**: Spécification initiale
**p**: programme exécutable

La mise en place de l'approche transformationnelle n'est possible que si le langage cible fait partie du langage de spécification: $Lc \subseteq Ls$.

Le processus de synthèse d'un programme par la méthode transformationnelle consiste à choisir à chaque moment une transformation parmi un ensemble dont dispose le système; son application sur la spécification en produit une autre, et ainsi de suite, jusqu'à l'obtention d'un programme exécutable.

Nous allons utiliser la notation informelle de [Clark, Darlington 78]. La spécification pour notre exemple serait:

*(s0)* $f(x) <- y$ ***tel que*** $u \in y \Leftrightarrow [u \in x \wedge atom(u)]$

où le symbole "<-" se lit "est défini par".

## *A.-Expression des connaissances*

Les connaissances sont décrites par des équations récursives[1]; il s'agit souvent de définitions où le côté gauche contient seulement le prédicat ou la fonction à définir, et le côté droite est un terme qui donne la définition, parfois récursivement.

Les connaissances utilisées pour la synthèse de notre exemple sont[2]:

*(c1)* $u \in nil = faux$
*(c2)* $l=nil \Rightarrow \neg x \in l$
*(c3)* $u \in a \bullet l = (u=a) \vee u \in l$
*(c4)* $x \neq nil \Leftrightarrow x=car(x) \bullet cdr(x)$
*(c5)* $(a \Leftrightarrow c) \wedge (b \Leftrightarrow d) \Rightarrow (a \vee b \Leftrightarrow c \vee d)$

où l'opérateur "•" est le "cons" habituel de lisp.

L'ensemble de connaissances ci-dessus est redondant; par exemple, $c_2$ vient directement de $c_1$ et de la propriété de substitutivité de l'égalité. Ceci est en accord avec notre notion de connaissances, où les théorèmes sont effectivement disponibles, et non pas seulement déductibles en théorie.

---

[1] [Burstall, Darlington 77] considèrent aussi l'application de lois (commutativité, ...)

[2] ...plus les propriétés de l'égalité et les lois (distributivité...)

## *B.- Injection des connaissances*

L'application de ces connaissances sur une spécification se fait par remplacement d'une instance du côté gauche par l'instance appropriée du côté droit ("*dépliage*") ou vice-versa ("*pliage*"). (Voir exemples dans le paragraphe suivant).

## *C.- Guidage -Présentation de l'exemple*

La stratégie suivie par [Clark et Darlington 78] est de tester l'effet de plusieurs entrées possibles; il ne s'agit pas de valeurs particulières de l'entrée, mais plutôt d'entrées symboliques dont l'ensemble couvre toutes les entrées possibles. Pour notre exemple, l'axiome x=nil ∨ x≠nil en combinaison avec la connaissance (c4) garantit que toutes les entrées sont soit *nil* soit *car(x)* • *cdr(x)*. En considérant ces deux cas, la spécification prend la forme:

$(s_1)$ $f(x) <$- $y$ ***tel que***
$x=nil \Rightarrow [u \in y \Leftrightarrow u \in x \wedge atom(u)]$
$x=car(x)\bullet cdr(x) \Rightarrow [u \in y \Leftrightarrow u \in x \wedge atom(u)]$

En choisissant les cas des listes nil et car(x) • cdr(x), on a laissé de côté d'autres alternatives, comme nil, [u] et $x_1 + x_2$ (concaténation des listes). Dans [Darlington 75], [Clark, Darlington 78], etc, on ne trouve aucun critère pour faire un choix plutôt qu'un autre[1].

Après quelques manipulations[2], on arrive à:

$(s_2)$ $f(x) <$- ***si*** $x=nil$ ***alors*** $t$ ***sinon*** $z$ ***tel que***
$u \in t \Leftrightarrow u \in nil \wedge atom(u)$
$u \in z \Leftrightarrow u \in car(x)\bullet cdr(x) \wedge atom(u)$

D'après $(c_1)$, u∈ nil = faux, donc u∈ t doit être faux. Par l'application de la connaissance $(c_2)$, on peut proposer la condition t=nil, qui nous donne t. On aura:

$(s_3)$ $f(x) <$- ***si*** $x=nil$ ***alors*** $nil$ ***sinon*** $z$ ***tel que***
$u \in z \Leftrightarrow u \in car(x)\bullet cdr(x) \wedge atom(u)$

A présent nous abordons la partie la plus délicate de la synthèse, c'est-à-dire, l'introduction de la récursion. La stratégie suivie par [Darlington 75] est d'effectuer un dépliage, suivi d'une suite de transformations, lesquelles permettront éventuellement de faire le pliage, donc l'appel récursif.

En appliquant (c3) sur la partie **tel que** (dépliage):

$(s_4)$ $u \in z \Leftrightarrow [\,[u=car(x) \vee u \in cdr(x)] \wedge atom(u)]$,

---

[1] Notons que le but de [Clark, Darlington 78] est justement d'utiliser la synthèse comme un moyen d'explorer de façon systématique les divers algorithmes possibles construits à partir d'une spécification.

[2] Rappel: ici nous nous restreignons à l'aspect *connaissance* de l'approche, donc les exemples ne sont pas présentés en détail.

puis, par distribution de $\wedge$ sur $\vee$:

*(s5)* $u \in z \Leftrightarrow [(u=car(x) \wedge atom(u)) \vee (u \in cdr(x) \wedge atom(u))]$;

Après une suite de manipulations (remplacements, simplifications booléennes, introduction des structures du langage cible...) on obtient:

*(s6)* *f(x)* <- ***si*** *x=nil* ***alors*** *nil* ***sinon***
***si*** *atom(car(x))* ***alors*** *v* ***sinon*** *w* ***tel que***
$u \in v \Leftrightarrow [u=car(x) \vee u \in cdr(x) \wedge atom(u)]$
$u \in w \Leftrightarrow [u \in cdr(x) \wedge atom(u)]$

On reconnaît dans la seconde équivalence une instance de la spécification de *f*, ce qui permet d'introduire un appel récursif de la forme *f(cdr(x))*

*(s7)* *f(x)* <- ***si*** *x=nil* ***alors*** *nil* ***sinon***
***si*** *atom(car(x))* ***alors*** *v* ***sinon*** *f(cdr(x))* ***tel que***
$u \in v \Leftrightarrow u=car(x) \vee u \in cdr(x) \wedge atom(u)$.

Dans l'exemple présent, un second appel récursif sera introduit pour résoudre *v*. Quelques transformations sont nécessaires pour y parvenir.

La partie **tel que** ressemble beaucoup à (c3), donc v pourrait être une liste de la forme $car(x) \bullet l$:

*(s8)* *f(x)* <- ***si*** *x=nil* ***alors*** *nil* ***sinon***
***si*** *atom(car(x))* ***alors*** $car(x) \bullet l$ ***sinon*** *f(cdr(x))* ***tel que***
$u=car(x) \vee u \in l \Leftrightarrow u=car(x) \vee u \in cdr(x) \wedge atom(u)$.

La partie **tel que** de (s8) peut être remplacée par la condition suivante, en utilisant (c5):

*(s9)* $u \in l \Leftrightarrow u \in cdr(x) \wedge atom(u)$,

Or, (s9) est une instance de la spécification de *f* avec $l=f(cdr(x))$, ce qui donne finalement le programme:

*(p)* *f(x)* <- ***si*** *x=nil* ***alors*** *nil* ***sinon***
***si*** *atom(car(x))* ***alors*** $car(x) \bullet f(cdr(x))$ ***sinon*** *f(cdr(x))* ♦

Il est clair, d'après cet exemple, que la partie où le guidage de l'utilisation des connaissances est le plus précaire est la suite de transformations précédant le pliage. Quel aurait été, par exemple, le résultat de cette synthèse sans la connaissance propositionnelle (c5) ? Comment savoir qu'elle était nécessaire ?

*Discussion*

L'ensemble des manipulations permises par la méthode transformationnelle engendre un espace énorme d'états intermédiaires, donc une combinatoire importante qu'il est très difficile de maîtriser.
De plus, il est possible de faire des "retours en arrière" dus à la nature bidirectionnelle des connaissances équationnelles.
Par ailleurs, l'emploi du "pattern matching" et du simple remplacement équationnel comme seul moyen pour incorporer les connaissances est assez limitatif, car cela exige que les connaissances se trouvent exactement dans la forme syntaxique qui convient.

D'une façon générale, le degré d'initiative attendue de l'utilisateur, l'utilisation des connaissances y comprise, est assez important. Nous allons citer ici [Clark, Darlington 78] qui, après avoir présenté les connaissances à utiliser au cours de la synthèse d'un programme, écrivent: "Anticipating that such facts will be useful, and using them at the appropriate time, is where some of the cleaverness comes into the program synthesis". Cette citation se passe de commentaires.

## La méthode des tableaux

Les tableaux déductifs de [Manna, Waldinger 80] sont des arrangements à trois colonnes: assertions, buts et programmes. Des règles d'inférence sont fournies pour ajouter des lignes à ces tableaux, en cherchant à obtenir une ligne où le but est identiquement vrai ou bien l'assertion est identiquement fausse: le terme de la colonne "sortie" est le programme désiré. Parmi les règles d'inférence l'on peut distinguer notamment des variantes non-clausales de la résolution.

### *A.- Expression des connaissances*

Les connaissances sont des égalités conditionnelles appelées *transformations*. Elles sont de la forme:

$$r \rightarrow s \textbf{ si } P$$

Ceci signifie que r est équivalent à (et pourra être remplacé par) s lorsque la condition P est vérifiée. Par exemple, une connaissance sur les listes est:

$$a \in l \rightarrow a=car(l) \vee a \in cdr(l) \textbf{ si } l \neq nil$$

### *B.- Injection des connaissances*

Le mécanisme d'injection des connaissances est *l'application de transformations*. Une transformation peut s'appliquer sur une assertion ou bien sur un but. Si on a une ligne *(A, B, O)* et *A* contient un sous- terme *a* unifiable avec *r* par la substitution $\theta$, alors on peut ajouter au tableau la ligne:

*(si $P\theta$ alors $A[a\theta <- s\theta], B\theta, O\theta$)*

où $A[a\theta <- s\theta]$ est l'assertion obtenue en remplaçant $a\theta$ par $s\theta$ dans $A$. Si dans une ligne *(A, B, O)* le but $B$ contient un sous-terme $b$ unifiable avec $r$ par la substitution $\theta$, alors on peut ajouter au tableau la ligne $(A\theta, P\theta \wedge B[b\theta <- s\theta], O\theta)$.

Une fois qu'une ligne a été ainsi ajoutée au tableau, elle peut entrer en rapport avec les autres lignes de celui-ci à l'aide des règles d'inférence du système.

### *C.- Guidage*

Nous n'allons pas développer notre exemple pour cette méthode. Il est clair qu'on peut incorporer au tableau n'importe quelle connaissance unifiable avec le but ou les assertions, ce qui entraîne une combinatoire très importante qui, en absence d'une stratégie globale, pourrait seulement être maîtrisée en utilisant des heuristiques ponctuelles.

### *Discussion*

Par rapport à l'approche transformationnelle, l'utilisation de l'unification au lieu du simple "pattern matching" et de la résolution au lieu du simple remplacement équationnel donne un net avantage aux tableaux déductifs du point de vue des outils formels employés. Ceci se traduit par une flexibilité accrue dans l'exploitation des connaissances.

Plus systématique dans la forme que l'approche transformationnelle, elle demeure, néanmoins, sujette aux avatars de l'utilisation "judicieuse" des connaissances disponibles.

## La méthode descendante

La programmation descendante, issue des techniques du Génie Logiciel, a été utilisée par [Smith 85] comme le point de départ de son approche pour la synthèse de programmes. Elle consiste à découper un problème donné en sous-problèmes tels que leur solution puisse après être recomposée. Si les sous-problèmes sont assez simples, alors ils sont directement résolus; sinon, ils sont à leur tour découpés en sous-problèmes, et ainsi de suite.

Dans [Smith 85] il est proposé de générer des sous-problèmes à l'aide de *schémas de programme*, qui sont des structures du langage cible avec des symboles à instantier par des sous-programmes. Par exemple, le schéma pour la "design theory" appelée "diviser pour règner" est:

```
Div-pour-reg(x) = si primitive(x) alors résoudre-directement(x)
                  sinon
                        (x1, x2) := décompose(x);
                        y1 := Div-pour-reg(x1);
                        y2 := Div-pour-reg(x2);
                        Div-pour-reg := compose(y1, y2).
```

## *A.- Expression des connaissances*

Les axiomes et théorèmes sont des formules de la logique du premier ordre. Dans le système CYPRESS elles sont exprimées en notation préfixée (à la lisp). Par exemple: (gt (plus 1 $i) 0), c'est-à-dire, i+1 > 0.

Les théorèmes sont groupés dans une *Base de Connaissances sur les structures de données* (DSKB). La DSKB contient aussi les définitions de types de données et d'opérateurs utilisables dans le langage de spécification Ls.

## *B.- Injection de connaissances*

Chez Smith, l'injection de connaissances dans la synthèse ne se fait pas "directement", mais par l'intermédiaire d'un sous-système déductif appelé *moteur de préconditions* ("preconditions engine"). Celui-ci sert à résoudre un problème déductif plus général que la démonstration de théorèmes, défini dans [Smith 82a] et appelé *Dérivation de préconditions*: étant donnée une formule B (le but) et des hypothèses H, trouver les conditions additionnelles P telles que B soit une conséquence logique de $P \wedge H$. Une solution triviale pour P est la constante faux, mais on va s'intéresser à des préconditions le plus faibles et simples possible.

Dans le CP1, un raffinement additionnel est introduit: une $\{x_i,\ldots, x_j\}$-précondition d'un but B, sous les hypothèses H est une formule P telle que $[H \wedge P \Rightarrow B] \wedge var(P) \subseteq \{x_i,\ldots, x_j\}$ est valide, où var est l'ensemble des variables libres de son argument. On peut donc limiter les variables de la précondition à obtenir.

[Smith 82a] propose un système formel de déduction naturelle pour le calcul de préconditions. Dans le système CYPRESS, le module qui effectue le calcul de préconditions s'appelle RAINBOW. C'est lui qui fait appel directement aux connaissances.

## *C.- Guidage*

La formation des problèmes de dérivation de préconditions est une démarche entièrement mécanisée, une fois que l'utilisateur a choisi le schéma de programme et les stratégies à utiliser. Ceci sera évident dans l'exemple ci-après.

Néanmoins, au niveau de la dérivation de préconditions elle-même, le problème du guidage de l'utilisation des connaissances apparaît de nouveau. Là, les réponses apportées par Smith sont beaucoup plus maigres.

*Présentation de l'exemple*

Nous rappellons la spécification de notre exemple:

$f(x) = z$ ***tel que***

(1) $\forall\ [u \in z \Leftrightarrow u \in x \wedge atom(u)]$

*où* $f$: *liste* $\rightarrow$ *liste* (type de f)

Nous allons instantier le schéma "Diviser pour règner" en choisissant les opérateurs de décomposition (car et cdr), de la composition (concatenation, représentée par "+") et du cas de base (nil). La récursion sera, elle, déjà introduite dans le schéma, qui aura l'allure suivante:

$f(x) =$ ***si*** $(x=nil)$ ***alors*** $f_0(x)$

***sinon***

$x_1 := car(x);$

$x_2 := cdr(x);$

$y_1 := f_1(x_1);$

$y := f(x_2);\ y_2 := f_2(y);$

$f := y_1 + y_2$

***tel que***

(2) $x=nil \Rightarrow \forall u\ [u \in f_0(x) \Leftrightarrow u \in x \wedge atom(u)\ ]$

(3) $x \neq nil \Rightarrow \forall u[u \in f_1(car(x)) + f_2(f(cdr(x))) \Leftrightarrow u \in x \wedge atom(u)]$

Les formules (2) et (3) caractérisent les fonctions $f_0$, $f_1$ et $f_2$. Si (2) nous conduit immédiatement à la relation $f_0(x)$=nil, (3) présente l'inconvénient de contenir une occurrence de f, qui sera inconnue jusqu'à la fin de la synthèse; il faut donc s'en débarraser. Pour cela nous pouvons utiliser le calcul de préconditions. En effet, nous pouvons changer le problème d'occurrence de fonction en un problème d'occurrence de variables, en introduisant des variables pour les résultats intermédiaires. Ainsi, la spécification de $f_1$ est une $\{x_1,y_1\}$-précondition de (3), et similairement, une $\{y_2,y\}$-précondition de (3) est la spécification de $f_2$. Des solutions possibles sont:

(4) $u \in y_1 \Leftrightarrow (u=x_1) \wedge atom(u)$

(5) $u \in y_2 \Leftrightarrow u \in y$

Une analyse fine de l'utilisation des connaissances chez Smith nous obligerait à rentrer dans les détails de son système déductif pour la dérivation de préconditions (RAINBOW), car c'est celui-ci qui fait appel directement aux connaissances. Nous nous limiterons à dire que le système RAINBOW utilise en général les théorèmes (sauf les axiomes) en tant que règles de remplacement (pas forcément réécriture de gauche à droite). Par exemple, à partir d'une formule $R(\ldots S\ldots)$, une équivalence $P \Leftrightarrow Q$, avec $S\theta=P\theta$ peut permettre d'obtenir $R(\ldots Q\theta\ldots)$.
L'application itérée des remplacements génère une combinatoire considérable. Celle-ci est limitée partiellement par quelques heuristiques mises en place par Smith.

Le travail de Smith présente l'originalité d'avoir isolé les questions relatives à l'utilisation de connaissances dans le cadre du calcul de préconditions, les séparant ainsi de la gestion du processus même de synthèse. Chaque partie possède son propre système formel, donc ses

propres règles d'inférence, et même ses propres heuristiques, ce qui permet de mieux maîtriser les difficultés concernant chacune d'elles. Ces idées vont influencer largement notre approche pour la synthèse de programmes.

## L'approche stratégique

[Bibel 80] propose une méthode transformationnelle pour la synthèse de programmes, basée sur l'application combinée d'un ensemble de stratégies. En partant d'une formule de la logique du premier ordre spécifiant le problème à résoudre, le but de chaque stratégie est d'introduire des transformations. Le programme obtenu à la fin est, lui aussi, une formule logique à laquelle il est possible d'imposer un contrôle (ordre d'exécution) pour en tirer un programme "traditionnel". La synthèse proprement dite se déroule donc entièrement dans le CP1.

On va concentrer notre attention sur les connaissances telles qu'elles sont utilisées par une de ces stratégies, GET-G (Goal Equivalence Transformation, avec but G), et ses instances: GET-DFN, GET-REC, etc.

### *A.- Expression des connaissances*

Les connaissances sont des équivalences, qui peuvent être conditionnées par une formule logique P:

$$P: (t_1 \Leftrightarrow t_2)$$

Exemple:

*(c1)* $car(x) \notin y \Rightarrow$
$[u \in y \Leftrightarrow u \in x \wedge atom(u)] \Leftrightarrow [u \in y \Leftrightarrow u \in cdr(x) \wedge atom(u)]$

### *B.- Injection des connaissances*

Les équivalences s'appliquent sur la spécification courante comme des règles de réécriture conditionnelles; le résultat en est une autre spécification.

### *C.- Guidage*

La façon de guider l'application des connaissances se comprend mieux à l'aide d'un exemple, qui sera le même que pour les cas précédents, à savoir, le calcul d'une sous-liste ne contenant que les éléments atomiques de la liste argument.

*Présentation de l'exemple.*

Rappelons la spécification de départ:

*(1)* $f(x) = y$ ***tel que*** $\forall u[u \in y \Leftrightarrow u \in x \wedge atom(u)]$

Comme dans [Biermann 85] on ne va pas s'occuper du cas trivial x=nil, qui produit la liste vide; on va plutôt définir une fonction g(x) pour le cas général:

(2) $f(x) = \textbf{si } x=nil \textbf{ alors } nil \textbf{ sinon } g(x)$

(3) $g(x) = y \textbf{ tel que}$
$x \neq nil \wedge [u \in y \Leftrightarrow u \in x \wedge atom(u)]$

Cette spécification est transformée par les stratégies GUESS-DOMAIN et GET-DFN (Disjunctive Normal Form) en:

(4) $g(x) = y \textbf{ tel que } x \neq nil \wedge$
$[u \in y \Leftrightarrow u \in x \wedge atom(u)] \wedge car(x) \in y$
$\vee$
$[u \in y \Leftrightarrow u \in x \wedge atom(u)] \wedge car(x) \notin y$

Maintenant la stratégie GET-REC (RECursion) va essayer de réécrire cette spécification de façon à introduire des appels récursifs. Pour que la récursion termine, il faut qu'elle s'effectue sur une entrée chaque fois plus petite; celle-ci s'obtient en enlevant de l'entrée l'élément objet de la conjecture: x\car(x) = cdr(x). La réécriture cherchée doit produire, à partir de (4), une instance de la spécification de départ avec x substitué par cdr(x).

Pour l'alternative car(x) ∈ y (le cas "réussite") on définit y=car(x)•y'; on suppose car(x) ∉ y'.

Pour l'alternative car(x) ∉ y, on observe le fait que si car(x) ne fait pas partie de la sortie, alors on peut l'éliminer de l'entrée, ce qui est exprimé par la connaissance ($c_1$).

Si le système ne dispose pas directement de cette connaissance, ce qui est, d'ailleurs, très vraisemblable, alors c'est un démonstrateur de théorèmes qui essaie de la trouver. [Bibel 80] argumente qu'en pratique l'espace de recherche des équivalences est restreint par le fait que le nombre de littéraux concernés est relativement petit.
On fait un raisonnement similaire pour y', puisque car(x)∉y', donc on a:

(5) $g(x) = y \textbf{ tel que } x \neq nil \wedge$
$y = car(x) \bullet y' \wedge [u \in y' \Leftrightarrow u \in x \wedge atom(u)] \wedge car(x) \in y$
$\vee$
$[u \in y \Leftrightarrow u \in x \wedge atom(u)] \wedge car(x) \notin y$

La recherche d'une récursion réussit car on trouve des intances de la spécification de f, ce qui nous permet d'introduire des appels récursifs:

(6) $g(x) = y \textbf{ tel que } x \neq nil \wedge$
$y = car(x) \bullet y' \wedge y'=f(cdr(x)) \wedge car(x) \in y$
$\vee$
$y=f(cdr(x)) \wedge car(x) \notin y$

et immédiatement:

(7) $g(x) = y$ ***tel que*** $x \neq nil \wedge$
$y = car(x) \bullet f(cdr(x)) \wedge car(x) \in y$
$\vee$
$y = f(cdr(x)) \wedge car(x) \notin y$

Finalement, la stratégie GET-EP (Evaluable Predicate) essaie de transformer la condition car(x)∈y pour en obtenir une qui ne dépend pas de la sortie; en particulier, on voudrait savoir quelle propriété de car(x) contrôle son appartenance à y, c'est-à-dire:

(8) $\Pi(car(x)) \Leftrightarrow car(x) \in y$

Si une telle équivalence n'est pas disponible dans le système, alors il fait appel au démonstrateur de théorèmes pour la trouver. Si ceci s'avère encore impossible, alors un "explorateur de modèles" est invoqué pour générer des exemples et les généraliser, pour en arriver au théorème désiré. Supposons qu'on trouve Π=atom.

En imposant un ordre d'exécution, le programme final sera:

$f(x) =$ ***si*** $x=nil$ ***alors*** $nil$ ***sinon*** $g(x)$
$g(x) =$ ***si*** $atom(car(x))$ ***alors*** $car(x) \bullet f(cdr(x))$ ***sinon*** $f(cdr(x))$ ♦

Il est intéressant de remarquer que la stratégie GET-EP de LOPS ne fait rien d'autre que résoudre un problème de calcul de préconditions; dans l'exemple présenté, c'était la recherche de la condition (8). Or, en posant $x_1$=car(x), ceci revient à chercher une plus faible $\{x_1\}$-précondition de car(x) ∈ y.

*Discussion*

Les spécificités du guidage de la recherche des connaissances nécessaires dépendent de la stratégie en cours d'application; presque chaque stratégie de LOPS fait appel aux connaissances. Nous ne disposons pas d'informations précises sur les méthodes de Bibel pour caractériser les connaissances nécessaires, si ce n'est cette idée générale de rechercher les connaissances nécessaires à un instant donné à partir du choix des lignes générales pour la synthèse (stratégies).

## L'approche par réécriture

[Dershowitz 83], [Perdrix 84], [Kodratoff, Picard 83] ont proposé d'appliquer la procédure de complétion de [Knuth, Bendix 70] pour synthétiser des programmes. Ceux-ci sont des systèmes de réécriture[1] où les entrées et sorties sont mises dans une forme prédicative, comme en PROLOG. Par exemple, un programme pour notre exemple pourrait être:

(1) $P(nil, nil) \rightarrow vrai$
(2) $P(nil, x \bullet l) \rightarrow faux$
(3) $P(x \bullet l, nil) \rightarrow \neg atom(x) \wedge P(l, nil)$
(4) $P(x \bullet l, y \bullet m) \rightarrow \neg atom(x) \wedge P(l, y \bullet m)$
$\vee (x=y) \wedge atom(x) \wedge P(l, m)$

[1] Voir définitions et propriétés dans [Huet, Oppen 80]

L'interprète de ces programmes est aussi la procédure de complétion. Pour trouver une valeur de sortie z, étant donnée une entrée x, on ajoute au programme la règle (but):

$$P(x, z) \rightarrow réponse(z),$$

et on fait tourner l'algorithme de complétion, qui calculera les paires critiques et ajoutera des nouvelles règles, jusqu'à ce qu'il y en ait une de la forme:

$$réponse(t) \rightarrow vrai,$$

où t est le résultat. Par exemple, pour l'entrée (a (b)), c'est-à-dire, a•((b•nil)•nil) on aurait la séquence de calcul:

(5) $P(a•((b•nil)•nil), z) \rightarrow réponse(z)$

Par (5) et (4) avec x / a, l / (b•nil)•nil, z / y•m:[1]

(6) $P((b•nil)•nil, m) \rightarrow réponse(a•m)$

Par (6) et (3) avec x / (b•nil), l / nil, m / nil:

(7) $réponse(a•nil) \rightarrow vrai$[2] ♦

Pour la synthèse de programmes on exprime la spécification sous la forme de règles de réécriture. Pour notre exemple ce serait:

$(s_1)$ $u \in y \Leftrightarrow u \in l \wedge atom(u) \rightarrow P(l, y)$

## A.- Expression des connaissances

Les théorèmes nécessaires à la synthèse sont, eux aussi, exprimés comme des règles de réécriture. La spécification, les connaissances et le programme obtenu sont donc tous des règles de réécriture. Par exemple:

$(c_1)$ $u \in nil \rightarrow faux$[3]

## B.- Injection des connaissances

Le mécanisme d'insertion de connaissances devient trivial à cause de cette uniformité syntaxique des spécifications, connaissances et programme. Les théorèmes nécessaires (définitions et théorèmes auxiliaires) sont tout simplement introduits dans le même système de réécriture que la spécification. Le système de réécriture ainsi formé constitue l'entrée de l'algorithme de

---

[1] Après l'évaluation des prédicats et la simplification booléenne

[2] L'ordre des termes, pour donner un sens aux règles engendrées, doit être tel que réponse(x) soit inférieur à tout autre terme sauf à la constante vrai

[3] ceci fait partie de la définition de $\in$.

complétion. A la sortie on obtient un programme correct vis-à-vis de la spécification, si la complétion termine normalement.

Afin d'illustrer brièvement la méthode, on va seulement calculer un des cas de base du programme. On utilise la connaissance ($c_1$), ainsi que:

*(c2)* $faux \wedge P \rightarrow faux$
*(c3)* $faux \Leftrightarrow faux \rightarrow vrai$

En superposant ($s_1$) et ($c_1$) sur l'occurrence $u \in l$,
*(s2)* $u \in y \Leftrightarrow faux \wedge atom(u) \rightarrow P(nil,y)$

Par ($c_2$) et ($s_2$) :
*(s3)* $u \in y \Leftrightarrow faux \rightarrow P(nil,y)$

Par ($s_3$) et ($c_1$) :
*(s4)* $faux \Leftrightarrow faux \rightarrow P(nil,y)$

Finalement, par ($s_4$) et ($c_3$) :
$P(nil, nil) \rightarrow vrai$ ♦

Pour cela, il faut un ordre sur les termes tel que (faux ⇔ faux) > P(nil, y) mais par contre P(nil, y) > vrai. Le choix d'un ordre affecte largement le programme qui en résulte [Dershowitz 83].

### *C.- Guidage*

Le problème dans l'approche par réécriture, telle qu'elle est présentée par [Dershowitz 83], est qu'il faut fournir tout au début la totalité des connaissances nécessaires. Les difficultés d'un tel pré-requis ont été évoquées plus haut.

Dans [Perdrix 84] sont proposées des stratégies permettant de guider l'introduction de la récursion similaires à celles de [Smith 85], mais on ne trouve toujours pas les critères pour faire appel à de connaissances spécifiques.

## L'état de l'art.- Couclusion

Les différentes formes d'expression des connaissances que nous avons présentées sont toutes des variantes du calcul de prédicats du premier ordre ou des équations algébriques. Nous croyons que, en ce qui concerne ce point-ci, les différences ne sont pas substantielles par rapport à la puissance et la capacité des systèmes de synthèse.

On trouve davantage de différences pour les mécanismes d'injection de connaissances; celles-ci sont tout simplement ajoutées à la spécification dans [Dershowitz 83], et y sont directement appliquées dans [Manna, Waldinger 80], ou traitées par un mécanisme intermédiaire dans [Smith 85]. Cette diversité reflète la différence des formalismes utilisés.

Finalement (last, but no least), là où l'on apprécie les différences les plus importantes c'est dans

la question du contrôle de l'utilisation des connaissances. En essayant d'en faire une classification, l'on va grouper les approches pour la synthèse de programmes de la façon suivante:

• Systèmes de chainage avant[1]. Les connaissances nécessaires sont données au début de la synthèse et ensuite mises en rapport avec la spécification pour en produire des variantes. On place dans cette catégorie la méthode transformationnelle, les tableaux déductifs, et la réécriture à la Dershowitz. A cette catégorie appartiennent la plupart des systèmes proposés jusqu'à présent.

• Systèmes de chainage arrière (guidage par le but)[1]. Etant donnée une situation au cours d'un processus de synthèse, le système est capable de caractériser les connaissances dont il a besoin pour continuer. Ceci est le cas, dans une certaine mesure, des systèmes CYPRESS et LOPS. En effet, à partir du choix de quelques lignes générales du déroulement de la synthèse (par exemple, le choix d'un schéma de programme), ils essayent de récupérer les connaissances nécessaires à celle-ci. Dans CYPRESS le choix d'un schéma de programme et d'une "design method" détermine les problèmes de calcul de préconditions qui feront appel aux connaissances. Dans LOPS ce sont les stratégies qui fixent les modalités d'appel aux connaissances.

Le chaînage avant présente les inconvénients suivants:

i) Il faut fournir au départ toutes les connaissances nécessaires à la synthèse. Ceci est possible dans les exemples des livres car quelqu'un a fait d'abord la synthèse et ensuite recensé les connaissances utilisées. En fait, aucun moyen n'a jamais été proposé pour déterminer à l'avance toutes les connaissances qui seront nécessaires à une synthèse.

ii) La nature générative de ces approches entraîne une explosion combinatoire incontrolable. Même s'il est possible de mettre en place des stratégies pour limiter les variantes engendrées à partir de la spécification, une augmentation de la taille du problème (p.ex., quantité de connaissances) amplifie énormement l'espace de recherche.

Le guidage par le but présente, néanmoins, un paradoxe: le choix des lignes générales pour le déroulement de la synthèse (p.ex. le schéma de programme à instancier) doit se fonder sur les connaissances disponibles concernant le problème à résoudre. Or, quelles connaissances faut-il consulter si l'on n'a pas encore fait le choix d'une stratégie de synthèse? "Ce paradoxe, à notre avis, n'en est pas un: les connaissances nécessaires au choix peuvent être d'un niveau plus général que celles nécessaires à l'obtention du programme. Par exemple, une preuve par récurrence ne peut s'appliquer que sur des objets ayant une structure inductive. Savoir qu'un domaine peut être défini inductivement est plus général que connaître exactement la propriété récursive nécessaire à la dérivation du programme." [Potet 88]. Ceci donne l'idée de ce qui pourrait être la "caractérisation des connaissances nécessaires", idée qui sera développée dans les chapitres suivants.

---

[1] On établit ainsi une analogie entre les techniques de la synthèse de programmes et les stratégies générales de l'IA pour la recherche d'une solution dans un espace d'états [Bar, Feigenbaum 81]. Le guidage par le but a été incontestablement une découverte importante pour l'efficacité des systèmes d'IA.

*Le second [précepte], de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre [Descartes 1636]*

# 2.- Notre approche pour la synthèse de programmes

Ici nous décrivons les caractéristiques principales de notre méthode pour la synthèse de programmes. Le lecteur pourra en trouver une description plus détaillée dans [Potet 88]. D'autres papiers décrivant notre travail en synthèse de programmes sont [Brena et al. 85], [Jacquet, Potet 86].

*Rappel.- Cadre de notre recherche*

Nous considérons le problème de la construction de programmes à partir de spécifications déclaratives et complètes, ce qui nous place dans la ligne des travaux de [Darlington 75], [Manna, Waldinger 80], [Bibel, Horning 84], [Dershowitz 83], [Smith 85], etc.

Dans [Potet 88] on parle de *synthèse assistée de programmes* pour mettre en valeur le rôle de l'utilisateur dans notre approche. Nous récupérons ainsi des résultats de la recherche en construction assistée de programmes [Gresse 84]. Ceci souligne aussi notre volonté de faire des propositions effectivement applicables en pratique.

## 2.1.- Lignes générales de notre approche

Nous pouvons qualifier notre approche de déductive dans le sens où elle cherche à prouver l'énoncé de façon à ce que de la trace de cette preuve on puisse extraire un programme; nous parlons donc de *preuves constructives*.. A la base de ce genre d'approches est l'assimilation des programmes à des preuves d'énoncés [Howard 80]. Aborder la construction comme un processus déductif permet de maitriser ce qui est fait au moins au niveau conceptuel [Potet 88].

*Guidage par le but*

Par rapport à l'utilisation des connaissances, décrite dans la section 1.4 pour les différentes approches de la synthèse de programmes, notre système est *guidé par le but* dans le sens où on fait un choix "a priori" des lignes générales de la preuve de l'énoncé, et à partir de ces lignes générales on détermine les propriétés du domaine du problème nécessaires à la synthèse en cours.

Plus concrètement, le choix des lignes générales d'une preuve revient à sélectionner un schéma de programme et une stratégie d'instanciation de ce schéma. Les parties non encore instanciées du schéma représentent les sous-problèmes qu'il reste à résoudre.

L'utilisation d'un schéma de programme, pour un problème donné, nécessite la vérification d'un ensemble de "conditions d'applicabilité" du schéma. Ces conditions sont codées dans les règles d'inférence de notre système, sous la forme de *prémisses à vérifier*. Elles sont utilisées pour caractériser les sous-problèmes faisant partie du schéma; cette caractérisation permettra de former des énoncés de sous-problèmes, lesquels pourront par la suite entraîner un processus de synthèse, etc. C'est donc, comme [Smith 85] une méthode descendante et guidée par le but dans le sens où, à partir du choix d'une forme de programme, les sous-problèmes à résoudre sont engendrés.

Toutefois, le choix d'un schéma de preuve n'est pas tout à fait arbitraire, et dans [Potet 88] sont ébauchés des critères pour fonder ce choix. (Par exemple, un schéma récursif ne sera intéressant que pour des domaines définis inductivement).

Nous allons distinguer deux types de prémisses à vérifier:

a) Conditions relatives aux sous-problèmes qu'il reste à résoudre;
b) Conditions relatives aux propriétés du domaine du problème.

La preuve des prémisses du type (a) doit s'effectuer dans le système de preuves constructives, car il s'agit de construire des sous-programmes répondant à ces conditions. Par contre, la preuve des prémisses du type (b) peut être menée dans n'importe quel système logique du premier ordre classique; elle fera appel aux connaissances sur le domaine du problème considéré.

Nous mettons ainsi en évidence trois niveaux de la preuve d'un énoncé:

i) Choix des lignes générales pour la preuve de l'énoncé;
ii) Preuve sur la construction;
iii) Preuve sur le domaine du problème.

## *Connaissances requises*

Les preuves pour la construction du programme font appel aux preuves sur le domaine du problème. Ceci conforme une situation où des connaissances de nature différente interagissent de façon complexe. Ces connaissances sont résumées dans le tableau suivant:

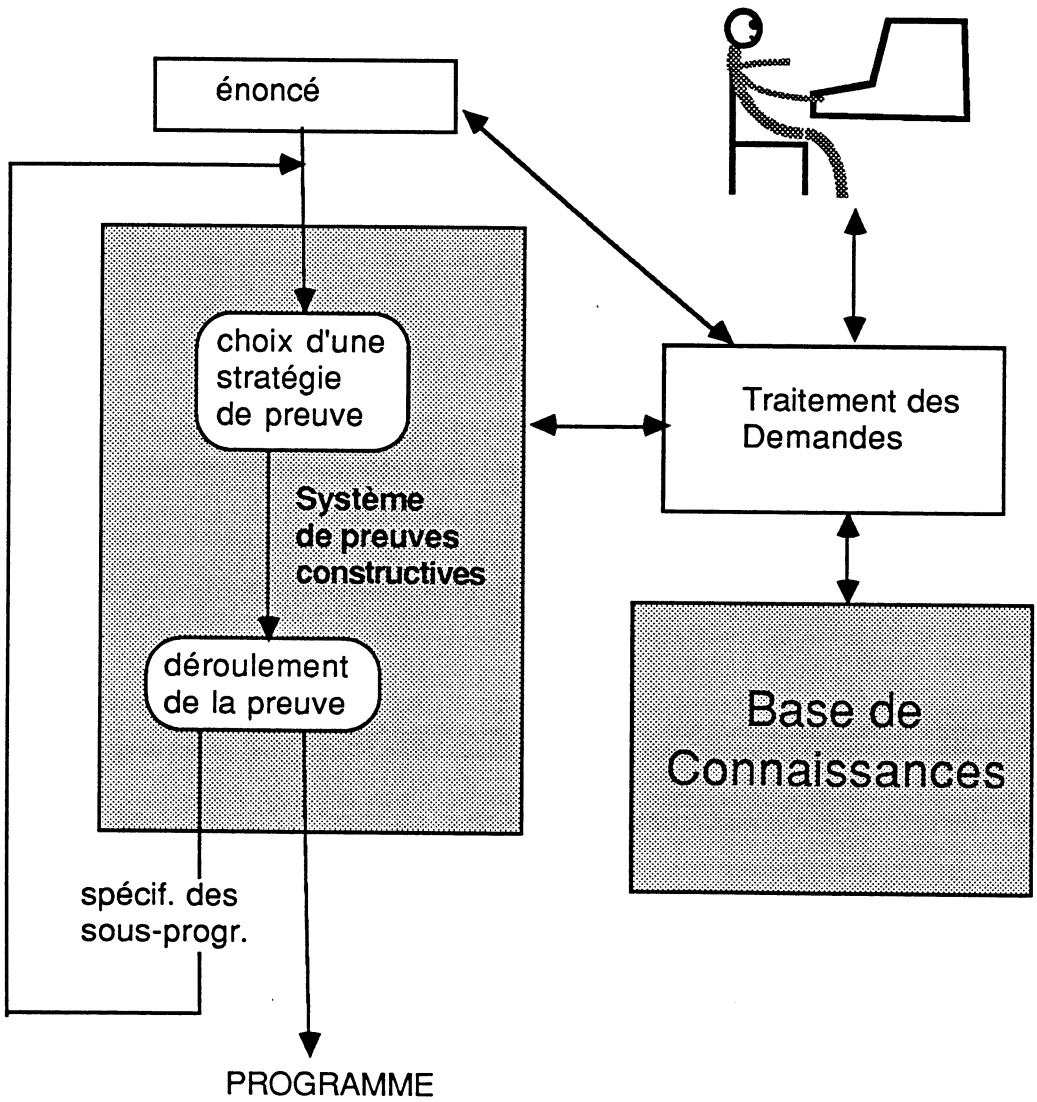
**Niveau de Preuve**

| Connaissances relatives à la programmation | Connais.relatives au domaine du problème |
|---|---|
| **Choix des lignes générales** | |
| Schémas de programmes; stratégies d'instanciation des schémas; | Connaissances sur la structure du domaine.- types de données; |
| **Déroulement de la preuve** | |
| Connaissance du système formel employé et de ses mécanismes pour générer des sous-problèmes | Propriétés des types de données concernés (théorèmes intéressants) |

Le fait de distinguer les différentes connaissances nécessaires à chaque étape de la synthèse peut aider à guider la recherche des connaissances et à réduire ainsi la combinatoire [Gresse 84].

## *Architecture*

Les différent niveaux de la synthèse et les types de connaissances mises en jeu sont matérialisés dans l'architecture de notre système, comme on peut le constater dans la figure suivante:




Les points qu'il faut retenir de cette architecture sont les suivants:

- Nous avons défini deux modules principaux, le module de *preuves constructives de programmes* (PCP) et la *base de connaissances* (BC), qui s'occupent respectivement des aspects relatifs à la construction de programmes et aux connaissances sur les domaines d'application. Ces deux modules réflètent la distinction entre les connaissances relatives à la programmation elle-même et les connaissances relatives au domaine du problème, distinction qui a été évoquée dans la section 1.2.

- Présence au centre même du système d'un module de *traitement de demandes*, centralisant l'échange d'information entre l'utilisateur, le PCP et la BC. Ceci met en valeur le caractère interactif de notre système.

Le découpage de notre système en modules, et plus particulièrement le couple PCP-BC, vise donc à permettre la *caractérisation des connaissances* nécessaires au cours d'une synthèse. Nous prolongeons ainsi de façon conséquente le guidage par le but vers l'utilisation des connaissances.

La modularité de notre architecture se traduit par le fait que chacun de ses composants n'est pas concerné par les méthodes mises en œuvre par les autres; ainsi, par exemple, un changement de l'organisation des connaissances de la BC ne serait apprécié de l'extérieur que par un changement de performance.

La vocation de notre système pour la *synthèse assistée* veut que l'utilisateur puisse se substituer à n'importe quel de ses composants.

## 2.2.- Description de notre méthode

Une fois établis les critère de base définissant notre approche pour la synthèse de programmes, il faut effectuer des choix permettant de détailler les mécanismes concrets pour sa mise en œuvre.

Il faut, notamment: [Potet 88]

1.- Préciser les langages source et cible;

2.- Définir une logique liant programmes et propriétés sous-jacentes au domaine du problème;

3.- Proposer un système déductif permettant de rechercher des instances de propriétés dans une théorie du premier ordre;

4.- Préciser des critères pour déterminer *a priori* les lignes générales de la preuve;

5.- Définir des stratégies permettant de guider le déroulement de la preuve.

### 2.2.1.- Langage cible

Le langage cible est un langage fonctionnel comportant un ensemble de *fonctions de base*, la composition fonctionnelle, la conditionnelle et la récursion. La forme générale de ces quatre constructions est la suivante:

*Fonctions de base*

$f(x) \leftarrow b(x),\ b \in B$

*Conditionnelle*

$f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$ **sinon** $f_2(x)$

*Composition fonctionnelle*

$f(x) \leftarrow g(h_1(x),\ldots,h_n(x))$

*Récursion*

$f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$
**sinon** $g(f(h_{1,1}(x),\ldots h_{1,n}(x)),\ldots,f(h_{k,1}(x),\ldots,h_{k,n}(x)),\ x)$

où "$\leftarrow$" est le symbole de définition de programme; B est un ensemble de fonctions connues par les système. Les "mots clefs" du langage sont en gras. Pour le reste, les notations ont le sens habituel.

Un programme est un ensemble de définitions de la forme décrite:

$$\mathcal{P} = \bigcup \{f_i(x_i) \leftarrow T_i\}$$

où tout symbole fonctionnel dans $T_i$ est soit une fonction primitive soit est définie dans $\mathcal{P}$.

## 2.2.2.- Langage des énoncés

Les spécifications de programmes que nous considérons sont de la forme:

PROFIL : $f(x : T) \rightarrow t$
PREC : $P(x)$
POST : $Q(x, f(x))$

où f est le nom du programme à synthètiser; T et t sont respectivement les "types" de l'entrée et de la sortie[1]. Les types pourront être *polymorphes* [Milner 78], ce qui nous donne la possibilité de paramétrer certains types par d'autres types; ainsi on pourra avoir des ensembles de naturels, noté *ens[nat]*. P et Q sont des formules du calcul des prédicats du premier ordre typé,[2] appelées respectivement *précondition* et *postcondition*. L'extension de P constitue l'ensemble des entrées valides, et Q dénote la relation entre l'entrée et le résultat. Les symboles qui apparaissent dans P et Q sont pris d'une bibliothèque de "symboles connus", c'est-à-dire, de symboles pour lesquels on dispose d'une définition.

L'interprétation d'un énoncé est la suivante: Un énoncé est vrai ssi il existe un programme $\mathcal{P}$ tel que:

$$P(x) \wedge y=\text{eval}(f,\mathcal{P},x) \Rightarrow y \neq \perp \wedge Q(x,y)$$

---

[1] Pour le moment "type" est identifié à "sorte", dans le sens [Goguen et al 78], c'est-à-dire, identificateur de type.

[2] Le choix du CP1 typé pour le langage d'énoncés est justifié dans [Potet 88] d'après des critères d'expressivité et de complexité des preuves.

Cette interprétation inclut proprement la formule habituelle $P(x) \Rightarrow Q(x,y)$, mais prend en plus en compte les fonctions partielles.
Par exemple, une spécification pour un programme de tri d'une liste est:

PROFIL : tri(l : liste[t]) $\rightarrow$ liste[t]
PREC : vrai (aucune)
POST : perm(l, tri(l)) $\wedge$ ord(tri(l)),

où le type t possède un ordre total dénoté $<$, et les prédicats perm et ord ont le sens suggéré intuitivement.

Cette spécification est *générique* dans ce sens qu'elle peut être instanciée pour s'adapter aux listes d'entiers avec l'opérateur $\leq$, aux listes de mots avec l'ordre lexicographique, etc.

## 2.2.3.- Le système de preuves

Nous avons voulu expliciter le système formel sous-jacent à notre approche pour pouvoir, sur cette base, clarifier le rôle des stratégies de synthèse. Cependant, dans une réalisation il pourrait rester implicite, si cela convient pour des raisons d'efficacité.

Nous définissons un système formel constitué par les composants suivants:

- Le langage est celui des énoncés;

- Les axiomes sont les énoncés d'un ensemble de fonctions de base de la forme:

$$P(x) \rightarrow Q(x,b(x)),\ b \in \mathfrak{B}$$

- Les règles d'inférence sont associées à des schémas de programme. Pour un schéma de la forme:

$$f(x) \leftarrow F(f_1(x),\ldots,f_n(x)),$$

la règle d'inférence correspondante est de la forme:

$$\frac{\begin{array}{c} P_1(x) \rightarrow Q_1(x, f_1(x)) \\ \ldots\ \ldots \\ P_n(x) \rightarrow Q_n(x, f_n(x)) \\ \\ P(x) \Rightarrow PREC(x) \\ P(x) \wedge POST(x,y) \Rightarrow Q(x,y) \end{array}}{P(x) \rightarrow Q(x,\ f(x))}$$

où $P_i(x) \rightarrow Q_i(x, f(x))$ sont des énoncés de sous-programmes, et PREC et POST sont les

formules décrivant respectivement la précondition et la postcondition associées à la construction $F(f_1(x),\ldots,f_n(x))$ décrite par la règle. PREC et POST font intervenir les prédicats $P_i$ et $Q_i$ qui caractérisent les fonctions $f_i$. Puisque les $P_i$ et $Q_i$ sont inconnus, nous allons nous servir des prémisses de la forme $P(x) \Rightarrow PREC(x)$ et $P(x) \wedge POST(x,y) \Rightarrow Q(x,y)$ pour les caractériser. Celles-ci deviendront les énoncés des sous-programmes engendrés.

Le sens intuitif des règles d'inférence est le suivant: si l'on prouve toutes les prémisses de la règle, alors la conclusion $P(x) \rightarrow Q(x, f(x))$ est automatiquement prouvée.

Les prémisses correspondent aux "conditions d'applicabilité" de [Smith 85].

## *Construction du programme*

Les constructions du langage cible sont associées aux preuves de la façon suivante:

- Aux axiomes correspond le programme vide $\varnothing$;

- Aux règles d'inférence R faisant intervenir comme prémisses les énoncés $P_i(x) \rightarrow Q_i(x, f_i(x))$, elles-mêmes associées aux programmes Pi, correspond le programme $P_1 \cup,\ldots, \cup P_n \cup \{f(x) \leftarrow F(f_1(x),\ldots,f_n(x))\}$. Ainsi, "l'image" des règles d'inférence, du point de vue de la construction de programmes, est de la forme:

$$\frac{\begin{array}{c}\mathcal{P}_1 \\ \ldots \\ \mathcal{P}_n\end{array}}{\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_n \cup \{f(x) \leftarrow F(f_1(x_1),\ldots,f_n(x_n))\}}$$

## *Exemple.- Règle de la conditionnelle*

Rappel: le schéma de la conditionnelle est:

$f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$ **sinon** $f_2(x)$

La règle d'inférence associée à l'introduction de la conditionnelle est la suivante:

$$\frac{\begin{array}{c}P_0(x) \rightarrow Q_0(x) \equiv (C(x)=\text{vrai}) \\ P_1(x) \rightarrow Q_1(x,f_1(x)) \\ P_2(x) \rightarrow Q_2(x,f_2(x)) \\ P(x) \Rightarrow P_0(x) \wedge ((Q_0(x) \wedge P_1(x)) \vee (\neg Q_0(x) \wedge P_2(x))) \\ P(x) \wedge Q_0(x) \wedge Q_1(x,y) \Rightarrow Q(x,y) \\ P(x) \wedge \neg Q_0(x) \wedge Q_2(x,y) \Rightarrow Q(x,y)\end{array}}{P(x) \rightarrow Q(x,y)}$$

où les trois première lignes sont des énoncés pour les programmes calculant C(x), $f_1(x)$ et $f_2(x)$ respectivement; la quatrième ligne exprime le fait que les préconditions de C(x), $f_1(x)$ et $f_2(x)$ doivent être compatibles avec P(x), et les deux dernières lignes assurent que dans les deux cas de la conditionnelle la condition liant la sortie à l'entrée est respectée.

L'image de cette règle au niveau de la construction du programme est la suivante:

$$\frac{\begin{array}{c}\mathcal{P}_0 \\ \mathcal{P}_1 \\ \mathcal{P}_2\end{array}}{\mathcal{P}_0 \cup \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{\, f(x) \leftarrow \textbf{si } C(x) \textbf{ alors } f_1(x) \textbf{ sinon } f_2(x) \,\}}$$

## *La génération de sous-programmes*

En supposant qu'un certain schéma est applicable à une spécification donnée, on peut caractériser logiquement les sous-programmes à trouver en se servant des prémisses de la règle d'inférence choisie. Nous allons donc appliquer les règles d'inférence du but vers les prémisses. Notre démarche est résumée par la formule suivante:

CHOIX D'UN SCHEMA $\Rightarrow$ ENSEMBLE DES PREMISSES A PROUVER

En effet, le choix d'une schéma de programme nous ramène à sa règle d'inférence associée, dont on connaît la conclusion $P(x) \rightarrow Q(x,y)$, qui est en fait l'énoncé du programme à construire. Il reste à trouver des instances pour les prémisses et prouver celles-ci.

Certaines prémisses, de la forme $P_i(x_i) \rightarrow Q_i(x_i, f_i(x_i))$, sont des énoncés de programmes qu'il faudra soit construire soit trouver dans la bibliothèque de programmes du système. Nous dirons que ces prémisses sont des *énoncés des sous-problèmes* engendrés. Elles deviendront ensuite des énoncés à prouver constructivement, ce qui itère le processus.

Les instances de $P_i$ et $Q_i$ devront se conformer aux principes suivants:

- Les $P_i$ doivent décrire le plus petit domaine pour lequel les fonctions $f_i$ doivent être définies.

- Les $Q_i$ doivent être:

    - Plus "simples" que la post-condition initiale;
    - Les plus faibles possibles
    - Avoir des solutions si l'énoncé initial est démontrable par la preuve choisie

La première de ces trois conditions tient au fait évident que si la spécification d'un sous-problème est plus compliquée que l'énoncé de départ, alors la règle d'inférence appliquée n'est guère adéquate à la situation.

## *L'instanciation des schémas*

Le processus de preuve constructive est un processus d'instanciation graduelle des règles, donc des schémas de programme. En effet, quand un schéma a été complètement instancié, il devient un programme.

Notre méthode pour instancier les schémas est, comme l'introduction des schémas elle-même, guidée par le but. Nous proposons, comme [Smith 85], de choisir *a priori* des instances pour quelques symboles du schéma retenu, permettant ainsi -en faisant une analogie avec les systèmes d'équations- de "résoudre" les autres.

Les différents choix d'instanciations *a priori* correspondent à autant de stratégies de contrôle du processus de synthèse.

Par exemple, pour la conditionnelle (voir schéma et règle plus haut) on peut choisir d'instancier soit la condition C(x), soit une des branches $f_1(x)$ ou $f_2(x)$.

Si l'on choisit de fixer la condition C(x), on pourrait avoir l'instanciation suivante de la règle d'inférence:

$P_0(x)$ : $P(x)$
$Q_0(x)$ : $C(x)=vrai$
$P_1(x)$ : $P(x) \wedge C(x)$
$P_2(x)$ : $P(x) \wedge \neg C(x)$

$Q_1$ et $Q_2$ seront caractérisés par les relations suivantes:

$(r_1)$ $P(x) \wedge C(x) \wedge Q_1(x,y) \Rightarrow Q(x,y)$
$(r_2)$ $P(x) \wedge \neg C(x) \wedge Q_2(x,y) \Rightarrow Q(x,y)$,

qu'il faut donc "résoudre" pour les inconnues $Q_1$ et $Q_2$. Une méthode pour faire cela est la dérivation de préconditions de [Smith 82].

## *Dérivation d'hypothèses*

Rappel: La dérivation de préconditions a été introduite dans la section 1.4.3, dans la description de *l'approche descendante* de D.R.Smith: étant donnée une formule B (le but) et des hypothèses H, trouver les conditions additionnelles P telles que B est une conséquence logique de $P \wedge H$. Une solution triviale pour P est la constante faux, mais on va s'intéresser à des préconditions le plus faibles et simples possible.

Dans le CP1, un raffinement additionnel est introduit: une $\{x_i,\ldots, x_j\}$-précondition d'un but B sous les hypothèses H est une formule P telle que $[H \wedge P \Rightarrow B] \wedge var(P) \subseteq \{x_i,\ldots, x_j\}$ est valide, où *var* obtient les variables libres de son argument. On peut donc limiter les variables de la précondition à obtenir.

Nous allons appliquer le terme *dérivation d'hypothèses additionnelles* plutôt que *dérivation de préconditions*, pour éviter des confusions avec les préconditions des énoncés de problèmes de synthèse.

La dérivation d'hypothèses s'applique à la recherche de valeurs pour les inconnues des prémisses, comme $Q_1$ et $Q_2$ de l'exemple de la section précédente. En suivant ce même exemple, nous pouvons obtenir à partir de ($r_1$) et ($r_2$), par des transformations du calcul propositionnel:

($r_3$) $\quad Q_1(x,y) \Rightarrow [P(x) \wedge C(x) \Rightarrow Q(x,y)]$
($r_4$) $\quad Q_2(x,y) \Rightarrow [P(x) \wedge \neg C(x) \Rightarrow Q(x,y)]$

Or, trouver des instances pour $Q_1$ et $Q_2$ revient à *trouver les conditions* , sur les variables x et y, nécessaires pour rendre valides les formules suivantes:

($r_5$) $\quad P(x) \wedge C(x) \Rightarrow Q(x,y)$
($r_6$) $\quad P(x) \wedge \neg C(x) \Rightarrow Q(x,y)$

## *Rôle architectural de la dérivation d'hypothèses*

La dérivation d'hypothèses joue un rôle primordial dans notre approche. Elle permet d'établir la communication entre les modules PCP et BC quand le premier a besoin des connaissances du second concernant le domaine du problème. En effet, le PCP doit former des problèmes de dérivation d'hypothèses et les envoyer à la BC. Un problème de dérivation d'hypothèses est composé d'un but B, un ensemble d'hypothèses H, et un ensemble de variables V. La réponse renvoyée par la BC est une formule logique P satisfaisant les contraintes définies plus haut, ou bien un message d'échec.

La dérivation d'hypothèses est pour nous, comme pour Smith, un mécanisme intermédiaire entre les techniques de construction de programmes et les connaissances sur le domaine du problème (voir §1.4.3). Néanmoins, ce fait n'est pas matérialisé dans l'architecture de Smith (§ 2.2.-Architecture). Il réunit dans un même module le *Contrôle de la Synthèse* et le *moteur de préconditions*, or c'est ce dernier qui résoud les problèmes de dérivation d'hypothèses. Chez nous, par contre, le fait d'utiliser la dérivation d'hypothèses comme interface entre les modules PCP et BC permet de caractériser cette dernière comme un résolveur de problèmes de dérivation d'hypothèses[1]. Il était important de délimiter de façon précise la partie correspondante à chaque module, dans une équipe où deux thésards travaillaient chacun sur un des modules...

[1] Toutefois, la dérivation de préconditions n'est pas la seule demande que peut faire le PCP à la BC; celle-ci doit aussi fournir des renseignements nécessaires aux choix des lignes générales de la synthèse (structure des domaines, entre autres).

## 2.3.- Un Exemple

Afin d'illustrer notre méthode, ainsi que l'interaction entre les modules de notre architecture, nous présentons ci-dessous un exemple de synthèse. Nous allons reprendre l'exemple de la section § 1.4, c'est-à-dire, la synthèse d'un programme qui élimine d'une liste ses éléments non-atomiques. Nous facilitons ainsi du même coup la comparaison de notre méthode avec les méthodes décrites précédemment.

Nous allons identifier la partie de la synthèse exécutée par les différents modules du système par les signes "**PCP :**", "**BC :**", "**UTIL :**" (utilisateur) et "**TD :**" (module de traîtement de demandes).

TD : Système XXX prèt...Que puis-je faire pour vous ?...
- Synthèse d'un programme à partir d'une spécification logique
- Aide à la formation d'un énoncé de programme
- ... ...

UTIL : Synthèse d'un programme...En voici l'énoncé :
**Profil :** $y = f(x : liste[t]) \rightarrow liste[t]$
**préc :** vrai
**post :** $u \in y \Leftrightarrow (u \in x \wedge atom(u))$

PCP : Il fait choisir une stratégie de synthèse. Je propose les suivantes:
- Décomposition de la donnée (schéma récursif)
- Construction du résultat (schéma récursif)
- ... ...

UTIL : Ce sera *Décomposition de la donnée.*

PCP : Très bien. Votre schéma de programme est de la forme:
$f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$ **sinon** $g(f(h_{1,1}(x),\ldots,h_{1,k}(x)),\ldots,f(h_{1;k}(x),\ldots,h_{n,k}(x)), x)$.
Il faudra choisir une décomposition des listes. Question à la BC: Quelles sont les décompositions disponibles pour les listes ?

BC : Je dispose des décomposition suivantes:
$x = car(x) \bullet cdr(x)$
$x = concat(l_1(x), l_2(x))$
... ...

UTIL : Je choisis la décomposition par car et cdr...

PCP : O.K. Je propose le schéma de programme suivant:
$f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$ **sinon** $g(f(cdr(x)), x)$.
qui contient un seul appel récursif, sur cdr(x).

PCP : Maintenant je vais chercher une instance pour la condition C (cas d'arrêt). Question à la BC: Quel est la plus petite liste[1] qui satisfait la précondition (vrai) ?

BC : C'est nil...

PCP : Si vous êtes d'accord, nous pouvons maintenant résoudre la branche du cas de base de la récursion. Une de prémisses de la règle de la récursion est de la forme:[2]
$\text{Précondition}(x) \wedge \text{Cas-d'arrêt}(x) \wedge Q_{f1}(x,y) \Rightarrow Q(x,y)$
où $Q_{f1}$ est le prédicat qui caractérise $f_1$. Nous allons donc obtenir un $Q_{f1}$ en calculant une hypothèse additionnelle, en termes de x et y, du but:
$\text{Précondition}(x) \wedge \text{Cas-d'arrêt}(x) \Rightarrow Q(x,y)$
C'est-à-dire, nous posons la suivante Question à la BC: Il faut résoudre un problème de dérivation d'hypothèses avec: B: $\forall u[u \in y \Leftrightarrow u \in x \wedge atom(u)]$, H: {x=nil}, V: {x, y}.
(Ceci revient à simplifier la spécification à l'aide de l'hypothèse x=nil)

BC : Voyons...
Mon résultat est : y=nil.
Explication intuitive:[3] Dans ma définition récursive de '$\in$' en termes de nil et cons je trouve :
$(t_1)$ $u \in nil \Leftrightarrow faux$
En instanciant le but B pour x=nil, on peut former une chaîne d'équivalences:
$u \in y \Leftrightarrow u \in x \wedge atom(u)$
$\Leftrightarrow faux \wedge atom(u)$
$\Leftrightarrow faux,$
donc $u \in y = faux$ . Ce résultat s'unifie avec $(t_1)$, donc une solution pour y est y=nil

PCP : L'énoncé de $f_1$ est donc:
profil : $y = f_1(x : liste[t]) \rightarrow liste[t]$
préc : x=nil
post : y = nil,
qui correspond évidemment à la fonction constante nil.
Le schéma devient donc:
$f(x) \leftarrow$ **si** x=nil **alors** nil **sinon** $g(f(cdr(x)), x)$.

---

[1] L'ordre est défini d'après les constructeurs: $l < a \bullet l$

[2] Les règles d'inférence sont présentées dans [Potet 88].

[3] Les techniques de manipulation de formules mises en œuvre dans la BC ne sont pas celles illustrées dans cet exemple (se référer aux chapitres 4 et 5). C'est la raison pour laquelle nous avons décidé d'utiliser le titre "Explication intuitive", qui confère aux développements qui suivent un caractère d'interface utilisateur.

PCP : Nous pouvons maintenant entreprendre la synthèse de g.
Soit z=f(cdr(x)). L'énoncé de g est de la forme:

profil : y = g(z : liste[t], x : liste[t]) → liste[t]
préc : $x \neq nil \wedge [u \in z \Leftrightarrow u \in cdr(x) \wedge atom(u)]$
post : $Q_g(y, z, x)$

Une des prémisses de la règle de la récursion est de la forme:
$Précondition(x) \wedge \neg Cas\text{-}d'arrêt(x) \wedge Hypoth.d'induction(z,x) \wedge Q_g(y,z,x) \Rightarrow Q(x,y)$
Donc:
$x \neq nil \wedge [u \in z \Leftrightarrow u \in cdr(x) \wedge atom(u)] \wedge Q_g(y, z, x) \Rightarrow$
$[u \in y \Leftrightarrow u \in x \wedge atom(u)]$
Pour résoudre $Q_g$ je pose une Question à la BC : Dérivation d'hypothèses avec :
B: $\forall u[u \in y \Leftrightarrow u \in x \wedge atom(u)]$,
H: $\{x \neq nil, \forall u[u \in z \Leftrightarrow u \in cdr(x) \wedge atom(u)]\}$,
V: {y, z, x}.

BC : Je suis arrivé au résultat :

P : $\forall u[(u=car(x) \wedge atom(u)) \vee u \in z]$

Explication intuitive: Par l'hypothèse $x \neq nil$ on a $x=car(x) \bullet cdr(x)$.Nous avons utilisé la définition récursive de '$\in$':
$(t_2)$ $w \in a \bullet l \Leftrightarrow (w=a) \vee w \in l$
Nous pouvons donc remplacer $car(x) \bullet cdr(x)$ dans $(t_2)$ instancié:
$u \in y \Leftrightarrow u \in car(x) \bullet cdr(x) \wedge atom(u) \Leftrightarrow [(u=car(x)) \vee u \in cdr(x)] \wedge atom(u)$
$\Leftrightarrow [u=car(x) \wedge atom(u)] \vee [u \in cdr(x) \wedge atom(u)]$
$\Leftrightarrow [u=car(x) \wedge atom(u)] \vee u \in z$

PCP : L'énoncé de g devient:

profil : y = g(z : liste[t], x : liste[t]) → liste[t]
préc : $x \neq nil \wedge [u \in z \Leftrightarrow u \in cdr(x) \wedge atom(u)]$
post : $u \in y \Leftrightarrow [(u=car(x) \wedge atom(u)) \vee u \in z]$

Question à l'utilisateur: Avez-vous une proposition pour la suite ? (Je vous montrerais la liste de stratégies disponibles sur simple demande).

UTIL : Hum...La disjonction me suggère l'introduction d'une conditionnelle.

PCP : Voulez-vous fixer la condition ou bien l'une des branches de la conditionnelle ?

UTIL : Je veux avoir y=z pour une des branches .

PCP : Je ne promets rien...Une des prémisses de la règle d'introduction de la conditionnelle est de la forme:
Précondition(x) $\wedge$ Test(x) $\wedge$ $Q_1(x,y) \Rightarrow Q(x,y)$
où Test(x) est le test de la conditionnelle, $Q_1$ est le prédicat caractérisant une des branches, et Q est la postcondition de la fonction à synthétiser. Ceci nous permet de poser une Question à la BC: Pour trouver le test de la conditionnelle, le problème suivant de dérivation d'hypothèses doit être résolu: B: $u \in y \Leftrightarrow [u=car(x) \wedge atom(u)] \vee u \in z$, H: $\{y=z, x \neq nil, u \in z \Leftrightarrow u \in cdr(x) \wedge atom(u)\}$, V: $\{x\}$

BC : L'hypothèse additionnelle trouvée est $\neg atom(car(x))$
Explication intuitive: Par l'hypothèse y=z et la substitutivité, on peut remplacer $u \in y$ par $u \in z$ dans le but:

$$u \in z \Leftrightarrow [u=car(x) \wedge atom(u)] \vee u \in z$$

Ceci force $\neg[u=car(x) \wedge atom(u)]$, puis $\neg atom(car(x))$

PCP : Il ne reste qu'à résoudre l'autre branche de la conditionnelle, c'est-à-dire, le cas atom(car(x)). Je pose donc une nouvelle Question à la BC:Soit B: $u \in y \Leftrightarrow [u=car(x) \wedge atom(u)] \vee u \in z$,
H: $\{x \neq nil, u \in z \Leftrightarrow u \in cdr(x) \wedge atom(u), atom(car(x))\}$,
V: $\{x,y,z\}$

BC : J'ai trouvé P: $\forall u[u \in y \Leftrightarrow u \in car(x) \bullet z]$
Explication intuitive: De atom(car(x)) et u=car(x) il suit atom(u), qui se simplifie à vrai, laissant donc:
$u \in y \Leftrightarrow u=car(x) \vee u \in z$, qui est équivalent à $u \in car(x) \bullet z$, par la définition de '$\in$'.

PCP : Bravo! Cela fournit la solution $y=car(x) \bullet z$, où z=f(cdr(x)). Le programme obtenu est donc:
f(x) $\leftarrow$ **si** x=nil **alors** nil **sinon** g(f(cdr(x)), x).
g(z,x) $\leftarrow$ **si** atom(car(x)) **alors** $car(x) \bullet z$ **sinon** z.

TD : Voulez-vous...
- Modifier le programme obtenu,
- Tester le programme obtenu,

... ... ...

*Today's algorithm-synthesis systems have typically synthesized one or a handful of examples for which each system has a small but appropriate knowledge base. The problem remains to build a large enough knowledge base with the necessary accompanying methods so that one system could synthesize more interesting algorithms [Green 85]*

# 3.- La Base de Connaissances

Pour déceler les caractéristiques d'une BC adéquate pour notre système, il nous a semblé utile de passer rapidement en revue quelques bases de connaissances proposées pour des systèmes de synthèse ou construction assistée de programmes.

## 3.1.- Quelques bases de connaissances pour la synthèse de programmes

Il est difficile de comparer les parties des systèmes de synthèse de programmes qui pourraient être identifiées à une "base de connaissances". En effet, ce qui dans une approche est appelé (ou pas appelé) connaissance peut ne pas avoir d'équivalent dans les autres. Cette difficulté provient du fait que des formes différentes de connaissance sont parfois mélangées.

### 3.1.1.- La base du système DEDALUS

Le système DEDALUS [Manna, Waldinger 79] consitue la première réalisation importante d'un système de synthèse automatique de programmes. Il est basé sur l'approche transformationnelle. Nous considérons ici ce système plutôt que les travaux en synthèse transformationnelle de Darlington, Burstall et Clark, analysés dans le chapitre 1. La raison en est que DEDALUS est une réalisation pratique, ce qui a obligé les auteurs à s'attaquer de façon détaillée aux problèmes de représentation et de recherche des connaissances dans le système, avec toutes les difficultés que cela entraîne.

Les connaissances du système DEDALUS sont appelées règles de transformation. Les types de règles sont:

1.- Règles exprimant l'univers sémantique des domaines connus par le système (nombres, listes, ensembles).
Exemple: $u\backslash v \wedge u\backslash w \Rightarrow u\backslash v \wedge u\backslash w\text{-}v$ où x\y se lit 'x divise y'

2.- Règles relatives au sens des constructions du langage de spécification et du langage cible.
Exemple: [Manna, Waldinger 79] $P(all(l)) \Rightarrow P(tête(l)) \wedge P(all(queue(l)))$
Ceci exprime une propriété de la construction *all* du langage de spécification, pour une propriété P des éléments d'une liste.

3.- Règles relatives à la programmation en général.

Par exemple: *Une analyse par cas est introduite quand les préconditions d'une règle ne peuvent pas être prouvées.*

Le système DEDALUS comporte environ une centaine de telles règles. Il y en a beaucoup du type 1, une certaine quantité du type 2, et peu de règles du type 3.

La structure retenue pour la base de règles de DEDALUS est arborescente; elle est issue directement de la méthode d'appel des règles. En effet, les règles sont sélectionnées par filtrage de leur partie gauche (pattern directed), puis les conditions associées à la règle sont vérifiées. Par conséquent, la base de règles est organisée comme un arbre de classification d'après les 'patterns' des parties gauches. Une règle $a \rightarrow b$ ***si*** $P$ est fille d'une règle $c \rightarrow d$ ***si*** $Q$ ssi a est filtrée par c. Pour des règles avec un même pattern à gauche, un ordre de préférence est utilisé.

Pour éviter d'appliquer une règle dans une situation où elle ne convient pas, Manna et Waldinger ont introduit les *contrôles stratégiques*. Ceux-ci sont des conditions nécessaires associées aux règles, portant sur la situation présente à un moment donné au cours de la synthèse. On doit, par exemple, éviter qu'un remplacement de $a$ par $b$ soit suivi d'un remplacement de $b$ par $a$.

Une règle $a \rightarrow b$ ***si*** $P$ sera appliquée seulement si:

- $a$ filtre un terme de l'énoncé courant, avec la substitution $\theta$;
- $P\theta$ évalue à vrai;
- Le contrôle stratégique de cette règle est vérifié.

*Discussion*

La cohabitation de différents types de connaissances, applicables à des moments différents de la synthèse, a été considéré comme une cause d'inefficacité [Gresse 84]. Ce problème peut s'aggraver à mesure que la taille de la base augmente.

Quant à la structure arborescente, notons qu'elle est strictement syntaxique et qu'elle n'ajoute aucune information nouvelle par rapport aux règles elles-mêmes. C'est donc une espèce de *base compilée,* facilitant un accés efficace par le mécanisme de recherche de règles par filtrage.

Manna et Waldinger eux-mêmes se posent des questions quant aux performances d'une structure limitée aux patterns et aux contrôles stratégiques, localisés au niveau de chaque règle, au lieu d'une stratégie globale: "...will this mechanism still be adequate when we increase the number of rules from one hundred to one thousand...?".

### 3.1.2.- La base du système CYPRESS

Le système CYPRESS [Smith 85] comporte une base (Data Structure Knowledge Base, DSKB) contenant des connaissances sur les types de données connus. La DSKB contient des descriptions des opérateurs du langage de spécification (profil, précondition et postcondition, et éventuellement définition calculatoire), ainsi que des théorèmes faisant intervenir ces opérateurs.

Nous allons concentrer notre attention sur ces théorèmes, lesquels sont utilisés pour la dérivation de préconditions, comme dans notre approche. Les ressemblances du système de Smith et de notre approche se font évidentes une fois de plus; c'est la raison pour laquelle l'analyse du système CYPRESS présente un intérêt tout particulier pour nous.[1]

Dans le système de Smith les théorèmes (environ une centaine) sont des formules de la logique du premier ordre. Les théorèmes disponibles dans la DSKB forment une théorie partielle, dans le sens où ils sont un sous-ensemble fini des théories concernées. Ils sont effectivement une forme de connaissance dans le sens où la DSKB contient les théorèmes que la pratique ou l'intuition de son créateur a jugés utiles.

Dans CYPRESS les théorèmes sont exprimés en notation prefixée (à la lisp). Nous trouvons les formes suivantes :[2]

- Axiomes (théorèmes sans connectives logiques ni égalités)
  Par exemple:
  ```
  (ordered nil)
  (le $i $i)
  (gt (plus 1 $i) 0)
  ... ...
  ```
  où les variables sont de la forme $*i* pour une lettre *i*.

- Implications, Equivalences, Négations, Conjonctions...
  Par exemple:
  ```
  (impl (le (length $x) 1) (ordered $x) 0.2)
  ```
  où le '0.2' est un coefficient servant à des heuristiques

- Simplifications booléennes
  Exemple: ( (conj $P true) $P)

- Egalités
  Exemple: (equal 0 (length nil) )

Il existe dans la DSKB une liste de théorèmes pour chacun de ces types de théorèmes. Cette classification tient à la façon d'utiliser les théorèmes dans le sous-système RAINBOW, chargé d'effectuer la dérivation de préconditions. Nous ne ferons pas ici une description détaillée de RAINBOW, elle sera faite dans un autre chapitre; nous nous bornerons à signaler ses caractéristiques concernant la présente discussion. C'est un système de "déduction naturelle" dont les règles d'inférence sont de trois types:

i) Règles de décomposition;
ii) Règles de remplacement;
iii) Règles pour les formules atomiques.

---

[1] Nous rappelons que, à la différence de CYPRESS, notre BC contient des information statiques *et aussi* des méthodes pour les exploiter, en particulier le sous-système pour la dérivation de préconditions.

[2] Cette information ne se trouve dans aucune publication. Nous remercions D.R.Smith de nous avoir fait parvenir le texte lisp de son système CYPRESS.

Nous allons considérer les règles ii) et iii), car ce sont elles qui permettent l'introduction des théorèmes de la DSKB. Ces formules sont utilisées de la façon suivante:

- Si un sous-but "colle" avec un *axiome* ou une hypothèse, alors il est évaluée à vrai.
- Sinon, il est toujours possible d'essayer une *simplification* pour arriver à une formule élémentaire.
- Sinon, on peut essayer d'appliquer une *implication*, *équivalence* ou *égalités* comme une règle de remplacement. En effet, ces trois types de théorèmes sont de la forme g ⊗ d, où '⊗' est ⇒, ⇔ ou =. Or, un terme t tel que gθ=t pour une substitution θ, peut être remplacé par dθ.[1]

La structure de la DSKB, dans la version dont nous disposons, est limitée aux listes de théorèmes. La recherche d'un théorème adéquat se fait par simple parcours linéaire des listes, pour trouver d'abord ceux qui sont compatibles syntaxiquement. Ceci est suivi d'une évaluation d'indices heuristiques, pour départager les cas où plusieurs théorèmes seraient applicables. Ce processus consomme un temps considérable.

Nous croyons que la faible structure de la DSKB est tout à fait au service de l'application des règles d'inférence de RAINBOW, au détriment de sa valeur en tant que modélisation intuitive du monde.

La toute nouvelle version de CYPRESS[2] comporte une structure beaucoup plus évoluée, où les information sont retrouvées par un mécanisme d'indexation multiple, que nous n'oserions pas décrire ici, en absence d'une documentation écrite.

### 3.1.3.- La Bibliothèque de types de CATY

Le système CATY [Gresse 84] de construction assistée de programmes comporte une base de connaissances adaptée à sa démarche pour aborder le processus de construction. Cette démarche est fondée sur l'idée d'utiliser un découpage des données pour engendrer des sous-problèmes plus simples de façon descendante. Cette idée avait déjà été suggérée par [Von Henke 75], qui constate que souvent la structure des programmes reflète fortement la structure des données que celui-ci manipule.

L'assistance fournie par CATY au programmeur consiste à:

1) Aider l'utilisateur à exprimer son problème en termes des types de données connus par CATY (l'utilisateur a la possibilité de définir des nouveaux types de données).

2) Gérer le processus de construction descendante du programme, dont les étapes sont:

2.1) Choix d'une stratégie de découpage du problème (transversalement au programme, transversalement aux données...).

2.2) Instanciation des schémas de programme issus de la phase 2.1 pour aboutir à

---

[1] Dans le cas des implications on remplace plutôt le côté droit par le côté gauche.

[2] présentée à la conférence AAAI'88

un programme.

La sortie du pas 1 (l'entrée du pas 2) est une forme d'énoncé ne portant que sur le typage des entrées et des sorties du programme à construire, par exemple:

```
(L:entier) <- LONGMAX(SUI:suite[entier])
```

La description logique complète du rapport entre entrée et sortie n'est donc pas considérée dans [Gresse 84]. La caractérisation des sous-problèmes dans CATY porte aussi uniquement sur les types d'entrée et de sortie.[1]

Puisque cette approche s'appuie fortement sur la structure des données, il est très important de caractériser celles-ci de façon adéquate. Gresse a choisi pour cela la spécification *abstraite* des types de données fournie par l'approche algébrique de [Guttag, Horning 78]. Une caractérisation abstraite des types de données est intéressante car elle permet de *repousser au maximum les problèmes d'implémentation liés aux contraintes technologiques* [Gresse 84].

Nous laissons pour plus tard la définition formelle des spécifications algébriques; pour le moment nous reprennons un exemple sur le type des séquences d'entiers présenté dans [Gresse 84]:

```
TYPE séquence[entier]

Opérations:
      vide       :    -> séquence[entier]
      videp _    :    séquence[entier] -> booléen
      deb _      :    séquence[entier] -> entier
      dem _      :    séquence[entier] -> séquence[entier]
      cons _ _   :    entier séquence[entier] -> séquence[entier]

Axiomes:
            videp(vide) = vrai
            videp(cons(n,s)) = faux
            deb(cons(n,s)) = n
            dem(cons(n,s)) = s

FTYPE séquence[entier]
```

où les types entier et booléen sont supposés être prédéfinis. Une telle présentation fait intervenir des opérations spécifiques appelées *constructeurs*; en l'occurrence ce sont les opérateurs *vide* et *cons*. La composition des constructeurs permet d'engendrer tous les objets du type; par exemple, la liste: cons(a,cons(b,cons(c,nil))) tient pour (a,b,c).

La base de connaissances de CATY comporte un ensemble d'unités de définition de types appelées *présentations*. Chaque présentation comporte, comme dans l'exemple, une partie de déclaration des profils syntaxiques des opérations (types d'entrée et de sortie), plus un

---

[1] Une extension de CATY pour prendre en compte des spécifications logiques est proposée dans [Brena et al.85].

ensemble d'*axiomes* définissant la sémantique de ces opérateurs.

La base de CATY contient également un ensemble de *schémas de décomposition* de données, qui est une forme de connaissance intermédiaire entre les stratégies de programmation et la caractérisation des domaines d'application (les types de données, en l'occurrence).

*Discussion*

La structuration des informations en unités modulaires constituées autour des types de données nous semble une forme très naturelle d'organisation. En effet, cette organisation repose sur les *propriétés intrinsèques des domaines*, caractérisés de façon abstraite, et non pas sur des critères relatifs aux procédés spécifiques d'utilisation ces informations, ou -pire encore- aux contraintes technologiques, comme c'est souvent le cas.

Toutefois, rappelons que les axiomes ne sont pas utilisés dans CATY pour les mettre en rapport avec les spécifications des programmes. Cette limitation élimine le problème relatif à la disponibilité des théorèmes nécessaires à la synthèse de programmes, tel que nous l'avons discuté précédemment.

### 3.1.4. Discussion générale

Les bases de connaissances de DEDALUS et CYPRESS sont adéquates par rapport à la forme d'exploitation que l'on en fait (et plus précisément, par rapport aux problèmes que le système synthétise correctement). Cette forme d'adéquation est appelée *Adéquation heuristique* par [McCarthy, Hayes 69]. Cet aspect pourrait être identifié à une *sémantique opérationnelle* de la base de connaissances, dans le sens où ce n'est que son fonctionnement qui compte, et non pas la signification du contenu de la base, qui serait sa *sémantique dénotationnelle* [1] L'adéquation "en principe" de la base en tant que représentation du monde est appelée *Adéquation métaphysique* par [McCarthy, Hayes 69]. Parmi les systèmes dont nous avons fait l'analyse, le seul qui répond à ce critère est CATY.

Cette constatation est flagrante en ce qui concerne la structure des bases de connaissances. Dans le cas de DEDALUS elle est déterminée par une forme de recherche des formules à partir d'un pattern, et dans le cas de CYPRESS elle est déterminée par les règles d'inférence du système de dérivation de préconditions.

Nous croyons que pour passer à des bases de connaissances d'un ordre de grandeur supérieur (de milliers de formules) sans pour autant ralentir excessivement le système, il faudra mettre en place des structures beaucoup plus riches, et des mécanismes de recherche plus sophistiqués permettant d'accéder directement aux informations.

Il est toutefois clair qu'une structure complexe n'aura une utilité quelconque que dans le cas où l'on sait exactement quelles informations nous voulons. Il faut donc être à même de préciser le plus possible ces informations.

---

[1] Ceci n'est qu'une analogie au sens de ces termes dans les langages de programmation. En logique, dénotation est *l'objet réel désigné par un signe* [Larousse 84].

## 3.2.- Lignes générales pour la conception de la BC

Afin d'entreprendre le dessin méthodique de la BC, nous allons commencer par mettre en rapport deux types de contraintes auxquelles la BC doit répondre:

$c_1$) Les fonctionalités de la BC, c'est-à-dire les questions auxquelles elle doit répondre;
$c_2$) Les critères généraux de conception.

### 3.2.1.- Les fonctionalités de la BC

En ce qui concerne le critère c1, nous avons donné, dans le chapitre précédent, un aperçu du rôle attribué à la Base de Connaissances dans notre approche. Nous rappelons ici ses principales fonctions:

- Réponse aux questions relatives à la structure des domaines connus par le système (schémas de décomposition, cas de base des récursions, etc.).
- Solution aux problèmes de dérivation d'hypothèses
- Trouver le cas de base des récursions[1];
- Récupération de définitions; consultation de la base par l'utilisateur;

Il est sous-entendu que la BC doit intervenir pour la validation des énoncés donnés au système (vérifier si les symboles de fonctions et prédicats sont connus et correctement utilisés). La BC est donc le dictionnaire des symboles et types de données connus. L'utilisateur doit également avoir la possibilité de consulter directement l'ensemble de symboles disponibles et leur définition, afin de l'aider à former ses énoncés de fonction.

Les difficultés entraînées par les différents problèmes posés à la BC ne sont pas de même nature. D'un côté, la récupération des schémas de décomposition disponibles peut se résoudre par consultation directe des informations stockées dans une base, comme dans [Gresse 84]. De l'autre côté, la résolution de problèmes de dérivation d'hypothèses par des méthodes de bases de données n'est pas envisageable, tellement le nombre d'entrées possibles est grand.

Nous voudrions pourtant résoudre tous ces problèmes dans un cadre uniforme, si ceci est possible, au lieu de proposer des solutions ad hoc et indépendantes pour chacun. Une même base pourrait ainsi être utilisée pour tous les problèmes propres à la BC, et ce serait la façon de s'en servir qui changerait pour chacun, de manière analogue aux *vues* des Bases de Données.

### 3.2.2.- Les critères généraux

A partir de la discussion générale du paragraphe 3.1, nous tirons les orientations suivantes:

*Adéquation métaphysique de la BC*

Il est très important de mettre en valeur l'aspect *représentation du monde* de la BC (son aspect dénotationnel) , parce que:

---

[1] Le travail que nous avons réalisé sur le sujet n'a pas été retenu pour la version définitive de cette thèse.

- Notre approche de synthèse *assistée* de programmes donne à l'utilisateur un rôle central, et il est donc indispensable que celui-ci *comprenne* le sens du contenu et de la structure de la base. Les approches "opérationnelles" pour définir la structure de la base (DEDALUS, CYPRESS) n'apportent guère cette facilité de compréhension.

- Une représentation naturelle de l'univers sémantique sous-jacent au langage des énoncés offre beaucoup de possibilités pour structurer la base de façon adéquate. Une structure *à la CATY*, adaptée pour tenir compte de la problématique des théorèmes, pourrait nous convenir. Notre pari consistera alors à prouver qu'une telle structure est adéquate par rapport à l'accès efficace aux informations.

*Une approche basée sur la connaissance*

D'après le premier point, la connaissance sera stockée dans la BC sous une forme déclarative décrivant explicitement l'univers sémantique connu par le système. Plus concrètement, la BC contiendra des ensembles de formules valides dans les modèles d'intérêt.

Dans l'approche syntaxique de la logique des prédicats, ces formules sont les théorèmes, qui sont obtenus à partir des axiomes en utilisant des règles d'inférence (voir §1.4.2). Or, dans le §1.4.2 nous avions évoqué la difficulté d'obtenir en pratique les théorèmes nécessaires à la synthèse de programmes, n'utilisant que les axiomes et les règles d'inférence.

Nous allons donc considérer des ensembles finis de formules valides (théorèmes, en supposant la complétude du système logique), stockés explicitement dans la BC. Si $\tau^M$ est l'ensemble de formules valides dans le modèle M, $\tau_A^M$ sont des axiomes définissant M, et $\tau_D^M$ sont les formules effectivement stockées dans la BC, nous aurons:

$$\tau_A^M \subset \tau_D^M \subset \tau^M$$

Nous allons appeler $\tau_D^M$ *la partie distinguée de la théorie de M*. Elle est *distinguée* dans le sens où elle contient les théorèmes disponibles de façon immédiate, sans avoir à effectuer d'inférences. Bien entendu, $\tau_D^M$ est redondante; nous ne cherchons donc pas à stocker dans la BC le strict minimum d'information. La motivation de cette attitude est exprimée dans la phrase suivante:

"L'efficacite en IA
revient a minimiser
le role de l'inference
pour maximiser
celui de la Connaissance"

La partie distinguée de la théorie, $\tau_D^M$, est une connaissance d'après les critères exposés dans le §1.4; elle l'est aussi d'après la définition de Lenat de connaissance comme "compiled search", car les théorèmes dans $\tau_D^M$ ne seront plus dérivés à partir des axiomes, une fois qu'ils sont dans la BC.

La recherche de formules dans $\tau_D^M$ peut se faire *modulo* un ensemble R de transformations qui

préservent l'équivalence, telles que la commutativité de certains opérateurs. Ceci peut être vu comme une extension de $\tau_D^M$, notée $(\tau_D^M)^R$, formée par la fermeture de $\tau_D^M$ par R. Il est essentiel que l'ensemble $(\tau_D^M)^R$ reste fini, afin d'autoriser l'emploi d'une recherche exhaustive des informations.

*Recherche des Connaissances à partir de leur caractérisation*

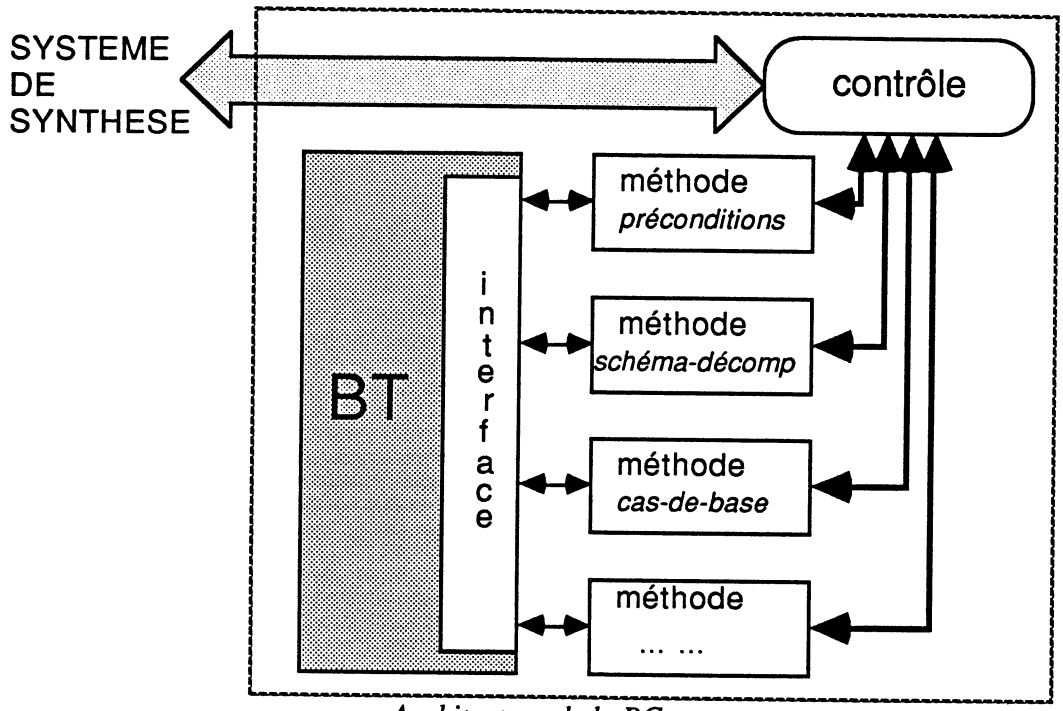
La définition précise des connaissances requises, condition *sine qua non* pour utiliser effectivement une structure complexe, est une forme de *spécification de connaissances*. Cette idée, suggérée à la fin du chapitre 1, provient du fait que la caractérisation des connaissances requises est d'un niveau plus général que les connaissance elles-mêmes. En effet, comment serait-il possible autrement de demander une connaissance que par définition on ne connait pas ? Le formalisme pour élaborer les demandes des connaissances constitue la définition de l'interface d'accès au module de stockage des connaissances. La question de la caractérisation des connaissances nécessaires à la synthèse sera discutée au §5.

*Modularité de la BC*

Imaginons ce qui se passerait si dans CYPRESS on remplace le sous-système RAINBOW pour un autre "dérivateur de préconditions": la DSKB deviendrait inutilisable, car sa structure dépend des règles d'inférence de RAINBOW! Nous voudrions avoir une certaine *indépendance* des connaissances stockées (structure y comprise) par rapport aux méthodes spécifiques d'utilisation de cette information. Nous devrons donc *modulariser* notre BC, en séparant les connaissances proprement dites des méthodes de solution des problèmes posés à la BC. Pour chaque module nous devrons définir l'interface de façon précise. Certains des modules pourront avoir une utilité générale, non restreinte à notre système de synthèse de programmes. Ainsi, par exemple, l'intérêt d'une bibliothèque contenant les propriétés formelles qui caractérisent les domaines d'application les plus courants en programmation a été mis en valeur par divers auteurs [Goguen 85].

### 3.2.3.- Architecture de la BC

Des proposition faites dans le paragraphe précédent découle une organisation générale de notre BC. Celle-ci comporte, d'une part, des connaissances déclaratives, statiques, qui sont groupées dans une "Bibliothèque de Théories" (BT), et d'autre part, des connaissances procédurales ou "méthodes" dédiées à la solution des problèmes spécifiques posés par le système de synthèse ou bien par l'utilisateur lui-même; Ces méthodes peuvent accéder à l'information contenue dans la BT par l'intermédiaire d'une interface. Cette architecture est montrée dans la figure suivante:




*Architecture de la BC*

Le système de synthèse envoie à la BC des problèmes à résoudre (p.ex. dériver une hypothèse). Le module de *contrôle* active alors la *méthode* (procédure spécialisée) appropriée pour le résoudre. A leur tour, les méthodes font appel à l'information sur les domaines d'application contenue dans la *Bibliothèque de Théories* (BT), à travers une *interface* qui fournit un langage de description des informations. Finalement, cette interface récupère les informations voulues en faisant usage des possibilités offertes par l'organisation de données choisie (relationnelle, orientée-objet,...).

Les méthodes représentent la partie *connaissance procédurale* de la BC. Ce sont des modules logiciels spécialisés, voués à la solution des problèmes spécifiques posés à la BC. Elles reçoivent les énoncés des problèmes à résoudre, et renvoient leur solution ou un message d'échec.

Nous ne rentrerons dans les détails des méthodes que pour le cas de la dérivation d'hypothèses (cf. § 5). En effet, pour tous les autre cas il s'agit surtout de problèmes de récupération des informations, donc ces méthodes remplissent des fonctions d'interfaçage, qu'il serait préférable de considérer dans une phase de réalisation de notre projet.

Les connaissances déclaratives sur les domaines d'application se trouvent dans la BT. Ce module sera décrit dans la section suivante.

## 3.3.- Conception de la BT

La Bibliothèque de Théories (BT) est le module qui contient les connaissances proprement dites. Nous étendons le sens de connaissance donné au chapitre 1 (ensemble de formules valides dans les modèles sous-jacents au langage des énoncés), pour englober aussi la structure qui sert à organiser ces formules, ainsi que d'autres informations sur les opérateurs de ces formules.

Pour entreprendre la conception de la BT d'une façon concrète, nous énumérons ci dessous les aspects qu'il reste à préciser:

• Informations élémentaires contenues dans la BT.- Nous allons considérer les aspects suivants de la caractérisation des données dans la BT:

• Définition des (types de) données
• Description des opérateurs faisant partie du langage des énoncés
• Théorèmes distingués appartenant aux théories sous-jacentes au langage des énoncés.

*Structure.*

Nous nous proposons de définir une structure riche tant en termes de récupération des informations élémentaires qu'en termes de sa sémantique, par rapport aux domaines modélisés. En ce qui concerne la récupération des informations, la BT doit permettre leur recherche par des critères différents, c'est-à-dire qu'une même information doit être repérée par différents indices. Ainsi, par exemple, le théorème: x + succ(y) = succ(x + y) pourrait être trouvé comme une partie de la définition de l'opérateur '+', ou bien comme un théorème de lien entre les opérateurs '+' et 'succ', ou bien comme une équation récursive dans les entiers. *La notion de typage peut donner un fil conducteur pour organiser la BT de façon naturelle.* Une organisation *à la CATY* (enrichie) peut fournir les moyens de modulariser et de hiérarchiser les informations..

*Interface.*

On peut accéder à la BT de l'extérieur par une interface fournissant un langage de description des informations. Celui-ci définit la *vue* que les méthodes (et le reste du système) ont de l'information contenue dans la BT, c'est-à-dire, la description externe de cette dernière. La représentation interne des objets et leur structuration au niveau physique peuvent être changées pour des raisons d'efficacité sans pour autant altérer cette vue externe, pourvu que les propriétés du langage de description soient maintenues. Nous voulons caractériser les informations demandées de façon abstraite, sans faire référence aux détails de l'organisation retenue pour la BT. Le langage d'interrogation de la BT est donc du type "logique", dans la terminologie des bases de données, par opposition aux langages dits "algébriques" [Delobel, Adiba 82], où l'on spécifie la façon d'obtenir les informations voulues par composition d'opérateurs d'extraction et combinaison de données. Ce langage permet de construire des formules logiques où les prédicats fixent des critères de sélection de l'information. Ainsi, les informations récupérées sont celles qui satisfont à la formule logique de la requête.

*Méthodes de recherche.*

Elles s'attachent à traduire les caractérisations abstraites de théorèmes en termes procéduraux, c'est-à-dire, comme des suites d'opérations d'accès à la base, en utilisant la structure de celle-ci, pour arriver à une information particulière.

## 3.3.1.- Caractérisation des données dans la BT

Nous voulons, comme Gresse, caractériser les données manipulées par les programmes de façon abstraite, pour repousser ainsi au maximum les problèmes de représentation liés à une implémentation particulière.

Dans la BT nous classifions les données en *types*. Le typage des données est important pour sélectionner dans la BT le modèle sous-jacent à un énoncé donné. En fait, par *sélectionner le modèle sous-jacent* nous voulons dire *déterminer exactement quels sont les types concernés*. Une fois cela fait, il sera possible d'utiliser toutes les connaissances sur ces types, sans consulter inutilement les connaissances sur les autres types. En effet, pour la BT il est très intéressant de pouvoir classer les informations de façon à savoir, étant donné un énoncé, lesquelles sont pertinentes.

Le typage permet également de guider le processus de formation de sous-problèmes, comme le montre le système CATY. Finalement, il est utile en tant que discipline de programmation. En effet, le typage fort est une façon de réduire les erreurs lors de la formation de spécifications, par le biais syntaxique.

### 3.3.1.1.- Premières définitions

Soit S un ensemble d'identificateurs de types appelés sortes.

Soit $\Sigma$ un ensemble de *symboles d'opération*, ou tout simplement *opérateurs*. Plus précisément, $\Sigma$ est une famille d'ensembles d'opérateurs, indexés par des chaines dans $S^+$. Pour un opérateur $\sigma \in \Sigma_{s_1 s_2 \ldots s_n}$, on dit que $s_1 s_2 \ldots s_n$ est le *profil* syntaxique. Celui-ci est souvent écrit comme $s_1, s_2, \ldots s_{n-1} \rightarrow s_n$, où $s_1 \ldots s_{n-1}$ sont les types des arguments, et $s_n$ est le type du résultat.

L'ensemble des termes typés $T_\Sigma$ est défini comme l'univers de Herbrand engendré à partir d'un ensemble $\Sigma$ d'opérateurs de la façon suivante:

- $\Sigma_S$ est inclu dans $T_{\Sigma S}$ pour toute $s \in S$;
- Si $\sigma \in \Sigma_{s_1 s_2 \ldots s_m s}$ et $t_1 \in T_{\Sigma s_1}$, $t_2 \in T_{\Sigma s_2}$,... $t_n \in T_{\Sigma s_m}$,, alors $s[t_1, t_2, \ldots t_m] \in T_{\Sigma S}$.

La sémantique des termes typés est donnée par la notion classique d'interprétation. Les domaines d'interprétation sont des *algèbres* (multi-sorte), c'est-à-dire, des familles A d'ensembles $A_S$, et des ensembles F de fonctions. Etant donnés des S et $\Sigma$ (comme définis plus haut), une algèbre <A,F> est une $\Sigma$-algèbre ssi à chaque sorte $\sigma \in S$ correspond un ensemble $A_S$, et pour chaque opérateur $\sigma \in \Sigma s$ il y a un élément a de $A_S$ qui lui correspond, et finalement pour chaque opérateur $\sigma \in \Sigma_{s_1 s_2 \ldots s_n s}$ il y a une fonction $f \in F$ de domaine $A_{S1} \times A_{S2} \times \ldots \times A_{Sn}$ et codomaine $A_S$.

Par exemple, si S={nat} et $\Sigma$={zéro:$\rightarrow$nat, un:$\rightarrow$nat, plus:nat$\rightarrow$nat}, alors les entiers naturels sont une $\Sigma$-algèbre avec l'addition '+', car:

- A la sorte *'nat'* correspond l'ensemble d'entiers naturels {0, 1, 2,...};

- Aux opérateurs *zéro* et *un* correspondent les éléments 0 et 1;
- A l'opérateur *plus* correspond la fonction calculée par l'addition (c'est-à-dire, {(3,2,5), (8,0,8), (3,17,20),...} ).

### 3.3.1.2.- Sémantique initiale

Il est très intéressant de noter que l'ensemble des termes typés forme lui-même une algèbre! En effet, pour exprimer $T_\Sigma$ en tant qu'algèbre, il suffit de définir des fonctions $\sigma_T = \lambda v_1, v_2, \ldots v_n \bullet \sigma[v_1, v_2, \ldots v_n]$, pour $\sigma \in \Sigma$. Toutefois, c'est une algèbre possèdant des caractéristiques intéressantes pour lesquelles elle est parfois appelée *le modèle standard.* Nous décrivons ci-dessous ces caractéristiques.

Les définitions nécessaires sont:

• Un homomorphisme $h:A \rightarrow B$ entre deux algèbres A et B est une famille $\{h_s\}$ d'applications $h_s:A_s \rightarrow B_s$ qui préservent les opérations, c'est à dire:

$h_s(c_A) = c_B$, pour des constantes $c_A \in A_s$, $c_B \in B_s$;

$h_s(f_A(a_1, a_2, \ldots, a_n)) = f_B(h_{s_1}(a_1), h_{s_2}(a_2), \ldots, h_{s_n}(a_n))$, pour $a_i \in A_{s_i}$

• On dit que deux algèbres sont isomorphes quand il existe deux homomorphismes, $h: A \rightarrow B$, $h^{-1}: B \rightarrow A$ tels que la composition $h^{-1} \circ h$ est égale à l'identité en A.

• Une algèbre J est *initiale* dans une classe C d'algèbres ssi il existe un homomorphisme unique $h: J \rightarrow A$ pour toute algèbre A dans C.

• Les algèbres initiales d'une classe d'algèbres forment une sous-classe d'algèbres isomorphes.

Proposition [Goguen et al.78].- L'algèbre des termes est initiale dans la classe $C_\Sigma$ de toutes les $\Sigma$-algèbres.

L'utilité de définir une algèbre initiale vient du fait que celle-ci est caractérisée d'une façon abstraite, par l'initialité, et non pas en termes d'une représentation quelconque. Néanmoins, nous pouvons visualiser de façon "concrète" l'algèbre initiale car nous disposons d'un représentant de la classe d'algèbres initiales isomorphes, à savoir, l'algèbre des termes $T_\Sigma$.

### 3.3.1.3.- La construction des types de données

Dans l'algèbre des termes chaque élément est différent. L'algèbre des termes est pour cela utile pour engendrer les éléments d'un domaine.

La propriété de l'algèbre des termes de ne posséder que des éléments distincts vient du fait qu'elle est une algèbre libre.

Définitions:

• Soit V un ensemble de variables, indexé par S. Si $v \in V_s$, alors on dit que v est une variable de type s.

• La Σ-algèbre librement engendrée par un sous-ensemble fini X de V est $T_{(\Sigma \cup X)}$, notée $T\Sigma(X)$. Exemple: Pour S={0,succ,+}, X={x,y}, $T\Sigma(X)$ est {0, succ(0),succ(succ(0)),... x, succ(x), succ(succ(x))... y, succ(y),... x+y,... x+succ(y)...} La caractéristique des algèbres libres est que étant donnée une affectation θ: X → A pour un ensemble de variables X et une algèbre A, il existe un homomorphisme unique $\underline{\theta}$: $T\Sigma(X)$ → A qui est une extension de θ dans le sens où $\underline{\theta}(x)=\theta(x)$ pour tout x ∈ A. Intuitivement, c'est le processus d'évaluation d'un terme étant données des valeurs pour les variables. Soit $Val(t, \theta)=\underline{\theta}(t)$.

Considérons maintenant les opérateurs suivants:

0: → entier
succ: entier → entier
+: entier,entier → entier

Les termes 0, succ(0), succ(succ(0)), 0+succ(0), succ(succ(0))+succ(0) sont tous des termes dans $T\Sigma_{entier}$. Mais si nous considérons la sémantique habituelle de ces opérations, nous aurions tendance à dire que 0+succ(0) est la même chose que succ(0), ou bien que succ(succ(0))+succ(0) est le même entier que succ(0)+succ(succ(0)), et ainsi de suite.

Nous n'allons donc maintenir la "liberté" que pour un ensemble distingué d'opérateurs appelés *constructeurs*. Les éléments de l'algèbre des termes formés exclusivement par des constructeurs seront effectivement distincts.[1] Les constructeurs permettent d'engendrer les ensembles de chaque type de données comme des univers de Herbrand.

Par exemple, les entiers peuvent être construits à partir des constructeurs 0 et succ: 0, succ(0), succ(succ(0)), succ(succ(succ(0)))... , pour 0, 1, 2,3...

*Dans la BT, les types sont introduits en donnant leurs constructeurs.* Ainsi, le type des entiers positifs peut être introduit par la déclaration des constructeurs 0 et succ:

```
0:    → entier
succ:entier → entier
```

### 3.3.1.4.- Les opérateurs dérivés

La description formelle des opérateurs autres que les constructeurs fait partie de la description des données, puisque celles-ci sont décrites par les opérations qu'on peut effectuer sur elles.

La sémantique des opérateurs non-constructeurs est donnée en identifiant d'une façon systématique les termes formés par eux avec des termes clos ne contenant que des constructeurs. Par exemple, nous dirons que:

e1) succ(0)+succ(succ(0)+0) = succ(succ(succ(0)))

Le terme à droite de e1 est *'la valeur'* ou la *'forme normale'* du terme à gauche. Pour décrire la

---

[1] Sauf dans le cas des équations sur les constructeurs.

valeur de tout terme contenant un certain opérateur, il est nécessaire d'établir des égalités comme e1, mais avec des variables qui seront instanciées pour les cas particuliers.

## *Définitions d'opérateurs*

• Les équations sont des égalités de la forme g = d où g,d$\in T\Sigma(X)$. Une Σ-algèbre A satisfait une équation g = d ssi pour toute affectation θ: X → A on a Val(g, θ) = Val(d, θ), où l'égalité indique qu'il s'agit d'un seul et même élément. Une Σ-algèbre satisfait un ensemble d'équations Ɛ ssi elle satisfait toute équation e ∈ Ɛ.

• L'ensemble de toutes les instances d'un ensemble d'équations Ɛ peut être vu comme une relation dans $T\Sigma(X) \times T\Sigma(X)$: on appelle la "congruence engendrée par Ɛ", noté $\equiv_\varepsilon$, la plus petite relation binaire qui contient Ɛ et qui a les propriétés de reflexivité, symétrie, transitivité et substitutivité (si $(a_i, b_i) \in \equiv_\varepsilon$ , $1 \le i \le n$, alors $(\sigma_A(a_1, \ldots, a_n), \sigma_A(b_1, \ldots, b_n)) \in \equiv_\varepsilon$ pour toute opération $\sigma_A$ avec le profil qui convient).

• L'algèbre quotient $A/\equiv_\varepsilon$ de A par la relation de congruence $\equiv_\varepsilon$ contient une partition de l'ensemble A telle que chaque classe d'équivalence E de la partition contient des éléments congrus par $\equiv_\varepsilon$, c'est-à-dire: $a_i, a_j \in E \Rightarrow (a_i, a_j) \in \equiv_\varepsilon$ (on note $a_i \equiv_\varepsilon a_j$). Les opérations de $A/\equiv_\varepsilon$ sont définis de la façon suivante: soit [a] la classe d'équivalence qui contient a; si $\sigma_A$ est une opération de profil $s_1, s_2, \ldots, s_n \rightarrow s$, alors pour $a_1, a_2, \ldots, a_n$ où $a_i$ est de sorte $s_i$,

$$\sigma_{A,\varepsilon}([a_1], [a_2], \ldots, [a_n]) = [\sigma_A(a_1, a_2, \ldots, a_n)]$$

• Pour un ensemble d'équations Ɛ on définit une classe $C\Sigma,\varepsilon$ (appelée *variété*) à laquelle appartiennent toutes les Σ-algèbres qui satisfont Ɛ. Cette variété a une algèbre initiale Jε, qui est le quotient de l'algèbre de termes par la relation de congruence $\equiv_\varepsilon$ engendrée par les équations.

Nous pouvons donc définir la sémantique des opérateurs non-constructeurs σ à l'aide d'équations faisant intervenir σ. Nous allons nous intéresser au modèle initial de ces équations.

Néanmoins, ce n'est pas n'importe quelle équation qui définit correctement un opérateur dérivé. Une bonne définition doit être cohérente et complète. Cohérente veut dire que deux éléments distincts d'un domaine doivent continuer à l'être une fois la définition faite. Une définition est complète quand elle prévoit tous les cas de termes possibles en vue de leur évaluation.

Le problème de la complétude des définitions, dans sa généralité, est indécidable. Heureusement, il n'est pas trop difficile en pratique de former des définitions cohérentes et complètes pour les opérateurs dérivés. Il suffit de respecter un nombre de contraintes, surtout syntaxiques, qui sont en fait des *conditions suffisantes* (mais pas nécessaires). Une façon d'exprimer ces contraintes est donnée par le principe de définition de [Huet, Hullot 80]:

## *Principe de définition*

Soit *G* l'ensemble des termes clos (sans variables) et *GC* le plus grand sous-ensemble de *G* que ne contient que des constructeurs. On dit que l'ensemble d'équations ε (avec la congruence associée $=_\varepsilon$) définit les opérateurs *D* sur les constructeurs *C* ssi pour chaque M dans *G* il existe un unique N dans *GC* tels que $M =_\varepsilon N$.

• Le principe de définition est séparé en deux contraintes afin de faciliter sa vérification:

1) Pour chaque M dans *G* il existe au moins un N dans *GC* tel que $M=_\varepsilon N$;
2) Pour tout M, N dans *GC*, $M=_\varepsilon N$ ssi M=N.

• Pour le point (1), une condition suffisante est la suivante [Huet et Hullot 80]:

1.1) Les équations ε interprétées comme un système de réécriture forment un système confluent et nœtherien;

1.2) La partie gauche des règles (équations) est de la forme $\sigma(S_i^1, S_i^2, \ldots S_i^n)$, avec σ dans *D* et $S_i^k$ dans *GC*;

1.3) L'ensemble $\{S_1, \ldots, S_k\}$ de n-uplet de termes de *GC* utilisés comme des arguments est "complet" sur les constructeurs, ce qui veut dire intuitivement qu'on a prévu tous les cas possibles d'arguments pour les règles. Cette condition garantit que si tous les arguments d'un opérateur défini (c'est à dire, dans *D*) sont des termes dans *GC*, alors il y aura toujours une règle applicable dans le système de réécriture.

Pour la vérification de ces points il est possible de donner des algoritmes de décision [Huet, Hullot 80], [Comon 86]. Leur automatisation dans le cadre de la BT peut constituer une aide à l'utilisateur pour former les définitions constructives.

Exemple.- Soit la définition suivante de la fonction à résultat booléen *perm*, qui teste si deux listes sont l'une une permutation de l'autre:

perm(nil, nil) = vrai
perm(nil, a•l) = faux
perm(a•l, b•m) = a∈ m ∧ b∈ l ∧ perm(elim(b,l), elim(a,m))

avec:

elim(a,nil) = nil
elim(a,b•l) = si (a=b) alors elim(a,l) sinon b•elim(a,l) [1]

Vérifier si cette définition de *perm* se conforme au principe de définition.

Preuve (pour *perm*):
- Point 1.1.- La confluence (locale) est vérifiée puisqu'il n'existe pas de paires critiques. Pour la terminaison, soit l'ordre défini par la longueur des listes; on a: elim(b,l) < a•l, elim(a,m) < b•m.

- Point 1.2.- Trivial.

- Point 1.3.- Les tuples des arguments de la définition sont: {(nil,nil), (nil,a•l), (a•l,b•m)}. Par l'algorithme de [Huet, Hullot 80] nous avons vérifié que la définition n'est pas complète; il manque le cas perm(a•l, nil). Nous pouvons la compléter en ajoutant l'équation:

perm(a•l, nil) = faux

---

[1] La sémantique des termes conditionnels est donnée dans [Potet 88].

ou bien en remplaçant la dernière équation de la définition par:

perm(a•l, x) = a∈ x ∧ perm(l, elim(a,x)).

Une propriété très importante des définitions formées en suivant le principe précédent est la possibilité d'utiliser ses équations comme des règles de réécriture pour calculer la valeur des termes. Ceci est fait dans le cadre des langages de programmation et spécifications tels que LPG [Bert et al. 87].

## 3.3.1.5.- La Généricité

Soient les types de données 'liste d'entiers' (liste-ent) et 'liste de caractères' (liste-car), introduits respectivement par:

$nil_1$: → liste-ent
$•_1$: entier, liste-ent → liste-ent

$nil_2$: → liste-car
$•_2$: entier, liste-car → liste-car

Est-il possible de définir les "listes de n'importe quoi" ? Ceci est une question pertinente car nous voudrions définir les types de données de la façon la plus générale possible. Une réponse à cette question est donnée par les types génériques [Bert, Jacquet 77], [Ehrig et al.84].

La définition d'un type générique comporte des *paramètres*, c'est-à-dire, des variables à instancier par des types 'concrets'. Par exemple, pour les listes on aurait:

```
nil: → liste[t]
•: t, liste[t] → liste[t]
```

Cette déclaration pourra par la suite être instanciée pour obtenir les listes d'entiers, des caractères, etc., par remplacement du paramètre `t` par les types entier, caractère; etc.

La sémantique d'un type générique correspond à l'union des modèles (initiaux) de toutes ses instantiations. (voir dans [Ehrig et al.84] une discussion détaillée de la sémantique des types paramétrables).

Avec les types génériques nous pouvons définir des opérations génériques. Par exemple, on peut avoir une opération de concaténation de listes:

```
+:   liste[t], liste[t] → liste[t]
```

avec les axiomes:

```
l + nil = l
l + a•m = a•(l + m)
```

L'opération '+' ainsi définie pourra par la suite être utilisée pour concaténer des listes d'entiers, d'ensembles, etc.[1] Les bénéfices de cette réutilisation des spécifications des opérateurs sont évidents.

Nous considérons un ordre partiel portant sur les instanciations des types génériques. Il est défini de la façon suivante:

$t_1 \leq t_2$ ssi il existe une substitution $\theta$ telle que $t_2\theta = t_1$,

où $t_1$ et $t_2$ sont des types génériques. Ainsi, par exemple, liste[ensemble[t]] ≤ liste[s].

Vis-à-vis d'un problème donné, nous essayons toujours de nous placer au bon niveau d'abstraction, c'est-à-dire, le plus élevé possible. Par 'plus élevé' nous voulons dire 'moins instancié', d'après l'ordre d'instanciation décrit plus haut. Par exemple, nous préférons synthétiser un programme de tri valable pour les listes d'éléments possédant un ordre total qu'un programme valable seulement pour les listes d'entiers.

Par ailleurs, il est intéressant de libérer l'utilisateur du système de synthèse de l'obligation de préciser le profil des fonctions à synthétiser. Le système de synthèse doit alors calculer le type le plus général à partir des types des opérateurs présents dans l'énoncé. Une telle *inférence de types* a été mise en œuvre dans le cadre d'un projet ENSIMAG [Dolle, Vigouroux 87], comme une partie d'un analyseur d'énoncés de problèmes de synthèse de programmes.

### 3.3.1.6.- Propriété exigée

Souvent on trouve des opérateurs génériques pour lesquels toutes les instanciations ne sont pas valables. Par exemple, l'opérateur `tri` qui ordonne une liste pourra être appliqué aux listes d'entiers ou de caractères, mais plus difficilement aux listes d'ensembles quelconques. Il faut donc caractériser les types 'concrets' pour lesquels un opérateur générique s'applique.

Cette caractérisation ne doit évidemment pas se faire par énumération des types valides. Il est possible de caractériser de façon abstraite les instanciations valides en donnant un ensemble d'axiomes qu'elles doivent satisfaire. Ces ensembles d'axiomes sont appelés *théories* dans [Goguen, Meseguer 84] et *propriétés* dans [Bert, Echahed 86]. Leur sémantique est tout simplement l'ensemble de leurs modèles. Par exemple, l'opérateur `tri` peut être appliqué aux listes d'éléments des types qui possèdent un ordre total, lequel peut être axiomatisé par:

| | |
|---|---|
| $a \leq a$ | (reflexivité) |
| $(a \leq b \wedge b \leq a) \Rightarrow a = b$ | (antisymétrie) |
| $(a \leq b \wedge b \leq c) \Rightarrow a \leq c$ | (transitivité) |
| $a \leq b \vee b \leq a$ | |

Les opérateurs génériques sont caractérisés par des propriétés également génériques; ainsi, dans l'exemple précédent, le type des variables a, b et c reste ouvert.

Nous attirons l'attention sur le fait que la sémantique des propriétés n'est pas initiale. Il est

---

[1] L'utilisation d'un même nom d'opération pour toutes ses instances donne lieu à une surcharge.

évident qu'avec l'opérateur '≤' de l'exemple précédent on ne peut pas construire le domaine d'un type. Pour les propriétés, au lieu de nous intéresser à un modèle en particulier (à savoir, le modèle initial[1]), nous nous intéressons à tous les modèles qui satisfont les axiomes. C'est la raison pour laquelle les restrictions propres aux définitions d'opérateurs en termes des constructeurs peuvent être levées.

Dans la BT les propriétés sont spécifiées par des équations. L'ensemble de modèles des spécifications équationnelles contient toujours des modèles initiaux, d'où l'intérêt de ce type de spécification, surtout si nous considérons que les propriétés sont justement utilisées pour caractériser des modèles initiaux (c'est le cas des *propriétés exigées*). Pour prouver qu'un type A satisfait une certaine propriété P, il faut prouver que tous les axiomes de P sont des théorèmes dans le type A. Cette preuve peut nécessiter de l'induction. Exemple: soit l'addition dans les entier naturels:

```
+ : nat,nat → nat
```

avec les axiomes:

```
x + 0 = x
x + succ(y) = succ(x + y)
```

Nous voudrions vérifier que l'opérateur + est commutatif, c'est-à-dire, que le type ENTNAT satisfait la propriété possédant l'axiome: `x % y = y % x` où `%` est un opérateur fictif. Il faut prouver que l'équation x+y = y+x (obtenue par remplacement de `%` par +) est un théorème dans le type ENTNAT. Preuve par induction:

*Lemme 1*: 0 + x = x.
*Cas de base:* 0 + 0 = 0 ;
*Pas d'induction*: 0 + succ(x) = succ(0 + x) = succ(x).
*Lemme 2*: succ(y) + x = succ(y + x).
*Cas de base*: succ(y) + 0 = succ(y) = succ(y + 0);
*Pas d'induction*: succ(y) + succ(x) = succ(succ(y) + x)) = succ(succ(y + x)) =
succ(y + succ(x)).
*Cas de base*: x + 0 = x = 0 + x (par le lemme 1);
*Pas d'induction*: x + succ(y) = succ(x + y) = succ(y + x) = succ(y) + x (lemme 2)
CQFD.

Pour le cas des propriétés décrites par des formules logiques, comme dans l'exemple précédent, il est nécessaire de former des équations dans un type 'booléen'. Les prédicats, et même les connectives, deviennent simplement des opérateurs du type booléen. La définition de ces opérateurs se fait dans le cadre d'une sémantique initiale. Par exemple, soit la définition des opérateurs booléens:

```
vrai, faux : → bool
non : bool → bool
et, ou : bool,bool → bool
```

avec les axiomes:

---

[1] Sa classe d'algèbres isomorphes.

```
non(vrai) = faux
non(faux) = vrai
x et vrai = x
x et faux = faux
x ou vrai = vrai
x ou faux = x
x imp y = non(x) ou y
```

Alors les axiomes de la propriété d'ordre partiel peuvent s'exprimer de la façon suivante:

```
x≤x = vrai
x≤y et y≤x = x≡y
(x≤y et y≤z) imp x≤z = vrai
x≤y ou y≤x = vrai
```

Comme une facilité de notation, nous allons souvent écrire les équations de la forme P(x) = vrai tout simplement comme P(x).

La sémantique d'une propriété contenant des opérateurs définis dans un type (comme les opérateurs et, ou... du type BOOL) se complique par le fait que dans les propriétés on considère normalement une sémantique équationnelle (ensemble de modèles), tandis que dans les types la sémantique est initiale. Une solution [Wirsing et al.83] consiste à restreindre les modèles de la propriété à ceux qui sont compatibles avec le type dont on utilise ses opérateurs.

L'utilité des propriétés n'est pas limitée à la vérification des instanciations des types génériques. Dans la BT nous allons définir également quelques propriétés mathématiques d'utilité générale. C'est le cas de la commutativité. Ce genre de propriété sont souvent nécessaires pour effectuer des transformations de formules. Quelques systèmes automatiques de manipulation de formules (démonstrateurs de théorèmes, etc.) ont des mécanismes spécialisés pour tenir compte opérationellement de ces propriétés (exemple.- l'unification associative-commutative de [Stickel 75]).

### 3.3.1.7.- Constructions alternatives des types

Dans le cadre de la synthèse de programmes il est souvent nécessaire de considérer les domaines des types ("carriers") de plusieurs façons différentes. Ainsi, par exemple, les listes sont classiquement construites à partir du • et de nil, mais quelquefois nous voudrions les regarder comme la concaténation de deux sous-listes $l_1$ et $l_2$. Tel est le cas du "mergesort", où une liste est éclatée en deux sous-listes qui sont par la suite ordonnées et fusionnées.

D'une façon générale, le choix des constructeurs pour engendrer un type est plus ou moins arbitraire. Par exemple, les entiers peuvent être construits des façons suivantes:

1) 0: → entier
succ: entier → entier
Exemple: 4 = succ(succ(succ(succ(0))))

2) 0: → entier
1: → entier
+: entier, entier → entier
Exemple: 4 = (1+((1+1)+0)+1

3) 0: → entier
1: → premier
sp: premier → premier
(_): premier → entier
*: entier, entier → entier
Exemple: 4 = 1*((sp(1))*(sp(1)))

(La construction 3 des entiers se fait à partir des nombre premiers)

Pour que 1+0 représente le même élément du domaine que 0+1 dans la construction 2, il est nécessaire d'identifier ces éléments. Autrement il n'y aura pas une correspondence univoque entre les éléments de 1 et ceux de 2. Il faut donc former des classes d'équivalence dans $T\{0,1,+\}$. Pour ce faire, nous sommes obligés d'introduire des équations sur les constructeurs:

$$x + 0 = x$$
$$x + y = y + x$$
$$(x + y) + z = x + (y + y)$$

Le quotient $T\{0,1,+\}/\equiv$ , où $\equiv$ est la congruence engendrée par ces équations, peut alors être mis en correspondence avec $T\{0,succ\}$. Toutefois, l'introduction d'équations sur les constructeurs entraîne beaucoup de complications (Par exemple, le principe de définition constructive, tel qu'il a été donné, ne s'applique plus). Il en est de même pour la construction avec * et les nombres premiers.

L'équivalence des présentations décrites peut se faire en explicitant l'isomorphisme de leurs algèbres de termes. Soit $h_{1,2}$: $T\{0_1,succ\} \rightarrow T\{0_2,1,+\}$, $h_{2,1}$: $T\{0_2,1,+\} \rightarrow T\{0_1,succ\}$

L'isomorphisme est défini par les équations suivantes:

$$h_{1,2}(0_1) = 0_2$$
$$h_{1,2}(succ(x)) = 1 + h_{1,2}(x)$$
$$h_{2,1}(0_2) = 0_1$$
$$h_{2,1}(1) = succ(0)$$
$$h_{2,1}(x + y) = h_{2,1}(x) +_1 h_{2,1}(y)$$

où $+_1$ est l'opérateur + défini constructivement dans $entier_1$ de la façon habituelle.

Il est conceptuellement difficile de faire cohabiter des constructions différentes d'un "même" type. Peut-on parler du type des entiers, indépendamment de ses constructions particulières ? Le problème des différentes constructions des types a été étudié dans [Wirsing et al.83], [Wadler 87], [Goguen, Meseguer 87]. Nous allons suivre ici cette dernière approche. L'idée fondamentale est l'introduction de sous-sortes d'une sorte. Un ordre partiel $\leq$ est défini entre les sortes $s_i$, dont la sémantique est l'inclusion des algèbres $As_i$: si $s_1 \leq s_2$ alors $As_1 \subseteq As_2$; les

termes de sorte $s_1$ sont donc en même temps des termes de sorte $s_2$.

Toutes les définitions concernant la sémantique initiale des types [Goguen et al.78] sont étendues aux "order sorted algebras" (OSA) dans [Goguen, Meseguer 87]. Nous ne le ferons pas ici, nous limitant à renvoyer le lecteur au papier cité...

Dans ce contexte, nous assimilons les différentes constructions d'un type à des sous-sortes distinctes $s_1, s_2, \ldots s_n \leq s$, où s est le type d'intérêt. Ainsi, soient les entiers construits par 0 et succ, identifiés par la sorte $entier_1$:

$0_1: \rightarrow entier_1$
$succ: entier_1 \rightarrow entier_1$

Soient les entiers construits par 0, 1 et +, de sorte $entier_2$:

$0_2: \rightarrow entier_2$
$1: \rightarrow entier_2$
$+: entier_2, entier_2 \rightarrow entier_2$

Dans le cadre des OSA il est possible d'établir des équations $g = d$ où $g \in T\Sigma_{s1}$, $d \in T\Sigma_{s2}$, $s1 \leq s$, $s2 \leq s$. Les équations de l'isomorphisme de l'exemple précédent peuvent donc s'exprimer comme:

$\mathcal{E}$: $0_1 = 0_2$
$succ(x) = 1 + x$

Pour que ces équations soient syntaxiquement correctes,la signature est étendue de la façon suivante[1]:

$+ : entier, entier \rightarrow entier$
$succ: entier \rightarrow entier$

Les équations de $\mathcal{E}$ sont vues comme des équations dans un type $entier \geq entier_{1,2}$. La partition $T\Sigma_{entier}/\equiv_{\mathcal{E}}$ correspond à la même algèbre initiale que $T\Sigma_{entier1}$ où $T\Sigma_{entier2}/\equiv_{comm,assoc}$ [Goguen, Meseguer 87].

La méthode qui en découle est donc la suivante:

1.- Définir une sorte $s_i$ pour chaque construction particulière du type considéré.
Exemple: $liste_1$, $liste_2$.

2.- Donner les constructeurs pour chacune de ces sortes. Exemple:
$nil_1: \rightarrow liste_1$
$\bullet: t, liste_1 \rightarrow liste_1$
c'est-à-dire:
$\{nil_1, a \bullet nil_1, b \bullet (a \bullet nil_1), \ldots\}$

---

[1] Il en résulte une surcharge des opérateurs

$nil_2$: $\rightarrow$ $liste_2$
{ }: t $\rightarrow$ $liste_2$
+: $liste_2$, $liste_2$ $\rightarrow$ $liste_2$
avec:
$l+nil_2 = l$
$l_1+(l_2+l_3) = (l_1+l_2)+l_3$
c'est-à-dire:
{$nil_2$, {a}, {b}+{a},...}

3.- Donner la super-sorte $s \geq s_i$, et étendre les signatures en déclarant les opérateurs dans la super-sorte. Exemple:
liste$\geq$$liste_1$, liste$\geq$$liste_2$
•: t,liste $\rightarrow$ liste
+: liste,liste $\rightarrow$ liste

4.- Ecrire des équations $\mathcal{E}$ dans s pour mettre en rapport les opérateurs des $s_i$. Exemple:
$nil_1 = nil_2$
$a \bullet l_1 = \{a\}+l_2$

5.- Votre type $T_{\Sigma s,\mathcal{E}}$ est prêt.

Dans l'exemple, on aurait les classes d'équivalence (éléments de $T_{\Sigma s,\mathcal{E}}$) comme:{$nil_1$, $nil_2$}, {a•$nil_1$, {a}, $nil_2$+{a}...},...

Dans le type *entier* on ne distingue plus de façon explicite l'ensemble des constructeurs.[1] La définition de *entier* comme une super-sorte de *entier1* et de *entier2* apporte un niveau d'abstraction par rapport aux présentations constructives. Pour en profiter pleinement, les axiomes des entiers devront s'écrire dans le type *entier* [Goguen, Meseguer 87]; par exemple, (a*b)/a = b sera une équation dans la sorte *entier* plutôt que dans *entier1* ou *entier2*.

## 3.3.1.8.- Spécifications déclaratives des opérateurs

Dans la BT les opérateurs sont normalement définis en termes des constructeurs, selon les normes (équationnelles) du principe de définition. Néanmoins, il est souvent utile de caractériser les opérateurs d'une façon plus déclarative. Par "déclarative" nous voulons dire qu'on n'indique pas la façon de les calculer.

Les expressions utilisées pour spécifier un opérateur sont des équations qu'on peut réduire à la forme $P(x) \Rightarrow Q(x,f_S(x)) = vrai$, où $f_S$ est une fonction de Skolem.

Par exemple, soit la définition suivante de la fonction perm:

perm(x,y) = [count(w,x) = count(w,y)]

où la fonction count est définie par:

---

[1] Cette distinction n'est pas nécessaire pour la définition d'une algèbre initiale [Goguen et al.78]

count(a,nil) = 0
count(a,b•l) = si (a=b) alors succ(count(a,l)) sinon count(a,l)

Une telle formule peut spécifier une fonction dans un sens strict si l'ensemble de fonctions dans l'algèbre des termes qui satisfont cette formule comporte exactement un élément. Deux conditions doivent alors être satisfaites:

- La spécification possède au moins un modèle dans l'algèbre des termes parce que [P(x) $\Rightarrow$ Q(x,f(x))] = vrai est un théorème du type concerné;

- La fonction spécifiée est unique.

L'unicité de la fonction spécifiée n'a pas d'importance pour la synthèse de programmes; en effet, il nous suffit de trouver une fonction quelconque satisfaisant la spécification. Par contre, quand dans la BT nous présentons une formule comme spécification d'une fonction, il est indispensable que celle-ci soit le seul modèle de cette formule. En général la preuve d'unicité est assez difficile. Néanmoins, si la fonction spécifiée correspond à un opérateur disposant aussi d'une définition constructive, il suffit de prouver que les deux définitions sont équivalentes. Ceci veut dire que chacune est une conséquence logique de l'autre. La preuve de cette équivalence dans le sens récursif $\rightarrow$ déclaratif est la preuve de programmes.

Il est clair que les spécifications sont une forme de propriété, car la sémantique y est équationnelle.

L'utilité de spécifier déclarativement les opérateurs de la BT est justifiée par:

1) La nécessité de comparer les énoncés des (sous-) problèmes donnés avec les énoncés des fonctions de base disponibles dans la BT. En effet, étant donné un énoncé $P(x) \Rightarrow Q(x,f_S(x))$ pour la fonction $f_S$, nous voudrions savoir s'il existe dans le système une fonction $f_0$ avec un énoncé $P_0(x) \Rightarrow Q_0(x,f_0(x))$, telle que: [Potet 88]

$P(x) \Rightarrow P_0(x)$
$P(x) \wedge Q_0(x,y) \Rightarrow Q(x,y)$

Si ces conditions sont remplies, alors le programme qui calcule $f_0$ satisfait également l'énoncé de f.[1]

2) Une définition non-récursive d'un opérateur permet d'exprimer celui-ci en termes d'autres opérateurs. Dans notre exemple, *perm* est exprimé en termes de *count*. Si la BT dispose de connaissances pour count qui ne sont pas disponibles pour perm, alors il peut être intéressant de remplacer les occurrences de perm par sa définition. D'une façon générale, les définitions déclaratives dans la BT expriment l'opérateur en question en termes d'opérateurs *de plus bas niveau* dans une hiérarchie de complexité qu'il reste à définir de façon précise. Au niveau le plus bas, il est très difficile de donner des définitions déclaratives (exemple: essayez de donner une définition non-récursive de *count*).

---

[1] Dans ce cas les énoncés peuvent ne pas spécifier une fonction unique. C'est la raison pour laquelle dans [Potet 88] le terme *énoncé* est employé au lieu de *spécification*.

3) Les définitions déclaratives sont souvent plus naturelles pour l'utilisateur, qui pourrait ainsi mieux comprendre le sens des fonctions disponibles dans la BT.

La DSKB du système CYPRESS comporte des spécifications déclaratives des opérateurs, en plus des définitions calculatoires.

### 3.3.1.9.- Théorèmes distingués

En plus des définitions déclaratives et constructives, nous allons stocker dans la BT un ensemble fini de formules valides dans le modèle initial auquel elles font référence. D'après la discussion du §3.2, ces formules sont des théorèmes distingués. Ces théorèmes sont la forme typique de connaissance utilisée en synthèse de programmes (§1.4). Par exemple, la formule (a+b) - b = a dans les entiers n'est pas la définition de l'addition ni de la soustraction, mais elle peut être utile pour effectuer une simplification.

Il va de soi que cette forme de connaissance fait partie de la caractérisation des données, tout comme les définitions. La différence est que les formules considérées dans ce paragraphe sont toutes des conséquences logiques des axiomes. Dans ce sens les théorèmes distingués sont des informations redondantes par rapport aux axiomes; leur utilité vient du fait qu'ils permettent d'éviter le travail de les dériver à partir des axiomes par un processus déductif.

*Les théorèmes pertinents*

Les informations stockées dans la BT sont supposées être "utiles" au système de synthèse (utilisateur compris). Nous décrivons ci-dessous quelques formes de théorèmes et leur utilité.

1) Définitions constructives des opérateurs. Exemple:

$$concat(a \bullet l, m) = a \bullet concat(l,m)$$

Utilité.- En plus de leur valeur définitionnelle, ces formules sont utilisées pour l'introduction de la récursion. Plus spécifiquement, elles permettent d'utiliser l'hypothèse d'induction (issue de l'application d'un schéma récursif) pour résoudre des problèmes de dérivation d'hypothèses. Par exemple, soit la spécification du tri:

$$perm(x, tri) \wedge ord(tri)$$

A la suite de l'introduction d'un schéma récursif, les hypothèses suivantes sont engendrées (voir §2) :

| | | |
|---|---|---|
| $(h_1)$ | $x \neq nil$ | cas ...sinon de la récursion |
| $(h_2)$ | $perm(cdr(x),z)$ | } hypothèse d'induction |
| $(h_3)$ | $ord(z)$ | |

Soit la définition suivante de perm:

$(t_1)$ $perm(nil,x) = (x=nil)$

$(t_2)$ $perm(a \bullet l, x) = a \in x \wedge perm(l, elim(a,x))$

Soit finalement le théorème:

$(t_3)$ $\quad x \neq nil \Rightarrow x = car(x) \bullet cdr(x)$

Considérons la partie perm(x,tri) de la spécification:

$(b_0)$ $\quad perm(x,tri)$

Nous voulons trouver une hypothèse additionnelle en termes de x, z et tri permettant de prouver $b_0$. (Ce sera un énoncé pour la fonction de recomposition du schéma récursif).

Solution.- Par $(h_2)$ et $(t_3)$, nous avons:

$(b_1)$ $\quad perm(car(x) \bullet cdr(x), tri)$

Par $(t_2)$ sur $(b_1)$ :

$(b_2)$ $\quad car(x) \in tri \wedge perm(cdr(x), elim(car(x),x))$

Finalement, par unification de $(h_2)$ avec un sous-terme de $(b_2)$, on obtient la solution:[1]

$(b_3)$ $\quad car(x) \in tri \wedge z = elim(car(x),x)$

2) Spécifications déclaratives. Les formules constituant les spécifications déclaratives des opérateurs connus par la BT ont une utilité en dehors de leur valeur définitionnelle. En effet, à partir des définitions déclaratives il est souvent possible d'obtenir des théorèmes intéressants moyennant des manipulations simples.

Par exemple, soit la définition de la fonction booléenne perm:

(1) $\quad perm(x,y) \Leftrightarrow count(w,x) = count(w,y)$

Le théorème perm(x,x) est obtenu directement de (1) en raison de la substitutivité de l'égalité. Or, l'obtention de ce même théorème à partir de la définition constructive nécessite de l'induction.

3) Théorèmes de simplification.- Ce sont des égalités avec un sens d'application, utilisées comme des règles de réécriture, dans le but de simplifier des formules. Le choix du sens d'application dépend d'un ordonnancement des termes considéré par l'utilisateur. Nous sommes particulièrement intéressés par les ordres dits de *simplification* [Dershowitz 83], où la relation de *plongement* est inclue. Celle-ci est introduite par des égalités de ka forme $\sigma(\ldots t \ldots) = t$, pour un opérateur $\sigma$. Une autre forme de simplification est l'élimination de termes, effectuée par des règles de la forme $g \rightarrow d$ où $var(d) \subset var(g)$.

---

[1] C'est une application de la substitutivité de l'égalité.

Exemples:

$\neg(\neg P) \rightarrow P$ (plongement)
$car(x \bullet l) \rightarrow x$ (plongement, élimination de termes)
$P \vee \neg P \rightarrow$ vrai (élimination de termes)

4) Théorèmes de lien.- Il est souvent nécessaire de remplacer les occurrences d'un opérateur dans une formule par d'autres opérateurs pour lesquels on dispose d'hypothèses. En effet, pour pouvoir utiliser les hypothèses il est nécessaire que celles-ci "collent" avec la formule à transformer (le but). Or, si les opérateurs intervenant dans les hypothèses ne sont par les mêmes que ceux intervenant dans le but, alors les hypothèses ne pourront être utilisées. Par exemple, soit le but:

$(b_0)$ less(n,z) (Pour tout w dans z, $n \leq w$)

Soient les hypothèses:

$(h_1)$ first(m,z) (m est le premier élément de z)
$(h_2)$ ord(z)

Afin d'utiliser $(h_1)$ et $(h_2)$ pour transformer $(b_0)$, il faut faire en sorte qu'ils aient des opérateurs en commun; autrement tout essai de 'pattern matching' est voué à l'échec. Pour ce faire, nous pouvons faire appel aux *théorèmes de lien*, qui sont des théorèmes distingués mettant en rapport deux opérateurs différents. Pour notre exemple, soit le théorème:

(t) $ord(l) \wedge first(x,l) \Rightarrow less(x,l)$

Ce théorème met en rapport less avec ord et aussi avec first, ce qui permettra par la suite d'utiliser les hypothèses.[1]

Il est toutefois nécessaire de préciser la notion de théorème de lien, car n'importe quel théorème contenant au moins deux opérateurs pourrait devenir un théorème de lien. Les "bons" théorèmes de lien sont ceux qui permettent de faire un remplacement d'opérateurs sans introduire trop d'effets de bord négatifs. Par exemple, soient les théorèmes:

$(t_1)$ $[car(l)=car(reverse(l))] \Leftrightarrow perm(cdr(reverse(l)),cdr(l))$
$(t_2)$ perm(reverse(l), l)

Nous aurions tendance à dire que le théorème $(t_1)$ lie de façon "plus directe" les opérateurs perm et reverse que $(t_2)$. L'utilité d'un théorème en tant que théorème de lien est inversement proportionnelle à:

- sa complexité;
- la quantité d'opérateurs autres que les opérateurs liés.

Ce serait intéressant de disposer dans la BT d'un outil pour mesurer ces indices par des

[1] Cet exemple est continué dans le §4.

fonctions heuristiques, et calculer ainsi la "qualité" des théorèmes en tant que théorèmes de lien.

### 3.3.1.10.- Exemples d'entrée/sortie

Au §1.1 nous avons discuté l'emploi des couples d'entrée-sortie en tant que spécifications pour la synthèse de programmes. Cette forme de spécification peut s'avérer très utile dans le cadre de la BT.

En effet, considérons le problème de la reconnaissance des fonctions de base: nous voulons savoir si une fonction f pour laquelle on dispose d'une spécification logique, correspond à une des fonction $f_i$ connues par le système..[1] Supposons ensuite que pour chaque fi nous disposons d'un ensemble d'exemples d'E/S de la forme $\{(x_1,f_i(x_1)), \ldots,(x_n,f_i(x_n))\}$. Alors si l'on dispose également d'un tel ensemble pour f, il est possible d'éliminer toutes les $f_i$ dont les exemples ne coïncident pas avec les exemples de f. Cette opération de réduction de l'espace de recherche ne fait intervenir que des données statiques, et elle peut donc être réalisée en utilisant des techniques des bases de données. Même dans le cas où les exemples pour f ne sont pas directement disponibles, nous pouvons évaluer par réécriture la spécification (non-récursive) de f pour les exemples de $f_i$ stockés dans la BT. Exemple.- Soit f spécifiée par:

f: liste[t] → liste[t]
avant(x,y,l) ⇒ avant(y,x,l),

où la fonction booléenne *avant* est définie récursivement comme:

avant(a,b,nil) = faux
avant(a,b,x•l) = [(a=x ∧ b∈ l) ∨ avant(a,b,l)]

Supposons que les exemples suivant sont disponibles:

tri: liste[t] → liste[t]
exemples:
() → ()
(a) → (a)
(a b) → (a b)
(b a) → (a b)
(c a b) → (a b c)

r-rotate: liste[t] → liste[t]
exemples:
() → ()
(a) → (a)
(a b) → (b a)
(b a) → (a b)
(c a b) → (b c a)

[1] Nous nous restreignons aux fonctions avec le même profil.

l-rotate: liste[t] → liste[t]
exemples:
() → ()
(a) → (a)
(a b) → (b a)
(b a) → (a b)
(c a b) → (a b c)

reverse: liste[t] → liste[t]
exemples:
() → ()
(a) → (a)
(a b) → (b a)
(b a) → (a b)
(c a b) → (b a c)

Nous voulons savoir si f correspond à une de ces fonctions connues.

Cas 1.- Nous disposons des exemples $\{(x_k,f(x_k))\}$:

() → ()
(a) → (a)
(a b) → (b a)
(b a) → (a b)
(c a b) → (b a c)

Dans ce cas, nous pouvons éliminer tout de suite les candidats tri, l-rotate et r-rotate; la preuve formelle d'équivalence des spécifications ne se ferait donc que pour le cas de reverse.

Cas 2.- Nous ne disposons pas d'exemples. Il faut alors évaluer par réécriture les exemples des $f_i$ sur la spécification de f. Cette évaluation peut nécessiter des mécanismes d'énumération des liste, ensembles, etc.

Par exemple, pour le couple (c a b) → (b c a) de r-rotate, l'évaluation de avant(c,a,(c a b)) ⇒ avant(c,a,(b c a)) donne:

avant(c,a,(c a b)) →
[(c=c) ∧ a∈ (a b)] ∨ avant(c,a,(a b) → vrai

avant(a,c,(b c a)) →
avant(a,c,(c a)) →
avant(a,c,(a)) →
avant(a,c,nil) → faux

Donc avant(c,a,(c a b)) ⇒ avant(c,a,(b c a)) est faux. Ceci élimine la possibilité d'équivalence entre r-rotate et f.

L'évaluation des termes par réécriture reste beaucoup plus efficace que la preuve d'équivalence

des spécifications logiques. D'ailleurs, l'évaluation par réécriture des termes clos est fournie par des langages de programmation tels que LPG [Bert et al.87].

### 3.3.1.11.- La documentation dans la BT

Les renseignements en langue naturelle (le français, en l'occurrence) sont aussi une description (informelle) des types de données et de leurs opérations. Cette forme de description, si méprisée par tout théoricien qui se respecte, est pourtant très appréciée par l'utilisateur naïf (nous tous), en particulier dans un système qui est sensé assiter l'utilisateur dans la construction de programmes...

Le problème avec la documentation en français est qu'elle demeure marginale tant qu'il n'y a pas d'interface avec la partie du système compréhensible pour l'ordinateur. Cette remarque ne s'applique pas aux exemples, dont la valeur en tant que documentation, pour aider l'intuition de l'utilisateur, est indiscutable.

*Thus we feel that a better grip on the way to structure the theory in terms of which specifications are made is a prerequisite for raising verification and synthesis techniques above the toy problem level. [Bustall, Goguen 77]*

## 3.3.2.- Structure de la BT

L'importance d'une structure suffisamment riche pour notre BT a été évoquée aux §3.1.4, §3.2.2, §3.3.

Nous avons choisi une structure de la BT basée sur des unités modulaires de spécification, comme dans [Burstall, Goguen 77], [Gresse 84], [Bert et al.87]. L'idée générale de cette structure est exprimée dans l'équation suivante:

STRUCTURE = PRESENTATIONS + RELATIONS ENTRE PRESENTATIONS

Les présentations sont des modules de définition de types ou de propriétés. Elles sont à la base de la structure de la BT. Au dessus des présentations nous définissons un ensemble de relations établissant des liens entre les présentations.

D'autres structures étaient envisageables, en particulier une organisation du type relationnel. Une organisation relationnelle consiste à former des tables où les colonnes sont des noms de champs (ensembles) $N_1,\ldots ,N_m$ et les lignes sont des éléments de $N_1\times\ldots \times N_m$. On aurait, par exemple, une table pour le profil des opérateurs de la BT:

| NOM-OP | SORTE | RANG |
|---|---|---|
| + | nat | 1 |
| vrai | bool | 0 |
| succ | nat | 0 |
| faux | bool | 0 |
| 0 | nat | 0 |
| + | nat | 2 |
| succ | nat | 1 |
| + | nat | 0 |
| ... | ... | ... |

où le RANG indique l'index des sortes pour un profil $\sigma$: $s_1,\ldots,s_n \rightarrow s_0$.

Les systèmes relationnels bénéficient de la maturité des SGBD mis en œuvre à nos jours. Par contre, un de leur inconvénients est de provoquer un "éclatement" de l'information, assez visible, d'ailleurs, dans l'exemple précédent: le profil de '+' se trouve submergé dans une masse de profils dans une table; sa définition se trouve dans une autre table, ses propriétés dans une troisième, etc. Nous voudrions rassembler dans une même unité toutes les informations concernant un même objet (l'opérateur '+' en l'occurrence). Ce choix nous place dans le terrain des organisations "orientées-objet" [réf], au moins au niveau conceptuel. Dans une implantation, les informations pourraient être en réalité stockées dans des tables, mais ce fait serait invisible à l'utilisateur.

### 3.3.2.1.- Les présentations

Dans une présentation nous pouvons distinguer deux types d'informations: les informations de nature syntaxique, relatives à une signature -où sont donnés les noms des sortes et des opérations, ainsi que les profils des opérateurs-, et les axiomes définissant la sémantique des opérations.

Les présentations de types, où nous considérons une sémantique initiale, sont commencées par le mot **type**; les présentations de propriétés, où la sémantique est équationnelle, sont identifiées par le mot **propriété**. Exemples:[1]

```
type ENTIERS
sortes ent
opérateurs
      0: → ent
      succ, pred: ent → ent
      +: ent,ent → ent
axiomes
      succ(pred(t)) = t
      pred(succ(t)) = t
      a + 0 = a
      a + succ(b) = succ(a + b)
      a + pred(b) = pred(a + b)
ftype

propriété COMM1
sortes t
opérateurs
      *: t,t → t
axiomes
      a * b = b * a
fpropriété
```

Chaque présentation possède une étiquette (ENTIERS, COMM1,...) qui est essentielle pour lui faire référence.

### 3.3.2.2.- Combinaison de présentations

Comme [Burstall, Goguen 77], nous bâtissons des théories complexes en combinant plusieurs unités plus simples. Dans ce paragraphe nous définissons les moyens utilisés pour combiner des présentations.

La construction fondamentale pour combiner des présentations est la rubrique **contient**, où l'on donne le nom (étiquette) d'une présentation dont les sortes, les opérateurs et les axiomes sont ajoutés à la présentation considérée, au lieu du **contient**; ce remplacement se fait au niveau du

---

[1] Nous n'allons pas définir formellement la syntaxe des présentations; les exemples suffiront pour la saisir intuitivement. Elle est similaire à celles de LPG ou OBJ.

texte. Par exemple, soient les présentations:

```
type BOOL0
sortes bool
opérateurs
      faux, vrai: → bool
ftype

type BOOL1
contient BOOL0
opérateurs
      et: bool, bool → bool
      non: bool → bool
axiomes
      non(faux) = vrai
      non(vrai) = faux
      a et vrai = a
      a et faux = faux
ftype
```

Alors BOOL1 peut être développé comme:

```
type BOOL1
sortes bool
opérateurs
      faux, vrai: → bool
      et: bool, bool → bool
      non: bool → bool
axiomes
      non(faux) = vrai
      non(vrai) = faux
      a et vrai = a
      a et faux = faux
ftype
```

Une autre construction utilisée pour combiner des présentations est le renommage, qui ne sert qu'à changer les noms des sortes et des opérateurs d'une présentation. La notation utilisée est: P $[r_1 \rightarrow w_1, r_2 \rightarrow w_2, \ldots, r_n \rightarrow w_n]$ où $r_i$ sont des symboles de sortes et d'opérateurs de la présentation P, et $w_i$ sont les nouveaux nom affectés aux $r_i$; le symbole '→' se lit 'devient'. Par exemple, les deux présentations suivantes sont équivalentes:

```
type BOOL-ANGLAIS
sortes bool
opérateurs
      true, false: → bool
ftype
```

```
type BOOL-ANGLAIS
contient BOOL0 [vrai → true, faux → false]
ftype
```

Quelques constructions utilisées dans d'autres approches pour combiner des unités modulaires de définition, telles que [Burstall, Goguen 77], [Bert et al.87], peuvent s'implanter au moyen de contient et du renommage. Ainsi, la combinaison de [Burstall, Goguen 77], définie comme la simple adjonction de deux présentations $P_1$ et $P_2$, notée '$P_1 + P_2$', peut s'implanter avec la clause contient. Par exemple, une présentation réunissant les sortes et les opérations des présentations BOOL et NAT est la suivante:

```
type BOOL-NAT
contient BOOL, NAT
ftype
```

De manière similaire, l'enrichissement de [Bert et al.87] est immédiat à partir de **contient**.

### 3.3.2.3.- Présentation d'un type générique

Dans le cas des types génériques, la propriété qui doit être satisfaite par l'argument est spécifiée, en utilisant pour cela une rubrique **propriété exigée**. Exemple:

```
type LIST-ELEM
propriété exigée EGALITE
contient BOOL
sortes liste[t]
opérateurs
      nil: → liste[t]
      •: t, liste[t] → liste[t]
      ∈: t, liste[t] → bool
axiomes
      a∈nil = faux
      a∈ (b•l) = (a≡b) ∨ a∈l
ftype
```

L'équation `a∈ (b•l) = (a≡b) ∨ a∈l` de LIST-ELEM fait référence à l'opérateur '≡' de la propriété EGALITE.

### 3.3.2.4.- Différentes classes de présentations

Le rôle de chaque présentation dans la BT est distingué explicitement. En plus de la distinction entre les présentation de types et celles de propriétés, on peut avoir des cas de présentations de types nécessitant d'être considérées différemment. Ainsi, les présentations introduisant les constructeurs d'un type doivent être repérées pour traiter la question des décompositions des données, et elles ne doivent donc pas être mélangées avec d'autres informations telles qu'un théorème de simplification, par exemple.

Les classes de présentations que nous considérons sont:

• Présentations de constructeurs d'un type. Nous y présentons une sorte et ses constructeurs. Ce type de présentation est identifié par la marque '(Constructeurs)' après le mot clé **type**. Exemple:

```
type (Constructeurs) NAT-SUCC
sorte natsuc
opérateurs
      0:    → natsuc
      succ: natsuc → natsuc
ftype
```

• Présentations d'opérateurs dérivés en termes de constructeurs. Ces présentations comportent des définitions d'opérateurs dérivés en suivant les normes de la définition constructive (§3.3.1.4), ou bien des définitions d'opérateurs en termes d'autres opérateurs, ceux-ci étant définis constructivement. Ce type de présentation est identifiée par la marque '(DéfConstructive)'. Une telle présentation doit comporter la clause **contient** suivie du nom d'une présentation de constructeurs ou bien d'une présentation dont la normalisation contient les constructeurs du type considéré. Exemple:

```
type (DéfConstructive) PLUSMULT
contient NAT-SUCC
opérateurs
      +,*: natsuc, natsuc → natsuc
axiomes
      a + 0 = a
      a + succ(b) = succ(a + b)
      a * 0 = 0
      a * succ(b) = (a * b) + a
ftype
```

• Des présentations non-constructives de type; une telle présentation contient la déclaration de la super-sorte d'un ensemble de sortes dont il existe des constructeurs. Elles sont identifiées par la marque '(SurSorte)'. Exemple:

```
type (SurSorte) NAT
contient NAT-SUCC, NAT-PLUS, NAT-PREM
sortes nat > natsuc, nat > natplus, nat > natprem
ftype
```

• Des présentations de théorèmes, identifiées par la marque '(Théorèmes)'; elles ne comportent que la rubrique 'axiomes'. Exemple:

```
type (Théorèmes) MOINSPLUS
contient NAT
axiomes
      (a + b) - b = a
ftype
```

• Des présentations de spécifications déclaratives d'opérateurs, identifiées par la marque '(spécification)'. Exemple:

```
propriété (spécification) PERMUT
contient NAT, LISTES
opérateurs
      perm:      liste[t], liste[t] → bool
axiomes
      perm(x,y) = (count(w,x) = count(w,y))
fpropriété
```

### 3.3.2.5.- Déclaration de modèles

Nous pouvons déclarer qu'un certain type est un modèle d'une propriété donnée. Ceci se fait en plaçant dans la présentation du type la rubrique 'satisfait', suivie du nom d'une ou plusieurs propriétés. Cette déclaration revient à dire que les axiomes de la propriété sont vérifiés dans le type, et qu'ils sont donc des théorèmes du type. La vérification de cette condition suppose la preuve des axiomes de la propriété à partir de ceux du type; cette preuve est souvent inductive et pour le moment nous la laissons à la charge de l'utilisateur. Exemple:

```
type (déf-constructive) NAT
contient NAT-SUCC
opérateurs
      +: natsuc, natsuc → natsuc
axiomes
      a + 0 = a
      a + succ(b) = succ(a + b)
satisfait COMM [t→natsuc, *→+]
ftype
```

où COMM est:

```
propriété COMM
sortes t
opérateurs
      *: t,t → t
axiomes
      a * b = b * a
fpropriété
```

### 3.3.2.6.- Hiérarchies de propriétés

Certaines propriétés sont 'plus fortes' que d'autres, dans le sens que tous les modèles des premières sont aussi modèles des dernières. Par exemple, soient les propriétés:

```
propriété ORDRE-PARTIEL
sortes u
opérateurs
      ≤, eq : u,u
axiomes
      x ≤ x = vrai
      eq(x,y) = (x ≤ y et y ≤ x)
      (x ≤ y et y ≤ z) imp (x ≤ z) = vrai
fpropriété

propriété EGALITE
sortes u
opérateurs
      eq : u,u
axiomes
      eq(x,x) = vrai
      eq(x,y) = eq(y,x)
      (eq(x,y) et eq(y,z)) imp eq(x,z) = vrai
fpropriété
```

La propriété ORDRE-PARTIEL est plus forte que la propriété EGALITE, car tous les axiomes de EGALITE peuvent être prouvés à partir de ceux de ORDRE-PARTIEL. La vérification de ceci nécessite une preuve équationnelle (l'induction n'est pas nécessaire). Pour l'exemple précédent:

Preuve de l'axiome eq(x, x) :
eq(x,x) = (x≤y et y≤x) = (vrai et vrai) = vrai

Preuve de eq(x,y) = eq(y,x):
eq(x,y) = (x≤y et y≤x) = (y≤x et x≤y) = eq(y,x)

Preuve de (eq(x,y) et eq(y,z)) imp eq(x,z) = vrai:
(eq(x,y) et eq(y,z)) = (x≤y et y≤x et y≤z et z≤y) = ((x≤y et y≤z) et (z≤y et y≤x)),
donc (eq(x,y) et eq(y,z)) = ((x≤y et y≤z) et (z≤y et y≤x));
Par remplacement de cette équation et de l'axiome eq(x,z) = (x≤z et z≤x) dans l'axiome de transititivité de ≤, on a: (eq(x,y) et eq(y,z)) imp eq(x,z) = vrai

Etant données deux propriétés P1 et P2, la déclaration **subsume** P1, insérée dans P2 indique que P1 est plus forte que P2. Exemple:

```
propriété ORDRE-PARTIEL
      ... ...
axiomes
      ... ...
subsume EGALITE
fpropriété
```

Dans certains cas, toutes les équations d'une propriété P1 sont héritées d'une autre propriété P2 (par la clause **contient**). Alors P1 est automatiquement plus forte que P2, et la déclaration **subsume** n'est pas nécessaire. Par exemple, la présentation d'un monoïde contient la propriété d'associativité.[1]

La relation 'plus forte que' définit un ordre partiel dans les propriétés qui peut être visualisé graphiquement comme un graphe sans cycles (dag).

### 3.3.2.7.- Les relations inter-présentations

Nous distinguons deux types de liens entre les présentations: les liens explicites et les liens implicites.

Les liens explicites sont ceux qui proviennent d'une référence, dans une présentation, à une autre présentation. Les déclarations **contient, propriété exigée, subsume** et **satisfait** sont à l'origine de ces liens.

Nous allons définir les prédicats suivants pour dénoter les relations binaires établies par ces déclarations:

contientExp(A, B) ssi (**type/propriété** A **contient** B...) ∈ BT
propriétéExigée(A, P) ssi (**type** A ...**propriété exigée** P...) ∈ BT
subsumeExp(P, Q) ssi (**propriété** P...**subsume** Q...) ∈ BT
modèleExp(A, P) ssi (**type** A...**satisfait** Q...) ∈ BT

Les liens implicites sont essentiellement des extensions des liens explicites. Par exemple, si une propriété A subsume une propriété B, qui subsume à son tour une troisième propriété C, alors A subsume aussi C (transitivité du lien **subsume**). Ce sont des formes d'héritage associées à notre organisation orientée-objet.

Les marques associées aux différentes classes de présentations seront également utilisées comme des prédicats pour définir la structure de la BT; ainsi nous avons:

| | |
|---|---|
| typeConstructeurs(A) | ssi (**type** (Constructeurs) A ...) ∈ BT; |
| typeDéfConstructive(A) | ssi (**type** (DéfConstructive A ...) ∈ BT; |
| typeSurSorte(A) | ssi (**type** (SurSorte) A ...) ∈ BT; |
| typeThéorèmes(A) | ssi (**type** (Théorèmes) A ...) ∈ BT; |
| propriétéSpécification(P) | ssi (**propriété** (Spécification) P ...) ∈ BT. |

[1] ...si les présentations ont été écrites avec un minimum de sens commun

Nous définissons les prédicat d'extension en utilisant une notation (et sémantique[1]) à la Prolog. Les définitions suivantes sont nécessaires:

présentation(A) :- type(A).
présentation(A) :- propriété(A).

type(A) :- typeConstructeurs(A).
type(A) :- typeDéfConstructive(A).
type(A) :- typeSurSorte(A).
type(A) :- typeThéorèmes(A).

propriété(P) :- propriétéSpécification(P).[2]

Les relations entre les présentations de la BT, définissant la structure de celle-ci, sont les suivantes:

contient(A,B) :- contientExp(A,B).
contient(A,B) :- contientExp(A,C), contient(C,B).

MesConstructeurs(A,B) :- type(A), typeConstructeurs(B), contient(A,B).

subsume(P,Q) :- subsumeExp(P,Q).
subsume(P,Q) :- subsumeExp(P,R), subsume(R,Q).
subsume(P,Q) :- propriété(P), propriété(Q), contient(P,Q).

présentationSurSorte(A,B) :- typeSurSorte(A), typeConstructeurs(B), contient(A,B).

modèle(A,P) :- modèleExp(A,P).
modèle(A,P) :- modèleExp(A,Q), subsume(Q,P).

## *Discussion*

Malgré le fait que ces définitions ont un certain goût "relationnel", les relations étendues auraient pu également être définies en termes de mécanismes d'héritage, dont la formalisation serait plus compliquée.

L'adéquation conceptuelle d'une structure basée sur des unités modulaires et des combinaisons de ces unités, est discutée au §3.3.2 et surtout dans [Burstall, Goguen 77]. Son adéquation opérationnelle sera discutée dans les chapitres suivants, quand nous aborderons la question des réponses de la BT aux problèmes qui lui sont posés. Nous voudrions uniquement discuter ici les facilités offertes par la structure pour la consultation directe de la BT par l'utilisateur. Cet

---

[1] La définition des prédicats au moyen des clauses de Horn permet de considérer une sémantique initiale [réf]; nous nous intéressons au plus petit prédicat satisfaisant les relations données.

[2] Ceci ne veut pas dire que toutes les propriétés sont des spécifications; en effet, on peut avoir des propriétés telles que la commutativité, déclarée comme: propriété(comm).

aspect possède une grande importance dans un système voué à la construction *assitée* de programmes.

Pour faciliter la consultation directe de la BT par l'utilisateur, il convient de mettre en œuvre un ***outil de navigation***, avec les caractéristiques suivantes:

- Accès aux présentations par leur nom. La présentation concernée est alors montrée à l'utilisateur.

- Accès aux présentations *en utilisant la structure*: à partir d'une présentation montrée à l'utilisateur, il serait possible d'accéder aux présentations contenues par celle-ci, ou qui contiennent celle-ci, ou subsumées par elle, etc.

- Accès aux sortes et opérateurs "visibles" à partir d'une présentation (sortes et opérateurs définis dans les présentations contenues par la présentation considérée).

Cet outil de navigation a été partiellement réalisé en prolog (cf. §4 et annexe 2).

*It is worth observing ... that logic programming shows how to extend any query language to a programming language [Kowalski 76].*

# 4.- La recherche des Informations dans la BT

Dans ce chapitre nous nous intéressons à l'obtention d'informations utiles à partir du contenu de la BT, celui-ci ayant été décrit dans le chapitre précédent. Nous allons donner en fait la description d'une interface d'utilisation de la BT. Cette interface présente , pour ainsi dire, une vue externe de la BT dans le sens où l'on ne s'intéresse qu'aux transactions qu'on peut effectuer avec elle. A la fin de ce chapitre nous discuterons la question des mécanismes spécifiques de recherche des informations demandées.

## 4.1.- Introduction

L'interface de la BT doit assurer les fonctions de description puis d'obtention d'informations qui ont été décrites aux §3.2.1 et §3.3; nous rappelons ici les plus importantes:

- Consultation directe de la BT par l'utilisateur
- Recherche de théorèmes nécessaires à la synthèse de programmes.

L'importance de la première fonction dans un système de synthèse *assistée*, qui accorde un rôle central à l'utilisateur, n'admet pas le moindre doute. La seconde s'inscrit dans le cadre de notre démarche pour la synthèse de programmes, décrite au §2.
Les différentes fonctions à remplir par la BT seront traitées de façon uniforme, à l'aide d'un *langage d'interrogation* de la BT, permettant de décrire les informations voulues.

## 4.2.- Le langage d'interrogation

Le langage d'interrogation de la BT (LIBT) a été caractérisé comme "déclaratif" au §3.3, par opposition aux langages "procéduraux", où l'on donne la façon d'obtenir les informations voulues. Le LIBT doit donc permettre de *caractériser* les connaissances, c'est-à-dire, à donner des indices plus généraux ou abstraits que les connaissances elles-mêmes. C'est de la confrontation de cette caractérisation avec les connaissances concrètes de la BT que sont engendrées les réponses apportées par la BT. Par exemple, on pourra demander un théorème contenant les opérateurs '+' et '*' dans le type des entiers naturels; un tel théorème peut exister ou non dans la BT, suivant son contenu concret à un moment donné, ou bien il peut y en avoir plusieurs, comme par exemple:

$$a * succ(b) = a + (a * b)$$
$$a * (b + c) = (a * b) + (a * c)$$

*Une approche hybride*

Notre approche consiste donc à poser des questions dans un langage "logique" portant sur des entités qui ont été définies dans un formalisme algébrique. Nous trouvons une telle approche hybride dans [Brachman et al. 83], [Castelfranchi et al. 88], où elle est proposée, dans le cadre des systèmes pour la représentation de connaissances, comme un moyen de combiner les avantages des descriptions structurées algébriques, pour construire des objets complexes, et l'expressivité des langages logiques, pour poser des questions sur ces objets.

*Interrogation en clauses de Horn*

Une proposition très simple de langage, qui permet pourtant d'exprimer une classe très large de demandes de connaissances, consiste à considérer des conjonctions de prédicats, c'est-à-dire, des demandes de la forme:

$$P_1(x_1,\ldots,x_n) \wedge P_2(x_1,\ldots,x_n) \wedge \ldots \wedge P_m(x_1,\ldots,x_n).$$

où $P_1,\ldots,P_m$ sont des prédicats connus du système, et $x_1,\ldots,x_n$ sont des variables libres. L'intérêt de la conjonction est qu'elle permet *de combiner* des critères de sélection, chacun exprimé par un prédicat.

Dans ce cadre-là, la définition détaillée du langage revient à donner une liste de *prédicats disponibles*, que le système est effectivement capable de prendre en compte. Les prédicats disponibles seront présentés au §4.3.

*Sémantique des requêtes*

La solution d'une requête de la forme décrite est constituée de tous les tuples $(e_1,\ldots,e_n)$ formés d'éléments $e_1,\ldots,e_n$ existant dans la BT (présentations, sortes, axiomes,...), qui satisfont chacun des prédicats. Ceci suppose que pour chaque prédicat P du LIBT il existe une procédure de décision pour établir, étant donnée une (instance de) BT, si $P(e_1,\ldots,e_n)$ est vrai ou faux.

*Utilisation des clauses de Horn*

La forme considérée des requêtes est un sous-ensemble des clauses de Horn. En fait, les requêtes sont des buts dans un langage de programmation logique utilisant des clauses de Horn [Kowalski 76].

L'utilisation des clauses de Horn offre en plus beaucoup de possibilités intéressantes pour l'exploitation de la BT. En particulier, il devient possible pour l'utilisateur de définir ses propres prédicats à l'aide des clauses de Horn [Kowalski 76]. Ces définitions sont de la forme $T_1 \wedge \ldots \wedge T_n \Rightarrow H$ où H, $T_1$, ..., $T_n$ sont des littéraux. La valeur définitionnelle de cette expression est donnée par une sémantique initiale[1]; leur sémantique opérationelle est donnée par un mécanisme

[1] L'initialité (§3.3.1.2) des définitions en clauses de Horn vient du fait qu'elle ne donnent que la partie "seulement si" des équivalences; on s'intéresse au plus petit prédicat satisfaisant les clauses données. Cette initialité reflète l'hypothèse "du monde fermé" [Makowski 87], qui est également courante en Bases de Données [Nicolas, Gallaire 78]; cette uniformité permet de mélanger harmonieusement données et programmes en prolog [Kowalski 78].

de preuve des buts par résolution. Dans $T_1$, ..., $T_n$ peut apparaître un quelconque des prédicats connus de la BT (fournis par le système ou bien "user-defined"). Il est ainsi possible d'enrichir le LIBT. Nous en donnerons des exemples une fois que les prédicat connus seront présentés.

Dans la suite de ce chapitre nous allons utiliser la notation du langage Prolog pour les clauses de Horn, afin de préserver la compatibilité avec la maquette de BT que nous avons réalisé en Prolog. Les requêtes du LIBT seront de la forme $P_1, P_2, \ldots, P_m.$, pour des littéraux $P_1, P_2, \ldots, P_m$.

## 4.3.- La consultation directe de la BT

Nous présenterons ici les possibilités offertes par le LIBT à l'utilisateur pour consulter directement le contenu de la BT. Puisque nous nous sommes placés dans le cadre de l'interrogation de la BT en clauses de Horn, tout ce qu'il reste à faire est de présenter les prédicats connus disponibles. Ces prédicats seront groupés en deux classes:

1) Prédicats issus directement des déclarations faites dans les présentations de la BT;

2) Prédicats définis à l'aide des clauses de Horn..

### *1) Prédicats connus issus des déclarations*

Chacun des prédicats ci-dessous fait référence à une déclaration effectuée dans une présentation particulière. Ils sont définis en donnant un schéma de la présentation où la déclaration concernée apparaît.

typeConstructeurs(P) ssi (**type** (Constructeurs) P ...) ∈ BT;

typeDéfConstructive(P) ssi (**type** (DéfConstructive P ...) ∈ BT;

typeSurSorte(P) ssi (**type** (SurSorte) P ...) ∈ BT;

typeThéorèmes(P) ssi (**type** (Théorèmes) P ...) ∈ BT;

propriété(P) ssi (**propriété** P ...) ∈ BT;

propriétéSpécification(P) ssi (**propriété** (Spécification) P ...) ∈ BT.

contientExp(A,B) ssi (**type/propriété** A...**contient** B...) ∈ BT;

propriétéExigée(A,P,R') ssi (**type** A...**propriété exigée** P[R]...) ∈ BT;

où R est le renommage (éventuellement vide) appliqué à P pour identifier les sortes et les opérateurs de P à ceux du type A; R' est R exprimé sous la forme d'une liste de couples $[[x_1, y_1], \ldots, [x_m, y_m]]$, où $x_i$ sont les anciennes valeurs et $y_i$ les nouvelles.

subsumeExp(P,Q,R') ssi (**propriété** P...**subsume** Q[R]...) ∈ BT;

modèleExp(A,P,R') ssi (**type** A...**satisfait** Q[R]...) ∈ BT;

sorteExp(S,P) ssi (**type/propriété** P...**sorte** S...) ∈ BT;

sousSorteExp(S1,S2,P) ssi (**type** P...**sortes** S2>S1...) ∈ BT;

opérateurExp(O,profil([$s_1$,...,$s_n$],s),P)
ssi (**type/propriété** P... **opérateurs** ... O:$s_1$,...,$s_n$ → s...) ∈ BT;

axiomeExp(G=D,P) ssi (**type/propriété** P... **axiomes** G=D...) ∈ BT.

## *2) Prédicats définis à l'aide des clauses de Horn*

Quelques-unes des définitions ci-dessous ont déjà été données au §3.3.2.

```
présentation(A) :- type(A).
présentation(A) :- propriété(A).

type(A) :- typeConstructeurs(A).
type(A) :- typeDéfConstructive(A).
type(A) :- typeSurSorte(A).
type(A) :- typeThéorèmes(A).

propriété(P) :- propriétéSpécification(P).

contient(A,B) :- contientExp(A,B).
contient(A,B) :- contientExp(A,C), contient(C,B).

subsume(P,Q,Renom) :- subsumeExp(P,Q,Renom).
subsume(P,Q,Rcomp) :-
      subsumeExp(P,X,Rexp),
      subsume(X,Q,R),
      composeRenom(Rexp,R,Rcomp).
subsume(P,Q) :- propriété(P), propriété(Q), contient(P,Q).

modèle(A,P) :- modèleExp(A,P).
modèle(A,P) :- modèleExp(A,Q), subsume(Q,P).

displayPres(P). affiche sur l'écran la présentation de nom P.

sortePres(S,P) :- sorteExp(S,P).
sortePres(S,P) :- contient(P,Q), sortePres(S,Q).
```

Le prédicat sortePres donne l'ensemble de sortes accessibles à partir d'une présentation donnée, c'est-à-dire, les sortes définies dans la présentation considérée plus les sortes des présentations qu'elle contient (récursivement), ce qui correspond aux sortes accessibles à partir de la présentation considérée. Des définitions similaires sont données pour les opérateurs et les axiomes accessibles:

```
opérateurPres(F,P) :- opérateurExp(F,Profil,P).
opérateurPres(F,P) :- opérateurExp(F,Profil,Q), contient(P,Q).

axiomePres(A,P) :- axiomeExp(A,P).
axiomePres(A,P) :- contient(P,Q), axiomePres(A,Q).
```

D'autres prédicats, spécifiques pour la recherche de théorèmes, seront présentés au §4.4.

L'ensemble de ces prédicats remplit les fonctions de *l'outil de navigation* défini au §3.3.2.7. Pour permettre de mieux saisir intuitivement les possibilités de consultation de la BT offertes par cet outil, nous présentons ci-dessous le déroulement d'une session (annotée), en utilisant la BT donnée en annexe.[1] Le prédicat displayPres(P) permet d'afficher sur l'écran la présentation de nom P.

```
| ?- presentation(X).
 Quelles sont les présentations disponibles dans la BT ?

X = bool0 ;
X = natZeroSuc ;
X = listeNilCons ;
...  ...
X = assoc ;
...  ...

| ?- propriete(P).
 Quelles sont les propriétés dans la BT?

P = comm ;
P = assoc ;
P = monoide ;
P = commonoide :
...  ...

| ?- displayPres(commonoide).
 Affichage de la présentation "commonoide"

propriete commonoide
contient comm1, monoide
fpropriete


| ?- contientExp(commonoide,X).
 Quelles présentations sont explicitement contenues dans commonoide?
 (celles qui apparaissent dans la présentation)

X = comm1 ;
X = monoide ;


| ?- contient(commonoide,X).
```

[1] Nous avons effectivement réalisé cette session avec la maquette de BT en prolog sur une station sun, mais le "script" a été modifié avec l'éditeur pour en faciliter la lecture.

*Quelles présentations sont récursivement contenues dans commonoide? (présentations explicitement contenues plus présentations contenues par celles-ci, etc.)*

```
X = comm1 ;
X = monoide ;
X = assoc ;
... ...
```

```
| ?- displayPres(monoide).
```

*Affichage de 'monoide'; on voit qu'elle contient 'assoc'*

```
propriete monoide
contient assoc
operateurs
     z : [] -> t

axiomes
     X+z=X
     z+X=X
fpropriete
```

```
| ?- displayPres(assoc).
```

*...affichage d'assoc, etc. Il est ainsi possible de "naviguer" par la structure de la BT*

```
propriete assoc
sortes  t
operateurs
     + : [t,t] -> t
axiomes
     X+(Y+Z)=X+Y+Z
fpropriete
```

```
| ?- typeConstructeurs(X).
```

*Quelles sont les présentations de constructeurs?*

```
X = bool0 ;
X = natZeroSuc ;
X = listeNilCons ;
... ...
```

```
| ?- displayPres(bool0).

type bool0
sortes  bool
operateurs
     false : [] -> bool
     true : [] -> bool
ftype
```

```
| ?- contient(X,bool0).
```

*Présentations contenant la présentation bool0. Ce sont les présentations qui utilisent la sorte bool et/ou ses constructeurs.*

```
X = bool1 ;
... ...
```

```
| ?- displayPres(bool1).
```

```
type bool1
contient bool0
operateurs
     and : [bool,bool] -> bool
     iff : [bool,bool] -> bool
     not : [bool] -> bool
     or : [bool,bool] -> bool
     then : [bool,bool] -> bool
axiomes
     (not false)=true
     (not true)=false
     (X and false)=false
     ...  ...
satisfait
     commonoide[+ -> and, z -> true]
     commonoide[+ -> or, z -> false]
     idempotence[* -> and]
     idempotence[* -> or]
ftype
```

```
| ?- sortePres(bool,P).
```

*Présentations où la sorte bool est accessible. C'est une autre façon de poser la question précédente*

```
P = bool1 ;
P = booltheo ;
... ...
```

```
| ?- operateurExp(Op,Profil,bool1).
```

*Opérateurs explicitement définis dans bool1*

```
Op = not
Profil = profil([bool],bool) ;
Op = and
Profil = profil([bool,bool],bool) ;
Op = or
Profil = profil([bool,bool],bool) ;
Op = then
Profil = profil([bool,bool],bool) ;
Op = iff
Profil = profil([bool,bool],bool) ;
no
```

```
| ?- operateurPres(Op,bool1).
```

*Opérateurs accessibles dans bool1. Notez que les opérateurs true et false ont été hérités.*

```
Op = not ;
Op = and ;
Op = or ;
Op = then ;
Op = iff ;
Op = true ;
Op = false ;
no
```

```
| ?- operateurExp(+,Profil,Pres),type(Pres).
```

*Profil de l'opérateur "+" et présentations (de type) où il est défini.*

```
Profil = profil([nat,nat],nat)
Pres = natTous ;
```

```
Profil = profil([natPlus,natPlus],natPlus)
Pres = natPlus ;
no

| ?- operateurExp(Op,profil(Args,bool),Pres).
 Opérateurs à résultat booléen (sorte bool).
Op = =<
Args = [t,t]
Pres = ordrePartiel ;
...  ...
Op = not
Args = [bool]
Pres = bool1 ;
... ...
Op = vide
Args = [listeCons]
Pres = opsListeCons ;
... ...
```

## 4.4.- Demandes de théorèmes

Ici nous décrivons les prédicats disponibles dans le LIBT servant à caractériser les théorèmes nécessaires à la synthèse. Les stratégies pour formuler, à partir de ces prédicats, les caractérisations de théorèmes nécessaires dans une situation concrète, seront présentées dans le chapitre suivant (§5). Bien sûr, à ce moment-là il faudra montrer que les constructions ici offertes sont suffisantes pour former des caractérisations adéquates.[1]

Les requêtes de théorèmes sont, comme toutes les autres requêtes, des buts Prolog. Les prédicats connus utilisés pour la recherche de théorèmes sont les représentants des différents critères de sélection de théorèmes, et la conjonction de ces prédicats représente la combinaison de ces critères de sélection.

### 4.4.1.- Les critères de sélection de théorèmes et leurs prédicats associés

Nous décrirons maintenant les critères de sélection de théorèmes et les prédicats connus utilisés pour les mettre en œuvre.

*Critère 1.- Unification avec un schéma de théorème équationnel*

Un schéma de théorème est un formule G=D où G et D sont des termes, pouvant contenir des variables à instancier pour des termes. A partir d'un tel schéma, sont cherchés les théorèmes s'unifiant avec ce schéma.

Un prédicat permettant d'utiliser cette forme de requête est axiomeExp(F,P), qui indique que l'axiome F est explicitement déclaré dans la présentation P. Exemple:

```
| ?- axiomeExp(A+B=C,P).

P = arithSuc
A = X
```

[1] Rappel.- Les caractérisations de théorèmes sont élaborées par le module de dérivation d'hypothèses.

```
B = 0
C = X ;

P = arithSuc
A = X
B = succ(Y)
C = succ(X+Y) ;
...  ...
```

Vis-à-vis du processus d'unification des théorèmes et des schémas, deux cas extrèmes peuvent se présenter:

• Le théorème est plus instancié que le schéma, comme dans l'exemple précédent.

• Le schéma est initialement plus instancié qu'un des théorèmes. Exemple:

```
| ?- axiomeExp((concat(t,u) and true) = concat(t,u),P).

P = bool1
```

Dans la pratique, des instanciations dans les deux sens peuvent se produire. Exemple:

```
| ?- axiomeExp(1+B=C,P).

B = 0
C = 1
P = arithSuc ;
...
```

Il est important de réaliser que, du point de vue de l'unification des schémas de théorème dans les requêtes avec les théorèmes de la BT, ces derniers sont regardés tout simplement comme des termes, et le signe "=" des équations est pris en tant que leur opérateur principal. Il est donc valable de faire des demandes axiomeExp(A,P) où A est une variable.

## *Critère 2.- Unification des sous-termes*

Un terme X s'unifie avec un sous-terme Y d'une formule T, noté T[Y] , s'il existe une substitution θ telle que Xθ=Yθ, où Y n'est pas une variable[1]. Le prédicat connu associé est sousTerme(X,T). Ce prédicat peut s'utiliser en donnant X et en cherchant un théorème T dans la BT. Exemple:

```
| ?- type(P),axiomeExp(T,P),sousTerme(Y+succ(Z),T).

P = arithSuc
T = X+succ(Y)=succ(X+Y)
```

[1] La condition que Y ne soit pas une variable est imposée pour éviter de récupérer n'importe quel axiome contenant des variables, qui serait une solution triviale.

```
Y = X
Z = Y
```

La caractérisation d'un théorème en donnant un de ses sous-termes est plus flexible que celle obtenue par simple unification avec un schéma du premier ordre. En effet, dans le premier cas il n'est pas nécessaire de préciser la position du sous-terme dans le théorème.

## *Critère 3.- Opérateurs intervenant dans les théorèmes*

Le prédicat operateurDansTerme(Op,T) est utilisé pour tester l'occurrence d'un certain opérateur Op dans un terme T. Le terme T sera normalement un théorème ou un sous-terme d'un théorème; les théorèmes sont ainsi caractérisés par les opérateurs y intervenant. Exemple:

```
| ?- axiomeExp(A,P), operateurDansTerme(+,A), operateurDansTerme(*,A).

A = X*succ(Y)=X+X*Y
P = arithSuc ;
```

Un autre prédicat servant à caractériser les formules par leurs opérateurs est operateurDeTerme(Op,Terme), qui indique que Terme est de la forme Op(...t...), c'est-à-dire, qu'il est l'opérateur le plus externe. C'est donc un prédicat plus fort que operateurDansTerme:

operateurDeTerme(Op,T) ⇒ operateurDansTerme(Op,T)

La caractérisation des théorèmes par le prédicat operateurDansTerme est plus flexible que celle du point précédent (par les sous-termes), car dans le premier l'ordre de composition des opérateurs n'est plus considéré. Le prédicat operateurDeTerme, par contre, établit en fait une espèce de pattern (du second ordre), et il est en dernière instance une facilité syntaxique vu le fait qu'en Prolog on ne peut pas écrire des expressions comme Op(T).[1]

## *Critère 4.- Types de données intervenant dans le théorème*

Nous dirons qu'un type (une sorte, en réalité) intervient dans une formule quand il existe dans celle-ci un sous-terme de ce type. Le prédicat qui permet de tester si un type S intervient dans une formule F est termeDuType(S,F). Ce prédicat peut être calculé à partir des profils des opérateurs. Nous pouvons définir un prédicat theoremeDuType comme:

```
theoremeDuType(S,T) :- axiomeExp(T,P), type(P), termeDuType(S,T).
```

Exemple:

```
| ?- theoremeDuType(bool,A).
A = (X=Y imp Y=X)=vrai ;
A = (X=Y et Y=Z imp X=Z)=vrai ;
...  ...
| ?- theoremeDuType(natSuc,A),theoremeDuType(listeCons,A).
```

[1] Une expression de la forme T = Op(X) peut s'écrire en Prolog de la façon suivante: T=..[Op,X].

```
A = (#nil)=0 ;
A = (#X<+L)=succ((#L))
```

## *Critère 5.- Caractère du théorème*

Les théorèmes contenus dans la BT sont supposés être utiles; c'est pour cela qu'ils s'y trouvent. Nous appelons "caractère du théorème" le critère de pertinence qui justifie la présence d'un théorème dans la BT (§3.3.1.9).

Les classes de théorèmes utiles (et leurs critères de pertinence) sont:

• Définitions constructives d'opérateurs, qui sont repérées par le prédicat `axiomeDefConstruc(Axiome, Operateur, Presentation)`, dont la définition (exécutable) est:

```
axiomeDefConstruc(G=D, Operateur, Presentation) :-
     typeDefConstructive(Presentation),
     axiomeExp(G=D,Presentation),
     operateurDeTerme(Operateur,G).
```

Exemple:

```
| ?- axiomeDefConstruc(Ax,+,Pres).

Ax = X+0=X
Pres = arithSuc ;

Ax = X+succ(Y)=succ(X+Y)
Pres = arithSuc ;
```

• Spécifications déclaratives des opérateurs, répérées par le prédicat axiomeSpecif(Ax, Op, Pres), qui indique que l'axiome Ax de la présentation Pres est une spécification déclarative de l'opérateur Op (cf.3.3.2.4). Il peut être calculé de la façon suivante:

```
axiomeSpecif(Ax,Op,Pres) :-
     proprieteSpecification(Pres),
     operateurExp(Op,Profil,Pres),
     axiomeExp(Ax,Pres),
     operateurDansTerme(Op,Ax).
```

Cette expression suppose que la BT est bien formée, c'est-à-dire, que les spécification sont faites dans les présentations de "proprieteSpecification", et que l'opérateur Op y est déclaré effectivement, etc.

• Théorèmes de simplification. Ils sont identifiés par le prédicat theoremeSimplif(Theo, Pres). Ce prédicat, comme les autres, peut être donné explicitement pour chaque théorème[1], mais il

---

[1] Voir §4.5.

peut également être calculé dans certains cas. Par exemple:

```
theoremeSimplif(G=D,Pres) :-
     axiomeExp(G=D,Pres),
     sousTerme(D,G).
```

Cette expression exprime l'ordre de simplification par "plongement" d'un côté de l'équation dans l'autre, c'est-à-dire, G = D[G] (G est un sous-terme de D).

• Théorèmes de lien. Ce sont les théorèmes dont leur présence dans la BT se justifie par le fait qu'ils servent à mettre en rapport des opérateurs (§3.3.1.9.1). Dans ce cas il est beaucoup plus difficile d'en donner une définition calculatoire, si ce n'est en indiquant les opérateurs qui doivent intervenir dans le théorème.[1]

## 4.5.- Les mécanismes de recherche des informations

Nous entendons par *mécanismes de recherche des informations* la façon, décrite en termes procéduraux, de trouver effectivement les informations existantes dans la BT, qui satisfont une requête écrite dans le LIBT.

Nous soulignons d'abord le fait que, jusqu'à présent, nous n'avons effectué aucun choix concernant cette question. En effet, même si pour le LIBT le choix des clauses de Horn et la notation de Prolog ont semblé nous convenir, rien ne nous oblige à retenir aussi sa sémantique opérationnelle, et il est parfaitement envisageable de mettre en œuvre un tout autre mécanisme de recherche des informations, si l'efficacité y gagne quelque chose.

Dans la suite de ce paragraphe nous allons développer ces deux volets, à savoir: le mécanisme de recherche des informations propre à Prolog, et d'autres mécanismes possibles.

### 4.5.1.- L'utilisation directe de Prolog pour répondre au requêtes

L'utilisation du langage prolog en tant que langage de représentation de données et/ou langage d'interrogation des bases de données a été très étudiée, à partir du papier de [Kowalski 78]. Nous distingons deux types d'utilisation de Prolog dans le cadre des Bases de Données: 1) Prolog est utilisé pour représenter les données, pour exprimer les requêtes et pour chercher les informations demandées; 2) Prolog est le langage hôte d'une BD conventionnelle. Pour des raisons de performance, la plupart des réalisations pratiques sont basées sur ce couplage BD-prolog. Nous allons discuter d'abord la première alternative.

#### 4.5.1.1.- L'environnement nécessaire

L'utilisation directe[2] de Prolog en tant que mécanisme de recherche des informations dans la BT suppose que les conditions suivantes ont été préalablement remplies:

---

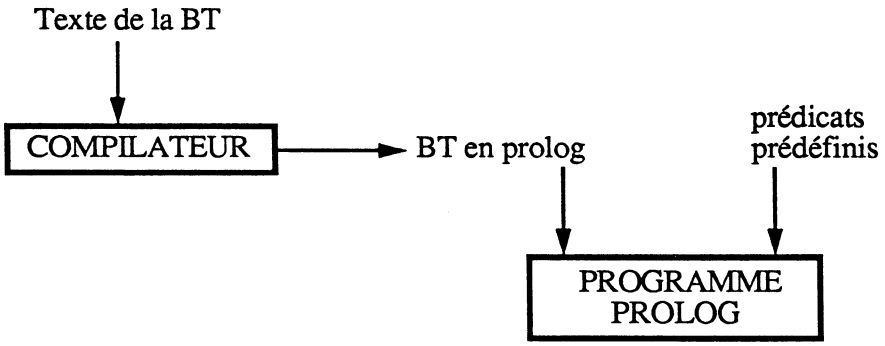
[1] D'autres critères à remplir par un "bon" théorème de lien ont été donnés dans §3.3.1.9.1.

[2] Ce n'est donc pas l'utilisation de prolog en tant que langage de programmation, avec lequel on peut mettre en œuvre un mécanisme quelconque de recherche des informations.

1) Les informations de la BT sont disponibles sous une forme clausale, c'est-à-dire, elles font partie du *programme* (dans le sens [Pereira 88]);

2) Il existe une définition exécutable en Prolog pour les prédicats connus.

Ces deux aspects sont fortement liés, à tel point que dans beaucoup de cas la solution de l'un entraîne la solution de l'autre. Le point 1 implique une traduction des informations de la BT vers une forme clausale, qui serait la représentation interne des présentations de la BT. Cette traduction (ou "compilation") serait assurée par un outil, le "compilateur" de la base, lequel, recevant en entrée un texte [1] de BT, générerait le code Prolog correspondant, et l'intégrerait au code des prédicats connus, pour le rendre accessible à ces derniers. Cette architecture est représentée dans la figure suivante:




Malgré le fait que la mise en œuvre du compilateur ne présente aucune difficulté particulière, nous n'avons pas eu le temps de le réaliser. En fait, pour nos expériences avec la maquette en Prolog, nous avons effectué "à la main" la génération de code Prolog à partir du texte des présentations de la BT. Les deux versions de la BT (texte, clauses Prolog) sont données en annexe.[2]

La traduction de la BT se fait en termes des prédicats connus suivants, dont la définition a été donnée au §4.3:

typeConstructeurs(A)
typeDéfConstructive(A)
typeSurSorte(A)
typeThéorèmes(A)
propriété(P)
propriétéSpécification(P)
contientExp(A,B)
propriétéExigée(A,P,R)
subsumeExp(P,Q,R')
modèleExp(A,P,R')
sorteExp(S,P)
sousSorteExp(S1,S2,P)
opérateurExp(O,profil([$s_1$,...,$s_n$],s),P)
axiomeExp(G=D,P)

Chacun de ces prédicats est associé de façon bijective à une déclaration précise des présentations (voir définitions de ces prédicats). La traduction est donc triviale.

La BT qui en résulte est un ensemble de clauses positives. Ainsi, par exemple, la présentation

---

[1] ou une autre représentation équivalente

[2] En réalité, nous avons procédé dans le sens inverse, puisque le texte de la BT présentée en annexe a été engendré à partir de la version prolog de la BT, en utilisant le prédicat prédéfini displayPres.

"bool1" et sa traduction en Prolog sont montrées ci-dessous:

*texte:*

```
type (Definition constructive) bool1
contient bool0
operateurs
     and : [bool,bool] -> bool
     iff : [bool,bool] -> bool
     not : [bool] -> bool
     or : [bool,bool] -> bool
     then : [bool,bool] -> bool
axiomes
     (not false)=true
     (not true)=false
     (X and false)=false
     (X and true)=X
     (X0 iff X1)=((X0 then X1) and (X1 then X0))
     (X or false)=X
     (X or true)=true
     (X then Y)=(not X or Y)
satisfait
     commonoide[+ -> and, z -> true]
     commonoide[+ -> or, z -> false]
     idempotence[* -> and]
     idempotence[* -> or]
ftype
```

*traduction Prolog:*

```
typeDefConstructive(bool1).
contientExp(bool1,bool0).
operateurExp((not),profil([bool],bool),bool1).
operateurExp((and),profil([bool,bool],bool),bool1).
operateurExp((or),profil([bool,bool],bool),bool1).
operateurExp((then),profil([bool,bool],bool),bool1).
operateurExp((iff),profil([bool,bool],bool),bool1).
axiomeExp((not true) = false,bool1).
axiomeExp((not false) = true,bool1).
axiomeExp((_1 and true) = _1,bool1).
axiomeExp((_1 and false) = false,bool1).
axiomeExp((_1 or true) = true,bool1).
axiomeExp((_1 or false) = _1,bool1).
axiomeExp((_1 then _2) = (not _1 or _2),bool1).
axiomeExp((_1 iff _2) = ((_1 then _2) and (_2 then _1)),bool1).
modeleExp(bool1,commonoide,[[t,bool],[&,and],[z,true]]).
modeleExp(bool1,commonoide,[[t,bool],[&,or],[z,false]]).
modeleExp(bool1,idempotence,[[t,bool],[&,and]]).
modeleExp(bool1,idempotence,[[t,bool],[&,or]]).
```

En ce qui concerne la mise en œuvre en Prolog des prédicats connus, quelques-uns d'entre eux ont été déjà trivialement réalisés par la traduction de la BT (liste de prédicats ci-dessus). Quelques autres ont été déjà définis en Prolog (type, contient, subsume...). Il ne reste que quelques prédicats de bas niveau dont la programmation fait appel aux acrobaties permises dans la version de Prolog utilisée; leur définition est pour cela donnée en annexe (operateurDeTerme, sousTerme...).

## *Compilation non-minimale de la BT*

La traduction de la BT en Prolog (BT-Prolog), telle qu'elle vient d'être décrite, correspond à la plus petite BT-Prolog, en terme des prédicats connus, contenant toute l'information des présentations de la BT. Il en est ainsi parce c'est une traduction littérale de chacune des déclarations faites dans les présentations de la BT.

L'utilisation d'une telle BT-Prolog suppose une exécution "descendante" des prédicats définis en Prolog.[1] Ceci correspond au mécanisme habituel d'exécution de Prolog. Néanmoins, il est également envisageable d'exécuter ces définitions de façon "ascendante" [Kowalski 78], en les utilisant de façon générative pour engendrer des nouvelles clauses qui vont enrichir la BT-Prolog. Si ceci est fait, les définitions en question n'auront plus à être considérées lors de l'exécution des prédicats qu'elles définissent, lesquels seront consultés directement dans une liste de clauses instanciées de la BT-Prolog enrichie.[2]

Par exemple, soient les clauses suivantes dans la BT-Prolog:

```
contientExp(bool1,bool0).
contientExp(opListe,bool1).
contientExp(egalite,bool1).
```

Soit ensuite la définition du prédicat `contient`:

```
contient(P,Q) :- contientExp(P,Q).
contient(P,Q) :- contientExp(P,R), contient(R,Q).
```

L'exécution "ascendante" de cette définition se fait à partir du corps des clauses vers la tête, de la façon suivante:

• Pour chaque clause contientExp(P,Q) on engendre une clause contient(P,Q);
• Chaque fois qu'on trouve une clause contientExp(P,R) et une clause contient(R,Q), on engendre une clause contient(P,Q), si celle-ci n'existe pas déjà.

Ce mécanisme produit les clauses suivantes:

```
contient(bool1,bool0).
contient(opListe,bool1).
contient(egalite,bool1).
contient(opListe,bool0).
contient(egalite,bool0).
```
[3]

---

[1] par exemple, les prédicats contient, subsume, etc.

[2] On peut alors dire que la définition de ces prédicats-là est donnée en extension.

[3] Cet ensemble de clauses peut être engendré automatiquement en prolog avec le programme :

```
contient2(P,Q) :- contientExp(P,Q).
contient2(P,Q) :- contientExp(P,R), contient2(R,Q).
```

et le but:

```
:- contient2(P,Q), assert((contient(P,Q))),fail.
```

Le choix entre une exécution ascendante (générative, en chaînage avant) ou descendante (en chainage arrière) des prédicats connus implique un compromis entre le temps d'exécution et l'espace mémoire occupé. En effet, dans l'approche générative les instances de base des prédicats connus sont engendrés lors de la compilation de la BT, et leur consultation ultérieure devient extrèmenent rapide. L'exécution ordinaire (descendante) des prédicats, par contre, est plus lente, mais aussi plus simple et moins consommatrice de place mémoire. Dans la pratique, des compromis intermédiaires sont envisageables, en engendrant les instances de quelques prédicats lors de la compilation de la BT, et en exécutant les autres, lors de la consultation de celle-ci. Dans l'élaboration de notre maquette de BT nous avons utilisé, par simplicité, l'exécution descendante, car notre BT est pour le moment réduite, mais pour une réalisation en vraie grandeur, il pourrait falloir changer le sens d'exécution de quelques prédicats.

### 4.5.1.2.- L'exécution des requêtes en Prolog

Nous allons ici discuter les mécanismes de recherche des informations par exécution directe des requêtes en Prolog. Par "directe" nous voulons dire que nous utilisons vraiment le mécanisme de recherche de solutions de Prolog comme mécanisme de recherche des informations, sans passer par des représentations auxiliaires. Une utilisation "indirecte" de Prolog serait de s'en servir en tant que langage de programmation pour mettre en œuvre un mécanisme quelconque de recherche des informations.

Nous commencerons par une brève description du mécanisme de recherche de solutions du langage Prolog.

#### *Le mécanisme d'exécution de Prolog*

Une description formelle de la sémantique opérationnelle de Prolog se trouve dans [Komorowski 82]. Il nous suffira la description informelle donnée dans [Pereira 88]:

Pour exécuter un but (une requête est une suite de buts), le système cherche à partir du début du "programme" (prédicats définis en Prolog plus clauses de la BT) la première clause dont la tête s'unifie avec le but. De cette unification résulte l'instance la plus générale commune aux deux termes. Si cela se produit, la clause (instanciée) est alors "activée" en exécutant, à son tour, de gauche à droite, chacun des sous-buts du corps de la clause. Si le système ne réussit pas à trouver une clause dont la tête s'unifie avec le but, alors il revient en arrière (backtracking), c'est-à-dire, il rejette la dernière clause activée, en défaisant toute substitution produite par l'unification. Puis, il reprend le but qui a activé la clause rejetée, et essaie de trouver après celle-ci une autre clause s'unifiant avec le but.

Du fait que la BT-Prolog n'est composée que de clauses positives (faits), il en résulte que leur activation n'engendre pas de sous-buts. Les sous-buts ne sont engendrés que lors de l'activation d'une clause faisant partie de la définition d'un prédicat (fourni par le système ou bien défini par l'utilisateur).

Dans le cas des prédicats définis en extension dans la BT-Prolog (p.ex. axiomeExp, sorteExp,...) le mécanisme de Prolog est donc réduit à la recherche séquentielle d'un littéral s'unifiant avec le but.

## *Efficacité de l'exécution directe des requêtes en Prolog*

L'exécution des buts en Prolog revient à faire un parcours en profondeur-d'abord d'un arbre ET-OU de recherche de solutions dans lequel la partie ET est le séquencement de sous-buts d'une clause activée (ou de la requête), et la partie OU correspond aux différentes clauses qu'il est possible d'activer. Le mécanisme d'exécution de Prolog est efficace dans la mesure où cette stratégie de recherche de solution est adaptée à la recherche des informations de la BT à partir des requêtes du LIBT.

D'une façon générale, l'efficacité d'une stratégie de recherche de solutions est mesurée par son aptitude à couper dans l'arbre de recherche, les branches inutiles, réduisant ainsi la taille de celui-ci.

Les moyens pour couper les branches ET et les branches OU présentent les particularités suivantes:

1) Quand une solution est trouvée, les branches OU à gauche ont été déjà explorées; si l'on ne s'intéresse pas aux autres solutions, les branches OU à droite sont abandonnées;

2) Quand un échec se produit dans une branche ET, les branches à gauche ont été déjà explorées, et les branches à droite sont abandonnées.

Pour controler le rapport branches explorées/branches coupées, on peut, pour chaque cas:

1) Ordonner les clauses du programme en cherchant à mettre d'abord les clauses qui doivent être explorées les premières; couper éventuellement les branches alternatives en utilisant le "cut" (!).

2) Chercher à faire échouer le plus tôt possible et au moindre coût possible les branches qui devront échouer en fin de compte. Pour cela, placer d'abord les littéraux les plus "sélectifs".

La performance des requêtes composées de plusieurs littéraux est particulièrement influencée par le second point. Par exemple, soient les requêtes:

**R1)**
```
axiomeExp(A,P),
operateurDansTerme(<+>,A),typeConstructeurs(P).
```
**R2)**
```
typeConstructeurs(P),
operateurDansTerme(<+>,A),axiomeExp(A,P).
```

Les ensembles de solutions des requêtes R1 et R2 sont identiques; par contre le temps de calcul de R2 est sensiblement plus court que celui de R1.[1] La raison en est que dans le premier cas le prédicat operateurDansTerme, qui est assez couteux en temps d'exécution, est appliqué à beaucoup d'axiomes qui ne seront pas retenus pour la solution finale.

---

[1] Avec le système C-prolog sur un Sun et la BT-prolog donnée en annexe, les temps d'exécution ont été de 11,1 et 1,7 secondes, respectivement.

Il en va de même pour les clauses des définitions de prédicats (du système ou user-defined); par exemple, si nous changeons la définition du prédicat axiomeDefConstruc (§4.4.1, *Critère 5*) de la façon montrée ci-dessous, les performances vont se dégrader:

```
axiomeDefConstruc(G=D, Operateur, Presentation) :-
    axiomeExp(G=D,Presentation),
    operateurDeTerme(Operateur,G),
    typeDefConstructive(Presentation).
```

Il est toutefois difficile de savoir à priori quels sont les sous-buts les plus sélectifs, pour les placer d'abord dans le corps de la clause. Cette question est souvent dépendante des arguments instanciés et même du contenu concret de la BT. Par exemple, soient les deux définitions suivantes du prédicat sortePres (sortes accessibles à partir d'une présentation):

**(1)**
```
sortePres(S,P) :- sorteExp(S,P).
sortePres(S,P) :- contient(P,Q), sortePres(S,Q).
```

**(2)**
```
sortePres(S,P) :- sorteExp(S,P).
sortePres(S,P) :- sorteExp(S,Q), contient(P,Q).
```

En utilisant notre BT, la définition (1) est plus efficace[1], car la définition (2) fait le test récursif de contient(P,Q) pour beaucoup de présentations Q qui n'ont aucun rapport avec P. Néanmoins, en augmentant le nombre de liens **contient** dans la BT et en diminuant le nombre de sortes, la définition (2) devient meilleure.

Malheureusement, toutes ces stratégies pour restreindre l'espace de recherche d'une solution s'avèrent, d'une façon générale, très limitées pour le problème de la recherche des informations dans une BT structurée de taille importante. En effet, nous aurions voulu utiliser la structure de la BT pour cloisonner les informations de façon à restreindre la recherche des informations aux parties de la BT concernées par une requête particulière.

Le critère le plus important pour compartimenter la BT est le typage. Or, la consultation de la BT-Prolog par les mécanismes décrits plus haut ne permet pas de restreindre la recherche des informations (par exemple, la recherche de théorèmes) aux types concernés. En effet, l'ordonnancement des sous-buts n'est pas applicable aux clauses de la BT, car elles sont toutes positives (faits), et l'ordonnancement des clauses ne l'est guère plus, car nous ne savons pas à priori quel type de données sera concerné par les requêtes.[2]

Une utilisation limitée de la structure pour améliorer l'efficacité de l'exécution des requêtes est néanmoins possible. En particulier, on peut utiliser l'information de typage en sélectionnant les présentations contenant récursivement celles où le type en question est introduit. Ainsi, par exemple, une définition du prédicat theoremeDuType(Sorte,Theoreme) a été donnée au §4.4.1 (*Critère 4*), de la façon suivante:

---

[1] Environ trois fois plus rapide.

[2] Si c'était le cas, tous les autres types seraient inutiles; le mieux qu'on peut faire à ce sujet est de placer d'abord les informations des types les plus fréquemment utilisés.

```
(1)  theoremeDuType(S,T) :-
       axiomeExp(T,P),
       type(P),
       termeDuType(S,T).
```

Or, en exigeant que la présentation P contienne récursivement la présentation où la sorte S est introduite, nous arrivons à la définition suivante:

```
(2)  theoremeDuType(S,T) :-
       sorteExp(S,Origine),
       contient(P,Origine),
       axiomeExp(T,P),
       type(P),
       termeDuType(S,T).
```

La seule différence entre ces deux définitions est que dans la définition (2) deux lignes ont été ajoutées au début du corps de la clause. Les solutions apportées par les deux définitions sont exactement les mêmes; par contre, la seconde est plus rapide d'un ordre de grandeur.[1]

En fait, la partie que nous avons ajoutée dans la seconde définition agit comme un "filtre": la variable P n'est instanciée que par les présentations effectivement concernées par le type (sorte) S, et ainsi le couteux prédicat termeDuType[2] n'est appliqué qu'aux théorèmes de ces présentations-là.

Nous remarquons que pour les deux définitions l'espace des théorèmes de la BT est entièrement et séquentiellement parcouru jusqu'à l'endroit où la solution est trouvée. La vrai différence entre ces deux définitions réside dans le fait que les échecs de la première se produisent soit lors du calcul du prédicat type(P), soit -bien pire- lors du calcul de termeDuType; par contre, les échecs de la seconde définition se produisent presque toujours plus tôt, lors de l'essai d'unification de axiomeExp(A,P) avec les axiomes de la BT, puisque P est déjà instanciée à ce moment-là. Or l'exécution de l'unification est infiniment plus rapide que celle des prédicats qui font échouer la première définition.

## 4.5.2.- D'autres mécanismes de recherche des informations

Pour surmonter les limitations de l'exécution directe des requêtes en Prolog, discutées dans le paragraphe précédent, nous avons envisagé d'autres mécanismes possibles d'exécution des

---

[1] En instanciant S, la première définition nous a donné un temps de réponse moyen (pour tous les types disponibles dans la BT)de 3,25 secondes, tandis que le temps moyen de la seconde a été de 0,06 secondes... un rapport de 50/1 ! En plus, le temps de réponse de la première dépend de l'endroit de la BT où se trouve l'information pour le type concerné, ce qui nous fait penser que pour une BT plus importante, la différence des performances serait encore plus prononcé. Il faut dire au passage que dans la seconde définition le littéral type(P) n'est plus nécessaire; nous ne l'avons gardé que pour rendre plus nette la comparaison; par contre, le prédicat termeDuType est, lui, indispensable pour que les solutions apportées par les deux définitions soient les mêmes (Il peut y avoir des présentations qui contiennent un type pour l'utiliser dans quelques-uns des axiomes y données, mais pas pour tous).

[2] termeDuType fait l'analyse des profils des opérateurs contenus dans le terme donné.

requêtes. La motivation principale pour se pencher sur la question est un souci d'efficacité qui prend de l'importance lorsqu'on envisage la mise en œuvre d'une BT en vrai grandeur.[1]

Nous allons classer les différentes possibilités alternatives en deux catégories:

• Des variations sur l'exécution en Prolog;
• D'autres mécanismes possibles.

### 4.5.2.1.- Des variations sur l'exécution des requêtes en Prolog

Il est possible d'utiliser le langage Prolog pour répondre aux requêtes autrement que pour leur exécution directe (§4.5.1). Entre cette exécution directe et la simple utilisation de Prolog en tant que langage de programmation pour réaliser un mécanisme quelconque, il existe tout un éventail de possibilités, parmi lesquelles nous en discuterons les deux suivantes:

1) Séparation des prédicats pour les différents types;
2) Utilisation de la Base de données interne de C-Prolog [Pereira 88].

*Séparation des prédicats pour les différents types*

Considérons le problème de la recherche des axiomes d'après les types y intervennant. Nous voudrions organiser les informations de façon à ce que, lors de la recherche des théorèmes concernant un certain type, les informations non concernées ne soient pas parcourues. Un moyen pour cloisonner les informations référentes à chaque type est d'utiliser un prédicat différent pour les axiomes de chaque type, au lieu de "`axiomeExp`". Ainsi nous pourrions avoir les prédicats:

- `axiomeBoolExp(A,P)`, pour les théorèmes sur les booléens;
- `axiomeNatExp(A,P)`, pour les entiers naturels;
- `axiomeListeExp(A,P)`, pour les listes, etc.

Ainsi, la recherche de théorèmes d'après le type devient plus efficace. En particulier, le prédicat théorèmeDuType est alors défini de la façon suivante:

```
theoremeDuType(bool,T) :- axiomeBoolExp(T,P).
theoremeDuType(nat,T) :- axiomeNatExp(T,P).
theoremeDuType(liste,T) :- axiomeListeExp(T,P).
...etc.
```

Cette solution naïve présente plusieurs inconvénients:

• C'est à l'utilisateur d'introduire dans la BT l'information de typage des théorèmes, avec le travail additionnel et les erreurs que cela entraîne;

---

[1] Il ne faut pas perdre de vue que notre maquette de BT, tout en étant utile pour faire des expériences, ne permet pas de mettre en évidence tous les problèmes entraînés par une réalisation en vrai grandeur (10 fois plus grande que notre maquette).

• Les axiomes faisant intervenir plusieurs types simultanément devront être introduits plusieurs fois dans la BT.

• L'adjonction d'un nouveau type à la BT nécessite la mise à jour des prédicats concernés par le mécanisme décrit, p. ex. `theoremeDuType`.

Surmonter ces inconvénients n'est pas trivial; il est nécessaire de faire appel aux astuces spécifiques de la programmation en Prolog (p. ex., engendrer et détruire des clauses). Nous préférons ne pas entamer ces questions d'implantation ici.

*Utilisation de la Base de données interne de Prolog*

Il s'agit d'une base de données indexée qui est maintenue séparée du programme. Dans cette base on distingue trois champs: *clé*, *terme* et *référence*. Un triplet (clé,terme,référence) est appelé un *enregistrement*. La clé de tout enregistrement est un atome.[1]; c'est en fait l'index qui sert à classer les informations en sous-classes disjointes. La référence est un identificateur engendré par le système, servant à repérer de façon unique chaque enregistrement. Finalement, le champ *terme* est destiné à l'information elle-même, mise sous la forme d'un terme Prolog.

En supposant qu'une base indexée Prolog a été déjà formée, son accès est assuré par le predicat recorded(<clé>, <terme>, <référence>), où la clé doit forcément être instanciée à un atome. Les enregistrements de la base dont la clé est <clé> sont alors parcourus séquentiellement en cherchant ceux dont le contenu du champ *terme* s'unifie avec <terme>. En cas de réussite, la variable <référence> est instanciée par le système à un identificateur qui répère l'enregistrement. Nous voyons donc que, dans l'ensemble d'enregistrements possédant la clé requise, le mécanisme de recherche redevient l'unification habituelle.

L'intérêt de cette base de Prolog vient du fait qu'elle permet dans une certaine mesure de profiter de la structure pour compartimenter les informations de la BT. Plus particulièrement, il est possible d'indexer les théorèmes de la BT en utilisant comme clé les sortes y intervenant.[2] De cette façon, les théorèmes d'un certain type peuvent être instantanément repérés, sans avoir à parcourir les informations des autres types. Dans le champ <terme> nous proposons de stocker des couples [Axiome,Présentation], qui est l'information contenue dans le prédicat axiomeExp(Axiome, Présentation).

Les modifications nécessaires pour l'exploitation de cette base indexée dans le cadre de notre BT peuvent se faire de façon transparente pour l'utilisateur. La création de la base indexée à partir de la BT-Prolog ne présente aucune difficulté particulière.[3] Il faut ensuite adapter la définition de quelques prédicats, en particulier theoremeDuType, qui devient:

---

[1] La possibilité d'utiliser le foncteur d'une expression complexe n'est pas utilisée dans notre cas.

[2] Voir prédicat theoremeDuType, §4.4.1, Critère 4)

[3] Elle est automatiquement engendrée par le but:

```
:- sorteExp(S,Origine),
   ((P=Origine);contient(P,Origine)),
   axiomeExp(A,P), type(P), A = (G=D),
   (termeDuType(S,G);termeDuType(S,D)),
   recordz(S,[A,P],Ref), fail.
```

```
theoremeDuTypeIndex(SorteCle,Theoreme) :- recorded(SorteCle, [Theoreme,_], _).
```

*Remarques:*

Le prédicat theoremeDuTypeIndex ne fonctionnera correctement que si la clé (sorte) est effectivement instanciée lors de l'appel. Ceci réduit la flexibilité d'utilisation. Ce problème peut s'éliminer en mettant en place un mécanisme trivial d'énumération des sortes, ou bien en faisant appel éventuellement à l'ancienne définition de theoremeDuType, qui s'exécuterait dans la BT-Prolog habituelle.[1]

Il faut remarquer que, du fait que certains théorèmes font intervenir plusieurs sortes, ils vont se trouver répétés à plusieurs endroit de la base indexée.

Nous avons effectué des expériences pour comparer les performances de la recherche des informations dans la base indexée et de la version Prolog pure. Pour la recherche de théorèmes (prédicat theoremeDuType), en instanciant la sorte (premier argument), le temps d'obtention d'une première réponse est dans les deux cas de l'ordre des centièmes de seconde[2]; par contre, l'obtention de toutes les réponses est sensiblement plus rapide pour la BT indexée[3]. Les requêtes faisant intervenir plusieurs types simultanément s'exécutent 50 fois plus rapidement avec la BT indexée[4]. En instanciant le second argument (le théorème) pour chercher les sortes y intervenant, la BT indexée est 20 fois plus rapide en moyenne que la BT Prolog[5]. La recherche de tous les théorèmes de tous les types[6] a été presque cent fois plus rapide sur la BT indexée[7].

## 4.5.2.2.- D'autres mécanismes possibles

L'adaptation de Prolog au contexte des structures hiérarchiques des présentations modulaires de la BT ne sera jamais que partielle. En effet, l'organisation du programme Prolog est "relationnelle", tandis que celle de notre BT est plutôt "orientée-objet".[8] C'est la raison pour

---

[1] La définition dans le premier cas serait:

```
theoremeDuTypeIndex(Sorte,Theo) :-
        sorteExp(Sorte,_),
        recorded(Sorte,[Theo,_],_).
```

et dans le second cas:

```
theoremeDuTypeIndex(Sorte,Theo) :-
var(Sorte) ->
        theoremeDuType(Sorte,Theo);
        recorded(Sorte,[Theo,_],_).
```

[2] Une moyenne de 0,06 s pour la BT prolog contre 0,02 s pour la BT indexée.

[3] Moyenne de 1,25 s pour la BT prolog contre 0,03 s pour la BT indexée, don cun rapport de plus de 30.

[4] Par exemple, pour les types nat et liste, la requête serait:

```
theoremeDuType(nat,A),theoremeDuType(liste,A).
```

Une réponse à cette requête est le théorème # nil = 0 (la longueur de la liste vide est zéro). Les temps moyens de recherche ont été de 3,33 s et de 0,09 s pour la BT prolog et la BT indexée, respectivement.

[5] 1,03 et 1,84 s (première réponse et toutes les réponses) pour la BT prolog contre 0,05 et 0,08 s pour la BT indexée.

[6] par la requête theoremeDuType(S,A), fail.

[7] 9,26 s / 0,10 s.

[8] Pour illustrer la différence entre ces deux types d'organisation, nous présentons un exemple: soient les assertions suivantes:

```
Minou mange des souris;
```

laquelle il nous paraît plus cohérent d'envisager une recherche des informations dans un contexte orienté-objet.

## *Les systèmes orientés objet*

Les mécanismes privilégiés des systèmes orientés-objet sont *l'héritage* et l'envoi et réception de *messages*.[(((référence)))].

L'héritage permet de transmettre un *attribut* d'un objet aux objets placés en dessous de lui dans une *hiérarchie*. Un attribut représente une caractéristique qu'un objet peut posséder ou non; à certains attributs il est affectée une *valeur*.

Un exemple d'attributs hérités dans les présentations de la BT sont les sortes, opérateurs et axiomes accessibles dans une présentation donnée; en effet, ils sont tous, soit déclarés dans la présentation considérée, soit "hérités" des présentations contenues récursivement. Ainsi, l'adjonction d'un axiome dans une présentation contenue par d'autres présentations, entraîne automatiquement l'adjonction de cet axiome dans l'ensemble d'axiomes accessibles à ces dernières.

Les messages sont le mécanisme qui permet de faire coopérer le objets afin d'effectuer un calcul. Quand un objet reçoit un message, il déclenche l'action correspondante à ce type de message; pour cela, il dispose d'un ensemble de *méthodes*, indexé par les types de messages.

## *Mécanismes de recherche*

La recherche descendante dans une arborescence est un mécanisme qui découle très naturellement d'une structuration hiérarchique d'un ensemble d'objets. Une des hiérarchies de la BT est issue de l'inclusion des présentations (déclaration "contient"). Les racines de cette hiérarchie sont des présentations où les types (leur sorte) sont introduits; c'est la raison pour laquelle cette hiérarchie se prète bien à la recherche des informations en utilisant le typage.

---

Togo est fidèle;

Togo et Minou se lèchent les pattes.

Une organisation relationnelle groupe dans une même table les entités possédant une même caractéristique; on aurait les tableaux suivants:

| Mange des souris |
|---|
| Minou |
| . . . |

| Est fidèle |
|---|
| Togo |
| . . . |

| Lèche ses pattes |
|---|
| Togo |
| Minou |
| . . . |

Dans une organisation orientée-objet, on attache aux individus leurs caractéristiques:

| Minou: |
|---|
| Mange des souris |
| Lèche ses pattes |

| Togo: |
|---|
| Est fidèle |
| Lèche ses pattes |

Il est possible d'ajouter une présentation vide *presentation(racine)* dont les fils seraient les anciennes racines, de façon à avoir une racine unique pour toute la hiérarchie.

La recherche arborescente dans une telle hiérarchie est illustrée par l'algorithme que nous présentons ci-dessous, pour des requêtes X dans un petit sous-ensemble du LIBT ne comportant qu'un des prédicats présents dans la BT Prolog (sorteExp, operateurExp,...)

***algorithme*** *diffusion(**entrée**: une hiérarchie de présentations, une requête X;*
*sortie: une réponse R );*
*1.- Envoyer un message requête(X) à la racine de la hiérarchie;*
*2.- Pour chaque présentation P qui reçoit un message requête(X):*
*2.1.- Si X est satisfait dans P, avec la substitution θ, envoyer message solution(Xθ) à son père;*
*2.2.- Dans le cas contraire:*
*2.2.1.- Si P a des fils, envoyer-les requête(X);*
*2.2.2.- Sinon, envoyer le message échec(X) au père.*
*3.- Quand une présentation reçoit un message solution(S), elle retransmet le même message à son père;*
*4.- Quand une présentation reçoit un message échec(Y):*
*4.1.- Si elle a déjà reçu un message solution(Y), ce message d'échec est ignoré;*
*4.2.- Si tous ses autres fils lui ont aussi envoyé un message échec(Y), envoyer le même message à son père.*
*5.- L'algorithme s'arrête quand la racine reçoit un des messages solution(Xθ) ou bien échec(X), lesquels deviendront la valeur de la sortie R.*
***fin algorithme.***

## *Remarques*

Dans le point 2.1, dire qu'un but X est satisfait dans P avec une substitution θ veut dire que X s'unifie avec une des clauses de l'équivalent Prolog de la présentation P.

Cet algorithme calcule, d'après le pas 5, *une* des solutions de la requête.

L'extension de l'algorithme à tout le LIBT n'est pas trivial.[1]

[1] Il semble nécessaire de mettre en place des mécanismes de marquage et de retour en arrière, comme en prolog.

*"Virtually everyoune is now agreed that knowledge about the problem domain must be used in the logic" [Reiter 76]*

# 5.- Le système de dérivation d'hypothèses

La description du système chargé de dériver des hypothèses demandées par le système de synthèse fait l'objet de ce chapitre.

Rappel.- L'énoncé d'un problème de dérivation d'hypothèses est un triple <B,H,V> où B est le *but* (une fbf), H un ensemble d'*hypothèses* (des formules logiques) et V est un ensemble de variables; la solution de ce problème est une condition P telle que: $(P \wedge H) \Rightarrow B$, et $var(P) \subseteq V$, où var(P) calcule l'ensemble de variables libres de P.

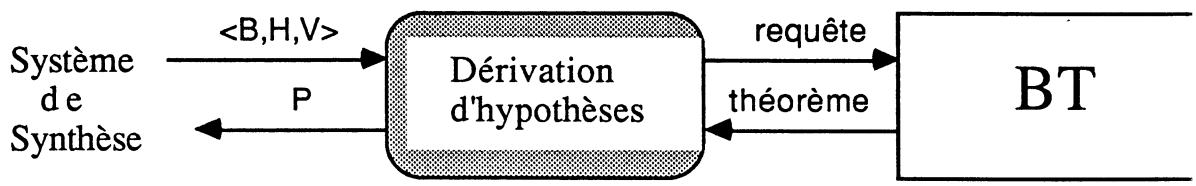
La spécification externe du sous-système déductif pour la dérivation d'hypothèses (ses entrées et sorties) est la suivante (voir §2.2.3):

a) Il reçoit du système de synthèse une demande <B, H, V> de dérivation de préconditions;

b) Il renvoie au système de synthèse une hypothèse dérivée P satisfaisant les contraintes données ci-dessus;

c) Il est en communication avec la BT décrite au §4, à qui il envoie des requêtes spécifiant les théorèmes dont il a besoin; ceux-ci sont envoyés en réponse, s'ils s'y trouvent. La recherche détaillée d'un théorème satisfaisant la requête est à la charge de la BT.

Cette architecture est rappelée dans le schéma suivant:




C'est la partie centrale du dessin qui concerne ce chapitre.

## 5.1.- La dérivation d'hypothèses

La définition du problème de la dérivation d'hypothèses a été donnée au chapitre 2, où nous avons repris la définition de [Smith 82]. Il s'agit de trouver, à partir d'un but B (une fbf) et d'un ensemble d'hypothèses, les conditions additionnelles P nécessaires pour que B soit une conséquence logique de H: $P \Rightarrow (H \Rightarrow B)$, avec $var(P) \subseteq V$. Dans [Smith 82] la dérivation d'hypothèses (appelée "calcul de préconditions") est définie de façon heuristique, car on demande que P soit une condition aussi faible que possible (écartant ainsi la solution trivial P=faux), mais également aussi "simple" que possible, pas trop difficile à calculer, etc... Smith propose même d'appliquer des mesures heuristiques pour évaluer chaque critère, ce qui permet d'obtenir un compromis "adéquat".

### 5.1.1.- Définition formelle de la dérivation d'hypothèses

Du point de vue méthodologique, la définition de Smith présente des faiblesses, car bien qu'il soit envisageable d'utiliser des heuristiques pour effectuer un calcul afin de réduire un espace de recherche trop grand, nous mettons en question cette démarche consistant à définir un calcul en donnant des heuristiques, car il devient impossible de prouver des propriétés de l'objet ainsi défini. Dans le cas de Smith, un problème supplémentaire est qu'il est impossible de savoir si l'on s'est éloigné de la meilleure solution.

Nous allons examiner d'abord une définition de "plus faible condition" au sens littéral du terme.

Définition 1.- Une hypothèse dérivée P pour un problème $<B,H,V>$ est la plus faible ssi pour toute autre hypothèse P' satisfaisant $(P' \wedge H) \Rightarrow B$ et $var(P') \subseteq V$, on a $P' \Rightarrow P$.

L'inconvénient de la définition 1 est qu'elle ne donne pas une méthode pour calculer effectivement la plus faible hypothèse dérivée. Un autre problème est que prouver, à partir de cette définition, qu'une hypothèse dérivée est la plus faible possible, est assez difficile même pour des exemples simples. Une des difficultés de cette preuve vient du fait que l'on peut être amené à utiliser des théorèmes quelconques de la théorie sous-jacente au problème traité. Par exemple, la condition $b \leq c$ de l'exemple traité plus haut est plus faible que $b<c$ à cause des théorèmes:

$$x \leq y \Leftrightarrow (x<y \vee x=y)$$
$$P \Rightarrow P \vee Q.$$

dans les entiers et les booléens.

Smith définit, pour le cas du calcul propositionnel, une "plus faible précondition", qui est équivalente à $H \Rightarrow B$:

$$P \Leftrightarrow (H \Rightarrow B)$$

En effet, le sens $P \Rightarrow (H \Rightarrow B)$ de l'équivalence étant trivial à partir de la définition 1, il reste à prouver $(H \Rightarrow B) \Rightarrow P$. Or, puisque $(H \Rightarrow B) \Rightarrow (H \Rightarrow B)$, toute précondition $P' \Leftrightarrow (H \Rightarrow B)$ serait plus faible que P, ce qui contredit l'hypothèse que P est la plus faible possible.

Pour le CP1 cette assertion n'est plus valable, car il peut se produire qu'aucune des solutions équivalentes à $H \Rightarrow B$ ne satisfasse le critère $var(P) \subseteq V$. Exemple: Soit, dans les entiers, un but B: a<c, l'hypothèse H:a<b et l'ensemble de variables V: {b,c}. Dans ce cas-ci, l'hypothèse dérivée b≤c est la plus faible que nous avons pu trouver, mais elle n'est pas équivalente à a<b $\Rightarrow$ a<c, ce qui est facilement vérifiable à l'aide du contre-exemple a=2, b=4, c=3.
Pour remédier à tous ces inconvénients, nous proposons la notion de plus faible hypothèse dérivée *par rapport à une théorie explicite.* La définition présentée ci-dessous est en fait un sous-produit de nos recherches d'une méthode de dérivation d'hypothèses basée sur la résolution. Le but (nié) et les hypothèses sont mis sous une forme clausale, en utilisant des transformations bien connues [Loveland 78]. Le principe de résolution de clauses est décrit dans [Robinson 65].

Définition 2.- Soit $\neg B$ et H exprimés comme des ensembles de clauses. Soit T un ensemble de clauses toutes valides dans la théorie considérée. Soit une suite de clauses $R_1,\ldots,R_k,\ldots,R_n$ où $R_1,\ldots,R_k$ sont les clauses du but nié, et $R_i$, $k<i\leq n$, est le résolvant[1] de deux clauses dont une est dans $\{R_1,\ldots,R_{i-1}\}$ et l'autre dans $\{R_1,\ldots,R_{i-1}\} \cup H \cup T$. La plus faible hypothèse dérivée par rapport à T, notée $P_T$, est définie comme la limite de:

$$\neg\left(\bigwedge_{i=1}^{n} R_i \mid var(R_i) \subseteq V\right)$$

Ceci revient à prendre l'ensemble de solutions vérifiant la condition sur les variables. L'ensemble ainsi calculé peut être infini.

La définition 2 est une définition constructive, mais pas encore un algorithme pour calculer $P_T$, car le calcul ne s'arrête pas toujours, sauf pour le cas du calcul propositionnel, où l'ensemble de résolvants est toujours fini.

Exemple.- Soit, dans le calcul propositionnel, B: $S \wedge R$, H: {T, S}. On a les clauses suivantes:

$H_1$: T
$H_2$: S
$R_1$: $\neg S \vee \neg R$

On ne peut engendrer qu'un seul résolvant:

$R_2$: $\neg R$

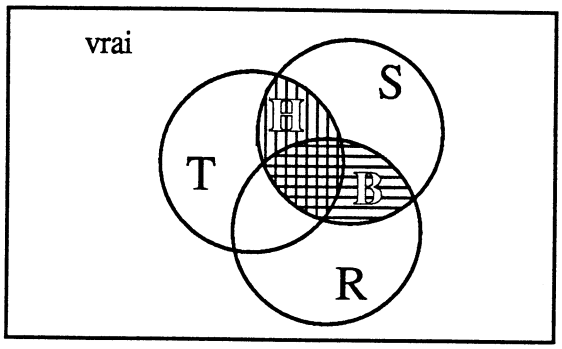
Donc $P = \neg(R_1 \wedge R_2) = \neg(\neg R \wedge (\neg S \vee \neg R)) = R \vee (S \wedge R) = R$.

D'après l'exemple, nous voyons qu'un sous-ensemble des $R_i$ engendrés peut être équivalent à la totalité. Ceci est intéressant puisqu'on cherche à obtenir des ensembles finis de clauses pour exprimer les hypothèses dérivées. En fait, tout sous-ensemble des clauses intervenant dans $P_T$ donne une hypothèse dérivée, mais pas la plus faible en général.

Nous pouvons visualiser graphiquement cet exemple à l'aide des diagrammes de Venn (nous représentons les extensions des prédicats):

---

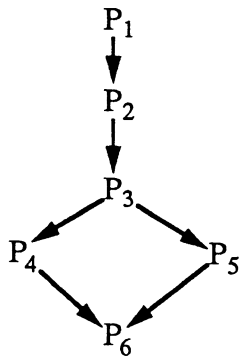
[1] Les variables libres de H et de B ne s'instancient pas lors de la résolution; elles sont considérées comme des constantes.




H est dans l'intersection de S et T, et B est dans l'intersection de S et R. Supposer les hypothèses H revient à restreindre l'univers à la région H. L'hypothèse additionnelle recherchée doit encore restreindre H vers la partie commune avec B ($S \cap T \cap R$). Par conséquent, tout ensemble tel que son intersection avec H est égale à $S \cap T \cap R$ fera l'affaire; en particulier, on trouve les solutions suivantes:

$P_1$: $S \cap T \cap R$
$P_2$: $R \cap S$
$P_3$: $R$
$P_4$: $Sc \cup R$
$P_5$: $T^c \cup R$
$P_6$: $(S \cap T)^c \cup R$

Il existe un ordre partiel d'inclusion dans l'ensemble des solutions, représenté graphiquement par le treillis suivant:




où les flèches vont d'une condition plus forte à une condition plus faible.

La définition de Smith ne nous indique pas clairement laquelle, parmi ces solutions possibles, il faut choisir: est-ce $P_6$, la plus faible, où bien $P_3$, la plus simple ? Notre définition 2, par contre, nous amène tout de suite à la solution $P_3$.

Nous pouvons voir intuitivement les pas de résolution comme des "élimination de redondances" entre le but et les hypothèses.

*Correction de la définition*

Il faut prouver que la solution obtenue par la définition 2 répond bien au problème de la dérivation d'hypothèses, c'est-à-dire, est telle que:

$$P_T \wedge (H \wedge T) \Rightarrow B$$

La preuve (par réfutation) consiste à ajouter $P_T$ à l'ensemble de clauses de H et T, et à prouver que ceci est inconsistant vis-à-vis de $\neg B$, en engendrant la clause vide. Or, ceci est évident, car $P_T$ est par définition le complément des clauses $\{R_i \mid var(R_i) \subseteq V\}$ dérivables de H, T et $\neg B$.

## 5.1.2.- La solution de problèmes de dérivation d'hypothèses

La dérivation d'hypothèses est un problème déductif très compliqué. Elle est strictement plus compliquée que la preuve de théorèmes, car en plus de prouver $H \wedge P \Rightarrow B$, il faut trouver P. En fait, la démonstration de théorèmes est un cas particulier où les hypothèses additionnelles sont réduites à vrai. Nous pourrions dire que la preuve de théorèmes est à la dérivation d'hypothèses ce que la vérification de programmes est à la synthèse de programmes.

Jusqu'à présent, la seule proposition de système de dérivation d'hypothèses a été celle de [Smith 82]. Smith en présente une réalisation en lisp, appelée RAINBOW, dans [Smith 85]. Ce système sera examiné au §5.3. Une version "parallélisée" sera présentée au §5.4.

Nous nous sommes penchés sur d'autres possibilités de calcul d'hypothèses dérivées. D'un point de vue sémantique, la solution recherchée est égale à la somme (c'est-à-dire, la disjonction) de tous les contre-exemples de la formule donnée. Par conséquant, toute méthode de preuve basée sur la recherche de contre-exemples, telle que la résolution ou les tableaux sémantiques, pourrait être adaptée à la dérivation d'hypothèses.Nous allons présenter au §5.2 la dérivation d'hypothèses par résolution, issue directement de la définition donnée au §5.1.1.

## 5.2.- Dérivation d'hypothèses par résolution

Nous pouvons adapter la définition 2 du §5.1.1 pour obtenir un algorithme qui calcule effectivement des plus faibles hypothèses dérivées par rapport à une théorie explicite. L'avantage de cette méthode est qu'elle s'appuie sur la déduction par résolution et la stratégie "de l'ensemble de support" qui ont été très étudiées et pour lesquelles il existe des réalisations très efficaces. Pour ce faire, il faut ajouter une condition d'arrêt qui nous permettrait de savoir quand la plus faible hypothèse dérivée (par rapport à une théorie explicite) a été atteinte.

Une première solution consiste à arrêter la production de clauses plus ou moins arbitrairement, quand on a déjà obtenu quelques solutions partielles $R_j,\ldots,R_k$, et former avec celles-ci une approximation $\neg R_j \vee \ldots \vee \neg R_k$ de $P_T$ . L'inconvénient de cette méthode est qu'on ne peut jamais savoir si la solution obtenue est la plus faible.

Nous présentons maintenant un exemple de cette méthode de dérivation d'hypothèses par approximation .

Soient B: a<c, H: {a<b}, V: {b,c}, où a,b,c sont des variables libres[1]. On forme les clauses:

$H$: $a<b$
$T_1$: $r\leq s \vee \neg r<s$
$T_2$: $r<t \vee \neg r\leq s \vee \neg s<t$
$T_3$: $t<s \vee \neg r=s \vee \neg t<r$
$R_1$: $\neg a<c$

où $R_1$ est $\neg B$, et $T_{1,2,3}$ sont des théorèmes dans les entiers; r,s,t sont des variables quantifiées universellement, et peuvent être instanciées. On dérive les clauses suivantes:

$R_2$: $\neg a\leq s \vee \neg s<c$ $(R_1/T_2)$
$R_3$: $\neg a<s \vee \neg s<c$ $(R_2/T_1)$
$R_4$: $\neg b<c$ $(R_3/H)$

Nous avons trouvé une clause, $R_4$, qui satisfait la contrainte de variables. On peut choisir, soit de récupérer l'hypothèse dérivée b<c, qui n'est pas la plus faible, soit de continuer la production de clauses, sans aucune garantie de trouver après une meilleure solution. Décidons de continuer:

$R_5$: $\neg r=c \vee \neg a<r$ $(R_1/T_3)$
$R_6$: $\neg b=c$ $(R_5/H)$

Encore une hypothèse de trouvée! Une approximation de $P_T$ sera la disjonction des solutions déjà trouvées, c'est-à-dire, $P = b<c \vee b=c$.

Il se trouve que celle-ci est une plus faible hypothèse dérivée[2]; autrement dit, la suite de la production de clauses n'apportera rien. Le seul problème est que nous ne pouvons pas le savoir à ce stade du développement du calcul: nous voilà dans la semi-décidabilité!

Avec le théorème additionnel:

$T_4$: $y<z \vee y=z \vee \neg y\leq z$

on peut arriver dans la suite (en résolvant contre $R_4$ et $R_6$) à la solution:

$R_7$: $\neg b\leq c$

Cette solution est équivalente à la précédente, c'est-à-dire:

$\neg R_4 \vee \neg R_6$
$\Leftrightarrow \neg R_7$
$\Leftrightarrow \neg R_4 \vee \neg R_6 \vee \neg R_7$
$\Leftrightarrow \neg R_4 \vee \neg R_6 \vee \neg R_7 \vee \ldots$

La solution $R_7$ peut être considérée la meilleure vis-à-vis d'un critère de "simplicité" comme dans [Smith 82].

---

[1] Elles ne pourront être instanciées lors de l'unification de littéraux.

[2] Nous n'en faisons pas la preuve

## 5.3.- Le système déductif de Smith

Le système déductif proposé dans [Smith 82] est une extension des systèmes dits de 'déduction naturelle' tels que [Bledsoe 77], [Reiter 76], [Loveland 78].

*Déduction Naturelle*

Nous allons nous épargner la peine de donner une définition de 'déduction naturelle'. Comme [Bledsoe 77], nous allons plutôt l'associer avec des idées qui -peut-être pour des raisons circonstantielles- ont été développées en rapport avec la déduction naturelle.

Les système de déduction naturelle sont guidés par le but. Ceci veut dire qu'ils permettent de prouver des expressions de la forme:

$$H_1, \ldots, H_n \vdash G$$

où $H_1, \ldots, H_n$ sont les hypothèses (ou prémisses) et G est le but; le symbole "$\vdash$" veut dire "le côté droit est déductible du côté gauche".

La plupart des systèmes de déduction naturelle travaillent par génération de sous-buts, c'est-à-dire qu'ils créent des sous-problèmes d'un problème donné de façon à ce que l'assemblage des solutions à ces sous-problèmes permette de construire une solution au problème initial.

Les systèmes de déduction naturelle sont caractérisés par une nette séparation entre les hypothèses et le but. Toute interaction entre eux s'effectue à travers la barrière "$\vdash$", et au moyen des règles d'inférence précises. Par contre, les systèmes basés sur le principe de résolution de Robinson mèlent le but avec les hypothèses, et ils engendrent ensemble une quantité quelconque de clauses intermédiaires dans la plus complète promiscuité.

Nous croyons qu'il existe une connexion entre cette distinction but - hypothèses, et le raisonnement cause - effet utilisé extensivement en sciences naturelles. C'est peut-être ceci qui rend intéressants les systèmes de déduction naturelle du point de vue de la compréhension intuitive du processus de déduction[1]. Ce côté intuitif est très important dans le cadre d'un système de synthèse assistée de programmes. En effet, un des objectifs de notre projet est de permettre à l'utilisateur du système de se substituer à un quelconque de ses composants, le sous-système déductif en l'occurrence; l'utilisateur doit donc être capable de réaliser lui-même les tâches de celui-là.

Les systèmes de déduction naturelle ont un grand nombre de règles d'inférence, en contraste avec les systèmes basés sur la résolution où une seule règle s'avère suffisante. Le choix des règles peut entraîner une explosion combinatoire en absence de méthodes efficaces pour diriger la preuve. Néansmoins, dans les systèmes de déduction naturelle il est possible d'adapter des heuristiques 'humaines' empruntées à la pratique des mathématiciens [Bledsoe 77]. Nous citons ici Nevins cité déjà dans [Bledsoe 77]: "A point worthy of stress is that a deductive system is

---

[1] L'avantage intuitif des systèmes de déduction naturelle a été parfois contesté par les "résolutionistes", qui argumentent qu'il est toujours possible d'ajouter des interfaces adéquates entre le système et l'utilisateur pour faciliter à ce dernier la compréhension des action entreprises par le système.

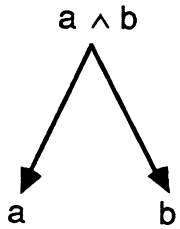
not 'simpler' merely because it employs fewer rules of inference. A more meaningful measure of simplicity is the easy with which heuristic considerations can be absorbed into the system".

## *Description du système de Smith*

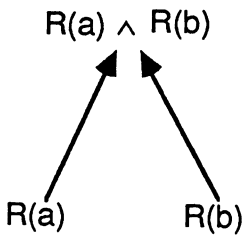
Comme on l'a dit plus haut, les règles du système de Smith sont pour la plupart des extensions de celles des systèmes typiques de déduction naturelle [Bledsoe 77], [Reiter 76], [Loveland 78].

Le système de Smith exécute le calcul des hypothèses dérivées (appelées "préconditions par Smith) en deux étapes: "Dans la première étape les règles sont appliquées aux buts de façon répétée, réduisant ceux-ci en sous-buts, ce qui a pour effet d'engendrer un arbre de sous-buts avec des buts primitifs aux feuilles. La seconde étape concerne la recomposition ascendante des préconditions. Initialement chaque but primitif délivre une précondition; ensuite, chaque fois qu'une précondition est trouvée pour chacun des sous-buts d'un but, une précondition composée est calculée pour ce but-là, d'après la règle de réduction / composition utilisée"[1].

La première étape est donc la décomposition des buts commune à tous les systèmes de déduction naturelle. Pour ceux-ci, l'étape de recomposition est très simple, car elle a à voir avec des réponses uniquement affirmatives ou négatives; dans le système de Smith, par contre, il est nécessaire de fournir des procédures adéquates de recomposition de résidus pour former les préconditions. En fait, chaque règle de Smith est une règle double, comportant une part utilisée pendant l'étape de réduction des buts, tandis que l'autre part est vouée à la recomposition ascendante des préconditions. Par exemple, la partie *décomposition* associée à la règle "and" de Smith est:




Elle divise un but formé d'une conjonction en ses composants. Soient R(a), R(b) des hypothèses dérivées à partir de a et b respectivement; alors la partie de recomposition de cette règle est:




c'est-à-dire, $R(a \wedge b) = R(a) \wedge R(b)$ [2]

---

[1] Extrait traduit de [Smith 82]

[2] Nous n'avois pas considéré les difficultés liées à la composition des substitutions.

Le système de Smith est en communication avec une "Data Structure Knowledge Base" [1] contenant des définitions d'opérateurs ainsi que des listes de théorèmes "utiles". Celles-ci sont parcourues séquentiellement, en cherchant un théorème unifiable avec les formules du problème à résoudre.

Dans le tableau 4.1 (page suivante) nous présentons les règles d'inférence du système de Smith. Celles-ci sont mises sous la forme $a_1, \ldots, a_n \rightarrow b_1, \ldots, b_m$, où les $a_i$ sont des buts (sous la forme $H \vdash G$) ou bien des hypothèses ou des théorèmes de la BT, et les $b_j$ sont les sous-buts générés.

| | Nom de la règle | Génération de sous-buts | Préconditions |
|---|---|---|---|
| R1 | Réduction de buts conjonctifs | $H \vdash (P \wedge Q) \rightarrow H \vdash P, H \vdash Q$ | $R(H \vdash P) \wedge R(H \vdash Q)$ |
| R2 | Réduction de buts disjonctifs | $H \vdash (P \vee Q) \rightarrow H \vdash P, H \vdash Q$ | $R(H \vdash P) \vee R(H \vdash Q)$ |
| R3 | Réd. de conjonctions d'hypothèses | $\{B \wedge C\} \cup H \vdash A \rightarrow \{B,C\} \cup H \vdash A$ | $R(\{B,C\} \cup H \vdash A)$ |
| | | $\{B \wedge C\} \cup H \vdash A \rightarrow \{B\} \cup H \vdash A,$ $\{C\} \cup H \vdash A$ | $R(\{B\} \cup H \vdash A) \vee R(\{C\} \cup H \vdash A)$ |
| R4 | Réd.de disjonction d'hypothèses | $\{B \vee C\} \cup H \vdash A \rightarrow \{B\} \cup H \vdash A,$ $\{C\} \cup H \vdash A$ | $R(\{B\} \cup H \vdash A) \wedge R(\{C\} \cup H \vdash A)$ |
| R5 | Application d'équivalences | $H \vdash A, C \Leftrightarrow A \rightarrow H \vdash C$ | $R(H \vdash C)$ |
| R6 | Application d'implications | $H \vdash A, C \Rightarrow A \rightarrow H \vdash C$ | $R(H \vdash C)$ |
| R7 | Modus ponens à partir des hypothèses | $\{B\} \cup H \vdash A, B \Rightarrow E \rightarrow \{B,E\} \cup H \vdash A$ | $R(\{B,E\} \cup H \vdash A)$ |
| R8 | Dualité Buts/Hypothèses | $H \vdash (\neg B \vee A) \rightarrow \{B\} \cup H \vdash A$ | $R(\{B\} \cup H \vdash A)$ |
| | | $\{B\} \cup H \vdash A \rightarrow H \vdash (\neg B \vee A)$ | $R(H \vdash (\neg B \vee A))$ |
| R9 | Remplac.,de termes égaux | $H \vdash A(r), r=s \rightarrow H \vdash A(s)$ | $R(H \vdash A(s))$ |
| R10 | Remplacement par des égalités conditionnelles | $H \vdash A(s), B \Rightarrow s_1=s_2 \rightarrow H \vdash A(s_2), H \vdash B$ | $R(H \vdash A(s_2)) \wedge R(H \vdash B)$ |
| P1 | Buts Primitifs | $H \vdash A, A$ | T (constante "vrai") |
| P2 | Buts Primitifs | $H \vdash A, \neg A$ | F |
| P3 | Buts Primitifs | $\{\} \vdash A$ | A if $var(A) \subseteq V$ |

*Tableau 4.1*

## *Discussion*

Le système RAINBOW présente plusieurs caractéristiques typiques des systèmes de l'IA. Ceci est mis en évidence par les point suivants:

- Du fait qu'il est souvent très difficile de garantir qu'un certain résidu est le plus faible possible,[2] RAINBOW cherche plutôt des "bons" résidus, qui doivent être, certes, aussi faibles que possible, mais aussi syntaxiquement simples et faciles à obtenir. Tous ces critères sont combinés par des heuristiques pour obtenir une mesure quantitative de la "qualité" d'un résidu.

---

[1] Voir architecture dans le chapitre 1

[2] Cf. §5.1, §5.1.1.

- La profondeur maximale de l'arbre de preuve[1] est aussi ajustée par des heuristiques.

- La sélection des théorèmes dans la DSKB se fait par des mesures de priorité ajustées par des heuristiques.

- L'architecture Système déductif / Base de Connaissances évoque bien évidemment les systèmes d'IA.

- Le choix d'un système de "déduction naturelle" renforce le côté intuitif du système. Cette facilité de compréhension est nécessaire du moment que RAINBOW inter-agit largement avec l'utilisateur[2], lui permettant, par exemple, d'insérer des résidus proposés par celui-ci.

En ce qui concerne le troisième point, il apparait que RAINBOW recherche des théorèmes dans la DSKB en mode "chaînage avant", c'est-à-dire qu'il ne caractérise pas les théorèmes nécessaires à un moment donné. Ainsi ses promesses d'un "guidage par le but" [3] sont cruellement trahies. D'autre part, la recherche séquentielle de théorèmes se traduit en une limitation sevère de la taille de la DSKB, si l'on veut rester dans des temps de recherche "raisonnables".

Le côté "démonstration de théorèmes" de RAINBOW souffre de faiblesses importantes, notamment:

- Incomplétude pour le calcul propositionnel. Par exemple: il est incapable de prouver la formule $A \Rightarrow B, B \Rightarrow C \vdash A \Rightarrow C$. [4]

- Pas de démonstration par induction. Celle-ci est la plus limitative des restrictions pour les preuves concernant des domaines définis inductivement (listes, entiers,...). Ceci oblige à stocker dans la DSKB la plus grande quantité possible de théorèmes.

- Peu de flexibilité dans l'utilisation des théorèmes de la DSKB. Pas de prise en compte de la commutativité ou associativité des opérateurs. Par exemple, il peut prouver $l \neq nil \vdash \#l > \#cdr(l)$, où # calcule la longueur d'une liste, mais non pas $\#(a \bullet l) > \#l$, qui en est une variante.

Nous pouvons donc dire que RAINBOW est un système "basé sur la connaissance", celle-ci étant les théorèmes de la DSKB.

Le premier des inconvénients ne se présente pas dans notre proposition de dérivation d'hypothèses par résolution.

Dans le paragraphe suivant nous faisons une proposition de "parallélisation" du système de

---

[1] Si cette profondeur est dépassée, un échec, qui déclenche le backtracking, est déclaré.

[2] Plusieurs niveaux d'interaction sont proposés à l'utilisateur par des menus.

[3] Cf. §1.3

[4] Le système formel présenté dans [Smith 82] est complet pour le calcul propositionnel, mais les stratégies de recherche d'une solution mises en place dans le système RAINBOW ne le sont pas.

Smith, dans laquelle nous ne cherchons pas à remédier aux inconvénients de celui-ci, mais plutôt à en améliorer les performances par le biais de l'exécution simultanée de plusieurs opérations.

La démarche suivie pour la recherche de théorèmes dans le système de Smith diffère de nos propositions de caractérisation de théorèmes faites au §5.6; néanmoins, ces dernières sont relativement indépendantes du système déductif utilisé, et pourraient s'adapter directement au système de Smith.

## 5.4.- La 'parallélisation' du système de Smith

Au cours de notre travail de thèse nous avons fait l'analyse des règles d'inférence de [Smith 82] sur des exemples exprimés graphiquement, pour mieux visualiser intuitivement le processus déductif. Sur cette représentation, des possibilités de fonctionnement en parallèle, divergeant radicalement du système de Smith, sont apparues.

Nous décrivons d'abord les motivations et contexte de cette recherche.

### 5.4.1.- Le parallélisme en IA

Il a été reconnu que le parallélisme porte la promesse d'accroître la performance des systèmes d'IA de plusieurs ordres de grandeur [Haynes et al.82]. De plus, certains problèmes d'IA exigent, par leur nature et par le volume de calcul qu'ils impliquent, une solution avec un degré important de parallélisme dans les calculs; tel est le cas de la vision par ordinateur.

Paradoxalement, l'introduction de machines hautement parallèles a entraîné des grandes difficultés pour leur utilisation effective en IA, c'est-à-dire, pour la solution de problèmes pratiques. En effet, pendant longtemps les chercheurs en IA ont fait obstinément le travail de faire rentrer toute solution algorithmique dans le cadre restreint de l'ordinateur séquentiel. A force de répéter toujours cette démarche, nous avons conformé nos mentalités à la machine de Von Neumann, en identifiant la notion d'algorithme à celle de programme séquentiel. Maintenant, avec l'introduction de machines parallèles, nous nous trouvons autant optimistes que déboussolés, sans trop savoir s'il faut adapter les algorithmes séquentiels ou bien tout recommencer...

Pour quelques problèmes d'IA, tels que le traitement d'images, il est relativement facile de voir comment arriver à un traitement en parallèle : étant donné un arrangement rectangulaire de "pixels", il est naturel de l'associer à un arrangement de processeurs avec des connexions nord-sud-est-ouest avec leurs voisins. Par contre, l'utilisation du parallélisme pour le traitement de chaînes de symboles peut s'avérer beaucoup moins évident. Le point difficile pour arriver à un traitement symbolique parallèle est de savoir comment diviser un certain problème en sous-problèmes relativement indépendants qui seraient exécutés en parallèle.

Il existe des méthodes générales pour faire cela, comme le parallélisme ET/OU [Bibel et al.87]. Il est aussi possible de "paralléliser par force" les algorithmes séquentiels existants [Haynes et al.82] [Zima et al.88], en faisant une analyse des dépendences dans les données. Ces méthodes sont souvent inefficaces et/ou combinatoires, et sont donc appelées "naive methods" par [Bibel

et al.87]. Une analyse en profondeur du problème donné s'avère nécessaire pour détecter son parallélisme "intrinseque" ("external parallelism" dans [Naran'yani 85]) et pouvoir donc le diviser en sous-problèmes de façon naturelle.[1]

*Parallélisme et déduction*

Les progrès en systèmes parallèles ont été adaptés à la déduction automatique surtout pour le cas restreint des Clauses de Horn [Bibel et al.87], soit pour l'exécution en parallèle des programmes PROLOG, soit pour permettre la programmation parallèle en PROLOG [Quinou, Trilling 88].

Pour la logique du premier ordre (disons, démonstration automatique de théorèmes) nous ne trouvons guère de propositions dans la littérature [Bibel, Aspetsberger 85]. Dans [Jorrand 87] est proposée une machine parallèle abstraite pour le CP1, basée sur la méthode des "connexions" de [Bibel 82]. A notre connaissance il n'existe aucune proposition de systèmes parallèles de déduction naturelle.

## 5.4.2.- Notre méthode pour l'introduction du parallélisme

Notre méthode pour "paralléliser" le système de Smith est fondée sur une représentation des formules logiques par des graphes orientés. En partant de la représentation des formules par des graphes, pour comprendre intuitivement notre proposition de système parallèle il suffit, au lieu de voir ces graphes comme des données mortes, de les percevoir comme des entités actives où les nœuds, dotés d'une capacité d'action, sont liés par les arcs servant de voies de communication entre les nœuds connexes. Cette idée, évoquée par [Bibel et al.87] revient à associer à chaque nœud du graphe un *automate cellulaire* (un processeur virtuel), et à chaque arc une voie pour l'échange de messages. Elle fournit une méthode pour "incruster" le parallélisme dans la structure même du problème.

Les inférences dans notre système vont prendre la forme de *transformations* du graphe représentant les formules du problème à résoudre. Ces transformations sont faites par interaction des automates associés aux nœuds avec leurs voisins. Ils travaillent tous simultanément dans le graphe, en effectuant des transformations concernant leur environnement proche; ils effectuent donc des transformations locales. Les effets de ces transformations peuvent se *propager* à travers le graphe, pouvant ainsi exercer une influence sur des parties du graphe non voisines.

La détection, pour une cellule donnée, des situations où une règle d'inférence peut être appliquée, n'ayant connaissance que de son propre état et de celui des cellules voisines, est rendu possible grâce à l'introduction d'un ensemble d'attributs attachés aux nœuds. Les attributs permettent donc de traduire des informations globales en des informations locales, qui peuvent par la suite être analysées au niveau de chaque cellule. Ainsi, par exemple, une cellule qui possède un attribut appelé 'h' sait qu'elle se trouve à la racine d'une hypothèse, sans avoir à examiner la morphologie du graphe entier.

---

[1] Nous disons "naturel" dans le sens où cette division dépend de la nature même du problème donné.

Le fonctionnement du système par des transformations locales introduit une forme de parallélisme "ET" dans le sens où il est possible d'effectuer plusieurs transformations simultanément dans des parties différentes d'un même but.

Nous proposons en plus une forme de parallélisme "OU" qui consiste à traiter en parallèle plusieurs buts (ou versions d'un but) qui partagent un même ensemble d'hypothèses.

Le parallélisme OU est utilisé dans notre proposition pour éviter les retours en arrière (backtracking) dus aux transformations infructueuses. En effet, chaque fois qu'une transformation non simplificatrice est appliquée sur un but, on fait une forme de *duplication* de celui-ci qui permet d'en conserver l'ancienne version. *Les nouvelles et anciennes versions des buts poursuivent en parallèle leur propre évolution.*

La demande de place mémoire entraînée par la création des nouveaux buts est réduite en grande partie par le partage des sous-expressions communes fourni par notre représentation des formules. Nous proposons de plus les deux mécanismes suivants:

1) Duplication implicite des buts. En utilisant des nœuds spéciaux "duplicateurs" nous évitons la duplication de chaque nœud du but.
2) Destruction des buts les moins prometteurs[1], si cela s'avère nécessaire, pour obtenir de la place mémoire et pouvoir ainsi continuer le processus.

Du fait que les buts à basse priorité peuvent être éliminés, il découle que la stratégie de recherche d'une solution est déterminée par les heuristiques d'évaluation des buts.[2]

### 5.4.2.1.- Notation graphique

Les représentations de formules par des graphes ont été proposées comme un moyen pratique pour stocker de façon économique des expression dans le contexte de la programmation dite 'applicative', car elles permettent de partager des sous-expressions apparaissant en même temps dans plusieurs expressions, évitant ainsi de faire une copie pour chaque expression [Staples 1980]. D'autres avantages seront discutés plus tard.

Pour commencer, nous donnons quelques définitions.

Un graphe orienté est un pair <N, A> où N est un ensemble de nœuds et A est un ensemble d'arcs de la forme $(n_1, n_2)$ allant du nœud $n_1$ au nœud $n_2$.

Soit $\lambda$ la fonction qui obtient l'étiquette d'un nœud, et $\delta$ la fonction qui en obtient le degré de sortie (le nombre d'arcs qui partent de ce nœud).

Les successeurs d'un nœud n dans un graphe (N,A) sont les nœuds x tels qu'il existe un arc $(n,x) \in A$. Soit $n_{[i]}$, pour $1 \leq i \leq \delta(n)$ le i-ème successeur de n.

Un chemin dans un graphe G = <N, A> est une liste de la forme $(n_1,e_1,\ldots,n_{m-1},e_{m-1},n_m)$, avec

---

[1] Des mesures heuristiques d'évaluation doivent être fournies pour les buts.

[2] Voir discussion

$n_i \in N$, $1 \leq i \leq m$ et $1 \leq e_i \leq \delta(n_i)$ un numéro de succeseur, tel que pour toute sous-liste $(n_i, e_i, n_{i+1})$ nous avons $n_{i\,[e_i]}=n_{i+1}$.

Un chemin $(n_1, \ldots, n_m)$ est un cycle si $n_1=n_m$. Nous allons nous intéresser plus particulièrement aux "graphes orientés sans cycles", appelés "dags" (de "directed acyclic graphs" en anglais).

On définit les sous-graphes d'un graphe G=<N,A> avec racine r, noté G|r, comme le plus grand graphe qui satisfait:

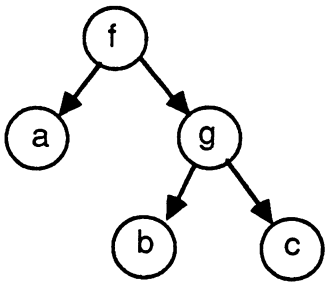
- chaque nœud de G|r est un nœud de G;
- pour chaque nœud n de G|r il existe un chemin (r,...,n).

Nous appelons dag avec racine un dag G=<N,A> tel que, pour un certain $r \in N$ on a G|r=G. Les arbres sont une sous-classe des dags avec racine.

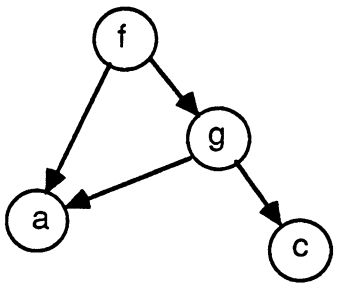
Nous pouvons représenter les termes par des arbres étiquetés de la façon classique:

- Les étiquettes des nœuds sont des symboles de fonctions ou de variables;
- Les successeurs sont les arguments de ces fonctions.

Par exemple, le terme f(a, g(b, c)) serait représenté par l'arbre de la figure ci-dessous:




Les arbres sont un cas particulier des dags. Ces derniers sont spécialement utiles pour exprimer des termes avec plusieurs occurrences d'une même sous-expresion; par exemple, le terme f(a,g(a,c)) est représenté par le graphe ci dessous:



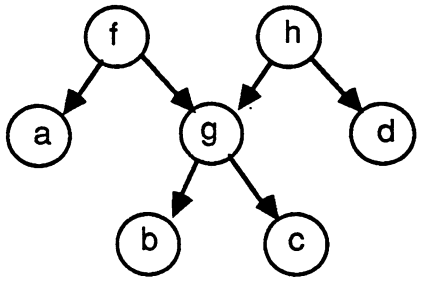

Il s'avère pratique de pouvoir dénoter les dags par des termes, ceci dans le but d'en faciliter la description formelle. On nomme de manière optionnelle les nœuds à l'aide des "identificateurs de nœud"; les occurrences répétés d'un identificateur de nœud représentent un même nœud.

La grammaire suivante fournit une notation pour les dags avec racine :

$V_T$ : {ensemble des symboles de fonctions et de variables}
NŒUD-ID : {ensemble des identificateurs de nœud}
DAG ::= NŒUD-ID : DAG
| $V_T$ ( LISTE-DAG )
| $V_T$
LISTE-DAG ::= DAG | DAG , LISTE-DAG

Les nœuds sont dénotés par un 'identificateur de nœud', <nœud-id>; les occurrences répétées des nœud-id font référence au même nœud, ce qui permet d'exprimer le partage des nœuds. Par exemple, f(x:a, g(x,c)) dénote le graphe dessiné ci-dessus.

Les ensembles de termes sont représentés par l'union des dags qui les représentent. Cette union est dénotée dans notre notation linéaire par l'opérateur '+'. Exemple: le graphe f(a, x:g(b,c)) + h(x,d) est dessiné ci-dessous:




Le terme concat(cons(a,b),cons(a,b)) peut être représenté par les trois dags suivants:

i) concat(cons(a,b),cons(a,b)) -un arbre;
ii) concat(cons(x:a,y:b),cons(x,y)) -avec partage de a et b;
iii) concat(x:cons(a,b),x) -avec partage de cons(a,b).

Ainsi, pour un même terme il est possible de construire plusieurs dags qui le représentent. Nous allons définir une relation d'ordre partiel sur toutes les représentations d'un même terme, à l'aide d'un homomorphisme de graphes défini de la façon suivante:

Etant donnés deux graphes $G_1=<N_1,A_1>$, $G_2=<N_2,A_2>$, un homomorphisme h de $G_1$ vers $G_2$ est une application $h:N_1 \rightarrow N_2$ telle que pour toute $n \in N_1$:

- $\lambda(h(n)) = \lambda(n)$;
- $h(n)_{[i]} = h(n_{[i]})$, $0 \leq i \leq \delta(n)$;

c'est-à-dire, l'homomorphisme préserve les étiquettes et les successeurs.

Pour chaque terme t on considère la classe Graphe(t) de tous les graphes qui le représentent 'correctement', avec tous les homomorphismes entre eux. L'élément *terminal* dans cette classe, que nous notons R(t), sera la *représentation standard* du terme t.

Proposition: La représentation standard d'un terme est celle avec le plus grand partage de sous-graphes communs, donc avec le plus petit nombre de nœuds.[1]

---

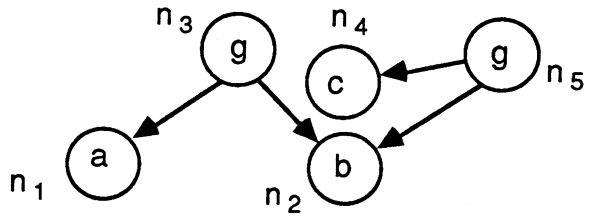
[1] Ceci vient du fait que les homomorphismes sont des fonctions, donc ils ne peuvent que réduire le nombre de nœuds mis en correspondance.

Il se trouve qu'une méthode pour obtenir la représentation standard d'un terme découle de la "Fermeture de Congruence" de [Nelson, Oppen 80], définie de la manière suivante:

Soit R une relation binaire sur l'ensemble N des nœuds d'un graphe, $R \subseteq N \times N$; alors deux nœuds n et m sont congrus sous R si:

$$\lambda(m) = \lambda(n), \text{ et } \forall i \in [1, \delta(n)], (m[i], n[i]) \in R \text{ [1]}$$

Une relation R est fermée par congruence si pour m et n congrus sous R, $(m,n) \in R$. La fermeture de congruence FC(R) est l'unique extension minimale de R qui est une relation d'équivalence fermée par congruence.[2] Exemple: Soit le graphe:

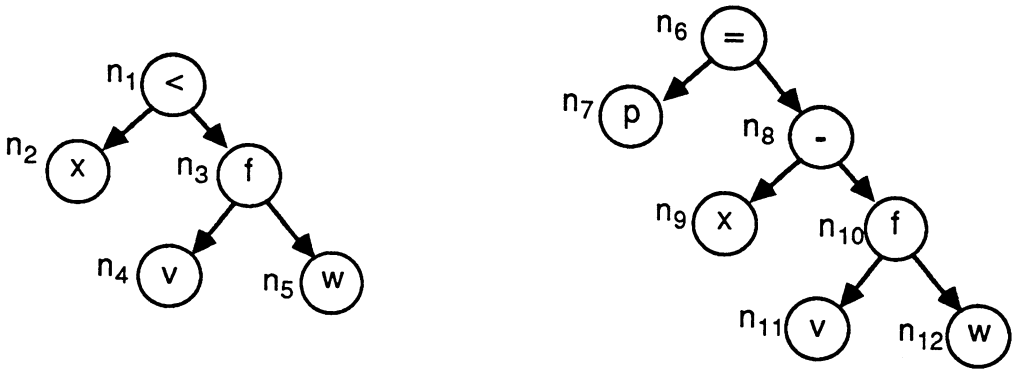



Soit R= $\{(n_1, n_4)\}$; alors $n_3$ et $n_5$ sont congrus. La fermeture FC(R) est:
$\{(n_1,n_1), (n_4,n_4), (n_1,n_4), (n_4,n_1), (n_3,n_5), (n_5,n_3), (n_3,n_3), (n_5,n_5)\}$,
avec la partition associée: $\{\{n_1,n_4\}, \{n_3,n_5\}, \{n_2\}\}$. ♦

Pour le cas de la représentation standard d'un terme, à partir des arbres abstraits de termes on définit $\equiv_f$ comme la relation d'équivalence sur les nœuds qui découle de l'application de la fermeture de congruence pour la relation R exprimant l'égalité syntaxique des constantes et des variables, Nous avons donc $(m,n) \in R \Rightarrow (\lambda(m) = \lambda(n) \wedge \delta(m)=\delta(n)=0)$. Ainsi, pour des nœuds $n_1$, $n_2$ avec $\delta(n_1) = \delta(n_2) = k$, $n_1 \equiv_f n_2$ ssi $\lambda(n_1) = \lambda(n_2)$ et $\forall i \in [1, k]$, $n_{1[i]} \equiv_f n_{2[i]}$.[3]

Cette relation d'équivalence entraîne une partition de N. Conceptuellement, nous allons *superposer* ou *fusionner* les nœuds équivalents, avec les arcs qui leur sont propres. Les sous-graphes congrus deviennent ainsi sous-graphes *partagés* ou *communs* à plusieurs termes.

Exemple: pour les expressions $\{x < f(v,w), p = x - f(v,w)\}$, nous avons initialement les arbres:



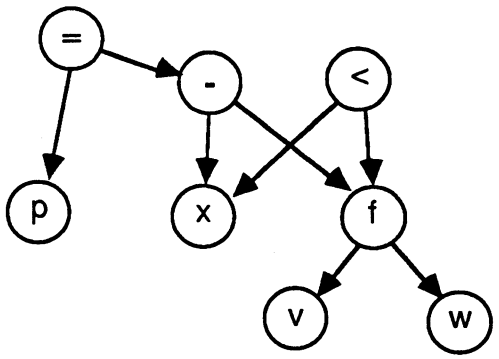

---

[1] On doit avoir $\delta(n) = \delta(m)$ si le graphe est bien formé.

[2] [Nelson, Oppen 80] donnent des algorithmes séquentiels pour calculer la fermeture de congruence d'une relation, avec des temps $O(m^2)$, où m est le nombre d'arcs du graphe, pour le cas le plus défavorable. L'algorithme de [Downey et al. 80] a un temps $O(m \log^2 m)$ pour le cas le plus défavorable.

[3] Le lecteur peut vérifier que celle-ci est bien une relation d'équivalence.

On voit que $n_4 \equiv_f n_{11}$ et $n_5 \equiv_f n_{12}$; par conséquent nous avons aussi $n_3 \equiv_f n_{10}$. En faisant la fusion des nœuds congrus, nous obtenons la représentation:standard (r:x<s:f(v,w)) + (p=r-s), dessinée ci dessous:




## 5.4.2.2.- Représentation des problèmes de dérivation d'hypothèses

Nous présentons ici le système d'attributs qui permet de reconnaître, au niveau de chaque cellule, le rôle de celle-ci au cours d'une dérivation d'hypothèses.

Un problème de dérivation d'hypothèses <B,H,V> est représenté par un graphe construit de la façon suivante:

i) Construire le graphe qui représente H∪{B};
ii) Attacher les attributs suivants aux nœuds concernés:

- Attribut 'B' à la racine des buts;
- Attribut 'noir' à tous les nœuds des buts;
- Attribut 'h' à la racine des hypothèses;
- Attribut 'blanc' aux nœuds des hypothèses;
- Attribut 'V' aux variables dans V.

L'attribut 'noir' est (intentionnellement) redondant par rapport à l'attribut 'B'. Il indique en fait qu'un certain nœud est la racine d'un sous-terme d'un but. Cette information relève de la structure *globale* du but concerné. L'utilité de l'attribut 'noir' consiste à placer cette information au niveau de chaque nœud. Il en est de même pour l'attribut 'blanc', par rapport à 'h'. En cas de partage, un nœud appartenant à une hypothèse et à un but simultanément sera noir. [1]

Au cours d'une dérivation d'hypothèses, un attribut 'noirci' (variante de l'attribut 'noir') peut se présenter. Il sera utilisé pour indiquer qu'un certain nœud vient de changer de blanc à noir. C'est dans ce sens que nous le considérons comme une variante de l'attribut 'noir'.

En utilisant les noms des attributs comme des symboles de prédicat, nous avons les relations suivantes, qui sont des "contraintes d'intégrité":

---

[1] Cette priorité du noir sur le blanc vuet dire que le blanc n'est finalement que l'absence du noir. En effet, le rôle de ce deriner est de permettre d'identifier les cellules des buts dont le traitement n'est pas encore fini (cf.§5.4.3.2).

(1) noir(n) ⇒ ¬blanc(n)
(2) B(n) ⇒ noir(n)
(3) noirci(n) ⇒ noir(n)

Les attributs des nœuds peuvent être incorporés à la notation linéaire définie pour les dags. Pour ce faire, nous les écrivons, entourés par des crochets, après l'identificateur et l'étiquette des nœuds, par exemple, n:f[h,noirci] (m:x[V]). Quand un attribut ne se trouve pas explicitement dans la liste d'un nœud qui vient d'être défini, sa valeur n'est pas déterminée[1].

Dans les dessins représentant des graphes, les noms des attributs sont placés à proximité des nœuds concernés, sauf pour les attributs de couleur nœuds noirs, blanc et noirci. Les cercles des nœuds blancs seront dessinés en utilisant un trait léger, tandis que pour les nœuds noirs nous utilisons des cercles à trait épais, lesquels sont en plus hachurés pour marquer les nœuds noircis.

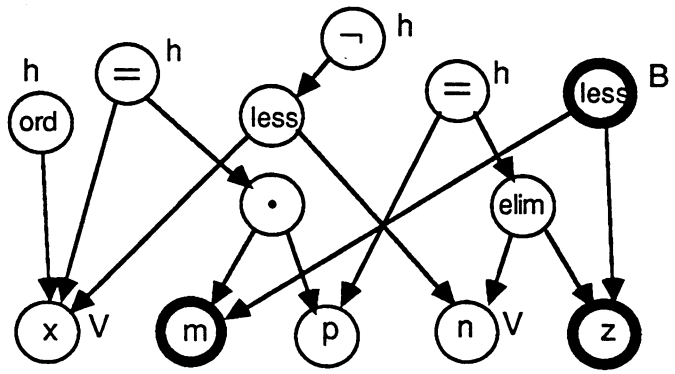
Par exemple, soit le problème de dérivation d'hypothèses ci-dessous:

B = less(m,z), V = {n,x}
H = {x=m•p, ¬less(n,x), p=elim(n,z), ord(x)}

Il peut être représenté par le dag:

**ord**[h] ($n_1$:**x**[V])
+ $n_1$ =[h] ($n_2$:**m**[noir] • $n_3$:**p**)
+ $n_3$ =[h] **elim**($n_4$:**n**[V], $n_5$:**z**[noir])
+ ¬[h] **less**($n_1$, $n_4$)
+ **less**[B,noir] ($n_2$, $n_5$) [2]

Ce graphe est dessiné ci-dessous:




[1] sauf si elle peut être déterminée à partir des autres attributs par les 'contraintes d'intégrité'.

[2] Nous avons utilisé des caractères **gras** pour les symboles de la formule provenants de B et de H, afin d'en faciliter la lecture.

## 5.4.3.- Les règles d'inférence

Les règles d'inférence de notre système sont des règles de transformation de graphes, c'est-à-dire, des opérateurs R: $G_1 \rightarrow G_2$ où R est la règle, $G_1$ le graphe donné et $G_2$ le graphe modifié. Ceci veut dire que les formules représentées par $G_2$ sont des conséquences logiques des formules représentées par $G_1$.

Plus concrètement, les règles d'inférence sont exprimées comme des "règles de réécriture de graphes" [Barendregt et al. 87]; ce formalisme sera décrit ci-dessous.

Il était également possible de formaliser les règles comme des descriptions de l'interaction entre les nœuds du graphe, dans un style "orienté-objet". Nous avons choisi les règles de réécriture de graphes surtout à cause de leur concision et de leur meilleure compréhension intuitive. Par contre, une version orienté-objet serait plus facilement implantable dans un vrai environnement parallèle.

### 5.4.3.1.- Réécriture de graphes

Pour décrire les règles, nous allons prendre la syntaxe et la sémantique du langage de programmation LEAN [Barendregt et al. 87], avec quelques extensions.[3]

Une règle de réécriture a la forme:

$$\alpha \rightarrow \beta\,\delta$$

où $\alpha$, appelé "redex pattern", est la représentation standard d'un graphe sans cycles (dag) avec des variables, laquelle va filtrer un sous-graphe du graphe reçu en entrée, exactement comme le côté gauche d'une règle de réécriture de termes. $\beta$ est appelé "contractum pattern", et son rôle est équivalent à celui du côté droit d'une règle de réécriture de termes. La partie $\delta$ est optionnelle, et décrit un ensemble de "redirections" de la forme:

$$\mathrm{red}[o \rightarrow \eta]$$

qui signifie que tous les arcs $(x,o) \notin \beta$ seront remplacés par des arcs $(x,\eta)$. Quand la partie des redirections n'est pas présente dans une règle, il ne sera effectué qu'une "redirection de racine", c'est-à-dire, les arcs dirigés vers le côté gauche de la règle seront redirigés à la racine du côté droit de la règle.

Par exemple, la règle de simplification true $\wedge$ P $\rightarrow$ P prendrait la forme: $n_1$:true $n_2$:$\wedge$ $n_3$:P $\rightarrow$ $n_3$.[4] Quand cela ne se prête pas à confusion, nous allons utiliser l'étiquette comme identificateur de nœud, comme dans true $\wedge$ P $\rightarrow$ P.

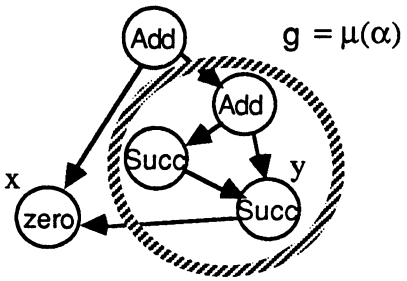
Appliquer une règle de réécriture $\alpha \rightarrow \beta\,\delta$ à un graphe d'entrée G nécessite les pas suivants, qui seront illustrés avec le graphe G=Add(x:zero,Add(Succ(y:Succ(x)),y) et la règle Add(Succ(w),z) $\rightarrow$ Succ(Add(w,z)):

---

[3] Dans cet exposé nous ne supposons pas une connaissance du langage LEAN.

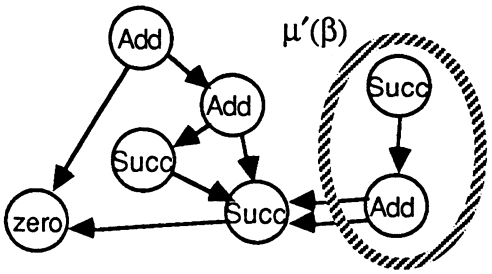
[4] Nous nous permettons d'utiliser des opérateurs infixés, qui n'étaient pas prévus par la grammaire des dags.

- Mettre en correspondance $\alpha$ et un sous-graphe g de G; cette mise en correspondance définit un homomorphisme $\mu:\alpha\rightarrow g$ qui préserve les fonctions et les constantes; g est appelé redex;

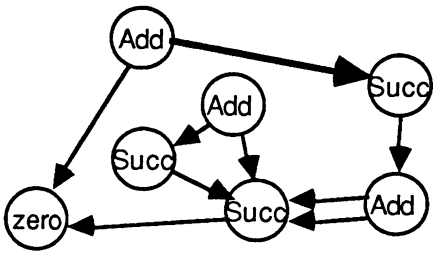



$\mu = \{w\rightarrow y, z\rightarrow y\}$

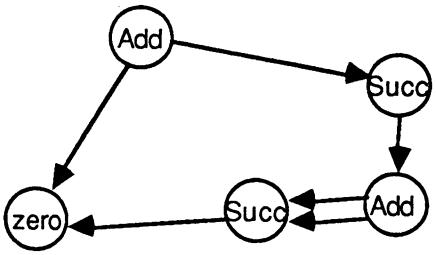
- Créer une copie isomorphe $\mu'(\beta)$ des parties de $\beta$ qui ne se trouvent pas dans $\alpha$; les arcs allant d'un nœud $n\in\beta$ vers $m\in\alpha$ auront comme image les arcs allant de $\mu'(n)$ à $\mu(m)$:




- Appliquer les redirections $\delta$ (en parallèle) aux arcs non contenus dans $\beta$; s'il n'y a pas de redirections explicites, rediriger les arcs allant vers la racine de g pour les diriger vers la racine de $\mu'(\beta)$:




- Les nœuds du nouveau graphe qui ne sont pas accessibles à partir de la racine sont éliminés:



## 5.4.3.2.- Présentation des règles d'inférence

Pour rendre plus compréhensible l'éxposé, nous allons classer les règles d'inférence de la façon suivante:

a) Règles de transformation du but;
   a.1) Egalités, équivalences et implications;
   a.2) Gestion des attributs
b) Règles concernant les hypothèses.

Pour chaque règle nous présentons son nom, la règle correspondante dans le système de Smith, sa définition formelle, et une explication avec éventuellement un exemple.

### *Règles de transformation du but*

Les égalités sont utilisées dans notre système, comme dans celui de Smith, pour effectuer des remplacements (raisonnement équationnel). Néanmoins, nous faisons un classement des égalités pour différencier le traitement donné à chaque classe; ainsi nous avons

- Des égalités (théorèmes ou hypothèses) simplificatrices;
- Toutes les autres égalités.

Nous supposons que l'application des égalités "simplificatrices" est toujours bénéfique, dans le sens qu'on ne risque pas de perdre des solutions. Dans le cas général, par contre, l'application d'un remplacement issu d'une égalité risque de nous éloigner de la solution. C'est là où le parallélisme intervient: nous proposons de conserver les versions du but avant et après le remplacement, de telle façon que toutes les deux seront traitées simultanément (parallélisme OU). Nous allons d'abord traiter le cas des égalités simplificatrices.

**Règle RWS** (correspond aux règles R5 et R9 de Smith, remplacements par des équivalences ou par des égalités)

Pour une égalité g=d telle que d<g par l'ordre de simplification < [Dershowitz 83], nous avons la règle de réécriture de graphes g' → d', où g' et d' sont les graphes représentant les termes g et d. Notez qu'il n'y a pas de redirection explicites, donc il n'est effectué qu'une redirection "de racine".

Beaucoup des règles simplificatrices son utilisées pour simplifier des expressions booléennes, commes les suivantes:[1]

$\neg$ true $\rightarrow$ false
$\neg$ false $\rightarrow$ true
$\neg (\neg p) \rightarrow p$
$\neg (p \wedge q) \rightarrow \neg p \vee \neg q$, etc.

---

[1] Prises du système RAINBOW de Smith.

Quelques autres règles sont des théorèmes distingués utilisés fréquemment dans certaines théories, comme a + 0 → a pour les entiers. Le codage des théorèmes distingués dans les règles d'inférence peut aider à accélérer le processus de preuve [Manna, Waldinger 85].

**Règle DUP** (correspond aux règles R5 et R9 de Smith, remplacements par des équivalences ou par des égalités)

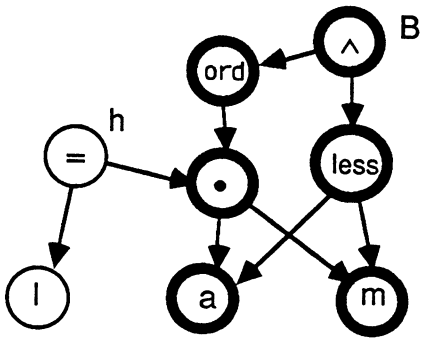
p[blanc] ≡[h] q[noir]) →

p[noirci] ≡[h] q[noir]) + p ≺ q

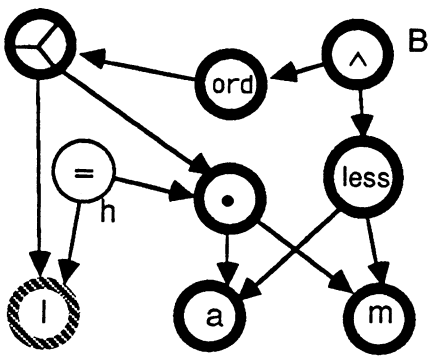
red[q[noir] → ≺]

où ≡ est ⇔ ou =, p et q sont des nœuds -q appartennant à un but (il est noir), et ≺ est un symbole special appelé "duplicateur" dont la sémantique est un choix non-déterministe entre ses successeurs. Notez la présence de l'attribut "noirci", indiquant que la couleur du nœud p vient de devenir noire.

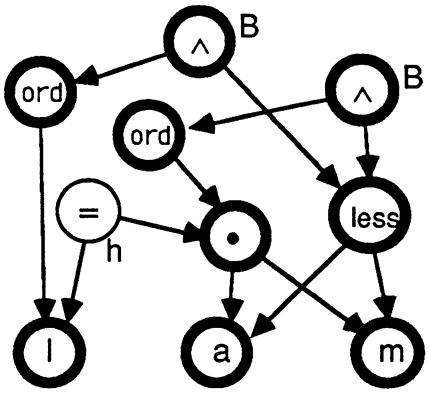
Exemple: soit le graphe:




Il peut être transformé par la règle DUP en:




La règle DUP effectue une duplication implicite du but concerné. En effet, le duplicateur est une bifurcation dans laquelle chaque chemin se dirige vers la racine d'un sous-graphe du but. Un but avec un duplicateur est considéré comme deux buts; la partie partagée de ces buts-là est placée sur le duplicateur. Ainsi, le graphe présenté ci-dessus représente les deux buts suivants:



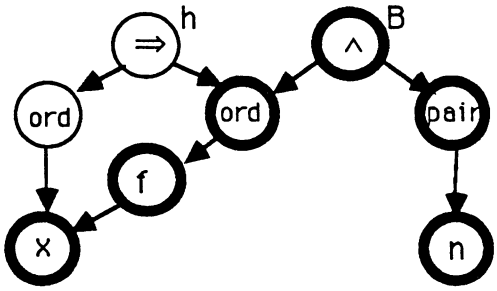

Nous avons introduit ce mécanisme de duplication implicite des buts au lieu d'une duplication explicite (qui reviendrait à dupliquer tous les nœuds de la partie partagée) pour les raisons suivantes:

1) Economie de place mémoire, puisqu'on évite de dupliquer la partie partagée.

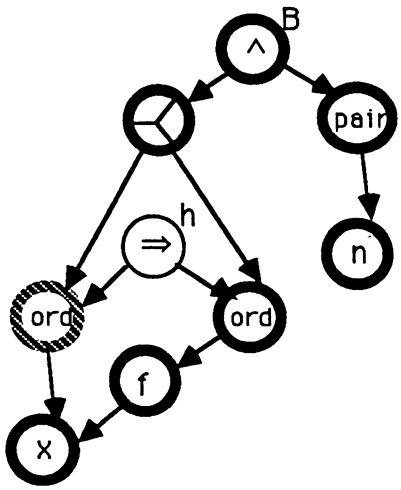
2) Utilisation de transformations locales. La duplication implicite des buts, avec l'introduction d'un duplicateur, n'affecte qu'un petit groupe (quatre) de nœuds voisins. La duplication explicite des buts, par contre, aurait entraîné des modifications de sous-graphes d'une extension quelconque, et aurait été en contradiction complète avec l'idée de partage adoptée.

L'application des implications (règle R6 de Smith) peut se faire à l'aide de la règle DUP, avec le symbole $\Rightarrow$ à la place de $\cong$. Ceci permet d'effectuer des remplacements d'une expression booléenne par une condition *plus forte* (antécédent de l'implication).

Exemple: soit le graphe:




qui est transformé par DUP en:




Quelquefois il est nécessaire de séparer explicitement la partie partagée de deux sous-buts, pour effectuer des traitements différents pour chacune. Pour cela, les règles suivantes sont proposées:

**Règle DR** ("duplicator rising", pas d'équivalent dans le système de Smith)

$$\sigma(s_1,\ldots,r\prec t,\ldots,s_n) \rightarrow (\sigma(s_1,\ldots,r,\ldots,s_n) \prec \sigma(s_1,\ldots,t,\ldots,s_n))$$

où $s_1,\ldots,s_n$, r et t sont des nœuds, et $\sigma$ est une étiquette (un opérateur). C'est un schéma de règle plutôt qu'une règle.

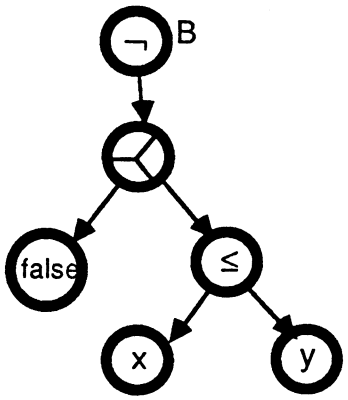
**Règle DD** (destruction du duplicateur, pas d'équivalent dans le système de Smith)

$$n_1[B] \prec n_2[B] \rightarrow n_1[B] + n_2[B]$$

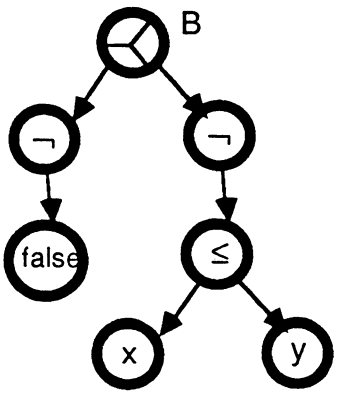
Notez que $n_1$ et $n_2$ sont des racines de but.

La règle DR fait remonter le duplicateur d'un niveau vers le haut, en duplicant les parents de celui-ci. Si on est arrivé à la racine du but, la règle DD détruit le nœud duplicateur, qui devient inutile.

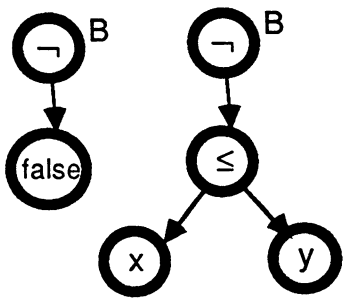
La nécessité d'une duplication explicite d'un sous-graphe (ou la totalité) de la partie partagée au dessus du duplicateur est illustrée par l'exemple suivant: Soit le but:




Pour pouvoir appliquer la simplification booléenne ¬false → true il faut d'abord déplacer le duplicateur à l'aide de la règle DR:




La simplification booléenne peut alors être effectuée. Puisque le duplicateur est à la racine, il peut être éliminé:




Les règles DR et DD donnent une sémantique opérationnelle du symbole duplicateur "$\prec$", car leur application permet de transformer un graphe contenant des duplicateurs en un autre n'en contenant pas.

Notez que le déplacement et la destruction du duplicateur sont toutes deux des transformations locales.

Le processus de séparation graduelle de buts par DR et DD rappelle curieusement le phénomène biologique de la réproduction des molécules d'ADN.

**Règle CER** (remplacement par une égalité conditionnelle, règle R10 de Smith)

$b \Rightarrow [h](s_1 = s_2[noir]) \rightarrow$
$b \Rightarrow [h]\ (s_1[noirci]=s_2[blanc]) + (b \wedge [B,noirci]\ root(s_2)[\neg B])$
$red[s_2[noir] \rightarrow s_1]$

où la fonction "root" obtient la racine du nœud donné en argument.[1]

## *Gestion d'attributs*

Rappel.- Dans notre représentation graphique des problèmes de dérivation d'hypothèses, les attributs "de couleur" noir, noirci, et blanc servent à distinguer les nœuds des buts actifs (noirs ou noircis) des nœuds des hypothèses ou des sous-buts dont le traitement est déjà fini. En effet, une hypothèse dérivée est un but qui a été "blanchi" jusqu'à la racine. Le buts blanchis sont appelés "résidus". Changer l'attribut d'un nœud de noir à blanc signifie qu'il sera incorporé au résidu, et qu'il ne sera plus concerné par les règles d'inférence. En fait, le blanchissement est la transformation graduelle des buts en résidus.

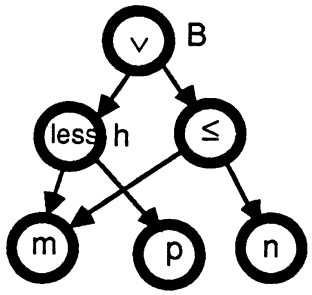
**Règle HY**.- (correspond à P1 de Smith, but qui coincide avec une hypothèse ou un théorème connu)

$$p[h,noir] \rightarrow n{:}true + p[h,blanc]\ \ red[p \rightarrow n]$$

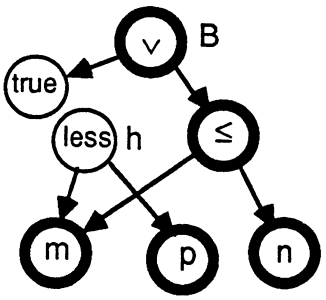
La règle HY permet l'utilisation effective des hypothèses. Un nœud qui est simultanément la racine d'une hypothèse (attribut h) et fait partie d'un but actif (attribut *noir*) est remplacé dans le but par la constante true. Par exemple, soit le graphe:

---

[1] Voir *discussion* pour le calcul de la fonction root. Notez que la règle CER ne cause pas une duplication du but, comme c'était le cas de DUP.




Notez que *less(m,p)* est à la fois une hypothèse et un but. Par application de la règle HY on obtient:




**Règle NHY**.- (correspond à P2 de Smith, un but coïncide avec la négation d'une hypothèse ou théorème connu)

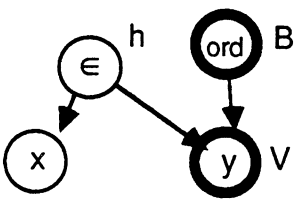
$$\neg[h]\ p[noir] \rightarrow n: false + \neg[h]\ p[blanc]\ red[p \rightarrow n]$$

**Règle R**.- (correspond à P3 de Smith, formation de résidus élémentaires)

$$n:[noir] \rightarrow n[blanc]$$

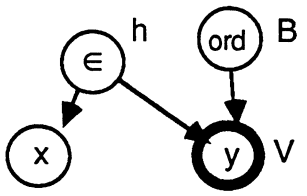
pour un nœud n tel que:

- $\lambda(n)$ est un symbole de prédicat;
- $var(n) \subseteq V$, où var fournit les variables accessibles à partir de n, et V est l'ensemble de variables donné dans l'énoncé du problème de dérivation d'hypothèses[1]. Exemple: A partir du graphe:




Par application de R nous obtenons:

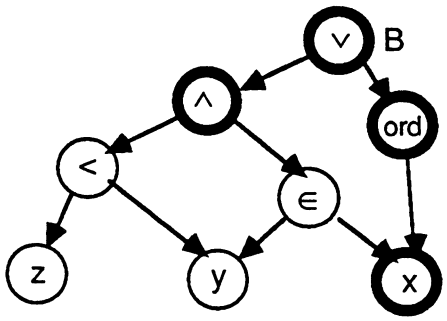
[1] La fonction var peut être calculée en utilisant des mécanismes de communication entre les nœuds voisins.



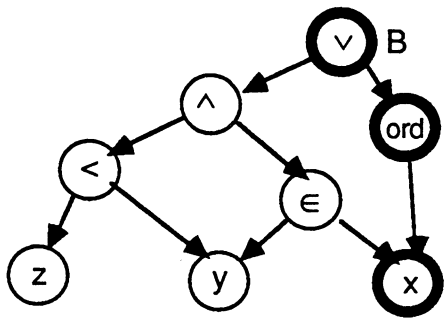

**Règles WB**.- ("white boolean", correspond à la partie ascendante des règles R1 et R2 de Smith, réduction de buts conjonctifs et disjonctifs, respectivement)

$b_1$[blanc] ∧[noir] $b_2$[blanc]) → $b_1$[blanc] ∧[blanc] $b_2$[blanc]
$b_1$[blanc] ∨[noir] $b_2$[blanc]) → $b_1$[blanc] ∨[blanc] $b_2$[blanc]

Chacune des règles WB permet de propager l'attribut "blanc" à travers les connectives ∧ et ∨ lesquelles sont ainsi incorporées aux résidus. Exemple, soit le graphe:




Par application de WB, nous obtenons:




**Règles BB**.- (sans équivalent dans le système de Smith; ces règles complètent l'action de DUP - duplication de buts-, en propageant la couleur noire au sous-graphe en dessous d'uh connective possédant l'attribut *noirci*.)

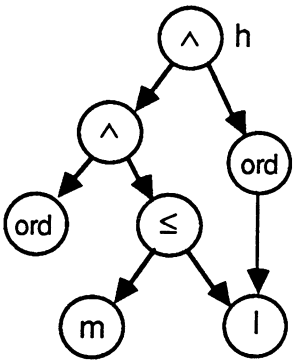
$n_1$ ∨[noirci] $n_2$ → $n_1$[noirci] ∨[noir] $n_2$[noirci]
$n_1$ ∧[noirci] $n_2$ → $n_1$[noici] ∧[noir] $n_2$[noirci]
¬[noirci] n → ¬[noir] n[noirci]

## *Règles concernant les hypothèses*

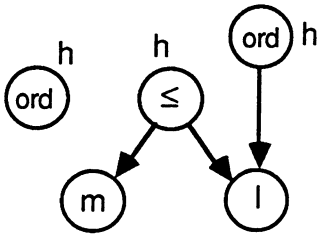
**Règle RCH**.- ("Reduce Conjunctive Hypotheses", R3 de Smith)

b ∧[h] c → b[h] + c[h].

Par exemple, soit le graphe:



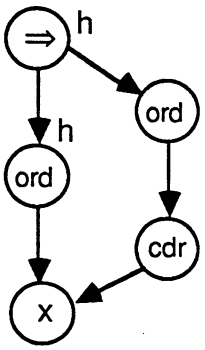

Il peut être transformé en:



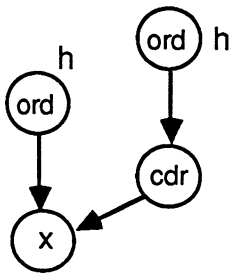

**Règle FIH**.- ("Forward Inference from Hypotheses", R7 de Smith.- modus ponens)

$x[h] \Rightarrow[h] y \rightarrow x[h] + y[h]$

Par exemple soit le graphe:




Par FIH nous obtenons:




**Adjonction de théorèmes**.- (sans équivalent dans le système de Smith)

L'opération d'adjonction de théorèmes au graphe a un statut spécial dans notre système. A la différence du système de Smith, où chaque règle (de remplacement) prend l'initiative d'appeler un théorème "adéquat", dans notre système nous avons isolé une opération spéciale pour l'adjonction de théorèmes, alors que le reste des règles se borne à manipuler le graphe sans faire appel à l'extérieur.

Ajouter un théorème T au graphe se fait de la façon suivante: Soient un graphe G et un théorème T, instancié par une substitution $\theta$.[1]

1) Construire T', la représentation standard de $T\theta$;
2) Faire la fusion des sous-expressions communes à G et T'.

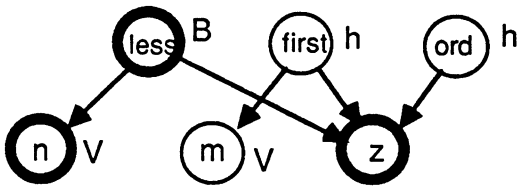
---

[1] La substitution $\theta$ est obtenue lors de la mise en correspondance de T avec la caractérisation utilisée pour la recherche de T (cf. §5.6).

Une fois que les théorèmes sont insérés au graphe, il sont traités comme s'ils étaient des hypothèses. Le problème reste de trouver dans la BT les théorèmes appropriés et de les instancier correctement. Le choix de théorèmes doit être d'autant plus sélectif que leur adjonction au graphe risque de faire exploser la combinatoire, laquelle est contrôlable en se restreignant aux hypothèses. Au §5.6 nous discuterons le problème de la caractérisation la plus fine possible les théorèmes nécessaires.

## 5.4.4.- Un exemple

Dans cette section nous allons développer un exemple complet de dérivation d'hypothèses. Soient B = less(n,z), H = {first(m,z), ord(z)}, et V= {n,m}, où le prédicat less(n,z) est vrai ssi $n \leq x$ pour $x \in z$ d'après un ordre total $\leq$; first(m,z) est vrai ssi m est le premier élément de la liste z; ord(z) indique que z est une liste ordonnée d'après l'ordre $\leq$. Le graphe représentant ce problème est le suivant:




Aucune règle de transformation n'est applicable, donc il est nécessaire d'ajouter des théorèmes au graphe. Supposons qu'on dispose des théorèmes suivants:[1]

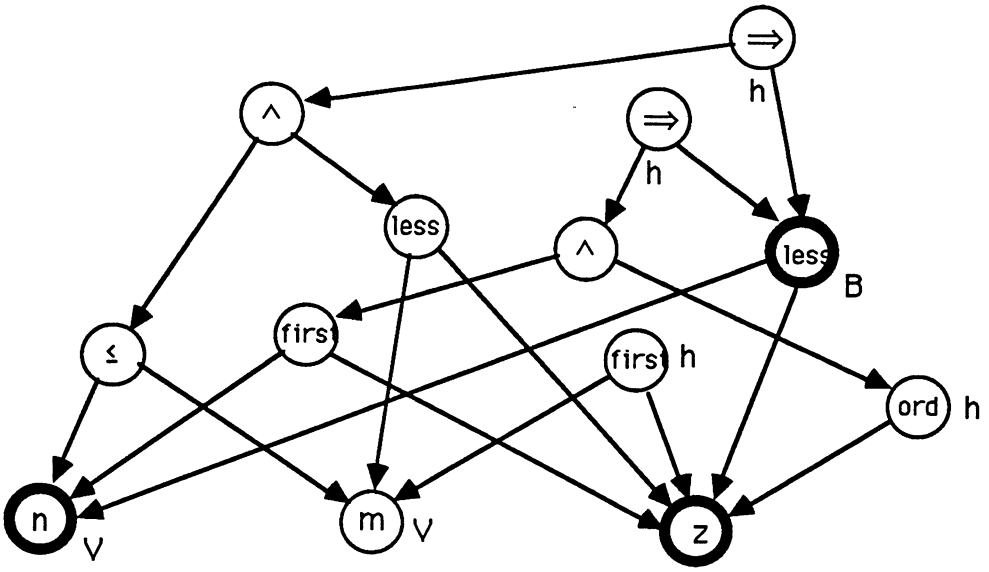
(t1) $ord(l) \wedge first(x,l) \Rightarrow less(x,l)$
(t2) $less(x,l) \wedge y \leq x \Rightarrow less(y,l)$

Ces théorèmes sont adaptés au problème en instanciant leurs variables:

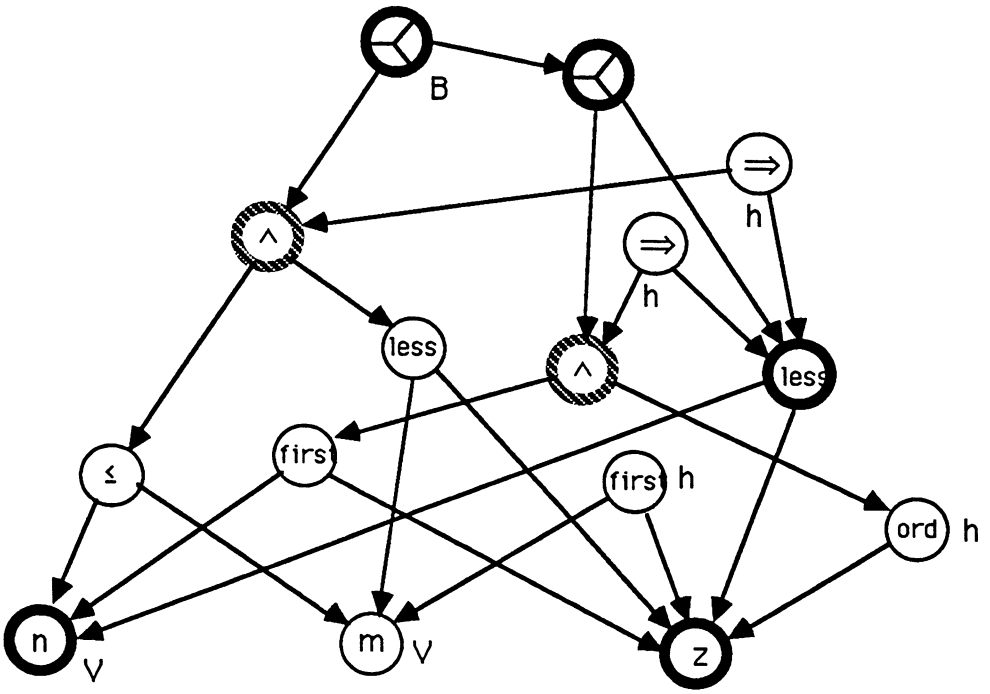
(t1') $ord(z) \wedge first(n,z) \Rightarrow less(n,z)$
(t2') $less(m,z) \wedge n \leq m \Rightarrow less(n,z)$

[1] Le problème de caractériser puis trouver les théorèmes "appropriés" est discuté au §5.6.

Ils sont ensuite ajoutés au graphe, en faisant la fusion des sous-expressions communes:

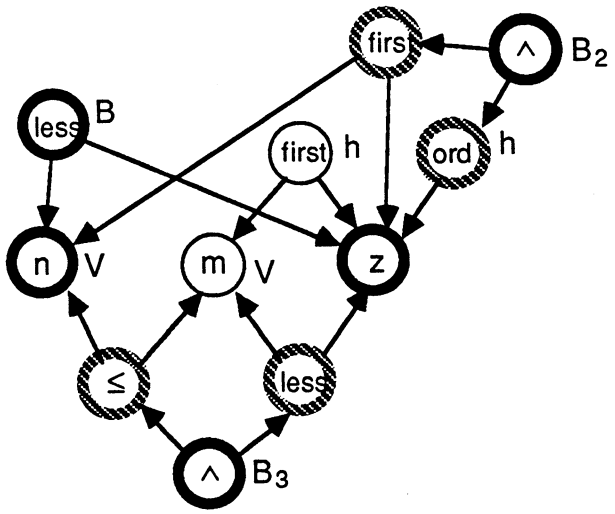



Il devient alors possible d'appliquer la règle DUP, en obtenant: [1]




[1] Remarquez la propagation du noir en utilisant l'attribut "noirci"

Nous pouvons éliminer le duplicateur (règle DD), et continuer la propagation de la couleur noire (règle BB):[1]



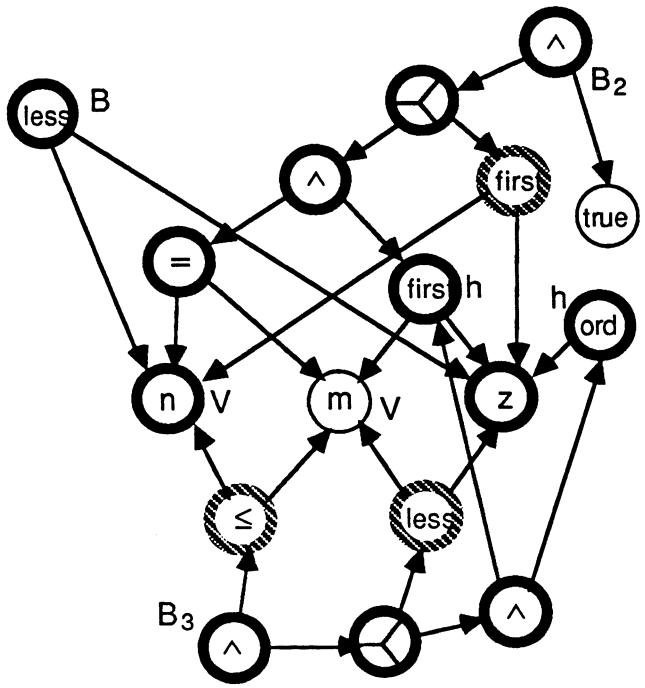

A ce moment nous pouvons effectuer (simultanément) les opérations suivantes:

1.- Appliquer HY au nœud 'ord' de $B_2$ en obtenant la constante "true";

2.- Nous avons le théorème: $n=m \wedge first(m,z) \Rightarrow first(n,z)$ issu de la substitutivité de l'égalité. Il est incorporé au graphe, puis appliqué par DUP à $B_2$;

3.- Appliquer $t_1$ à $B_3$, instancié comme:$ord(z) \wedge first(m,z) \Rightarrow less(m,z)$ par la règle DUP;

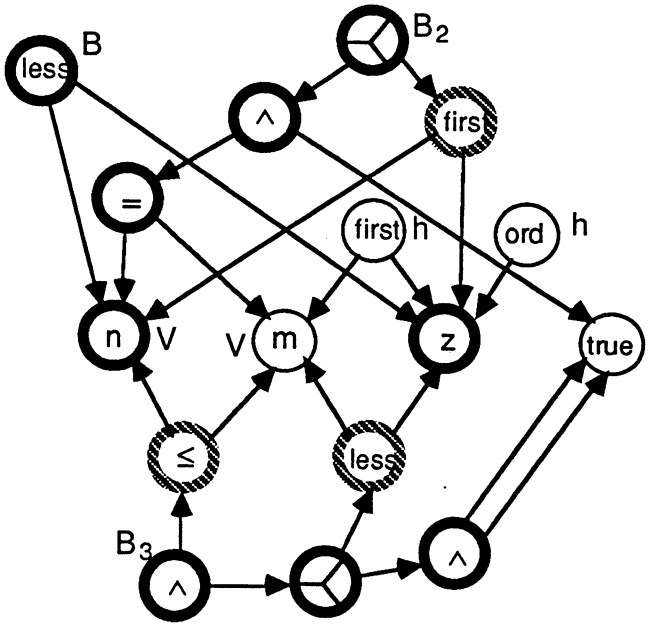
Nous obtenons alors:



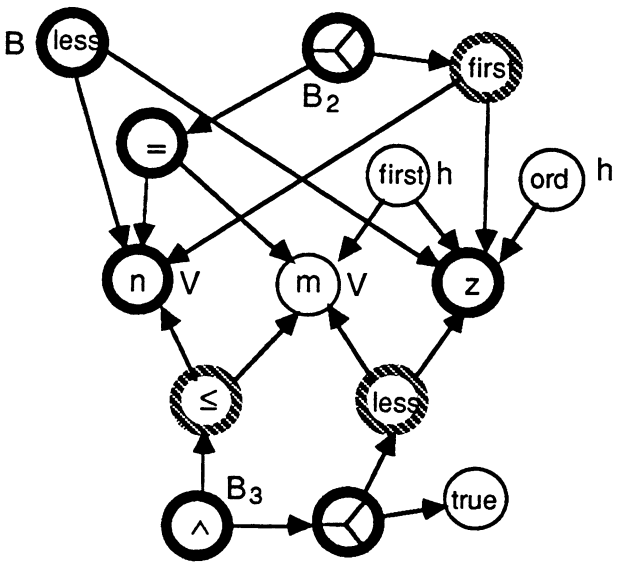

[1] Nous éliminons du graphe les théorèmes déjà utilisés. Cette tactique ne serait pas forcément retenue dans une réalisation du système, mais elle est utile pour rendre le dessin moins touffu.

Nous pouvons maintenant appliquer:

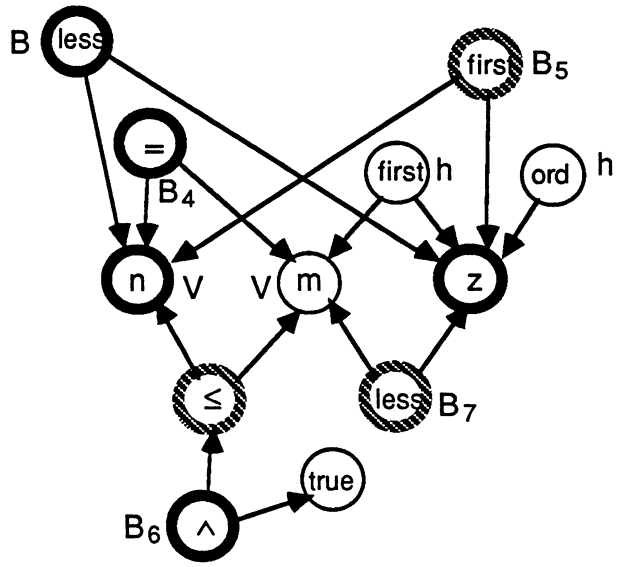
- Une simplification booléenne (a ∧ true → a) à $B_2$;
- HY à la racine des deux hypothèses (nœuds avec l'attribut 'h').




La simplification booléenne P ∧ true → P peut être appliquée à deux endroits du graphe:



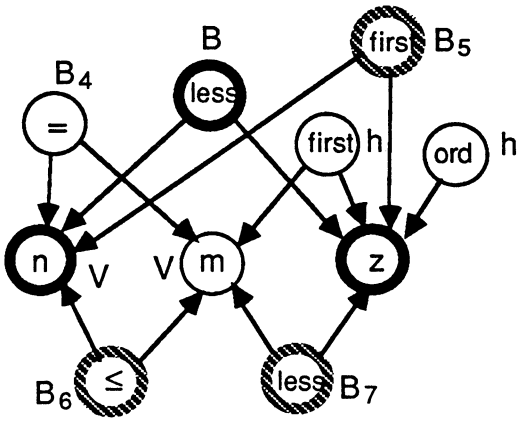
Nous appliquons alors DD sur $B_2$ et DR suivie de DD sur $B_3$:




Par R appliquée à $B_4$ nous obtenons un premier résidu valable:

($R_1$) n=m

Nous pouvons arrêter le système après l'obtention de ce premier résidu. Ici nous allons continuer le processus pour voir si éventuellement nous arrivons à une meilleure solution. Pour l'instant nous effectuons une simplification booléenne sur $B_6$:




Par l'application de R à $B_6$ nous obtenons un deuxième résidu:

($R_2$) n≤m

Nous arrêtons ici le développement de l'exemple. Il se trouve que le second résidu est plus faible que le premier, car (n=m) ⇒ (n≤m). Le résultat final est:

(R) n≤m

## 5.4.5.- Comparaison avec le système de Smith

La proposition de système parallèle que nous avons présentée est une *version* parallélisée du système de Smith; par conséquent, toute comparaison entre ces deux systèmes doit partir du fait que l'un est une variante de l'autre. Dans cette situation-là, la comparaison se réduit à l'analyse des différences existantes entre les deux, car la question des ressemblances est banalisée. Ceci est fait dans le tableau suivant:

| ASPECT<br>SYSTEME DE SMITH | NOTRE PROPOSITION |
|---|---|
| Génération de sous-buts<br>Par règles de décomposition suivant la structure syntaxique des buts | Il est possible de réduire des sous-termes (sous-graphes) d'un but, donc leur traitement ne nécessite pas de la génération des sous-buts pour les sous-termes d'un but. Dans notre système, la génération de buts revient à engendrer des variantes des buts existants |
| Recomposition des hypothèses dérivées<br>Phase ascendante de recomposition, effectuée par la partie recomposition des règles d'inférence | Blanchissement progressif des buts, à l'aide des règles de propagation de la couleur |
| Appel des théorèmes<br>Les théorèmes sont cherchés à l'initiative des règles d'inférence | Il existe une opération spécial pour l'appel d'un théorème et son insertion dans le graphe |
| Principe du fonctionnement<br>Ensemble de procédures (lisp) récursives | Interaction entre les cellules voisines dans le graphe |
| Parallélisme<br>N'est pas discuté par Smith | Traitement parallèle des sous-termes d'un même but (parallélisme ET); traitement simultané des différentes versions des buts (parallélisme OU) |

## 5.4.6.- Stratégies d'application des règles

Nous abordons ici la question du choix de la règle d'inférence à appliquer dans une situation donnée. C'est le problème du contrôle de l'application des règles.

Tout d'abord, le problème du choix des règles disparait de notre système dans les situations où il est possible, au lieu de choisir une règles entre plusieurs règles applicables, de les exécuter toutes en parallèle. Toutefois, dans certaines situations il est nécessaire de faire un tel choix.

Nous disons que deux règles entrent en conflit quand leurs redex[1] ont des nœuds en commun. Ceci veut dire intuitivement que la partie du graphe sur laquelle agit une règle "chevauche" celle sur laquelle agit une autre règle. C'est quand plusieurs règles applicables entrent en conflit qu'il faut faire un choix.

Une stratégie très simple pour choisir parmi les règles en concurrence est de les classer par ordre de priorité. Nous allons grouper les règles d'inférence en classes, chacune contenant les

[1] Cf. §5.4.3.1.

règles d'une même priorité. Les règles dans une classe de priorité plus haute sout toujours choisies parmi l'ensemble de règles applicables dans une situation donnée. Pour départager les conflits éventuels entre règles d'une même classe il serait nécessaire de définir des critères plus fins de choix, lesquels, en l'absence d'une réalisation du système risqueraient d'être spéculatifs.

Nous décrivons maintenant les classes de règles par ordre de priorité décroissante.

**Priorité 1.**
Les règles de priorité 1 effectuent des transformations qui n'affectent pas la sémantique du graphe, en ce sens que les formules représentées par celui-ci avant et après leur application sont les mêmes. Les règles BB et DD entrent dans cette catégorie. La règle BB (black boolean) se charge de la propagation de la couleur noire à la suite de la duplication d'un but; il est donc nécessaire de compléter sa propagation pour arriver à un état stable avant de faire quoi que ce soit d'autre. La règle DD (duplicator deletion) peut toujours s'appliquer sans interférence des autres règles.

**Priorité 2.**
Ce sont des règles qu'il est toujours bénéfique d'appliquer. Les règles WB, HY, NHY, RWS, FIH et RCH sont de priorité 2. Les trois premières sont des règles de "blanchissement" des buts (donc formation de résidus). RWS permet de simplifier les buts, et FIH et RCH facilitent l'application des hypothèses.

**Priorité 3.**
Cette classe comprend des règles permettant d'appliquer des remplacements (DUP et CER). L'augmentation de l'espace mémoire utilisé pénalise l'application de ces règles, donc leur utilisation doit être "judicieuse". Il serait possible de raffiner davantage le contrôle de ces règles en mettant en place des mesures heuristiques capables de juger si un remplacement (CER), ou la création d'un certain but (règle DUP), est bénéfique ou non.

**Priorité 4.**
Ce sont des règles dont il faut repousser au plus tard possible l'application. Les règles DR et R appartiennent à cette classe. L'application de DR (duplicator rising) implique de réduire la partie partagée des buts, ce qui coûte en espace mémoire. La complication du point de vue du contrôle vient du fait que DR est parfois nécessaire *pour appliquer une autre règle* (voir §5.4.3.2). Dans ces cas-là on pourrait affecter à DR la priorité de la règle à l'origine de son déclenchement. La règle R termine définitivement la transformation d'un sous-graphe d'un but, qui sera alors integré sans modification au résidu final. Cette décision de stopper la transformation d'un sous-but équivaut à reconnaître qu'on ne peut plus obtenir un meilleur résidu, ce qui doit être étayé par des raisons valables.

Un critère heuristique pour l'application de la règle R pourrait être le simple "vieillissement" des cellules du graphe: après sa création, à chaque cellule d'un but est affectée une "durée de vie"; si cette durée est dépassée, un des deux cas suivants aura lieu: 1) si var(n) $\subseteq V^1$ pour un nœud n[noir], alors il est blanchi par R et devient partie d'un résidu; 2) dans le cas contraire, le but est éliminé du graphe et leur cellules rendues au "garbage collector".

---

[1] la fonction var fournit les variables accessibles à partir du nœud n en suivant les flèches du graphe.

Dans notre système, comme dans celui de Smith, le contrôle du processus déductif comprend, en plus du choix de règles, le problème de récupérer de la BT les théorèmes adéquats. Cette question sera discutée au §5.6.

## 5.5.- Des extensions au système parallèle

Notre version parallèle du système de Smith se prête bien à l'adaptation de la la procédure de décision de [Nelson, Oppen 80] pour des extensions libres de théories équationnelles non quantifiées. L'intérêt est de disposer dans le graphe de toutes les égalités qui sont des conséquances logiques des égalités données comme des hypothèses. Ceci nous évite d'effectuer des chaînes de remplacements de termes égaux.

La procédure en question est une application de la "fermeture de congruence" décrite au §5.3.3, quand la relation binaire de départ est un ensemble d'égalités. Nous commencerons pour montrer sur un exemple les caractéristiques de cette procédure de décision. Soit l'ensemble de formules:

H: { P(g(a,b)),
b = c,
g(a,c) = d }

Le graphe qui leur est associé d'après notre notation graphique est:

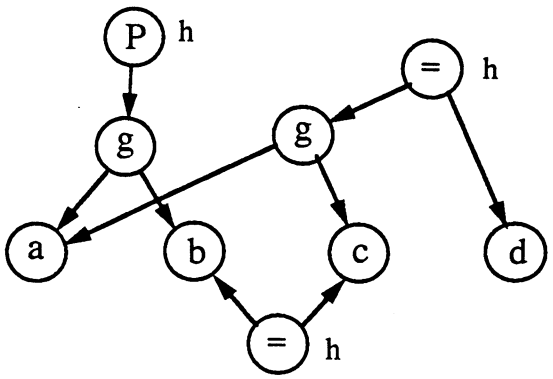



*Figure 5.1*

On calcule la fermeture de congruence à partir de la relation R={(b,c),(g(a,c),d)} contenant les termes égaux, et on observe que les termes congrus sont aussi égaux, à cause de la propriété de substitutivité de l'égalité:

g(a,b) = g(a,c);
g(a,b) = d

Dans [Nelson, Oppen 80] il est proposé de former des classes d'équivalence de termes égaux, ce qui permet de représenter de façon simple des ensembles d'égalités contenant plusieurs termes égaux. Nous allons utiliser le symbole ≡ comme pointeur des éléments appartenant à une de ces classes d'équivalence. Ainsi, pour l'exemple nous aurions:

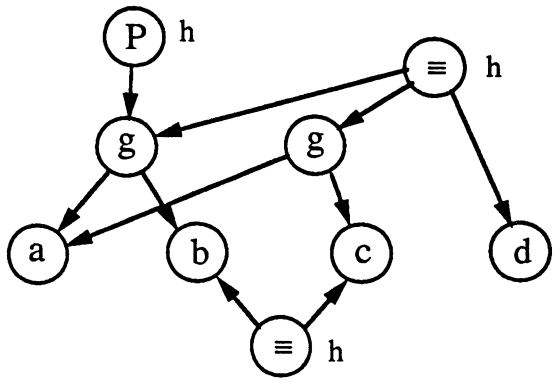



*Figure 5.2*

Les égalités engendrées par la fermeture de congruence (g(a,b) = g(a,c) et g(a,b) = d) sont dénotées par le graphe de la figure 5.2 dans le sens où leur adjonction au graphe ne modifie pas celui-ci.

Maintenant, si nous incorporons au graphe un but g(a,b) = d, il sera immédiatement prouvé par application de la règle HY (adaptée) au graphe suivant:

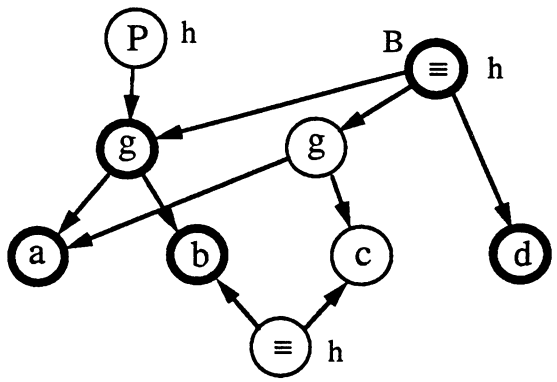



*Figure 5.3*

Ceci nous évite les péripéties des preuves par remplacement équationnel, typiques du système de Smith.

## *Une proposition: la propagation de contraintes équationnelles*

Nous proposons une extension au traitement des extensions libres de théories équationnelles de Nelson et Oppen, qui a été décrit plus haut. Nous montrons d'abord cette extension sur l'exemple précédent. Reconsidérons le graphe de la Figure 5.2. L'extension que nous proposons consiste à rediriger la flèche qui va de P à g, vers le pointeur ≡:

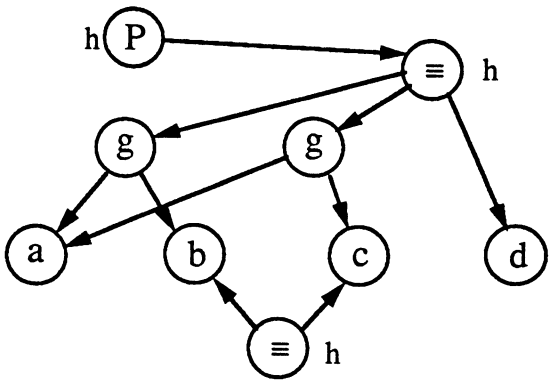



*Figure 5.4*

La transformation proposée est formellement décrite de la façon suivante: si $n_k$ est un successeur d'un nœud n et s'il existe un sous-graphe de la forme $n_0$: ≡[h]($n_1$,...,$n_k$,..., $n_m$), $n_0 \neq n$, alors remplacer l'arc (n,$n_k$) du graphe par (n,$n_0$).

Le nœud ≡ est un point de choix non-déterministe pour les flèches qui arrivent sur lui, de façon analogue au duplicateur "$\prec$" présenté dans les paragraphes précédants. Par exemple, en arrivant à "≡" à partir de P, on peut choisir entre $n_1$:g(a,c), $n_2$:g(a,b) ou d pour fils de P. Ce nouveau rôle du nœud "≡" est ajouté à son ancien rôle de pointeur des classes d'équivalence des termes égaux.

Le graphe de la figure 5.4 dénote, en plus des formules de départ et de celles engendrées par la fermeture de congruence, les formules suivantes:

P(g(a,c));
P(d)

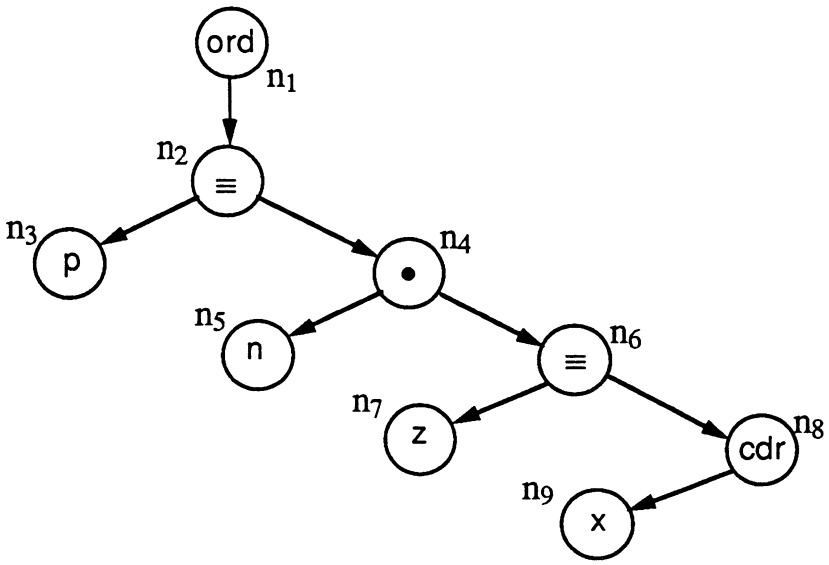
Ces formules sont dénotées par le graphe de la figure 5.4 dans le même sens que précédemment, c'est-à-dire que leur adjonction au graphe ne le modifie pas. ce critère fournit directement une procédure de preuve "ascendante" des buts: pour prouver un littéral B, ajoutez-le au graphe; s'il coincide, après fusion des sous-expressions communes, de la fermeture de congruence pour l'égalité et de notre extension, avec une des hypothèses, il sera automatiquement prouvé. Il est également possible de faire une comparaison du but avec une hypothèse *modulo les points de choix "≡"*, par une sorte de "matching" descendant. Ceci est précisé par l'algorithme suivant:

***fonction*** *test(n:nœud,t:terme):bool;*
***var*** *testbool : bool;*
***si*** *λ(n) = "≡"* ***alors***
*testbool := faux;*

***pour*** *i := 1* ***jusqu'à*** *δ(n)* ***faire***
*testbool := testbool ∨ test($n_{[i]}$,t)*
***sinon si*** *atom(t)* ***alors***
*testbool := λ(n)=t ∧ δ(n)=0*
***sinon soit*** *t=f($t_1$,...,$t_m$);*
*testbool := λ(n)=f ∧ m=δ(n);*
***si*** *testbool* ***alors***
***pour*** *i := 1* ***jusqu'à*** *δ(n)* ***faire***
*testbool := testbool ∧ test($n_{[i]}$,$t_i$);*
*test := testbool*
***fin.***

Cet algorithme vérifie que le terme t est dénoté par le sous-graphe avec racine n. Si le terme est un atome, l'algorithme vérifie simplement que l'étiquette de n est bien t, et que n est un nœud terminal (sans fils). Quand t est de la forme $f(t_1,\ldots,t_m)$, si l'opérateur principal f est l'étiquette de n et le nombre d'arguments de f est égal au nombre de fils de n, alors le test est effectué récursivement pour tous les fils. Finalement, si l'étiquette de n est "≡", alors il faut vérifier qu'au moins un des fils de n satisfait le test.

Nous conseillons au lecteur de suivre l'algorithme sur l'exemple suivant: Soient les hypothèses: p = n • z, z = cdr(x), ord(p) et le but ord(n • cdr(x)). La solution par remplacement équationnel nécessite deux remplacements. Notre algorithme commence par la construction du graphe suivant, qui représente les hypothèses:




L'appel de l'algorithme est test(n1,ord(n•cdr(x))). Le lecteur n'aura pas de problème pour suivre l'algorithme sur le dessin.

Du point de vue de l'IA, le traitement des égalités par la fermeture de congruence et notre extension, sont tous les deux des formes de raisonnement en chainage-avant, aussi appelé "propagation de contraintes". Dans notre cas, cela est efficace parce que la propagation termine toujours et parce qu'il existe des algorithmes très efficaces pour la calculer.

## 5.6.- Caractérisation des théorèmes nécessaires à la dérivation d'hypothèses

Dans cette section nous allons étudier les critères permettant de caractériser les théorèmes nécessaires à la dérivation d'hypothèses, ainsi que leur expression dans le LIBT.

La démarche suivie pour caractériser les théorèmes nécessaires est d'utiliser des *critères de sélection* combinés à l'aide de *stratégies*. Il est également possible de combiner les stratégies pour former des stratégies *composées*.

Les critères de sélection de théorèmes ont été discutés dans la section §4.4.1; ils sont exprimés par les prédicats du LIBT que nous avons présentés.

Rappel.- Ces critères de sélection sont:

1.- Unification avec un schéma de théorème;
2.- Unification de sous-termes des théorèmes, buts et hypothèses;
3.- Opérateurs intervenant dans le théorème;
4.- Types intervenant dans le théorème;
5.- Critère de pertinence du théorème.

Nous allons d'abord présenter des stratégies ponctuelles ("simples") de sélection de théorèmes, et ultérieurement nous discuterons les façons de les combiner.

### 5.6.1.- Les stratégies simples de sélection de théorèmes

*Stratégie de l'applicabilité au but*

Un critère syntaxique fondamental pour caractériser les théorèmes qui devront être incorporés au processus de dérivation d'hypothèses est celui de *l'applicabilité*.

Nous dirons que le théorème T est applicable à un but $\mathfrak{B}$ si $\mathfrak{R}$ permet d'utiliser T pour transformer $\mathfrak{B}$. Nous allons examiner plus en détail le critère d'applicabilité pour les règles du système concernant les remplacements.[1]

Règle RWS.- Un théorème G = D est applicable au but $\mathfrak{B}[X]$ par RWS si $G\theta = X$ pour une substitution $\theta$. Exemple: Le théorème vrai $\wedge$ P = P est applicable au but Q $\vee$ (vrai $\wedge$ ord(l)). Dans le LIBT, la requête d'un théorème applicable par RWS est de la forme:

```
sousTerme(X,B), axiomeExp(X = D,Presentation).
```

Règle DUP.- Un théorème $G \cong D$ où $\cong \in \{=, \Leftrightarrow, \Rightarrow\}$ est applicable à $\mathfrak{B}[X]$ par DUP si $D\theta$ = X. Exemple: le théorème ord(a•x) $\Rightarrow$ ord(x) est applicable au but less(k,m) $\wedge$ ord(m), avec $\theta$ = {P $\leftarrow$ ord(l)}. La requête d'un théorème applicable par DUP est d'une des formes suivantes:

---

[1] Ce sont les règles qui font appel aux théorèmes.

```
sousTerme(X,𝔅), axiomeExp(G = X, Presentation).
sousTerme(X,𝔅), axiomeExp((G then X) = true,
    Presentation).
sousTerme(X,𝔅), axiomeExp((G iff X) = true, Presentation).
```

Règle CER.- Un théorème C ⇒ (G = D) est applicable à 𝔅[X] par CER si Dθ = X. Exemple: le théorème ord(x) ⇒ (min(x) = car(x)) est applicable au but car(l) > a. La requête correspondante est de la forme:

```
sousTerme(X,𝔅), axiomeExp((C then (G = X)) = true, Presentation).
```

Remarque.- En prenant en compte la commutativité des équations, l'applicabilité de RWS et celle de DUP pour l'égalité sont équivalentes.

*Stratégies d'unification avec les hypothèses*

Pour les règles DUP et CER, un critère de sélection est l'utilisation effective des hypothèses: les théorèmes sont introduits pour permettre l'interaction des buts et des hypothèses, en fournissant les "maillons manquants".

Pour la règle CER, nous demandons que la précondition C du théorème soit unifiable avec une hypothèse, c'est-à-dire, si $h$ est une des hypothèses, la requête de théorème serait: `axiomeExp((`$h$` then (G = D)) = true, Presentation).`

Remarques:

1.-Une égalité conditionnelle peut s'appliquer autrement que par la règle CER. En effet, la règle FIH (modus ponens) permet d'appliquer les égalités conditionnelles, lesquelles sont sélectionnées par un critère d'unification de la précondition C avec une hypothèse.

2.-La condition sur l'hypothèse est restrictive dans le sens où C doit être directement unifiable avec une des hypothèses, et non pas seulement déductible de celles-ci.

Pour les théorèmes de la forme G ⇒ D ou G ⇔ D, appliqués par la règle DUP, un critère de sélection consiste à exiger qu'une des hypothèses $h$ s'unifie avec un prédicat apparaissant dans G. Ceci peut s'exprimer simplement par la requête du LIBT: `sousTerme (`$h$`, G)`. Un tel critère n'est évidemment pas applicable au remplacement par l'égalité.

*Stratégies basées sur les occurrences d'opérateurs.*

Il est possible de décrire le lien établi entre le but et les hypothèses de façons moins restrictives que par les stratégies d'applicabilité au but et d'unification avec les hypothèses.[1]

Nous proposons une stratégie consistant à exiger que le théorème contienne simultanément des opérateurs d'un but et des hypothèses. Les requêtes de cette stratégie sont de la forme:

```
axiomeExp(T,P), operateurDansTerme(σ1,T),
operateurDansTerme(σ2,T),...
```

[1] L'intérêt de cela sera mis en évidence au §5.6.3.

La plus simple des possibilités consiste à prendre un opérateur du but et un opérateur d'une des hypothèses pour $\sigma_1$ et $\sigma_2$ respectivement. Il reste le problème de choisir $\sigma_1$ et $\sigma_2$ parmi tous les opérateurs d'un but et d'une hypothèse. Nous pouvons éliminer du choix les connectives logiques, les égalités, et les constructeurs et les sélecteurs du type, en suivant le critère que ce sont des opérateurs "auxiliaires". Par exemple, si nous avons le but less(car(x),cdr(x)) et l'hypothèse ord(x), une (la seule) requête sera:

```
axiomeExp(T,P),operateurDansTerme(less,T),
operateurDansTerme(ord,T).
```

*Stratégies basées sur le typage*

Nous disons qu'un type S concerne un théorème à chercher s'il intervient dans un but B et dans une des hypothèses H, dans le sens où un de leurs sous-termes est de type S. Les requêtes de ce critère sont de la forme:

```
termeDuType(S,B), termeDuType(S,H), theoremeDuType(S,T).
```

La caractérisation des théorèmes par les types de données y intervenant est encore plus faible que celle des occurrences des opérateurs.

*Stratégies basées sur le critère de pertinence*

La notion de "caractère" ou "critère de pertinence" des théorèmes a été discutée au §4.4.1. Parmi les prédicats de sélection définis, nous allons utiliser pour la dérivation d'hypothèses `theoremeSimplif(Theoreme, Presentation)` et a `xiomeDefConstruc(Axiome, Operateur, Presentation)`. Le premier est complémentaire du critère de l'applicabilité (voir §5.6.2), tandis que le second indique que pour mettre en rapport un but et une hypothèse d'induction il est souvent nécessaire d'utiliser des définitions inductives. Exemple: pour utiliser l'hypothèse `ord(l)` sur le but `ord(a•l)` il faut une définition constructive de `ord` comme `ord(a•l) = less(a,l)` $\wedge$ `ord(l)`, où `less(a,l)` teste si `a` est inférieur ou égal à tout élément de `l`.[1]

## 5.6.2.- Les stratégies composées de sélection de théorèmes

L'utilisation combinée des stratégies "simples" décrites au §5.6.1 peut s'avérer utile pour restreindre davantage la sélection de théorèmes ou bien pour améliorer l'efficacité de l'exécution des requêtes.

*Combinaison Applicabilité-Critère de pertinence*

La recherche de théorèmes à appliquer par la règle de simplification RWS doit se restreindre aux

---

[1] La définition de less est:

less(x,nil = true

less(x,y•l) = x≤y $\wedge$ less(x,l).

théorèmes qui simplifient effectivement, appelés "théorèmes de simplification" au §4.4.1. Ceci justifie l'enchainement des stratégies d'applicabilité au but, pour RWS, et du critère de pertinence des théorèmes, pour les théorèmes de simplification. Une requête combinée, pour un but $\mathfrak{B}$, peut être la suivante: `sousTerme(X,𝔅), theoremeSimplif(X=D,P).`

Remarques:

- Si nous écrivons à la suite, d'abord la requête pour l'applicabilité: sousTerme(X,$\mathfrak{B}$), axiomeExp(X=D,P), et celle du critère de pertinence: theoremeSimplif(X=D,P), alors le prédicat axiomeExp(X=D,P) est redondant.
- L'ordre dans lequel les stratégies sont appliquées n'affecte pas le résultat, mais les performances peuvent changer.[1]

*Combinaison Applicabilité-unification avec les hypothèses*

Ces deux stratégies sont naturellement combinées pour l'application des règles DUP et CER. On a des requêtes comme la suivante, pour un but $\mathfrak{B}$ et une hypothèse $\mathfrak{H}$:

```
sousTerme(X,𝔅), axiomeExp((G then X) = true, P), sousTerme(ℌ,G).
```

Remarque.- La quantité de combinaisons de sous-termes de $\mathfrak{B}$ et des hypothèses $\mathfrak{H}$ peut être assez élevée, ce qui fait augmenter la combinatoire dans l'exécution de ces requêtes. Il faudrait être à même de sélectionner les sous-termes "importants" du but, plutôt que tous ses sous-termes.

*Combinaisons avec le typage*

La stratégie de sélection par typage peut être utilisée comme un premier filtre pour améliorer les performances des autres stratégies (applicabilité au but, unification avec les hypothèses, occurrences des opérateurs). Cette amélioration est due au fait que le prédicat theoremeDuType utilise la structure de la BT pour restreindre l'espace de recherche. Par exemple, soient les requêtes:

**(a)** `theoremeDuType(natSuc,T), operateurDansTerme(+,T), operateurDansTerme(-,T).`

**(b)** `axiomeExp(T,P), operateurDansTerme(+,T), operateurDansTerme(-,T).`

La requête (a) est plus efficace que (b), car elle permet d'éviter le test sur les opérateurs pour un grand nombre de théorèmes des types non concernés (4,5 s et 1,35 s respectivement).

## 5.6.3.- Choix parmi les stratégies de sélection de théorèmes

Au §5.6.1 et §5.6.2 nous avons présenté des stratégies différentes permettant toutes de caractériser les théorèmes nécessaires. Maintenant il faut les situer les unes par rapport aux

---

[1] Par exemple, la requête sousTerme(X,(ord(l) and true)),theoremeSimplif(X=D,P) a permis de trouver les solutions en un dixième du temps nécessaire à theoremeSimplif(X=D,P),sousTerme(X,(ord(l) and true)), soit 0,23 et 2,33 s respectivement.

autres, pour pouvoir discerner quand il faut en appliquer une plutôt qu'une autre. Ceci est particulièrement nécessaire du moment où plusieurs stratégies se chevauchent: ainsi, il est souvent possible d'utiliser soit la stratégie des occurrences d'opérateurs, soit celles utilisant l'unification (applicabilité au but et unification avec les hypothèses), ou même une combinaison des deux.

## *La sélectivité*

Le critère le plus important permettant de diriger l'application des stratégies est celui de la *sélectivité*. Il est défini de la façon suivante: une stratégie $S_1$ est plus sélective qu'une stratégie $S_2$ si l'ensemble de réponses aux requêtes de $S_1$ est un sous-ensemble de celui de $S_2$. Ainsi, les relations suivantes sont établies:

- Les stratégies issues de l'unification (applicabilité au but et unification avec les hypothèses) sont plus sélectives que celles d'occurrences d'opérateurs;

- Ces dernières sont, à leur tour, plus sélectives que celles de typage.

Cette relation d'ordre peut être utilisée pour choisir laquelle, parmi ces stratégies, il faut appliquer dans une situation donnée.

En effet, nous pouvons utiliser en première instance la stratégie la plus sélective, et seulement si aucun nouveau théorème n'est trouvé répondant à ce critère, la stratégie suivante dans l'ordre est employée.[1] Par "nouveau théorème" nous voulons dire qu'il est possible qu'une requête donnée ait déjà été utilisée, mais le théorème reçu en réponse par le système de dérivation d'hypothèses ne s'est pas avéré suffisant pour résoudre le problème en cours de solution. L'idée est d'utiliser la stratégie la plus sélective possible tant qu'elle continue à nous approvisionner de théorèmes; dans le cas contraire nous sommes obligés de nous rabattre sur la stratégie en dessous.

---

[1] Il est toujours possible de combiner la stratégie du typage avec une des stratégies en haut de la hiérarchie, pour améliorer les performances, voir §5.6.2.

# CONCLUSION

Notre travail s'inscrit dans le cadre d'une démarche déductive pour la synthèse assistée de programmes. Par rapport à d'autres démarches déductives, ce qui distingue la nôtre est qu'elle s'appuie sur des stratégies "guidées par le but": à partir des choix de lignes générales effectués par l'utilisateur, le système essaie de trouver ce qu'il faudrait pour que ce choix soit valable, c'est-à-dire, qu'il conduise effectivement à la construction d'un programme.

Dans cette thèse nous nous sommes occupés d'un des aspects décisifs permettant la mise en place d'une telle démarche: ce sont les connaissances sur le domaine d'application auquel le problème donné fait référence.

La prise en compte des domaines d'application dans le processus de construction de programmes a été jusqu'à présent très partielle, le typage dans les langages de programmation étant un pas très important dans cette direction. Néanmoins, faire intervenir des propriétés plus complexes des données dans le processus de construction, en particulier des théorèmes valides dans un modèle considéré, présente des difficultés qu'il n'est pas trivial de surmonter. En effet, pour contrer l'explosion combinatoire qui résulte de la quantité infinie de théorèmes dans les domaines définis inductivement, il faut être à même d'extraire les propriétés "intéressantes" de la masse de formules non significatives vis-à-vis d'un problème donné.

Pour avancer dans la solution de ces problèmes, nous avons fait dans cette thèse des propositions appartenant essentiellement à trois catégories.

1) Une démarche permettant de caractériser les connaissances sur le domaine d'application nécessaires pour résoudre un problème donné;

2) Des éléments pour la conception d'un outil d'aide à la programmation assistée, à savoir, une Base de Connaissances sur les domaines d'application;

3) Des propositions concernant le module de déduction de la BC.

Nous discutons ci-dessous chacun de ces aspects.

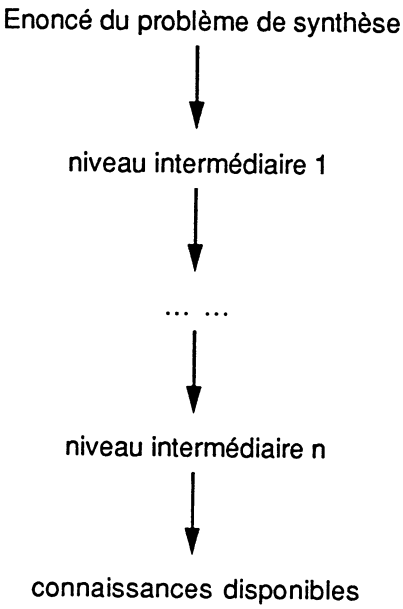
## *1) Caractérisation des connaissances nécessaires.*

Nous avons développé une méthode permettant de caractériser les connaissances nécessaires vis-à-vis d'un problème donné. C'est le prolongement naturel de notre démarche guidée par le but dans la question de l'utilisation des connaissances.

En effet, la plupart de méthodes de synthèse font une utilisation des connaissances en "chaînage avant": les programmes sont dérivés en introduisant des connaissances *ad hoc*, astucieusement choisies, au cours de la synthèse. Les inconvénients de cette démarche ont été discutés au §1: il est nécessaire de connaître *a priori* les connaissances nécessaires, puisque la seule caractérisation (implicite) des "bonnes connaissances" qui découle de ces approches est basée sur le critère d'une synthèse réussie. L'automatisation d'un tel critère est évidemment impossible: il serait nécessaire de faire la synthèse pour savoir si on peut la faire!

Il est donc nécessaire de donner une meilleure caractérisation des connaissances requises pour pouvoir ensuite les chercher dans un volume important de connaissances disponibles, ou bien les déduire. Par "caractériser les connaissances" nous voulons dire donner des indices plus généraux que les connaissances elles mêmes, tels que leur forme syntaxique ou leur type, etc.

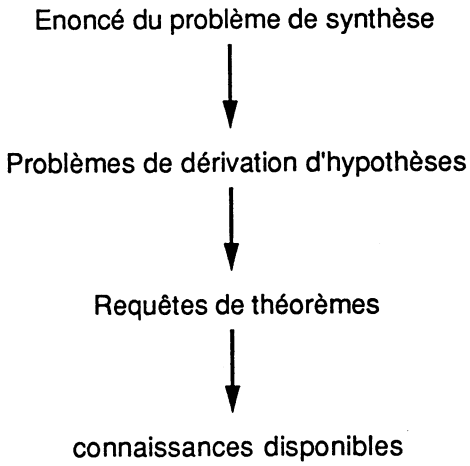
Dans notre démarche guidée par le but, cette caractérisation est obtenue par des raffinements successifs. C'est un processus descendant, dans le sens où il parcourt du haut vers le bas un "espace de caractérisation des connaissances". Nous plaçons en haut de cet espace l'énoncé du problème de synthèse à résoudre, et en bas, la connaissance concrète disponible. Les niveaux intermédiaires sont autant de formes de caractérisation, chacun étant plus détaillé que le niveau au dessus et moins détaillé que le niveau en dessous. Cet espace est représenté par la figure suivante:




L'énoncé initial caractérise partiellement la connaissance nécessaire parce qu'il permet de sélectionner les types de données concernés, et aussi parce qu'il contient des opérateurs qui pourront apparaître dans les connaissances nécessaires. Cette caractérisation est néanmoins beaucoup trop faible, car les connaissances disponibles répondant à ce critère seront (idéalement!) très nombreuses. Il est donc nécessaire d'élaborer des caractérisations

intermédiaires permettant de diriger autant que possible la recherche des connaissances.

Nous avons identifié deux étapes intermédiaires de caractérisation des connaissances, à savoir: les énoncés des problèmes de dérivation d'hypothèses, et les requêtes de théorèmes issues de l'application des stratégies de sélection de théorèmes; elles sont respectivement l'entrée et la sortie du module de dérivation d'hypothèses. Notre espace de caractérisation des connaissances prend la forme suivante:




Le caractère "guidé par le but" de notre approche est évident par le fait que chaque étape de raffinement (vers le bas) est issue des choix effectués à l'étape précédente.

Nous allons maintenant donner un aperçu récapitulatif et synthétique de cette démarche qui mène des problèmes de synthèse vers les connaissances concrètes disponibles.

## *De l'énoncé du problème de synthèse aux problèmes de dérivation d'hypothèses*

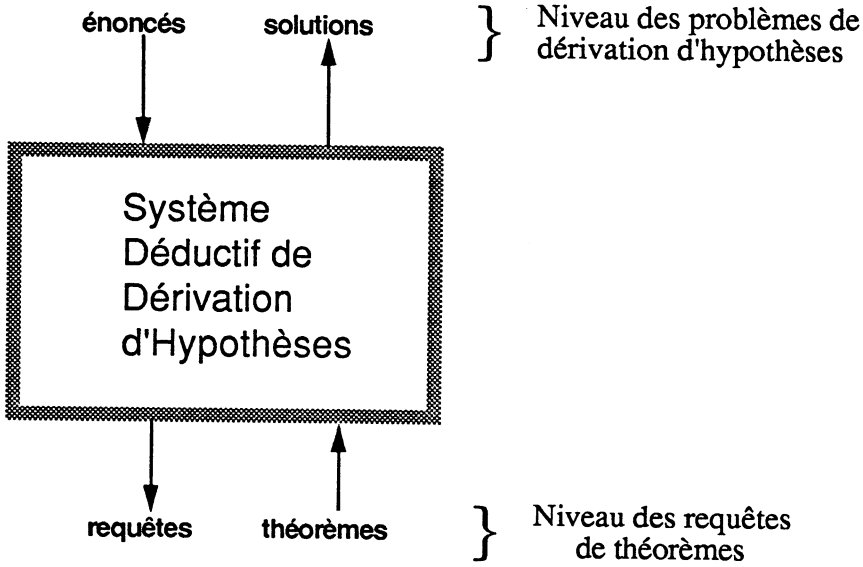
Ce passage est décrit au §2; il concerne la thèse de M-L Potet [Potet 88].

Notre approche commence par le choix, par l'utilisateur, des lignes générales de la synthèse, sous la forme d'une règle d'inférence. L'application "dirigée par le but" des règles d'inférence se fait à partir de la conclusion (l'énoncé à prouver constructivement) vers les prémisses, lesquelles sont alors partiellement instanciées, et deviennent des sous-problèmes à résoudre. Quelques-unes des prémisses font référence à des propriétés qui devront être satisfaites dans la théorie sous-jacente à l'énoncé. Si l'on ne trouve pas dans le domaine d'application des instances concrètes des propriétés requises, alors le choix de la règle d'inférence sera mis en question. La traduction du problème consistant à trouver des instances pour une de ces prémisses en un énoncé de problème de dérivation d'hypothèses est directe et entièrement syntaxique.

## *Des problèmes de dérivation d'hypothèses aux requêtes de théorèmes*

L'intervention des connaissances est ainsi repoussée à la solution des problèmes de dérivation d'hypothèses. En fait, un énoncé <B,H,V> de dérivation d'hypothèses est lui-même une forme de caractérisation des connaissances nécessaires dans le sens où l'on cherche un théorème de la

forme $H \wedge P \Rightarrow B$ tel que P vérifie $var(P) \subseteq V$. Si un tel théorème existe parmi les connaissances disponibles, la solution au problème est immédiate, mais en général ce ne sera pas le cas. Soit, par exemple, le problème $B = x \leq y \wedge y < z$, $H = y < z$, $V = \{x,y\}$; alors une solution évidente est $P = x \leq y$. Mais il n'est pas du tout raisonnable de stocker explicitement des théorèmes du style $x \leq y \wedge y < z \Rightarrow x \leq y \wedge y < z$ dans la BC ! Il est donc nécessaire de raffiner davantage la caractérisation des connaissances nécessaires.
Dans notre approche, un système déductif est utilisé pour résoudre les problèmes de dérivation d'hypothèses. Il reçoit du niveau au dessus des énoncés de problèmes de dérivation d'hypothèses et renvoie les solutions obtenues; il transmet au niveau en dessous des caractérisations des théorèmes nécessaires et reçoit en réponse ces théorèmes. Cette architecture est montrée dans la figure suivante:




La participation des théorèmes est nécessaire à l'élaboration de la solution des problèmes de dérivation d'hypothèses parce qu'en général les hypothèses ne suffisent pas, à elles seules, pour obtenir une solution. L'aide des théorèmes porte, soit sur la possibilité de simplifier des buts, soit sur la possibilité de mettre en rapport le(s) but(s) et les hypothèses, pour permettre à ces dernières d'être effectivement utilisées. Des stratégies ont été conçues pour traiter chacun de ces deux cas. Chaque stratégie apporte un critère différent de sélection de théorèmes.

L'application de certains de ces critères permet d'engendrer des schémas syntaxiques de théorèmes, lesquels pourront par la suite être instanciés par des formules spécifiques contenues dans la Base de Théorèmes. D'autres critères sont relatifs aux opérateurs qui devront apparaître dans le théorème recherché, ou bien portent sur le typage ou sur un "critère de pertinence" du théorème. Les différentes stratégies sont combinées pour former la caractérisation du théorème à chercher. Cette caractérisation est exprimée dans un "Langage d'Interrogation de la Base de Théorèmes" (LIBT), sous la forme d'une conjonction de prédicats, chaque prédicat exprimant un critère de sélection de théorèmes. Ces conjonctions sont les "requêtes de théorèmes".

## *Des requêtes aux théorèmes spécifiques*

La dernière étape à franchir dans l'espace de caractérisation des connaissances est la confrontation des requêtes avec le contenu concret de la BT, confrontation à l'issu de laquelle un ensemble (pouvant être vide) de théorèmes satifaisant la requête est rendu par la BT au

module de dérivation d'hypothèses. Le problème dans cette dernière étape consiste à effectuer une recherche (efficace) des informations demandées. Plusieurs mécanismes de recherche de théorèmes à partir de leur caractérisation ont été étudiés.

C'est dans cette démarche de caractérisation des connaissances par un processus de raffinements successifs guidé par le but, que nous plaçons la contribution principale de cette thèse. Cet aspect de notre travail a été présenté dans [Brena, Potet 89].

## *2) Conception d'une Base de Connaissances*

L'utilité d'une bibliothèque contenant des informations sur les domaines d'application utilisés couramment en programmation a été préconisée par [Goguen 85]. Nous avons essayé d'avancer dans la conception d'un tel outil pour la synthèse assistée de programmes.

En suivant une démarche descendante, à partir des fonctionalités attendues de la BC, nous avons défini le contenu et la structure de celle-ci.

Le contenu, c'est-à-dire, les connaissances proprement dites, sont essentiellement des sous-ensembles finis d'une théorie du premier ordre. L'ensemble de formules stockées n'est donc pas réduit aux axiomes de la théorie, et inclut un (grand) nombre de lemmes "pertinants" vis-à-vis de la solution de problèmes de dérivation d'hypothèses. C'est dans ce sens que notre approche peut être caractérisée comme *"basée sur la connaissance"*, proche donc des systèmes experts, l'expertise étant le choix "judicieux" des formules devant être stockées dans la base. Cette appréciation est étayée par le fait que le système déductif (pour la dérivation d'hypothèses) ne cherche pas la complétude (du moment où l'induction fait défaut), mais se borne à appliquer les connaissances disponibles pour résoudre les problèmes de dérivation de préconditions. Ces problèmes ne seront donc résolus que si l'on dispose des connaissances nécessaires.

Un critère essentiel pour la conception de la BC a été de permettre une augmentation importante de sa taille tout en conservant l'efficacité dans la recherche des informations et la compréhension intuitive de la connaissance stockée. Pour cela, nous avons souligné la nécessité de *structurer* les connaissances. L'organisation retenue est basée sur des unités modulaires, les présentations, au dessus desquelles un ensemble de liaisonss entre les présentations est défini, où l'on distingue la déclaration *contient* qui permet d'utiliser des présentations déjà existantes pour en définir de nouvelles, ceci de façon ascendante.

Le composant principal de la BC, la Bibliothèque de Théories (BT) a fait l'objet d'une maquette en Prolog. Cette maquette (décrite dans l'annexe 2) nous a permis de donner une validation pratique à nos propositions concernant la BT, ainsi que d'effectuer certains tests de performance relatifs à la recherche des informations à partir de leur caractérisation (requêtes). Nous avons ainsi vérifié la grande influence de la structure des connaissances sur l'efficacité de leur recherche. Cette maquette peut également être utilisée en tant qu'"*outil de navigation*", pour la consultation directe des connaissances disponibles par l'utilisateur. Ces facilités pour l'interaction avec l'utilisateur sont très importantes dans une approche ou la synthèse est *assistée.*

Dans la partie "ce qui reste à faire", nous voudrions mentioner tout d'abord la réalisation d'un prototype complet de la Base de Connaissances (en fait, de tout le système de synthèse, mais il

se trouve que notre thèse n'est concernée que par l'aspect *connaissances*). Il serait ainsi possible d'effectuer une quantité d'expériences impossibles à réaliser pour des exemples "faits à la main". La pertinence des formules contenues dans la BT trouverait alors une validation.

Une autre possibilité très intéressante qu'il reste à étudier est relative à une utilisation plus flexible des formules stockées dans la BT. En effet, les schémas de théorèmes des requêtes doivent "coller" exactement avec les formules stockées, et la moindre nuance syntaxique, telle que l'ordre des arguments d'un opérateur commutatif, suffit pour faire échouer le filtrage. La possibilité de considérer un ensemble de transformations élémentaires R capables d'engendrer des variantes valides des formules stockées, ou d'effectuer le matching *modulo R*, a été évoquée au §3.2. Ces transformations prendraient en compte la commutativité et l'associativité des opérateurs (comme c'est le cas du filtrage associatif-commutatif [Stickel 75]), ainsi que des variantes propositionnelles des théorèmes (p.ex., obtenir $(\gamma \wedge \neg x) \Rightarrow \neg y$ à partir de $(\gamma \wedge y) \Rightarrow x$), et le changement des constructeurs par des sélecteurs (p.ex., obtenir $(l \neq nil \wedge ord(l)) \Rightarrow ord(cdr(l))$ à partir de $ord(x \bullet l) \Rightarrow ord(l)$ ). La fermeture de l'ensemble des formules de la BT par R produirait une Base Etendue (BE) de théorèmes. Les théorèmes dans BT - BE ne seraient pas introduits par l'utilisateur; ils seraient pris en compte automatiquement par le système. Ce genre d'extension des informations disponibles en utilisant des formes limitées d'inférence est typique des Bases de Données Déductives [Gallaire et al. 84].

Il reste également à approfondir les possibilités d'utilisation d'exemples d'entrée/sortie comme une forme de description des opérateurs connus (cf. §3.3.1.10).

### *3) Propositions concernant la dérivation déductive d'hypothèses*

Nous avons d'abord donné une définition formelle du problème de la dérivation d'hypothèses, comblant ainsi un vide laissé par la définition "heuristique" de [Smith 82]. Ensuite, en reprenant le caractère constructif de notre définition, nous avons ébauché une méthode "approximative" de dérivation d'hypothèses par résolution.

Nous avons également fait des propositions autour du système de déduction naturelle de [Smith 82] pour la dérivation d'hypothèses. Une version "parallélisée" de ce système, utilisant une représentation des formules par des graphes, a fait l'objet d'une publication [Brena 88]. Dans le contexte de cette représentation des formules, nous avons proposé une extension du mécanisme de propagation des égalités basée sur la fermeture de congruence de [Nelson, Oppen 80]. En modifiant le graphe représentant le résultat de la fermeture de congruence, nous avons élargi l'ensemble de formules dénotées par ce graphe. Appliquée au système de déduction naturelle, cette extension élimine la nécessité d'effectuer des remplacements équationnels.

Toutes ces propositions restent à l'état conceptuel, et elles attendent donc d'être réalisées et testées dans la pratique.

Nous voudrions conclure avec une considération très générale: Depuis plus d'une dizaine d'années on parle d'une "crise du logiciel", évoquée au §1, caractérisée par un coût trop élevé et une fiabilité trop réduite. On dit que le développement de logiciel est la seule profession où l'on vous paie pour passer 70% de votre temps à corriger les erreurs que vous avez introduites vous-même. L'aide à la programmation sous la forme d'environnements de développement où l'ordinateur prend en charge une partie du travail pourrait améliorer cette situation. En

particulier, les méthodes formelles de construction de programmes pourraient contrer les problèmes de fiabilité. Néanmoins, nous croyons qu'une faiblesse chronique des environnements de programmation existants ou proposés jusqu'à présent est l'absence d'une prise en compte adéquate des domaines d'application. En effet, dans le cas des environnements "généraux" d'aide à la programmation on a toujours supposé que l'aide devrait se placer sur le plan de la programmation "pure", en faisant abstraction des domaines d'application. Le programmeur doit alors apporter *toute* la connaissance relative à l'application, toujours à partir de zéro, ce qui élève énormement les coûts de développement. Dans les environnements "verticaux", ou "générateurs d'application" (en réalité, configurateurs de logiciel), la connaissance sur le domaine d'application existe dans le système, mais elle est mise sous une forme procédurale et inextricablement mélée au code, ce qui rend impossible sa récupération pour d'autres problèmes, même dans un même domaine d'expertise. L'approche présentée dans cette thèse permet de considérer de façon explicite et modulaire les connaissances des domaines d'application, au moins dans le cadre de la synthèse déductive. Nous sommes convaincus que c'est dans cette direction qu'il faut avancer. Comme R. Waters l'a dit au Colloque en "Artificial Intelligence and Software Engineering" (Exeter, 1989), nous aimerions voir d'ici dix ans des scènes comme la suivante: *"Je n'arrive pas à intégrer le module de connaissances en statistique qu'on vient d'acheter, au module de connaissances en comptabilité, et ce programme doit être engendré pour demain!"*.

# ANNEXES

# Annexe 1.- Un exemple de BT

Nous présentons ici notre exemple de BT, sur la version Prolog avec laquelle nous avons effectué les tests décrits dans les §4 et §5. Nous avons donc écrit cette BT dans le but de rendre ces tests possibles, mais également pour donner un aperçu intuitif de ce que serait une BT remplissant les fonctions décrites aux chapitres précédents. Il faut toutefois garder en tête qu'une réalisation en "grandeur nature" serait peut-être dix fois plus importante.

Le lecteur averti notera que la question du traitement des exceptions n'a pas été considérée dans cet exemple de BT , pour éviter de distraire l'attention des points que nous voulons mettre en valeur, notamment la structuration de la BT.

## Listing commenté de la BT

Ce listing a été obtenu à partir de la version Prolog de la BT en exécutant le but:

presentation(P),displayPres(P),fail.

Ensuite, nous avons édité ce listing, en changeant l'ordre des présentations et/ou des opérateurs et des équations, ainsi qu'en remplaçant les noms de variables engendrés par le système (du genre _475, _290, etc.) pour des noms plus lisibles (comme S,Y, etc.).

*Les booléens*

Nous commençons avec la présentation du type booléen, qui est utilisé par la suite par d'autres types et propriétés.[1]

```
type (Constructeurs) bool0
sortes  bool
operateurs
     false :    -> bool
     true :    -> bool
ftype
```

[1] Nous rappelons que dans le formalisme algébrique utilisé, les booléens sont un type comme les autres, et c'est à l'utilisateur de préserver la cohérence en évitant de collapser les classes d'équivalence des termes congrus avec la constante "vrai" avec ceux congrus avec "faux". Dans le formalisme de la logique, par contre, les valeurs de vérité "vrai" et "faux" appartiennent au méta-langage [Kleene 67].

```
type (Definition Constructive) opsBool
contient bool0
operateurs
     iff : [bool,bool] -> bool
     and : [bool,bool] -> bool
     not : [bool] -> bool
     or : [bool,bool] -> bool
     then : [bool,bool] -> bool
axiomes
     (not false)=true
     (not true)=false
     (X iff Y)=((X then Y) and (Y then X))
     (X and false)=false
     (X and true)=X
     (X or false)=X
     (X or true)=true
     (U then V)=(not U or V)
satisfait
     commonoide[t -> bool, & -> and, z -> true]
     commonoide[t -> bool, & -> or, z -> false]
     egalite[t -> bool, = -> iff]
     idempotence[t -> bool, & -> and]
     idempotence[t -> bool, & -> or]
     ordrePartiel[t -> bool, =< -> then]
ftype
```

*Les propriétés*

Rappel.- La sémantique des présentations de "type" est relative au modèle initial, alors que celle des "propriétés" est relative à la classe des modèles. Quant un type est "contenu" par une propriété (déclaration *contient*), on considère que la classe de modèles de celle-ci se restreint à ceux qui respectent la sémantique initiale des opérateurs du type utilisés.

```
propriete comm
sortes  r, t
operateurs
     & : [t,t] -> r
axiomes
     (X & Y)=(Y & X)
fpropriete

propriete assoc
sortes  t
operateurs
     & : [t,t] -> t
axiomes
     (X &(W & R))=((X & W)& R)
fpropriete

propriete idempotence
sortes  t
operateurs
     & : [t,t] -> t
axiomes
     (X & X)=X
fpropriete
```

```
propriete monoide
contient assoc
operateurs
     z :   -> t
axiomes
     (X & z)=X
     (z & X)=X
fpropriete

propriete comm1
sortes  t
operateurs
     & : [t,t] -> t
axiomes
     (Z & X)=(X & Z)
subsume  comm[r -> t]
fpropriete

propriete assocomm
contient assoc, comm1
fpropriete

propriete commonoide
contient comm1, monoide
fpropriete

propriete transitivite
contient opsBool
sortes  t
operateurs
     = : [t,t] -> bool
axiomes
     (X=W and W=R then X=R)=true
fpropriete
```

(Nous n'avons choisi d'utiliser l'opérateur "=" dans la définition de la transitivité que pour récupérer cette définition dans la propriété "égalité" ci-dessous.- Le renommage dans la déclaration "contient" n'a pas été implanté).

```
propriete egalite
contient transitivite
axiomes
     (X=X)=true
     (T=U then U=T)=true
fpropriete

propriete ordrePartiel
contient opsBool
sortes  t
operateurs
     = : [t,t] -> bool
     =< : [t,t] -> bool
axiomes
     (X=<X)=true
     (S=T iff S=<T and T=<S)=true
subsume  egalite
fpropriete
```

```
propriete ordreTotal
contient ordrePartiel
axiomes
     (U=<V or V=<U)=true
fpropriete
```

*Les types de données*

```
type (Sur-sorte) nat
sortes  nat
operateurs
     1 :   -> nat
     + : [nat,nat] -> nat
     succ : [nat] -> nat
axiomes
     succ(S)=1+S
ftype
```

Le type des entiers naturels a dans la BT plusieurs présentations contructives possibles; nous avons retenu la construction par 0 et succ, et celle par 0, 1 et +, dont les sortes sont respectivement natSuc et natPlus. Ces deux sortes sont des sous-sortes de nat, qui n'a pas, elle, des constructeurs (cf.§3). Nous avons choisi de "contenir" la présentation nat dans les présentations des sous-sortes (zero, natSuc, natPlus,...) pour avoir une racine unique pour le type des naturels; ainsi, quand il faut chercher des informations sur les naturels, on commence toujours la recherche par cette racine.

```
type (Constructeurs) zero
contient nat
sortes  zero
sous-sortes  zero<nat
operateurs
     0 :   -> zero
ftype
```

Le type zero est un sous-type des naturels; son intérêt ici est qu'on peut avoir un seul zéro pour les différentes constructions des entiers.[1]

```
type (Constructeurs) natSuc
contient zero
sortes  natSuc
sous-sortes  natSuc<nat, zero<natSuc
operateurs
     succ : [natSuc] -> natSuc
ftype
```

C'est la construction des entiers naturels par 0 et succ; le constructeur 0 est hérité de la présentation zero.

---

[1] Il n'est pas possible de surcharger l'opérateur "0" pour les sortes natSuc et NatPlus sans perdre la propriété de régularité de la signature multi-sorte, laquelle revient à dire que pour tout terme il y a un plus petit type (cf.§3 et [Goguen, Meseguer 87])

```
type (Constructeurs) natPlus
contient zero
sortes  natPlus
sous-sortes  natPlus<nat, zero<natPlus
operateurs
     1 :   -> natPlus
     + : [natPlus,natPlus] -> natPlus
axiomes
     U+V = V+U
     X+0 = X
     (V+X)+Y = V+(X+Y)
satisfait
     commonoide[t -> natPlus, z -> 0, & -> +]
ftype
```

C'est la construction des entiers naturels par 0, 1 et +. Ensuite, des définitions récursives en termes de constructeurs:

```
type (Definition Constructive) arithSuc
contient natSuc
operateurs
     * : [nat,nat] -> nat
     - : [nat,nat] -> nat
     pred : [nat] -> nat
axiomes
     pred(0)  = 0
     pred(succ(X)) = X
     W*succ(R) = W+W*R
     X*0 = 0
     T+succ(U) = succ(T+U)
     X+0 = X
     V-succ(X) = pred(V-X)
     X-0 = X
satisfait
     commonoide[t -> natSuc, & -> *, z -> succ(0)]
     commonoide[t -> natSuc, & -> +, z -> 0]
ftype

type (Definition Constructive) arithPlus
contient natPlus
operateurs
     * : [nat,nat] -> nat
     - : [nat,nat] -> nat
     pred : [nat] -> nat
axiomes
     pred(zero) = zero
     pred(1+X) = X
     X*(Y+Z) = (X*Y) + (X*Z)
     X*1 = X
     X*0 = 0
     T-(U+V) = (T-U)-V
     X-0 = X
     X+1-1 = X
satisfait
     commonoide[t -> natPlus, & -> *, z -> 1]
ftype

type arithNat
contient arithPlus, arithSuc
ftype
```

Une présentation telle que **arithNat** est utile pour disposer d'une présentation où l'on a accès

à toutes les déclarations (en particulier les axiomes) accessibles dans **arithPlus** et toutes celles accessibles dans **arithSuc**, pour les utiliser en tant qu'informations sur les entiers naturels (sorte nat).

```
type (Definition Constructive) comparSuc
contient natSuc, opsBool
operateurs
     < : [nat,nat] -> bool
     =< : [nat,nat] -> bool
     > : [nat,nat] -> bool
     >= : [nat,nat] -> bool
axiomes
     (V<X) = (not V>=X)
     (0=<X) = true
     (succ(X)=<succ(W)) = (X=<W)
     (succ(X)=<0) = false
     (R>S) = (not R=<S)
     (T>=U) = (U=<T)
satisfait
     ordreTotal[t -> nat]
     ordreTotal[t -> nat, =< -> >=]
     transitivite[t -> nat, = -> <]
     transitivite[t -> nat, = -> >]
ftype

type (Definition Constructive) comparPlus
contient natPlus, opsBool
operateurs
     < : [nat,nat] -> bool
     =< : [nat,nat] -> bool
     > : [nat,nat] -> bool
     >= : [nat,nat] -> bool
axiomes
     (V<X) = (not V>=X)
     (0=<X) = true
     (X+W=<X+R) = (W=<R)
     (1+X=<0) = false
     (R>S) = (not R=<S)
     (T>=U) = (U=<T)
satisfait
     ordreTotal[t -> nat]
     ordreTotal[t -> nat, =< -> >=]
     transitivite[t -> nat, = -> <]
     transitivite[t -> nat, = -> >]
ftype

type comparNat
contient comparPlus, comparSuc
ftype

type (Definition Constructive) divSuc
contient arithSuc, comparSuc
operateurs
     / : [nat,nat] -> nat
     mod : [nat,nat] -> nat
axiomes
     (not Y<Z then Y/Z=succ((Y-Z)/Z)) = true
     (not T<U then T mod U=(T-U) mod U) = true
     (V<X then V/X=0) = true
     (R<S then R mod S=R) = true
satisfait
     divSpec
ftype
```

*Les listes*

```
type (Sur-sorte) liste
sortes  liste
operateurs
     <+ : [t,liste] -> liste
     <+> : [liste,liste] -> liste
     unit : [t] -> liste
axiomes
     (V<+X) = (unit(V)<+>X)
ftype
```

Nous suivons ici la même logique que dans le cas des entiers, en présentant d'abord la sorte liste et ensuite les présentations des sous-sortes avec des constructeurs.

```
type (Constructeurs) listeVide
contient liste
sortes  listeVide
sous-sortes  listeVide<liste
operateurs
     nil :   -> listeVide
ftype

type (Constructeurs) listeNilCons
propriete exigee egalite
contient listeVide
sortes  listeCons
sous-sortes  listeCons<liste, listeVide<listeCons
operateurs
     <+ : [t,listeCons] -> listeCons
ftype
```

Rappel.- La notation utilisée pour la généricité indique que le type t du profil de <+ est le même que t dans la propriété éxigée (égalité).

```
type (Constructeurs) listeAppend
propriete exigee egalite
contient listeVide
sortes  listeAppend
sous-sortes  listeAppend<liste, listeVide<listeAppend
operateurs
     <+> : [listeAppend,listeAppend] -> listeAppend
     unit : [t] -> listeAppend
axiomes
     (Y<+>(Z<+>X)) = ((Y<+>Z)<+>X)
     (X<+>Y) = (Y<+>X)
     (X<+>nil) = X
     (nil<+>X) = X
satisfait
     commonoide[t -> listeAppend, z -> nil, & -> <+>]
ftype
```

```
type (Definition Constructive) selecNilCons
contient listeNilCons
operateurs
     car : [listeCons] -> t
     cdr : [listeCons] -> liste
axiomes
     car(R<+S) = R
     cdr(T<+U) = U
ftype

type (Definition Constructive) opsListeCons
contient listeNilCons, natSuc, opsBool
operateurs
     # : [liste] -> natSuc
     <+> : [liste,liste] -> liste
     contains : [liste,liste] -> bool
     last : [liste] -> t
     member : [t,liste] -> bool
     vide : [liste] -> bool
axiomes
     (#nil) = 0
     (#Y<+Z) = succ((#Z))
     last(Z<+X) = last(X)
     last(X<+nil) = X
     vide(nil) = true
     vide(T<+U) = false
     (nil<+>X) = X
     (R<+S<+>T) = (R<+(S<+>T))
     contains(Z,X<+W) = (member(X,Z) and contains(Z,W))
     contains(Y,nil) = true
     contains(nil,U<+V) = false
     member(S,T<+U) = (S = T or member(S,U))
     member(X,nil) = false
satisfait
     commonoide[t -> listeCons, & -> <+>, z -> nil]
ftype

type (Definition Constructive) opsListeAppend
contient arithSuc, listeAppend, natPlus, opsBool
operateurs
     # : [liste] -> nat
     car : [liste] -> t
     contains : [lsite,liste] -> bool
     last : [liste] -> t
     member : [t,liste] -> bool
     vide : [liste] -> bool
axiomes
     #(X<+>W) = (#X+(#W))
     (#nil) = 0
     (#unit(Y)) = 1
     car(unit(X)) = X
     car((T<+>U)) = car(T)
     last(unit(X)) = X
     last((Z<+>X)) = last(X)
     vide(nil) = true
     vide(unit(X)) = false
     vide((Z<+>X)) = (vide(Z) and vide(X))
     contains(R,(S<+>T)) = (contains(R,S) and contains(R,T))
     contains(W,nil) = true
     contains(nil,unit(Y)) = false
     member(U,(V<+>X)) = (member(U,V) or member(U,X))
     member(T,unit(U)) = (T=U)
     member(X,nil) = false
ftype
```

```
type opsListe
contient opsListeAppend, opsListeCons
ftype

type (Definition Constructive) opsListeCons2
contient listeNilCons, natSuc, opsBool
operateurs
     count : [t,liste] -> bool
     elim : [t,liste] -> liste
     reverse : [liste] -> liste
axiomes
     reverse(nil) = nil
     reverse(T<+U) = (reverse(U)<+>T<+nil)
     count(X,nil) = 0
     elim(X,nil) = nil
     (not Y=Z then elim(Y,Z<+X)=Z<+elim(Y,X)) = true
     (not S=T then count(S,T<+U)=count(S,U)) = true
     (not member(V,X) then perm(V<+X,Y)=false) = true
     (X=Y then elim(X,Y<+Z)=elim(X,Z)) = true
     (R=S then count(R,S<+T)=succ(count(R,T))) = true
     (member(U,V) then perm(U<+V,X)=perm(V,elim(U,X))) = true
ftype

type (Definition Constructive) opsListeAppend2
contient listeAppend, natPlus, opsBool
operateurs
     count : [t,liste] -> bool
     elim : [t,liste] -> liste
     reverse : [liste] -> liste
axiomes
     reverse(nil) = nil
     reverse(unit(X)) = unit(X)
     reverse((V<+>X)) = (reverse(V)<+>reverse(X))
     count(Z,(X<+>W)) = count(Z,X)+count(Z,W)
     count(X,nil) = 0
     elim(X,(Y<+>Z)) = (elim(X,Y)<+>elim(X,Z))
     elim(X,nil) = nil
     (not V=X then elim(V,unit(X)) = unit(X)) = true
     (not Y=Z then count(Y,unit(Z))=0) = true
     (T=U then elim(T,unit(U))=nil) = true
     (V=X then count(V,unit(X))=1) = true
ftype

type opsListe2
contient opsListeAppend2, opsListeCons2
satisfait
     contains
     elim
     transitivite[t -> liste, = -> contains]
ftype
```

Ici nous déclarons des propriétés satisfaites dans une présentation sans déclaration d'opérateurs ni d'axiomes; ceci est valable mais présente l'inconvénient que cette information n'est pas accessible dans les présentations en-dessous (contenues par opsListe2), qui satisfont également ces propriétés.

Les propriétés *contains* et *elim* sont en fait des spécifications déclaratives d'opérateurs (voir plus bas).

```
type (Definition Constructive) opsListeCons3
contient bool0, listeNilCons
operateurs
     perm : [liste,liste] -> bool
axiomes
     perm(nil,nil) = true
     perm(nil,U<+V) = false
satisfait
     egalite[t -> liste, = -> perm]
ftype

type (Definition Constructive) orderedListCons
propriete exigee ordreTotal
contient comparNat, opsListeCons
operateurs
     less : [t,listeCons] -> bool
     ord : [liste] -> bool
axiomes
     ord(nil) = true
     ord(X<+Y) = (less(X,Y) and ord(Y))
     less(W,R<+S) = (W=<R and less(W,S))
     less(X,nil) = true
ftype

type (Definition Constructive) orderedListAppend
propriete exigee ordreTotal
contient comparNat, opsListeAppend
operateurs
     less : [t,listeCons] -> bool
     ord : [liste] -> bool
axiomes
     ord(nil) = true
     ord(unit(X)) = true
     ord((X<+>Y)) = ((ord(X) and ord(Y)) and last(X)=<car(Y))
     less(Y,(Z<+>X)) = (less(Y,Z) and less(Y,X))
     less(X,unit(Y)) = (X=<Y)
     less(X,nil) = true
ftype

type orderedList
contient orderedListAppend, orderedListCons
satisfait
     less
ftype
```

*Les théorèmes*

Ci-dessous nous présentons des théorèmes qui ne sont pas des définitions, et dont la présence dans la BT se justifie par les *critères de pertinence* discutés au §3. En particulier, on y trouve des théorèmes de simplification et des théorèmes de lien entre opérateurs.

```
type (Theoremes) booltheo
contient opsBool
axiomes
     (not not X) = X
     (not (X and W)) = (not X or not W)
     (not (R or S)) = (not R and not S)
     (Y and (Y or Z)) = Y
     (S and (T or U)) = (S and T or S and U)
     (X and not X) = false
     (V or V and X) = V
     (T or U and V) = ((T or U) and (T or V))
     (X or not X) = true
ftype

type (Theoremes) theoCompar
contient comparNat
axiomes
     (T=<U) = (T=U or T<U)
     (succ(X)>X) = true
     (V>=X) = (V=X or V>X)
     (Y+Z>=Y) = true
ftype

type (Theoremes) theoListe
contient opsListe2, orderedList
axiomes
     last(X) = car(reverse(X))
     reverse(reverse(T)) = true
     ((#elim(S,T))>=(#T)) = true
     member(car(S),S) = true
     perm(X,reverse(X)) = true
     perm(X,cdr(X)) = false
     (ord(X) then less(car(X),X)) = true
     (ord(X) then ord(elim(W,X))) = true
     (R=S then perm(R,S)) = true
     ((#Z)=1 then car(Z)=last(Z)) = true
     (cdr(W)=nil then ord(W)) = true
     ((#Y)=<1 then ord(Y)) = true
     (less(V,X) and Y=<V then less(Y,X)) = true
     (contains(W,R) then (#W)>=(#R)) = true
     (perm(U,V) then (#U)=(#V)) = true
     member(last(X),X)
ftype
```

*Les spécifications*

Rappel.- Les spécifications déclaratives des opérateurs sont des propriétés dans le sens où la sémantique considérée est celle de la classe de modèles, alors que pour les types c'était l'initialité. Il faut faire attention à ce que les opérateurs auxiliaires intervenant dans la spécification (p.ex. "*" ci dessous) soient accessibles dans la présentation de la spécification; l'opérateur définit, par contre, ne doit pas y être accessible, car dans ce cas-là la spécification ne serait pas vraiment une définition.

```
propriete (Specification) divSpec
contient arithNat
operateurs
     / : [nat,nat] -> nat
axiomes
     (Z*X) / X = Z
fpropriete
```

```
propriete (Specification) selecAppend
contient listeAppend
operateurs
     split : [liste] -> liste
axiomes
     cdr(cdr(split(X))) = nil
     (car(split(X))<+>car(cdr(split(X)))) = X
fpropriete
```

**Split** est le sélecteur du type **listeAppend**; il fait éclater une concaténation de listes en ses deux composants. Nous n'en avons pas donné auparavant une définition récursive; étant données les équations sur les constructeurs de **listeAppend**, la spécification de **split** définit une famille d'opérateurs. Par exemple, un membre de cette famille est l'éclatement d'une liste en deux sous-listes pour lesquelles la longueur de la première est égale ou d'une unité plus grande à celle de la seconde.

```
propriete (Specification) perm
contient opsListe2
operateurs
     perm : [liste,liste] -> bool
axiomes
     perm(X,W) = (count(R,X)=count(R,W))
fpropriete

propriete (Specification) elim
contient opsListe
operateurs
     elim : [t,liste] -> liste
axiomes
     member(Z,elim(Z,L)) = false
fpropriete

propriete (Specification) tri
contient opsListeCons3, orderedList
operateurs
     tri : [liste] -> liste
axiomes
     ord(tri(X)) = true
     perm(tri(X),X) = true
fpropriete

propriete (Specification) contains
contient opsListe, opsListe2
operateurs
     contains : [liste,liste] -> bool
axiomes
     contains(T,U) = (member(V,T) then member(V,U))
fpropriete

propriete (Specification) less
contient comparNat, opsListe
operateurs
     less : [t,liste] -> bool
axiomes
     less(T,U) = (member(V,U) then T=<V)
fpropriete
```

# Annexe 2.- Une maquette Prolog de la BT

Nous avons décidé de programmer cette maquette de la BT principalement pour pouvoir effectuer des tests et valider ainsi dans la pratique les définitions de prédicats permettant l'accès aux informations (cf. §3 et §4). La maquette permet également de mesurer la performance des mécanismes de recherche d'information mis en œuvre; et de comparer l'efficacité des différentes solution possibles.

Le choix du langage Prolog se justifie surtout par un critère de rapidité de prototypage. En effet, les commandes de l'utilisateur sont reçues et analysées syntaxiquement par l'interprète Prolog lui-même, et nous n'avons fait que fournir un ensemble de définitions Prolog mises à la disposition de l'utilisateur. Nous récupérons ainsi tous les avantages de l'environnement Prolog, permettant en particulier l'ajout par l'utilisateur de ses propres définitions de prédicats pour étendre la maquette. Il a été néanmoins nécessaire de supposer l'existence d'une version "en Prolog" des présentations de la BT (cf. §4); cette BT-prolog a été engendrée "à la main" par nous-mêmes. Un outil de traduction de la BT à Prolog serait nécessaire pour travailler plus confortablement.

La version de Prolog utilisée est C-Prolog [Pereira 88], sur une station Sun-350. Cet environnement de travail est assez performant à en juger par les tests présentés au §4. Le choix du langage Prolog n'a donc pas pénalisé excessivement le niveau de performance.

## *Rappel des fonctionnalités de la maquette*

Nous rappelons très brièvement les fonctionalités de la maquette, lesquelles sont discutées plus en détail au §4 :

- Consultation directe (par l'utilisateur) des informations contenues dans la BT et de leurs liens. Un ensemble de prédicats apporte les fonctionalités d'un outil de navigation, lequel permet l'affichage des présentations (les unités modulaires à la base de la structure de la BT) et la consultation des liens existant entre ces présentations.

- Réponse aux requêtes de théorèmes. Un ensemble de prédicats met en œuvre les critères de sélection de théorèmes décrits au §4.

## *Utilisation de la maquette*

Une fois dans le système C-Prolog, il faut charger l'ensemble de définitions de prédicats constituant la maquette. Ces prédicats ont été, par facilité, répartis en plusieurs fichiers, selon le type de fonctionalité qu'ils apportent; ainsi, nous avons un fichier "relations" contenant les définitions des liens entre les présentation, "accespres" pour l'accès aux informations contenues dans les présentations, etc. Tous ces fichiers sont néanmoins appelés automatiquement en chargeant le fichier "callbt", lequel contient un but qui déclenche le chargement des autres fichiers. La session commence donc de la façon suivante:

```
C-Prolog version 1.5
| ?- [callbt].
declops consulted 0 bytes 0.0666667 sec.
terme consulted 2812 bytes 0.3 sec.
bt consulted 28192 bytes 2.98333 sec.
relations consulted 1320 bytes 0.133334 sec.
accespres consulted 1720 bytes 0.166668 sec.
display consulted 5284 bytes 0.500001 sec.
theoadeq consulted 1180 bytes 0.0833369 sec.
aide consulted 3164 bytes 0.300001 sec.
callbt consulted 43672 bytes 4.53333 sec.

yes
```

Pour savoir quel sont les prédicats disponibles, l'utilisateur peut faire appel à la fonction d'aide, appelée par le prédicat **aide** (sans arguments) :

| ?- aide.

```
                PREDICATS PREDEFINIS DISPONIBLES:

displayPres(P)
        montre sur l ecran la presentation de nom P
presentation(NomPres)
        NomPres est le nom d une presentation
type(NomType)
        NomType est le nom d une pres.de type
propriete(NomProp)
        NomProp est le nom d une pres.de propriete
typeConstructeurs(Nom)
        Nom est le nom d une pres.de cons.de type
typeDefConstructive(Nom)
        Nom est le nom d une present.constructive d operateurs.
typeTheoremes(NomType)
        NomType est le nom d une pres.de theoremes d un type
typeSurSorte(NomType)
        NomType est le nom d une pres.de type avec sous-types
proprieteSpecification(Prop)
        Prop est une specification declarative d operateur
contientExp(P1,P2)
        Dans.P1 il est declare qu elle contient P2
contient(P1,P2)
        P1 contient recursivement P2
proprieteExigee(A,P,R)
        Le type generique A est parametre par la propriete P
        renommee par R (liste de couples [[o1,n1],[o2,n2],..])
subsumeExp(P,Q,R)
        La propriete P subsume Q, laquelle est renommee par R
subsume(P,Q,R)
        La propriete P subsume recursivement Q,renommee par R
modeleExp(A,Q,R)
        Le type A satisfait la propriete Q renomme par R
modele(A,Q,R)
        Le type A est modele declare ou herite de Q

FRAPPEZ RETOUR POUR CONTINUER...
```

```
sorteExp(S,P)
       La sorte S est declaree dans la presentation P
sortePres(S,P)
       La sorte S est accessible a partir de la pres.P

sousSorteExp(S1,S2,P)
       Dans P, S1 est declaree comme sous-sorte de S2
sousSorte(S1,S2)
       S1 est (recursivement) sous-sorte de S2
operateurExp(Op,profil([s1,..,sn],s),P)
       L operateur Op avec le profil Op:s1..,sn -> s
       est declare dans la pres. P
operateurPres(Op,P)
       L operateur Op est accessible dans la pres. P
operateurDansTerme(Op,T)
       Dans le terme T il existe une occurrence de
       l operateur Op
operateurDeTerme(Op,T)
       Le terme T est de la forme Op(...)
sousTerme(T1,T2)
       Le terme T1 est un sous-terme de T2
termeDuType(S,T)
       Le terme T contient un sous-terme de sorte S
axiomeExp(A,P)
       L axiome A est declare dans la pres.P
axiomePres(A,P)
       L axiome A est accessible (recursivement) dans P
theoremeDuType(S,A)
       Le theoreme A est de sorte S
axiomeDefConstruc(A,O,P)
       L axiome A dans la pres.P est une def.rec.de l oper.O
axiomeSpecif(A,O,P)
       L axiome A dans P est une def.declarative de l oper.O
theoremeSimplif(T,P)
       L axiome T de la pres.P est un theor.de simplification

FRAPPEZ RETOUR POUR CONTINUER...

yes
```

La façon d'utiliser ces prédicats pour la consultation de la BT ou pour la recherche de théorèmes a déjà été montrée au §4.

## *Description des fichiers et prédicats constituant la maquette*

Comme nous l'avons déjà dit, chacun des fichiers chargés au début de la session groupe des définitions / déclarations / buts avec un même rôle. Ainsi, la description des fichiers constituant la maquette est une façon d'aborder la description des prédicats définis.

Fichier ***callbt***:
Contient un but permettant de charger les autres fichiers.

```
:- ([declops,terme,bt,relations,accespres,display,theoadeq,aide]).
```

Fichier ***declops***.
Contient des buts avec le prédicat prédéfini op, qui permet de déclarer des symboles opérateurs infixés ou prefixés (cf. [Pereira 88]).

```
:- op(1000,xfx,and).
:- op(1050,xfx,or).
:- op(1000,fx,#).
:- op(900,xfx,<+).
```
...etc.

Fichier ***display***:
Contient la définition du prédicat *displayPres*, qui permet d'afficher la présentation d'un nom donné, ainsi que des définitions auxiliaires. C'est le noyau des facilités fournies pour la consultation de la BT.

*La clause suivante définit le prédicat displayPres pour le cas des présentations de type.*

```
displayPres(NomPres) :-
        type(NomPres),put(12),nl,nl,nl,
        write('type '),
        (typeConstructeurs(NomPres) -> write('(Constructeurs) '); true),
        (typeDefConstructive(NomPres) ->
             write('(Definition Constructive) '); true),
        (typeSurSorte(NomPres) -> write('(Sur-sorte) '); true),
        (typeTheoremes(NomPres) -> write('(Theoremes) '); true),
        write(NomPres),nl,
        (proprieteExigee(NomPres,Prop,Renom) ->
             (write('propriete exigee '),
             write(Prop),
             writeRenom(Renom),nl);
             true),
        (contientExp(NomPres,PresContient) -> writeContient(NomPres); true),
        (sorteExp(Sorte,NomPres) -> writeSort(NomPres); true),
        (sousSorteExp(PtiteSorte,GrandeSorte,NomPres) ->
             writeSousSorte(NomPres);true),
        (operateurExp(Op,Profil,NomPres) -> writeOps(NomPres); true),
        (axiomeExp(Ax,NomPres) -> writeAx(NomPres); true),
        (modeleExp(NomPres,Propriete,Ren) -> writeMod(NomPres);true),
        write('ftype'),nl,nl,nl.
```

*Définition de displayPres pour les propriétés*

```
displayPres(NomPres) :-
        propriete(NomPres),put(12),nl,nl,nl,
        write('propriete '),
        (proprieteSpecification(NomPres) ->
             write('(Specification) '); true),
        write(NomPres),nl,
        (contientExp(NomPres,PresContient) -> writeContient(NomPres); true),
        (sorteExp(Sorte,NomPres) -> writeSort(NomPres); true),
        (operateurExp(Op,Profil,NomPres) -> writeOps(NomPres); true),
        (axiomeExp(Ax,NomPres) -> writeAx(NomPres); true),
        (subsumeExp(NomPres,Pres2,Renom) -> writeSub(NomPres); true),
        write('fpropriete'),nl,nl,nl.
```

*Cette clause affiche le mot-clé "contient" suivi des nom des présentations explicitement contenues par la présentation de nom NomPres.*

```
writeContient(NomPres) :-
        buildContientList(NomPres,ContientList),
        write('contient '),
        writeList(ContientList),nl.
```

*Formation d'une liste avec les nom des présentations contenues par NomPres.*

```
buildContientList(NomPres,ContientList) :-
        setof(PresContient,contientExp(NomPres,PresContient),ContientList).
```

*Ecriture d'une liste de termes séparés par des virgules*

```
writeList([X]) :- write(X).
writeList([X|ResteList]) :-
        write(X), write(', '),
        writeList(ResteList).
```

Des définitions similaires tiennent pour *writeSort*, *writeOps* et *writeAx*. Les cas de *writeSousSorte* et de *writeMod* sont légèrement plus compliqués; nous présentons ci-dessous ce dernier:

```
writeMod(NomPres) :-
        buildModList(NomPres,ModList),
        write('satisfait  '),nl,
        writeModList(ModList).
```

*Construction de la liste de propriétés dont NomPres est un modèle*

```
buildModList(NomPres,ModList) :-
        setof(      (Propriete,Renom),
              modeleExp(NomPres,Propriete,Renom),
              ModList).
```

*Ecriture de cette liste*

```
writeModList([]).
writeModList([X|ResteModList]) :-
        write('     '),writeJustOneMod(X),nl,
        writeModList(ResteModList).
```

*Ecriture pour une de ces propriétés*

```
writeJustOneMod((Propriete,Renom)) :-
        write(Propriete),
        writeRenom(Renom).
```

*Ecriture des renommages*

```
writeRenom(Renom) :-
        (Renom=[] -> true;
              (write('['),
              writeListRenom(Renom),
              write(']'))).
```

*Eciture d'une liste de paires de renommage* $[[a_1,n_1], \ldots,[a_j,n_j]]$, *où* $a_i$ *sont les symboles à remplacer et* $n_i$ *sont les nouveaux symboles.*

```
writeListRenom([UnRenom]) :- writeJustOneRenom(UnRenom).
writeListRenom([UnRenom|ResteListRenom]) :-
        writeJustOneRenom(UnRenom),
        write(', '),
        writeListRenom(ResteListRenom).
```

*Ecriture d'une pare de renommage*

```
writeJustOneRenom([AncienSymb,NouvSymb]) :-
        write(AncienSymb),
        write(' -> '),
        write(NouvSymb).
```

Le calcul de *writeSub* est fait de manière analogue.

Fichier ***relations***:
Contient les définitions des prédicats relatifs à la structure de la BT (cf. §3.3.2.7).

```
presentation(P) :- type(P).
presentation(P) :- propriete(P).

type(A) :- typeConstructeurs(A).
type(A) :- typeDefConstructive(A).
type(A) :- typeTheoremes(A).
type(A) :- typeSurSorte(A).

propriete(P) :- proprieteSpecification(P).[1]

contient(A,B) :- contientExp(A,B).
contient(A,B) :- contientExp(A,C), contient(C,B).

subsume(P,Q,R) :- subsumeExp(P,Q,R).
subsume(P,Q,[]) :- propriete(P), propriete(Q), contient(P,Q).
subsume(P,Q,R) :-
        subsumeExp(P,PQ,R1),
        subsume(PQ,Q,R2),
        compose(R1,R2,R).[2]

compose([],X,X).
compose([A|L],X,[A|M]) :- compose(L,X,M).

presSupersorte(A,B) :-
        typeSuperSorte(A),
        typeConstructeurs(B),
        contientExp(A,B).

modele(A,P,Renommage) :- modeleExp(A,P,Renommage).
modele(A,P,Renommage) :-
        modeleExp(A,Q,Renommage),
        subsume(Q,P).
```

Fichier ***accespres***:
Contient les définitions des prédicats permettant d'accéder aux informations à l'intérieur d'une présentation.

*Sortes accessibles dans la présentation P*

```
sortePres(S,P) :- sorteExp(S,P).
sortePres(S,P) :- contient(P,P2), sortePres(S,P2).
```

*Fermeture transitive de sousSorteExp*

```
sousSorte(S1,S2) :- sousSorteExp(S1,S2,P).
sousSorte(S1,S2) :- sousSorte(S1,S3), sousSorte(S3,S2).
```

*Opérateurs F accessibles dans la présentation P*

---

[1] Rappel.- Les propriétés qui ne sont pas des spécifications sont déclarées explicitement, comme par exemple propriete(comm), pour la propriété "comm" (commutativité).

[2] "compose" est un prédicat auxiliaire qui calcule la composition de deux renommages comme leur concaténation.

```
operateurPres(F,P) :- operateurExp(F,Profil,P).
operateurPres(F,P) :-
        operateurExp(F,Profil,P2),
        contient(P,P2).
```

*Le prédicat axiomePres donne les équations FBF accessibles dans une présentation P. Nous avons d'abord les équations propres à P et l'héritage des équations des présentations contenues par P:*

```
axiomePres(FBF,P) :- axiomeExp(FBF,P).
axiomePres(FBF,P) :- contient(P,P2),axiomePres(FBF,P2).
```

*Ensuite nous y ajoutons les équations des propriétés satisfaites par P, en changeant les noms des symboles d'après les renommages*

```
axiomePres(FBF,P) :-
        modele(P,Q,Renommages),
        axiomeExp(FBFprop,Q),
        renommePlus(FBFprop,Renommages,FBF).
```

*Finalement, les trois clauses suivantes permettent d'y rajouter les équations de la subsitutivité des opérateurs déclarés dans P*

```
axiomePres(FBF,P) :-
        operateurExp(Op,profil([S1],S2),P),
        TermeX=..[Op,X],
        TermeY=..[Op,Y],
        FBF=((X=Y) then (TermeX=TermeY)).
axiomePres(FBF,P) :-
        operateurExp(Op,profil([S1,S3],S2),P),
        TermeX=..[Op,X,Z],
        TermeY=..[Op,Y,Z],
        FBF=((X=Y) then (TermeX=TermeY)).
axiomePres(FBF,P) :-
        operateurExp(Op,profil([S1,S3],S2),P),
        TermeX=..[Op,Z,X],
        TermeY=..[Op,Z,Y],
        FBF=((X=Y) then (TermeX=TermeY)).
```

Fichier *terme*:
Contient des définitions de prédicats de bas niveau, tels que tester qu'un terme est un sous-terme d'un autre.

```
sousTerme(X,T) :- (var(X),(\+testSousTerme(X,T))) ->
                  genSousTerme(X,T);
                  funcSousTerme(X,T).[1]
```

*Le terme T contient une occurrence de l'opérateur Op.*

```
operateurDansTerme(Op,T) :- atom(Op),testSousTerme(Op,T).
operateurDansTerme(Op,T) :-
        sousTerme(S,T),(funcTerm(S) -> functor(S,Op,_)).
```

*Le terme F est de sorte S. Ce prédicat est calculé par examen des profils des opérateurs*

---

[1] Il est nécessaire de différencier les cas où l'on veut tester si un terme donné est un sous-terme d'un autre, et le cas où l'on veut engendrer les sous-termes d'un terme donné. Le premier cas est pris en compte par funcSousTerme, alors que le second l'est par genSousTerme. Le ficher terme inclut les définitions de tous ces prédicats auxiliaires, que nous ne présentons pas ici par un souci de propreté.

*intervennant dans F.*[1]

```
termeDuType(S,F) :-
        operateurDansTerme(Op,F),
        operateurExp(Op,profil(Args,S),P), !.
termeDuType(S,F) :-
        operateurDansTerme(Op,F),
        operateurExp(Op,profil(Args,Resul),P),
        member(S,Args), !.
member(A,[A|L]).
member(A,[B|L]) :- member(A,L).
```

*Calculs de renommages de symbols*

```
renomme(T,OpOld,OpNew,Tnew) :- var(T),T=Tnew.
renomme(T,OpOld,OpNew,Tnew) :- nonvar(T), T=..[Head|Tail],
        renommeListe(Tail,OpOld,OpNew,RenamedTail),
        (Head==OpOld ->
              (Tnew=..[OpNew|RenamedTail]);
              (Tnew=..[Head|RenamedTail])).

renommeListe([],OpOld,OpNew,[]).
renommeListe([X|L],OpOld,OpNew,[RenamedHead|RenamedTail]) :-
    renomme(X,OpOld,OpNew,RenamedHead),
    renommeListe(L,OpOld,OpNew,RenamedTail).

renommePlus(FBF,[],FBF).
renommePlus(FBF,[[Old,New]|Tail],FBFnew) :-
        renomme(FBF,Old,New,X),
        renommePlus(X,Tail,FBFnew).
```

Fichier ***theoadeq***:
Contient les définitions de quelques cas particuliers de théorèmes.

*Le théorème A fait intervenir la sorte S. Cette définition utilise la structure de la BT.*

```
theoremeDuType(S,A) :-
        sorteExp(S,Origine),
        ((P=Origine);contient(P,Origine)),
        axiomeExp(A,P),
        type(P),
        A = (G=D),
        (termeDuType(S,G);termeDuType(S,D)).
```

*L'axiome G=D est une définition constructive de l'opérateur Op.*

```
axiomeDefConstruc(G=D,Op,Pres) :-
        typeDefConstructive(Pres),
        axiomeExp(G=D,Pres),
        functor(G,Op,_).
```

*La formule Ax est une spécification déclarative définissant l'opérateur Op.*

```
axiomeSpecif(Ax,Op,Pres) :-
        proprieteSpecification(Pres),
        operateurExp(Op,Profil,Pres),
        axiomeExp(Ax,Pres),
        operateurDansTerme(Op,Ax).
```

*Le théorème G=D est un théorème de simplification, ce qui veut dire que D<G d'après un ordre de simplification <. L'ordre considéré comprend uniquement le cas où D est un sous-terme*

---

[1] Le "cut" y est pour éviter des réponses répétées.

*propre de G, et quand D est un atome et pas G.*

```
theoremeSimplif(G=D,Pres) :-
        axiomeExp(G=D,Pres),
        testSousTerme(D,G),
        (not G==D).
theoremeSimplification(G = D) :- axiomeExp(G = D,P),(not atom(G)),atom(D).
```

*FBF est une équation sur les constructeurs (définition d'un quotient).*

```
eqSurConstructeurs(FBF) :-
        axiomeExp(FBF,P),
        typeConstructeurs(P).
```

Fichier ***aide***:
Contient la définition du prédicat (sans arguments) "aide" (affichage des prédicats disponibles).

Fichier ***bt***:
Contient la version Prolog de la BT. Nous en avons présenté un extrait au §4.

# RÉFÉRENCES


**Backus J. [78]**

*Can Programming be liberated from the Von Neumann style? A functional style and its algebra of programs*
CACM vol.21 no.8

**Bar A., Feigenbaum E.A. [81]**

*The Handbook of Artificial Intelligence*
Willliam Kaufmann Inc., Los Altos, California

**Barendregt H.P., van Eekelen M.C.J.D., Glauert J.R.W., Kennaway J.R., Plasmeijer M.J., Sleep M.R. [87]**

*Towards an Intermediate Language based on Graph Rewriting*
PARLE'87 Proc., LNCS 259

**Barstow D.R. [79]**

*Knowledge-based Program Construction*
Elsevier North Holland

**Barstow D.R. [84]**

*The Roles of Knowledge and Deduction in Algorithm Design*
dans: *Automatic program construction techniques* (Biermann ed.), McMillan

**Bert D. [79]**

*La programmation générique: construction de logiciel, spécification algébrique et vérification*
Thèse d'état, USMG, Grenoble juin 1979

**Bert D., Drabik P., Echahed R. [87]**

*Manuel de Référence du Langage LPG*
Rapport LIFIA RT-17

**Bert D., Jacquet P. [77]**

*Generic Abstract Data Types*
Proc. 5th Annual III Conference, Guidel, France (mai 1977)

**Bibel W. [80]**

*Syntax-directed, semantics-supported program synthesis*
Artificial Intelligence 14

**Bibel W. [82]**

*Automated Theorem Proving*
Viewag Verlag, Braunschweig (sec.ed. 1987)

**Bibel W., Aspetsberger K. [85]**

*A Bibliography on Parallel Inference Machines*
Journal of Symbolic Computation 1(1985)

**Bibel W., Hörning K.M. [84]**
*LOPS.- A system based on a strategical approach to program synthesis*
dans: *Automatic Program construction techniques* (Biermann, Guiho, Kodratoff eds.), MacMillan

**Bibel W., Kurfess F., Aspetsberger K., Hintenaus P., Schumann J. [87]**
*Parallel Inference Machines*
Rapport du projet Esprit-415

**Biermann A.W. [78]**
*The inference of regular lisp Programs from Examples*
IEEE Trans.on Sys.,man and Cyber.v.SMC-8(8)

**Biermann A.W. [85]**
*Automatic Programming: a tutorial on formal methodologies*
Journal of Symbolic Computation

**Bledsoe W.W. [77]**
*Non-resolution Theorem Proving*
Artificial Intelligence 9

**Brachman R.J., Fikes R.E., Levesque H.J. [83]**
*Kripton: A Functional Approach to Knowledge Representation*
IEEE COMPUTER, Octobre 1983

**Brena R. [84]**
*Etude le la synthèse de programmes à partir d'exemples*
Rapport de D.E.A., Grenoble

**Brena R [88]**
*Parallelizing a Natural Deduction System*
Dans: *Artificial Intelligence III: Methodology, Systems, Applications* (O'Shea & Sgurev eds.), Elsevier Sc.Pub. (North Holland)

**Brena R., Caferra R., Fronhöfer B., Gresse C., Jaquet P., Potet M-L [84]**
*Program Synthesis through Problem Splitting: a Method for Subproblem Caracterization*
Computers and Artificial Intelligence, vol.4 no.5. Aussi rapport LIFIA 11

**Brena R., Potet M-L. [89]**
*Knowledge Specification in computer-aided program construction*
Art.Int. & Software Eng. Workshop, 11-13 avril, Exeter,Angleterre

**Burstall R.M., Goguen J.A. [77]**
*Putting Teories together to make Specifications*
Proc. Fifth IJCAI

**Castelfranchi C., Aloisi D.D., Stock O., Touzzi A. [88]**
*Propositions in a Hybrid Knowledge Representation System*
Dans: *Artificial Intelligence III: Methodology, Systems, Applications* (O'Shea & Sgurev eds.), Elsevier Sc.Pub. (North Holland)

**Clark K.L., Darlington J. [78]**
*Algorithm Classification through Synthesis*
The Computer Journal, vol.23 no.1

**Colmerauer A., Kanoui H., van Caneghem M. [79]**
*Etude et réalisation d'un système Prolog*
Rapp.Technique, Groupe d'I.A., U.E.R. de Luminy, Univ.d'Aix-Marseille II

**Comon H. [86]**
*An anti-unification approach to decide the sufficient completeness of algebraic specifications*
Rapport RR 619 IMAG, 51 LIFIA -I-, Juillet 1986

**Darlington J. [75]**
*Application of Program Transformation to Program Synthesis*
Proc.Int.Symp.on proving and improving programs, Arc-et-Senans, Juillet 1975

**Delobel, Adiba [82]**
*Les Bases de Données Relationnelles*
Dunod

**Dershowitz N. [83]**
*Computing with Rewrite Systems*
Aerospace report ATR-83 (8478)

**Descartes R. [1636]**
*Le Discours de la Méthode*

**Dollé D., Vigouroux J.R. [87]**
*Projet SCALP, volume 1: rapport du projet*
Rapport de 3ème année ENSIMAG

**Downey P.J., Sethi R., Tarjan R.E. [80]**
*Variations on the common subexpression problem*
Journal of the ACM vol.27 no.4

**Gallaire H., Minker J., Nicolas J-M. [84]**
*Logic and Databases: A Deductive Approach*
Computing Surveys, Vol.16, No.2, Juin 1984

**Goguen J.A. [85]**
*Suggestions for using and organizing Libraries in Software Development*
Proc.First International Conf.on Supercomputing systems, St.Petersburg, FL, USA, Déc. 1985 (IEEE press)

**Goguen J.A. [86]**
*Extensions and Foundations of Object-Oriented Programming*
Dans: *Research Directions in Object-Oriented Programming* (B.Shriver & P.Wegner eds.), MIT press

**Goguen J.A., Meseguer J. [87]**
*Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems*
Proc. Second Symposium on Logic in Computer Science.

**Goguen J.A., Thatcher J.W., Wagner E. [78]**
*An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*
Dans: *Current Trends in Programming Methodology* (R.Yeh ed.), Prentice Hall

**Grawitz [86]**
*Dictionnaire de Sciences Sociales*

**Green C. [69]**
*The Application of Theorem Proving to Problem solving*
IJCAI 1969

**Green C. [76]**
*The design of the PSI Program Synthesis system*
Proc.2nd Int.Conf.on Soft.Eng., San Francisco 1976

**Gresse C. [84]**
*Contribution à la programmation automatique. CATY: un système de construction assitée de programmes.*
Thèse de Docteur d'Etat, Université de Paris-Sud, centre d'Orsay, mars 1984

**Guiho G. [83]**
*Automatic Programming using Abstract Data Types*
Proceedings 8ème IJCAI

**Guttag J.V., Horning J.J. [78]**
*The algebraic specification of abstract data types*
Acta Informatica 10,1

**Haynes L.S., Lau R.L., Siewiord D.P., Mizell D.W. [82]**
*A Survey of Highly Parallel Computing*
IEEE Computer vol.15 no.1

**Hillis W.D., Steele G.L. [86]**
*Data Parallel Algorithms*
CACM vol.29 no.12

**Hoare C.A.R. [69]**
*An Axiomatic Basis for Computer Programming*
CACM vol.12 no.10

**Howard W.A. [80]**
*The Formulæ-as-types notion of construction*
To H.B.Curry: Essays on Combinatory Logic, lambda calculus and formalism (Seldin & Hindley eds.)

**Huet G., Hullot [82]**
*Proofs by induction in Equational Theories with Constructors*
J. Comp. Sys. Sc. vol.25 no.2

**Huet G., Oppen D. [80]**
*Equations and Rewrite Rules: A survey*
Dans: *Formal Languages: Perspectives and open problems* (Book R. ed.), Academic Press

**Jacquet P., Potet M-L [86]**
*Program Synthesis = Proof method + Knowledge*
7th ECAI, Brighton

**Jorrand Ph. [85]**
*Term Rewriting as a basis for the design of a Functional and Parallel Programming Language. A case study: the language FP2*
dans: Fundamentals of AI -an advanced course (Bibel & Jorrand eds.) LNCS 232

**Jorrand Ph. [87]**
*Design and implementation of a paralle inference machine for first order logic: an overview*
PARLE'87 proceedings

**Jouannaud J.P., Kodratoff Y. [79]**
*Caracterization of a class of functions synthesized from examples by a Summers-like method using a BMW matching technique*
IJCAI 1979

**Kleene [67]**
*Mathematical Logic*
John Wiley & Sons

**Knuth D. [68]**
*The Art of Computer Programming*
Addison-Wesley

**Kodratoff Y., Picard M [83]**
*Complétion de système de réécriture et synthèse de programmes à partir de leur spécification*
Journées BIGRE 83

**Kowalski R. [76]**
*Algorithm = Logic + Control*
Research Report, Dept.of Comp.and Control, Imperial College, London

**Kowalski R. [78]**
*Logic for Data description*
Dans: Logic and Data Bases (Gallaire & Minker eds.), Plenum Press, N.Y.

**Kung H.T. [82]**
*Why systolic architectures ?*
IEEE Computer vol.15 no.1

**Loveland D.W. [78]**
*Automated Theorem Proving: A Logical Basis*
Dans la série: *Fundamental Studies in Comp.Sci.* vol.6, North Holland

**Makowski J.A. [87]**
*Why Horn Formulas Matter in Computer Science: Initial Structures and Generic Examples*
Journal of Computer and Sciences 34

**Manna Z., Waldinger R. [79]**
*Synthesis: Dreams fi Programs*
IEEE Trans.on Soft.Eng., vSE-5-4

**Manna Z., Waldinger R. [80]**
*A Deductive Approach to Program Synthesis*
ACM ToPLaS vol.2 no.1

**Manna Z., Waldinger R. [85]**
*Special Relations in Automated Deduction*
Technical Note 355, SRI International

**McCarthy J., Hayes P. [69]**
*Some Philosifical Problems from the standpoint of Artificial Intelligence*
Machine Intelligence 4

**Milner R. [78]**
*A Theory of Type Polymorphism in programming*
Journal of Computer and Systems Sciences n.16

**Naran'yani A. [86]**
*Parallelism vs. Knowledge Processing*
AIMSA'86 Conference Proc., Septembre 1986, Varna, Bulgarie

**Nelson G., Oppen D. [80]**
*Fast Decision Procedures Based on Congruence Closure*
Journal of the ACM vol.27 no.2

**Nicolas J.M., Gallaire N. [78]**
*Data Base: theory vs. interpretation*
Logic and Data Bases (Gallaire & Minker eds.), Plenum press, N.Y.

**Nicolas J.M., Minker J., Gallaire N. [84]**
*Logic and Databases: A Deductive Approach*
Computing Surveys vol.16, no.2

**Perdrix H. [84]**
*Synthèse de programmes à partir de leurs spécifications*
Rapport de recherche 187, Univ.de Paris-Sud

**Pereira F. [88]**
*C-Prolog User's Manual*

**Potet M-L [88]**
*Preuves et stratégies pour la synthèse déductive de programmes*
Thèse de Docteur de l'INPG, Grenoble, juin 1988

**Reiter R. [76]**
*Asemantically guided deductive system for automatifc theorem proving*
IEEE Trans.on Elec.Computing C-26; aussi dans 3rd.IJCAI proc.(1973)

**Smith D.R. [82]**
*Derived Preconditions and their use in Program Synthesis*
6th Conf.on Aut.Ded., LNCS 138

**Smith D.R. [85]**
*Top-down synthesis of Divide-and-Conquer Algorithms*
Artificial Intelligence Journal, vol.27 (septembre)

**Staples J. [80]**
*Computation on graph-like expressions*
Theoretical Computer Science 10

**Stickel M. [81]**
*A complete unification algorithm for Associative-Commutative fuctions*
Journal of the ACM, vol.28

**Wadler Ph. [87]**
*Views: A way for Pattern Matching to Cohabit with Data Abstraction*
14th Symposium on Principles of Programming Languages proceedings (S.Munchnik ed.), ACM

**Wirsing M., Pepper P., Partsch H., Dosch W., Broy M. [83]**
*On Hierarchies of Abstract Data Types*
Acta Informatica 20

**Zima H.P., Bast H.J., Gerndt M. [88]**
*SUPERB: A tool for semi-automatic MIMD/SIMD parallelization*
Parallel Computing 6, (North Holland)

# AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de Messieurs

. C. GRESSE , Professeur

. L. TRILLING , Professeur

**Monsieur Ramon BRENA**

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " **Informatique** "

Fait à Grenoble, le 1er juin 1989

Pour le Président de l'I.N.P.G.
et par délégation
*Le Vice-Président*

C. GAUBERT