Preuves et stratégies pour la synthèse déductive de programmes

Marie-Laure Potet

THESE

présentée par

MARIE-LAURE POTET

pour obtenir le grade de DOCTEUR

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

(spécialité : INFORMATIQUE)

==========

PREUVES ET STRATEGIES POUR LA SYNTHESE

DEDUCTIVE DE PROGRAMMES.

==========

Date de soutenance : 22 juin 1988

Composition du jury :

J. MOSSIERE	président
J. - L. REMY	rapporteur
M. SINTZOFF	rapporteur
P. JACQUET	
P. JORRAND	

Thèse préparée au sein du Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle (LIFIA).

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1988**

Professeurs des Universités

BARIBAUD Michel	ENSERG	JOUBERT Jean-Claude	ENSPG
BARRAUD Alain	ENSIEG	JOURDAIN Geneviève	ENSIEG
BAUDELET Bernard	ENSPG	LACOUME Jean-Louis	ENSIEG
BEAUFILS Jean-Pierre	ENSEEG	LESIEUR Marcel	ENSHMG
BLIMAN Samuel	ENSERG	LESPINARD Georges	ENSHMG
BLOCH Daniel	ENSPG	LONGEQUEUE Jean-Pierre	ENSPG
BOIS Philippe	ENSHMG	LOUCHET François	ENSIEG
BONNETAIN Lucien	ENSEEG	MASSE Philippe	ENSIEG
BOUVARD Maurice	ENSHMG	MASSELOT Christian	ENSIEG
BRISSONNEAU Pierre	ENSIEG	MAZARE Guy	ENSIMAG
BRUNET Yves	IUFA	MOREAU René	ENSHMG
CAILLERIE Denis	ENSHMG	MORET Roger	ENSIEG
CAVAIGNAC Jean-François	ENSPG	MOSSIERE Jacques	ENSIMAG
CHARTIER Germain	ENSPG	OBLED Charles	ENSHMG
CHENEVIER Pierre	ENSERG	OZIL Patrick	ENSEEG
CHERADAME Hervé	UFR PGP	PARIAUD Jean-Charles	ENSEEG
CHOVET Alain	ENSERG	PERRET René	ENSIEG
COHEN Joseph	ENSERG	PERRET Robert	ENSIEG
COUMES André	ENSERG	PIAU Jean-Michel	ENSHMG
DARVE Félix	ENSHMG	POUPOT Christian	ENSERG
DELLA-DORA Jean	ENSIMAG	RAMEAU Jean-Jacques	ENSEEG
DEPORTES Jacques	ENSPG	RENAUD Maurice	UFR PGP
DOLMAZON Jean-Marc	ENSERG	ROBERT André	UFR PGP
DURAND Francis	ENSEEG	ROBERT François	ENSIMAG
DURAND Jean-Louis	ENSIEG	SABONNADIERE Jean-Claude	ENSIEG
FOGGIA Albert	ENSIEG	SAUCIER Gabrielle	ENSIMAG
FONLUPTJean	ENSIMAG	SCHLENKER Claire	ENSPG
FOULARD Claude	ENSIEG	SCHLENKER Michel	ENSPG
GANDINI Alessandro	UFR PGP	SILVY Jacques	UFR PGP
GAUBERT Claude	ENSPG	SIRIEYS Pierre	ENSHMG
GENTIL Pierre	ENSERG	SOHM Jean-Claude	ENSEEG
GREVEN Hélène	IUFA	SOLER Jean-Louis	ENSIMAG
GUERIN Bernard	ENSERG	SOUQUET Jean-Louis	ENSEEG
GUYOT Pierre	ENSEEG	TROMPETTE Philippe	ENSHMG
IVANES Marcel	ENSIEG	VEILLON Gérard	ENSIMAG
JAUSSAUD Pierre	ENSIEG	ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G

BOREL Joseph

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

ROWE Alain

E.N.S.I.M.A.G

COURTIN Jacques

E.F.P.

CHARUEL Robert

C.E.N.G

CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T

DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

RESUME

Pour maitriser la combinatoire liée à la synthèse déductive de programmes nous avons choisi de décomposer le processus nécessaire à l'élaboration d'un programme en deux étapes principales : la première est relative au choix de la preuve à mettre en place et à son déroulement, la seconde est relative à la recherche des propriétés qui doivent exister sur le domaine du problème, afin de valider les choix faits. Cette décomposition nous a permis de développer des stratégies interactives, dans lesquelles l'utilisateur est responsable du choix de la forme du programme attendu. Le rôle des stratégies est alors d'exécuter la preuve sous-jacente à ces choix et de caractériser, le mieux possible, les propriétés nécessaires sur le domaine du problème. Cette caractérisation permet d'envisager la déduction de ces propriétés mêmes lorsqu'elles ne sont pas explicitement disponibles et donc de " découvrir " de nouveaux algorithmes. Les stratégies que nous proposons sont basées sur un système formel de preuve que nous avons développé et elles sont attachées à l'élaboration de schémas de programmes très simples (condition, récursion et composition fonctionnelle).

ABSTRACT

In order to restrict the combinatorial behaviour of deductive program synthesis, we chose to decompose program's elaboration process into two steps. The first one is related to choice of the proof and its development. The second one is related to search for domain properties, which validate the latter choices. This decomposition allows to develop interactive strategies in which the user is responsible for the expected program form. The aim of these strategies is to carry out the proof underlying user's choices and to characterize the necessary properties of the domain. This characterization allows to consider the deduction of these properties, even though they are not explicitly available and thus to 'discover' new algorithms. The proposed strategies are based on a formal proof system which we have developed and are linked with elaboration of very simple program schemas (conditional, recursion and functional composition).

REMERCIEMENTS

Je tiens à remercier les membres du jury :

• M. Sintzoff, Professeur à l'université catholique de Louvain, qui a accepté d'être juge de ce travail. Je le remercie vivement pour ses observations qui m'ont été, et me seront, très utiles.

• J-L. Rémy, Chargé de Recherche première classe au CNRS. Je le remercie chaleureusement d'avoir accepté d'être rapporteur sur ce travail et de ses observations sur la partie théorique de cette thèse.

• J. Mossière, Professeur à l'Institut Polytechnique de Grenoble, Directeur de l'ENSIMAG, qui m'a fait l'honneur de présider le jury de cette thèse.

• P. Jorrand, Directeur de recherche au CNRS. Je le remercie de sa participation au jury et de m'avoir accueillie dans son laboratoire le temps de cette thèse.

• P. Jacquet, Maitre de Conférence à l'ENSIMAG, qui a encadré ce travail. Je tiens à le remercier particulièrement pour la confiance qu'il m'a accordée et pour sa très grande disponibilité.

Je remercie également les membres du LIFIA, en particulier les participants du groupe SADA. Je tiens aussi à remercier P. Berlioux, Maitre de Conférence à l'ENSIMAG, d'avoir mis sa compétence à ma disposition.

Enfin je remercie tous mes proches et très proches, pour leur patience et pour leur soutien lors de la rédaction de cette thèse.

TABLE

DES

MATIERES

INTRODUCTION

" The time is clearly ripe for program-manipulation systems "
Knuth 1976.

L'élaboration d'un programme résolvant un problème donné se décompose classiquement en trois étapes principales [Liv 78]. Dans un premier temps le programmeur décrit précisément ce qu'il attend du programme, c'est à dire le problème auquel il s'intéresse. Cette description du problème est classiquement appelée *énoncé*. Il doit alors trouver une méthode opératoire décrivant comment répondre à ce problème. Cette méthode, liée aux propriétés du problème traité, doit non seulement décrire une manière de calculer une solution mais celle-ci doit être obtenue dans un temps jugé raisonnable. Une telle méthode de résolution est classiquement appelée *algorithme*. L'implantation d'un algorithme dans un langage de programmation déterminé, en utilisant au mieux les constructions offertes par ce langage, produit un programme. La séparation entre algorithme et programme n'est pas toujours aussi nette, le choix d'une méthode de résolution dépendant souvent des possibilités du langage choisi. Développer des outils apportant une aide dans ce processus est une nécessité non seulement pour faciliter la tâche du programmeur mais aussi pour des raisons évidentes de maintenance et de fiabilité. Parmi ces outils on peut citer bien entendu les compilateurs, les environnements de programmation (éditeurs, débogueurs) mais aussi les outils d'aide à la spécification ou à la transformation de programmes. Pour notre part nous nous sommes intéressés à la mécanisation possible de la recherche d'algorithmes à partir d'énoncés de problème. Nous supposons donc que le problème à traiter est complètement décrit et nous ne nous préoccupons pas des problèmes d'implantation.

Maitriser l'activité de programmation, vue comme le passage d'un énoncé à un programme, a été rapidement et reste toujours un thème de recherche important en informatique. Une première contribution a consisté à étudier des méthodes de programmation, sans parler déjà de méthodologie. Ceci a donné lieu à une nombreuse littérature : *Structured Programming* (Dahl, Dijkstra et Hoare 1972), *A discipline of Programming* (Dijkstra 1976), *The science of programming* (Gries 1981), *Les bases de la programmation* (Arsac 1983) et *Raisonner pour programmer* (Gram

86) pour ne citer que les grands classiques. Ces méthodes sont très variées et peuvent aller des principes de bon sens (structurer, décomposer, raffiner) et de l'étude des algorithmes associés aux structures de données (tableau, liste, pile ...) à la proposition de schémas généraux de programmes répondant à des classes de problèmes particuliers (les différentes formes d'itérations, les schémas dichotomiques ...). Nous devons constater que nous sommes loin de pouvoir parler d'une théorie de la programmation et il reste beaucoup à faire, si tant est que cela soit possible. Une seconde contribution est liée à l'étude des différents objets manipulés lors de la construction de programmes : les données, les programmes, les transformations et les développements. A cet égard les langages applicatifs, qui ont une sémantique mathématiquement bien définie [ScS 71], permettent de considérer les programmes comme des objets sur lesquels il est possible de raisonner (les algèbres de programmes [Bac 78]). Des résultats conséquents ont été obtenus dans ce domaine et on peut maintenant parler de théorie des programmes. De la même manière un courant actuel s'intéresse à la formalisation des développements de programmes. Ceux-ci décrivent l'historique du développement, c'est à dire les décisions de constructions adoptées, leurs paramètres, leurs conditions d'application ... Il devient possible d'envisager de " rejouer " le développement, à partir de sa description, pour des problèmes analogues. Le processus de construction de programmes que nous avons évoqué au début de cette introduction (énoncé, algorithme, programme) n'est, dans la pratique, jamais linéaire. La recherche d'un algorithme peut, par exemple, entrainer une remise en cause partielle de l'énoncé du problème. Il est alors nécessaire de pouvoir déduire les changements qui découlent des modifications apportées sans recommencer tout le travail. Ceci est envisageable si on dispose d'une représentation des développements sur laquelle il est possible de raisonner. L'étude de la mécanisation de la recherche d'algorithmes à partir d'énoncés de problèmes se situe au carrefour de ces domaines. Nous devons trouver une démarche systématique de construction permettant, à partir d'un énoncé, de déduire un programme correct.

L'approche déductive de la construction de programmes est basée sur l'assimilation des programmes à des preuves d'énoncés (Isomorphisme de Curry-Howard [How 80]). Aborder la construction comme un processus déductif permet de maitriser ce qui est fait au moins au niveau conceptuel, à défaut du niveau opératoire. L'énoncé devient une formule à prouver, la construction d'un programme est assimilée à la recherche d'une preuve dont le programme est une trace opérationnelle, ceci dans une logique adéquate. Les énoncés, les programmes et les méthodes de résolution sont alors des objets manipulables formellement. Cette manière de considérer la mécanisation possible de la construction de programmes a bien sur déjà été étudiée notamment en liaison avec les progrès en démonstration automatique (Greene 1969, Waldinger & Lee 1969). Ce processus reste néanmoins très complexe. D'une part deux logiques interviennent : celle nécessaire à l'élaboration des différentes constructions de programmes et celle relative au domaine du problème. Cette dernière permet de déduire les propriétés du domaine du problème garantissant la correction du programme construit, celles-ci n'étant pas nécessairement connues explicitement. D'autre part, il reste la difficulté liée à la recherche de la preuve à mettre en place, avec la contrainte supplémentaire que le programme obtenu doit être raisonnablement efficace. L'approche déductive, dans son état actuel, ne permet pas d'envisager raisonnablement la construction de programmes comme un processus entièrement automatisé. Elle doit donc être assistée du programmeur. La carence des travaux dans cette approche provient du fait qu'ils ne prennent pas en compte la nécessité

d'inclure le programmeur en tant que participant tout en lui apportant une aide pertinente. A l'opposé, des approches plus empiriques, basées sur l'utilisation de connaissances de programmation, ont été développées dans le souci d'une telle aide. Mais il manque cependant à celles-ci le cadre formel permettant de garantir la correction du programme engendré et le degré de généralité suffisant pour pouvoir potentiellement dériver tout programme.

Dans cette thèse nous décrivons une méthode de programmation, guidée par l'utilisateur, ceci dans un cadre déductif. Nous avons adopté une démarche de programmation par instanciation progressive de schémas de programmes. Ces schémas peuvent être assimilés à des méthodes de programmation très générales et sont les principales connaissances de programmation utilisées. Nous avons caractérisé les propriétés sur le domaine du problème nécessaires à l'instanciation correcte de ces schémas vis à vis de l'énoncé traité. Si ces propriétés s'avèrent vraies nous élaborons alors les énoncés des sous-problèmes introduits par ces schémas. Aborder la construction de programmes sous cet angle permet de distinguer la part d'intervention demandée à l'utilisateur de la part prise en charge par le système déductif, ceci dans une approche descendante de l'analyse du problème. En raison des objectifs que nous nous sommes fixés le cadre de notre travail est très restreint. Le langage de " programmation " contient le minimum de constructions permettant d'exprimer toutes les fonctions calculables et les schémas de programmation proposés sont très génériques (condition, composition séquencielle et récursion). Nous sommes donc loin des problèmes actuels du génie logiciel, notamment de la programmation " in the large " ! Etant dans un cadre formel, l'extension à un vrai langage de programmation et à des techniques de programmation plus sophistiquées ne pose à priori pas de problème au niveau théorique mais nous ne pouvons pas en dire de même au niveau pratique ... Néanmoins la maitrise de la construction de petits problèmes est, à l'heure actuelle, une étape nécessaire.

En raison du peu de travaux dans ce domaine nous avons tenu à présenter, dans une première partie, les différentes approches existantes. Cette présentation tend principalement à mettre en évidence les apports de chacune de ces approches, tant au niveau théorique qu'au niveau pratique. Dans la seconde partie nous présentons les fondements théoriques de notre travail, c'est à dire le système de preuve des énoncés sur lequel nous nous basons. Les énoncés sont des formules du premier ordre, les programmes construits sont de type fonctionnel et les preuves effectuées sont constructives. Le système que nous proposons est propre aux preuves nécessaires à la synthèse de programmes et prend en compte notamment le fait que les fonctions manipulées peuvent n'être que partiellement définies. D'autre part, il met en évidence de façon explicite les deux niveaux de preuves nécessaires à la construction de programmes (celui relatif à la construction et celui relatif au domaine du problème) et leur lien afin de pouvoir déterminer les propriétés sous-jacentes à chaque schéma de programme proposé. Enfin dans le dernier chapitre nous décrivons des stratégies de preuve dans ce système formel. Ces stratégies ont pour interlocuteur d'une part l'utilisateur et d'autre part un démonstrateur de théorèmes auquel est demandé des formes particulières de propriétés sur le domaine du problème. Dans ces stratégies nous avons cherché à justifier les demandes adressées à l'utilisateur et au démonstrateur en terme de la qualité du programme construit. Les critères utilisés ne permettent pas de juger si ce programme est le meilleur

possible mais si tous les calculs effectués sont nécessaires. Nous présentons en annexe une dérivation systématique du tri par insertion et du tri par sélection, par application de ces stratégies. Ces exemples, avec quelques autres (recherche du maximum d'un ensemble, division euclidienne, calcul du plus grand diviseur commun de 2 entiers) sont très classiques. C'est une des raisons pour laquelle nous les avons choisis, leur développement complet et rigoureux étant, à notre avis, nécessaire. D'autre part dans les stratégies que nous proposons nous nous sommes intéressés à la gestion de l'historique de la preuve en cours de développement. En effet, un sous-problème issu de l'énoncé initial devra être prouvé en tenant compte des hypothèses découlant des choix de preuve déjà faits. La synthèse du tri par insertion et du tri par sélection engendre de nombreux sous-problèmes, leur développement permet donc d'illustrer ce fait.

partie 1

PROGRAMMATION AUTOMATIQUE

ET

SYNTHESE DEDUCTIVE

DE PROGRAMMES.

CHAPITRE 1

LA PROGRAMMATION AUTOMATIQUE.

L'étude de la mécanisation du processus de construction de programmes, rêve ambitieux des informaticiens, n'a pour le moment donné que peu de résultats probants. Cependant les résultats obtenus pour de "petits" exemples, l'avancement des travaux sur les fondements théoriques de l'activité de programmation et les progrès réalisés dans différents domaines sont des raisons valables pour persévérer en ce domaine. Parmi les domaines apportant leur contribution dans cette étude citons notamment la démonstration automatique et l'étude de la construction raisonnée de programmes [Dij 76], [Gri 81], [Gra 86]. Les motivations à une telle étude sont nombreuses, donnons en brièvement quelques unes :

– Il semble naturel de chercher à libérer, au moins en partie, le programmeur du coté fastidieux du travail de programmation afin qu'il puisse se consacrer à la partie plus créative. Proposer des systèmes d'aide à la construction de programmes est donc une suite logique des outils de plus en plus puissants offerts aux programmeurs : langages évolués, environnements de programmation proposant des facilités d'édition, de mise au point ...

– L'exécution de tâches par une machine garantit, sauf imprévus, une certaine fiabilité du résultat, le traitement se déroulant de manière rigoureuse. Vis à vis de la construction de programmes, la correction du programme engendré ou des transformations effectuées peut être assurée, contraintes auxquelles le programmeur ne peut que très difficilement se plier. Lorsqu'on connaît le rapport du temps passé entre la conception d'un programme et sa mise au point, ceci est intéressant.

– Une motivation plus futuriste est liée à l'Intelligence Artificielle. Une machine " intelligente " devrait être capable de développer ses propres méthodes de résolution de problèmes et donc de se programmer elle-même. Un domaine d'application actuellement étudié est, par exemple, la programmation automatique des robots : on souhaiterait pouvoir engendrer automatiquement la séquence d'actions qu'un robot doit effectuer à partir d'une situation donnée pour atteindre un certain but. Il en est de même en conception de VLSI où, pour des contraintes données, nous voudrions savoir comment construire ces systèmes de manière optimum.

La programmation automatique est un vaste domaine qu'il est difficile de caractériser de manière très précise. En effet, le premier compilateur pouvait, à l'époque, être vu comme un système de programmation automatique. Il prenait en charge la traduction d'un programme écrit dans un langage évolué, Fortran, en un programme équivalent écrit dans un langage d'assemblage et déchargeait ainsi l'utilisateur d'une partie de son travail ! Actuellement les travaux en ce domaine visent la mécanisation des pas de la construction dans lesquels la prise en compte de la sémantique des domaines d'application des programmes est nécessaire et doit être utilisée de manière adéquate. Cela peut être la recherche d'une implantation efficace d'un algorithme, la recherche d'un algorithme à partir d'un énoncé descriptif du problème à résoudre ...

L'étude de la programmation automatique est assez récente et, en raison de l'ampleur de la tâche visée et de ses divers aspects, les travaux en ce domaine apportent chacun leur contribution. Avant de s'intéresser à une approche particulière, qui aura ses propres limites, il nous a semblé nécessaire de montrer rapidement le principe des différentes

approches existantes. Classifier les travaux n'est pas facile. En effet, d'une part ils sont peu nombreux, ce qui permet difficilement de dégager des caractéristiques communes. D'autre part, la limite avec d'autres domaines tels que la transformation de programmes ou les environnements de programmation sophistiqués n'est pas toujours très nette (Mentor, Cornell Synthesizer Program et the Programmer's Apprentice). Des synthèses sur l'état actuel de la programmation automatique ont été faites notamment par A. Barr & E.A. Feigenbaum [BaF 82], A.W. Biermann [Bie 85], G. Gresse [Gre 84]. Les critères de classement prennent généralement en compte la forme de l'information demandée à l'utilisateur, le but visé et la manière d'y arriver ... Pour notre part, nous distinguons principalement deux approches, très opposées dans la manière d'aborder le problème. La première utilise des méthodes formelles à partir d'un énoncé de problème lui aussi bien formalisé. Cette approche est communément appelée Synthèse de Programmes. La seconde est basée sur l'utilisation d'experts en programmation, la connaissance codée dans ces experts étant issue de l'expérience des programmeurs. Nous qualifions cette approche de *basée sur l'expertise.*

1. APPROCHE SYNTHESE DE PROGRAMMES.

On distingue classiquement trois types de travaux suivant la forme exigée des énoncés : un ensemble d'exemples de données et de résultats attendus, une trace d'exécution du programme à construire ou une spécification complète du problème à résoudre.

1.1. Synthèse à partir d'exemples.

L'énoncé est un ensemble fini de couples de données et de résultats associés à ces données. Le but est de construire une fonction, sur un domaine donné, vérifiant ces exemples. Cette approche repose sur l'existence d'une régularité dans les exemples, régularité qui sera étendue au domaine voulu. La fonction est obtenue à l'aide de méthodes d'inférence inductive : recherche de loi, de règle ou de schéma à partir de cas particuliers. Plusieurs méthodes ont été proposées (Biermann, Summers, Kodratoff, Jouannaud et Guiho ....). Elles permettent de construire différentes classes de programmes. Pour illustrer cette approche, nous développons un exemple emprunté à A.W. Biermann [Bie 85] en utilisant la méthode Synthèse à partir de relations récurrentes (Summers, Kodratoff et Jouannaud). Les programmes sont construits sous une forme LISP.

Exemple :

Soient les couples suivants :

	données	résultats
1	nil	nil
2	(C)	(C)
3	((B) C)	(C)
4	(A (B) C)	(A C)
5	((D) A (B) C)	(A C)

– Dans une première étape, chaque résultat est réécrit sous forme d'une expression en terme d'une donnée x quelconque. Pour les exemples ci-dessus, les expressions obtenues sont :

1	$f_1(x) = nil$
2	$f_2(x) = cons(car(x), nil)$
3	$f_3(x) = cons(car(cdr(x)), nil)$

4 $f_4(x) = cons(car(x), cons(car(cdr(cdr(x))), nil))$

5 $f_5(x) = cons(car(cdr(x)), cons(car(cdr(cdr(cdr(x)))), nil))$

– La seconde étape consiste à réécrire chaque fonction f_{i+1} en terme de f_i. C'est à dire :

1 $f_1(x) = nil$

2 $f_2(x) = cons(car(x), f_1(cdr(x)))$

3 $f_3(x) = f_2(cdr(x))$

4 $f_4(x) = cons(car(x), f_3(cdr(x)))$

5 $f_5(x) = f_4(cdr(x))$

– Le but est alors de trouver des similarités dans ces expressions, ceci par des techniques de généralisation et de filtrage de sous-arborescences. Nous obtenons ici deux relations de récurrence :

$f_i(x) = f_{i-1}(cdr(x))$

$f_i(x) = cons(car(x), f_{i-1}(cdr(x)))$

et le cas de base $f_1(x)$=nil lorsque x est la liste vide.

- On cherche alors une relation permettant de différencier les données associées à chacune de ces deux relations de récurrence. Le prédicat atome(car(x)) convient. Le programme obtenu est donc :

```
f(x) = si null(x) alors nil
       sinon  si atome(car(x)) alors cons(car(x), f(cdr(x)))
              sinon f(cdr(x))
```

fin-exemple.

Les exemples donnés ne peuvent jamais caractériser la fonction cherchée de manière non ambigüe. Par exemple le programme suivant vérifie aussi les exemples :

```
f ' (x) = si null(x) alors nil
          sinon  si impair(long(x)) alors cons(car(x), f ' (cdr(x)))
                 sinon f ' (cdr(x))
```

long(x) est une fonction calculant le nombre d'éléments de la liste x.

Les méthodes utilisées en Synthèse à partir d'exemples permettent de synthétiser sans trop de difficultés des classes particulières de fonctions, classes qu'il est possible de caractériser.

1.2. Synthèse à partir de traces.

Dans cette approche, l'énoncé décrit la suite des valeurs intermédiaires prises par les différentes variables du programme, ceci pour des données particulières. Par exemple, nous pouvons spécifier le problème précédent par une suite de couples (L, R) où L décrit l'état de la variable représentant la donnée et R l'état du résultat élaboré. Pour la donnée initiale ((D) A (B) C), cette suite peut être :

1	(((D) A (B) C), nil)
2	((A (B) C), nil)
3	(((B) C), (A))
4	((C), (A))
5	(nil, (A C))

Cet énoncé décrit, par exemple, les états intermédiaires de l'exécution de l'algorithme :

```
début
      L := x ;
      R :=  nil ;
      tantque L≠nil faire
      début
            si atom(car(L)) alors  R := append(R, list(car(L))) ;
            L := cdr(L) ;
      fin ;
fin.
```

nil, atom, car, cdr, list et append sont supposées être des fonctions prédéfinies ayant la même sémantique qu'en Lisp.

Cette approche n'a été que peu étudiée [BiK 76], [SyS 75]. Le principe de base est le même que dans l'approche précédente : recherche d'une induction à partir d'un ensemble fini d'exemples. Nous ne détaillons donc pas davantage la manière dont construire le programme.

1.3. Synthèse à partir de spécification complète.

L'énoncé est une formule logique qui décrit le lien entre l'ensemble des données et le résultat attendu pour chacune de ces données. Ceci suppose que l'utilisateur dispose d'un ensemble de prédicats et fonctions prédéfinis. Le langage d'énoncés est généralement un sous-ensemble du Calcul des Prédicats du Premier Ordre, il possède donc une sémantique clairement définie. Ce langage peut alors être considéré comme un langage de programmation de très haut niveau et le processus de synthèse joue le rôle d'un compilateur sophistiqué. Le programme est dérivé de manière formelle à partir de l'énoncé, en effectuant soit une preuve de cet énoncé (**Approche Déductive**) soit des transformations préservant sa sémantique (**Approche Transformationnelle**). Contrairement aux approches précédentes il n'y a pas, en théorie, de limitation sur les programmes qui peuvent être synthétisés. Différents systèmes ont été proposés, notamment par C. Greene, Z. Manna et R. Waldinger, W. Bibel ... Ces travaux seront présentés plus longuement

dans la suite. Nous illustrons ici informellement l'approche déductive sur l'exemple précédent.

Exemple.

Soit l'énoncé : $\forall x\ \exists y\ \forall w\ [\mathbf{atome}(w) \wedge \mathbf{dans}(w, x) \equiv \mathbf{dans}(w, y)]$

On suppose donc que le système connaît les prédicats **dans** et **atome**. Une preuve possible de cet énoncé est :

- Soit x est la liste vide, alors **dans(w, x)** est toujours faux quelque soit w. Il doit donc exister un y tel que **dans(w, y)** est faux. nil est une instance possible vérifiant cette condition.

- Soit x n'est pas vide et peut se décomposer en car(x) et cdr(x), **dans(w, x)** est alors équivalent à la formule w=car(x) ∨ dans(w, cdr(x)). Prouver l'énoncé initial revient à montrer :

$$\exists y\ \forall w\ [[\mathbf{atome}(\mathbf{car}(x)) \wedge w=\mathbf{car}(x)] \vee [\mathbf{atome}(w) \wedge \mathbf{dans}(w, \mathbf{cdr}(x))] \equiv \mathbf{dans}(w, y)] \quad (1)$$

Pour cela, nous utilisons l'hypothèse d'induction $\exists a\ \forall w\ [\mathbf{atome}(w) \wedge \mathbf{dans}(w, \mathbf{cdr}(x)) \equiv \mathbf{dans}(w, a)]$. L'ordre bien fondé associé à ce raisonnement par récurrence est basé sur la longueur des listes, le plus petit élément étant la liste nil. Par hypothèse d'induction nous supposons donc que **a** existe. En utilisant la propriété de substitutivité de l'équivalence, la formule (1) peut se réécrire en :

$$\exists y\ \forall w\ [[\mathbf{atome}(\mathbf{car}(x)) \wedge w=\mathbf{car}(x)] \vee \mathbf{dans}(w, a) \equiv \mathbf{dans}(w, y)] \quad (2)$$

– Si car(x) n'est pas un atome, alors (2) se réduit en $\exists y\ \forall w\ [\mathbf{dans}(w, a) \equiv \mathbf{dans}(w, y)]$ et **a** est donc une instance possible de y validant cette formule.

– Si car(x) est un atome, alors la solution y=cons(car(x), a) convient.

fin-preuve.

L'énoncé initial a donc été prouvé et le programme découlant de la preuve est :

```
f(x) = si null(x) alors nil
       sinon si atome(car(x)) alors cons(car(x), f(cdr(x)))
             sinon f(cdr(x))
```

null et atome sont supposées être des fonctions de base du langage cible.

fin-exemple.

Remarquons que l'énoncé initial est aussi ambigu. Par exemple le programme suivant vérifie aussi cet énoncé :

```
f ' (x) = si null(x) alors nil
          sinon
               si non(atome(car(x))) ou dans(car(x), f ' (cdr(x))) alors f ' (cdr(x))
               sinon  append(f ' (cdr(x)), cons(car(x), nil))
```

En effet il n'est pas précisé que le résultat doit avoir le même nombre d'éléments que x et que ceux-ci doivent apparaître dans le même ordre.

Bien qu'un énoncé puisse être considéré comme un programme logique [Kow 74], le but visé par le processus de synthèse est différent. Un programme logique est basé sur le paradigme PROGRAMME = LOGIQUE + CONTROLE [Kow 79]. La partie logique peut être vue comme énonçant les propriétés du problème qui seront utilisées pour le résoudre. La partie contrôle peut être vue comme décrivant le déroulement de l'exécution. Un énoncé du point de vue de la programmation logique contient donc une idée algorithmique alors que le même énoncé du point de vue de la Synthèse de Programmes décrit uniquement les propriétés que le programme doit vérifier. Le rôle du processus de synthèse consiste, en particulier, à trouver une " logique " adéquate à ce problème en exploitant les propriétés disponibles.

Par exemple soit l'énoncé :

$$\forall x \, \exists y \, [perm(x, y) \wedge ord(y)]$$

x et y sont des listes et y est le résultat obtenu après avoir trié la liste x, **perm(x, y)** est vrai si y est une permutation de x et ord(y) est vrai si y est ordonnée. Nous supposons d'autre part disposer d'une définition de **perm** et **ord**.

– Interprété comme un programme logique, cet énoncé décrit l'algorithme qui construit une permutation de x puis vérifie qu'elle est ordonnée, et ainsi de suite, jusqu'à trouver la bonne permutation.

– Donné comme un énoncé d'un processus de synthèse, cet énoncé peut être prouvé de différentes façons, par exemple par induction sur x en utilisant les sélecteurs car et cdr. Ceci revient à engendrer un algorithme de tri par insertion, ce qui est déjà plus efficace !

Pour cette raison, un énoncé pour la synthèse de programmes peut être qualifié à juste titre de déclaratif. Il n'indique pas, en théorie, la manière dont le problème sera résolu, ce qui n'est pas le cas d'un programme logique ...

2. APPROCHE BASEE SUR L'EXPERTISE

Cette approche regroupe les travaux utilisant de l'expertise sur la manière de construire les programmes, que ce soit au niveau de paradigmes de programmation, de l'implantation de types abstraits, de l'utilisation des caractéristiques d'un langage de programmation particulier ... Ces travaux sont peu nombreux et peuvent difficilement être présentés de manière générale. Nous avons choisi de décrire un système, PSI, qui est une bonne illustration de ce qui a été réalisé dans cette approche. D'autres systèmes assez similaires ont été proposés. Citons, entre autres, SAFE/TI (Specification Acquisition From Experts / Transformational Implementation) et APE (Automatic Programming Expert).

PSI a été développé principalement à l'université de Stanford par C. Green, D. Barstow et leur équipe [Gre 76]. Ce système permet de construire un programme LISP à partir d'un algorithme informel donné par l'utilisateur. Il est organisé sous la forme d'un ensemble d'experts ayant des tâches bien déterminées. Ces experts sont regroupés en deux modules :

– **le groupe d'acquisition.** Son rôle est de construire un modèle de programme à partir d'un dialogue mené avec l'utilisateur. Un modèle de programme peut être vu comme un algorithme de très haut niveau pouvant contenir des annotations telles que des contraintes sur le programme, des indications sur les données utilisées par telle partie du programme ...

– **le groupe de synthèse.** Son rôle est de construire une implantation efficace d'un modèle en choisissant les structures de données, les structures de contrôle ... Cette partie du travail se déroule sans intervention directe de l'utilisateur. Le terme Synthèse désigne ici le fait d'implanter un algorithme en choisissant les bonnes structures et n'a donc pas la même signification que précédemment.

Cette séparation du processus de construction en deux étapes permet d'obtenir, à partir d'un même modèle de programme, différentes implantations par exemple suivant la taille des données, les probabilités sur les données possibles. Nous décrivons rapidement les principaux experts de PSI en mettant principalement en évidence la connaissance relative à la construction de programmes qu'ils utilisent.

• **Le groupe d'acquisition.**

L'utilisateur peut communiquer à l'aide d'un sous-ensemble de l'anglais, d'un langage formel de haut niveau et d'exemples de données-résultats ou de traces d'exécution. L'utilisateur décrit donc ce que le programme doit faire et sa structure générale. L'interaction avec l'utilisateur est menée par le "Dialogue Moderator Expert" qui décide des questions à poser à l'utilisateur et gère le déroulement du dialogue. Les réponses fournies par l'utilisateur sont traitées par :

– Le "Parser/Interpreter Expert". Cet expert dispose de connaissances sur les structures de données (les ensembles, les records ...), sur les structures de contrôle (boucles, conditionnelles, procédures ...) et sur des idées d'algorithmes plus compliqués. Il est capable de comprendre environ 70 concepts de programmation.

– Le "Example/Trace Expert". Son rôle est de généraliser des structures de données ou des structures itératives. Il utilise des principes d'inférence inductive comme dans l'approche synthèse à partir d'exemples.

Le but de ces deux experts est de traduire l'information donnée par l'utilisateur en des termes plus orientés programmes que nous appelons *fragments*. Pour ce faire ils font appel à un autre expert qui dispose de connaissances sur le domaine d'application du programme. Son rôle est très important car il peut permettre de lever certaines ambiguïtés, ce qui évite de tout demander à l'utilisateur. L'élaboration d'un modèle de programme est faite de manière incrémentale, à partir des fragments, par le **Constructeur de Modèle** [Mcc 77] ceci à partir de différents types de connaissances portant sur :

– la construction de modèles à partir de fragments ;

– le langage de description de modèles : inférence et cœrcion de types, opérateurs permis ;

– la recherche de structures abstraites à partir des informations fournies par l'utilisateur. Par exemple celui-ci peut parler d'une liste d'objets qui sera abstraite, dans le modèle élaboré, en la structure de données Collection. Cette dernière pourra ultérieurement être implantée de différentes façons (tableaux, listes ...) suivant des critères d'efficacité.

• **Le groupe de synthèse.**

Ce groupe est constitué de deux experts, **PECOS** et **LIBRA**. Le premier est chargé du codage des modèles. Il s'appuie sur le second dont le rôle est de faire les " bons " choix en tenant compte de l'efficacité du programme engendré. PECOS procède par raffinements successifs et utilise environ 450 règles se rapportant à des concepts de programmation. Donnons des exemples :

- une collection d'objets peut être représentée par une liste ou un tableau ou une fonction dans le domaine des booléens (si un objet appartient à la collection on lui associe la valeur vrai sinon faux) ;

- une application sur le domaine des atomes peut être représentée par une propriété attachée à ces atomes ;

- si une collection est une donnée du programme, sa représentation peut être convertie en une autre représentation ;

- l'intersection de 2 collections peut être obtenue en énumérant les objets de l'un et en conservant ceux qui sont membres du second ;

- mettre en place une énumération revient à trouver : 1) un ordre d'énumération, 2) un moyen de conserver l'état courant de l'énumération, 3) une condition d'arrêt.

Lorsque plusieurs règles sont applicables, PECOS fait appel à l'expert LIBRA dont le rôle est de choisir une règle en fonction de critères d'efficacité (temps d'exécution, espace mémoire nécessaire, ...). Pour ce faire LIBRA utilise des techniques d'analyse d'algorithmes et des heuristiques fondées sur la taille des données, le coût des opérations, la probabilité associée aux tests ...

Les exemples traités dans le système PSI sont des programmes de tri, de formations de concepts par apprentissage, de calcul de nombres premiers ou des programmes résolvant des problèmes classiques de la théorie des graphes. Nous n'en présentons pas, ceux-ci nécessitant un long développement et l'utilisation d'un nombre important de règles.

3. AUTRES APPROCHES.

Peu d'autres méthodes ont été étudiées et elles sont à un stade encore plus exploratoire. Nous énonçons rapidement leur principe.

• La synthèse de programmes par analogie (Ulrich & Moll [UlM 77], Dershowitz et Manna [DeM 77]). Les données nécessaires à la synthèse sont alors d'une part l'énoncé D1 pour lequel un programme doit être construit et d'autre part un programme P2 vérifiant un énoncé D2 et une preuve de P2. L'établissement d'une analogie entre D1 et D2 permet alors d'obtenir un programme par transformation de P2 et de sa preuve associée, si cette analogie est pertinente. L'analogie peut être choisie par l'utilisateur ou établie par des critères syntaxiques à partir des énoncés. Cette approche, basée sur la recherche d'une preuve de l'énoncé, peut être vue comme un cas particulier de l'approche déductive. Le choix d'une analogie est un guide permettant de préciser la preuve qui sera faite.

• L'inversion de programmes. Cette technique peut être vue comme un cas particulier de construction de programmes. Citons par exemple les travaux de Eppstein [Epp 85] sur l'inversion de programmes LISP et les travaux de Shoham qui, pour sa part, cherche à caractériser les programmes Prolog réversibles [Sho 84].

• E. Kant propose une approche nouvelle qui est encore très spéculative [Kan 85], [KaN 83]. Elle vise la recherche d'un algorithme en l'absence d'axiomatisation formelle du problème et des opérateurs disponibles. Cette approche est basée sur l'étude de raisonnements utilisés par le programmeur, notamment l'exécution symbolique. La construction de l'algorithme résulte d'une interaction entre quatre espaces de connaissances relatives respectivement :

- à la construction de programmes ;
- au domaine d'application du problème ;
- à l'exécution d'algorithme sur des données symboliques ou constantes ;
- à la construction d'exemples.

Cette approche semble intéressante dans le sens où la connaissance codée dans les différents espaces est moins spécifique que celle nécessaire dans un système tel que PSI. D'autre part les techniques d'application de ces connaissances sont plus sophistiquées mais, d'après E. Kant, plus maîtrisables que celles utilisées dans les processus entièrement déductifs.

4. CONCLUSION.

Ces différentes approches sont difficiles à comparer car la forme de l'énoncé attendu et les buts visés ne sont pas les mêmes (énoncé algorithmique, énoncé déclaratif, données partielles et recherche d'un algorithme ou d'une implantation dans un langage donné). D'autre part, les travaux proposés sont encore très exploratoires et peu de résultats conséquents ont été obtenus. En pratique, ils permettent la construction de petits programmes bien adaptés aux méthodes utilisées et souvent au prix d'efforts importants ! Mais ces approches cherchent toutes à modéliser une partie du processus de construction de programmes et nécessitent des connaissances du même ordre qui sont :

- des connaissances sur le domaine des problèmes ;
- des connaissances sur les langages et les principes de programmation ;
- du raisonnement pour choisir et combiner ces connaissances et pour les enrichir.

Les connaissances sur les domaines possibles de problème doivent décrire la sémantique de ces domaines et leur propriétés " intéressantes ". Ces connaissances ne sont jamais caractérisées formellement, il est toujours supposé qu'elles sont disponibles au moment souhaité. Cette lacune provient en grande partie du fait que la programmation automatique est étudiée pour des domaines d'application quelconques. Cette supposition est, à notre avis, le reproche majeur qui peut être adressé aux résultats actuellement obtenus dans ce domaine. Manna et Waldinger ont recensés, par exemple, tous les théorèmes et définitions nécessaires à la synthèse d'un algorithme d'unification [MaW 81], ils s'avèrent d'un nombre impressionnant ! Dans le cadre de la réalisation d'un système général, il se pose donc le problème d'une part de disposer de ces nombreuses connaissances et d'autre part d'être capable de sélectionner, parmi un grand ensemble, celles qui sont appropriées au problème. Il est donc important de pouvoir caractériser les connaissances nécessaires à la construction d'un programme particulier. Ainsi nous pourrons étudier la possibilité de

les rechercher et même de les déduire, si elles ne sont pas directement disponibles. Ceci est l'une des motivations principales de notre travail.

Les connaissances relatives à l'activité de programmation elle-même sont, dans les travaux que nous avons présentés, d'ordre différent. Elles sont très génériques dans les approches Synthèse, puisqu'elles se réduisent à la connaissance des primitives du langage cible, ou au contraire très spécifiques et donc nombreuses dans des systèmes tels que PSI. Ces différences ne sont que très peu liées aux méthodes utilisées mais proviennent plutôt des buts visés : dans les approches Synthèse on s'intéresse à la recherche d'une " idée algorithmique " qui peut s'exprimer dans un langage abstrait et simple alors que le système PSI vise l'élaboration d'un programme efficace utilisant au mieux les possibilités offertes par le langage de programmation choisi.

La différence essentielle entre ces approches réside dans le mode de raisonnement mis en place, c'est à dire dans la manière d'utiliser les connaissances : méthodes formelles (preuves, transformations valides, inférence inductive ...), utilisation directe de ces connaissances (PSI) ou raisonnements moins modélisables tels que la recherche d'analogie, l'exécution symbolique. Ces différentes méthodes, intéressantes soit par leur caractère général soit, à l'opposé, en raison de leur spécificité permettent de traiter plus facilement des classes particulières de problèmes. Elles apportent donc chacune leur contribution à la modélisation d'une activité aussi complexe que la construction de programmes. La cohabitation et l'interaction entre ces différents modes de raisonnement sont, à notre avis, nécessaires pour simuler la créativité des programmeurs.

En conclusion de cette rapide présentation, nous aimerions soulever deux points, généralement peu abordés, qui sont très importants. Le premier est relatif à l'efficacité de la solution obtenue et le second concerne l'automatisation réellement envisageable de ce processus. Nous n'attendons pas d'un système automatique qu'il produise un programme génial, mais que celui-ci soit néanmoins utilisable, c'est à dire d'une efficacité raisonnable. Or, bien entendu, il n'existe pas de critères applicables à priori permettant de choisir la bonne solution. Par exemple, l'expertise utilisée dans le module LIBRA (PSI), est capable de prendre en compte l'efficacité locale lors du choix de l'implantation d'une structure, d'un type abstrait mais non l'efficacité du programme en entier. Bien qu'il existe des techniques permettant d'améliorer un programme à postériori (suppressions des appels redondants, réarrangements des tests ...) ceci n'est pas suffisant. La méthode utilisée pour résoudre le problème peut être intrinséquement plus ou moins efficace. A défaut, il peut être intéressant de chercher à évaluer la complexité du programme construit, au moins en donnant un ordre de grandeur. Ceci permet de porter un jugement sur la solution obtenue. L'évaluation de la complexité des programmes est un domaine plus maîtrisable et des règles d'évaluation de programmes, généralement basées sur la syntaxe, ont été mise en évidence notamment par Aho, Hopcroft et Ullman [AHU 74], [AHU 82]. Donnons des exemples de ces règles :

- le temps d'exécution d'une affectation est majoré par un facteur constant ;

- le temps d'exécution de la composition de 2 blocs d'instructions est le plus grand des temps d'exécution associés à ces blocs à un facteur constant près ;

- le temps d'exécution d'une itération est le produit du nombre de fois où l'itération sera exécutée et du temps d'exécution du corps de l'itération.

Des systèmes formels permettant de raisonner sur la complexité des programmes ont donc été proposés, principalement pour les langages impératifs. Par exemple H. Nielson [Nie 84] décrit une axiomatisation d'un langage impératif en logique de Hoare prenant en compte le temps d'exécution du programme. Les formules de son système formel sont de la forme P { S : R } Q. De telles formules sont considérées comme vraies si et seulement si, lorsque P est vrai et que le programme S termine alors la postcondition Q est vrai et le temps d'exécution de S satisfait la

formule R.

En dehors du fait qu'un système de programmation automatique n'est pas capable de choisir une bonne solution il est important qu'il puisse fournir une évaluation de sa solution, pour que l'utilisateur puisse juger de sa pertinence. Un tel système devra donc être capable de raisonner, par exemple en terme d'évaluation du temps d'exécution, sur des objets de type programme.

Deux difficultés empêchent d'envisager raisonnablement la mécanisation totale de la construction de programmes. La première que nous venons d'évoquer est relative à la pertinence de la solution fournie. La seconde est liée à la nécessité de disposer des propriétés adéquates sur le domaine du problème traité, propriétés que le système peut ne pas nécessairement connaître ou reconnaître parmi toutes ses connaissances. Les systèmes de programmation automatique doivent donc être vus comme des outils d'aide à la construction de programmes, l'utilisateur participant activement à l'obtention d'une solution. Pour que ces systèmes apportent une aide effective, ces systèmes doivent être capables d'interroger l'utilisateur à bon escient et de justifier l'influence de ses réponses vis à vis du programme élaboré. L'intégration, dans le processus de construction de programmes, de cette interaction nécessaire est rarement prise en considération. Ceci est regrettable puisque la pertinence des résultats dans ce domaine est entièrement liée à l'aide effective que peuvent apporter de tels systèmes. Pour envisager des réponses à ces problèmes, nous nous sommes intéressés à l'approche Synthèse Déductive dans laquelle le programme est obtenu à partir d'une preuve d'une formule logique décrivant le problème à traiter. Cette approche est fondée sur une formalisation du processus de construction de programmes, ceci en terme de preuves. Il devient donc possible de raisonner sur les pas nécessaires à l'élaboration d'un programme, en raisonnant sur les preuves : mise en évidence des propriétés nécessaires au déroulement d'une preuve, des informations que doit fournir l'utilisateur pour élaborer une preuve, évaluation de la solution ... Il restera bien sûr la difficulté liée au choix de la preuve à mettre en place. Mais, contrairement à Barstow [Bar 79], l'un des auteurs du système PSI, nous pensons que l'expertise doit être une aide à l'implantation d'une méthode formelle et non l'inverse.

L'objectif du travail que nous présentons dans cette thèse était de proposer une axiomatisation correcte du processus de synthèse de programmes afin de pouvoir caractériser les propriétés nécessaires à la synthèse d'un algorithme particulier. Ceci permet de répondre au problème " des connaissances disponibles " qui est à nos yeux la principale carence des résultats actuels. Dans cette première partie nous présentons d'abord deux logiques de la construction de programmes : la théorie intuitioniste des types de Martin-Löf et la logique de Hoare. Nous dressons ensuite un état de l'art de la Synthèse de programmes à partir d'une spécification logique dans le but de montrer les apports et les carences actuelles. Pour cela, nous avons choisi de présenter les principaux travaux en les illustrant sur le développement d'un même exemple. Dans la partie 2, nous présentons, après avoir décrit le langage cible et le langage d'énoncé utilisés, le système formel employé permettant de prouver les énoncés. Enfin, nous décrivons différentes stratégies, basées sur ce système formel, permettant d'exhiber d'une part les demandes faites à l'utilisateur et d'autre part les propriétés nécessaires à la synthèse d'un schéma d'algorithme particulier. Nous pourrons alors caractériser les connaissances dont le système doit absolument disposer et les processus déductifs nécessaires à leur recherche lorsqu'elles font défaut.

CHAPITRE 2

SYNTHESE LOGIQUE : LES FONDEMENTS THEORIQUES.

" It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance " J. McCarthy (1963).

Bien que la logique mathématique soit un cadre possible pour la logique algorithmique, cette dernière nécessite des contraintes qui lui sont propres. En particulier nous exigeons des preuves constructives, c'est à dire qui fournissent un moyen de calculer les objets cherchés. Par exemple le tiers exclus $A \vee \neg A$ n'est acceptable que si nous sommes capables de décider, pour une interprétation particulière, si A est vrai ou faux.

Dans ce chapitre nous présentons deux formalisations de la logique des programmes. La première repose sur la notion de type, elle est intéressante en particulier en raison de ses fondements théoriques. La seconde, plus classique, est la Logique de Hoare qui a été élaborée pour des raisons pragmatiques, la preuve de programmes. Dans la première approche, la sémantique des programmes est décrite en terme de type, le typage employé devant être assez puissant pour permettre une description précise du sens des programmes. La logique des types est donnée à l'aide de règles de typage associées à chaque contruction possible du langage de programmation. En logique de Hoare, la sémantique est décrite en terme des relations vérifiées par les objets manipulés par les programmes, les règles énoncent alors comment les différentes constructions du langage " transforment " ces relations. Ces deux logiques permettent de considérer un **programme** comme une **preuve** d'une formule logique et donc d'assimiler la **construction d'un programme à la recherche d'une preuve.**

1. LA THEORIE DES TYPES.

Des liens ont rapidement été établis, pour des besoins théoriques, entre les classes de fonctions calculables et la logique. Gödel, par exemple, propose une interprétation de l'arithmétique intuitioniste du premier ordre par les fonctionnelles récursives primitives de type fini (1958). De la même manière, Curry et Feys (1958) établissent une correspondance entre la Logique Combinatoire et le calcul propositionnel Intuitioniste basé uniquement sur l'implication que nous notons $P(\Rightarrow)$. La logique Intuitioniste, développée initialement pour éviter les paradoxes découlant de la Logique Classique, nous intéresse particulièrement en raison de son caractère intrinsèquement constructif.

Nous présentons brièvement les travaux de Curry et Feys dans le but d'illustrer le lien entre preuve et programme. Puis nous montrons comment la théorie des types peut alors être utilisée naturellement pour la Synthèse de Programmes, en enrichissant la définition des types.

1.1. Rappels sur la Logique Combinatoire Typée.

La Logique Combinatoire est un dérivé du λ-calcul. Elle permet de s'abstraire des variables, ce qui rend les règles de manipulation plus simples, c'est pour cette raison que nous l'avons choisie pour cette illustration. La présentation ci-dessous est fortement inspirée de " Introduction to Combinators and λ-calculus " de J. Hindley et J. Seldin [HiD 86]. Nous rappelons ci-dessous les notions principales sur la Logique Combinatoire.

Définition des termes (notés CL-termes) :

Soient V un ensemble infini dénombrable de symboles de variables et C un ensemble de symboles de constantes. C contient deux symboles particuliers K et S, appelés combinateurs de base, qui correspondent respectivement aux λ-termes $\lambda x . (\lambda y . x)$ et $\lambda x . (\lambda y . (\lambda z . (xz)(yz)))$.

- Toutes les variables et les constantes sont des CL-termes.
- Si X et Y sont des CL-termes alors (XY) est un CL-terme. (XY) est l'opération d'application.

fin-définition.

Par exemple, l'abstraction du λ-calcul peut être décrite récursivement à l'aide des combinateurs S et K de la manière suivante :

$-\ \lambda x. U \equiv KU$	si x n'est pas libre dans U.
$-\ \lambda x. x \equiv I$	où I=SKK
$-\ \lambda x. Vx \equiv V$	si x n'est pas libre dans V.
$-\ \lambda x. UV \equiv S\ (\lambda x. U)\ (\lambda x. V)$	sinon.

x est une variable et U et V sont deux CL-termes. Rappelons que x est une variable non libre dans U si et seulement si x n'apparait pas dans U ou si toutes les occurrences de x dans U sont sous la portée d'un λ.

L'assimilation des programmes aux preuves est basée sur la notion de type qui permet de parler des programmes. Nous présentons donc rapidement la Logique Combinatoire typée en donnant un système formel d'affectation de type, noté TA_C.

Définition des schémas de type :

Soit $\mathfrak{C}$ un ensemble de symboles de constantes de type et $\mathfrak{V}$ un ensemble infini dénombrable de symboles de variables de type. Un schéma de type est défini de la manière suivante :

- Tous les constantes de types et les variables de types sont des schémas de type.
- Si α et β sont des schémas de types alors $\alpha \rightarrow \beta$ est un schéma de type qui dénote l'ensemble des applications de α dans β.

Définition des formules d'affectation :

Une formule d'affectation est de la forme $X \in \alpha$ où X est un CL-terme et α un schéma de type. X est appelé le sujet de cette formule.

fin-définitions.

Le système formel TA_C décrit les formules d'affectation valides. Il comporte deux schémas d'axiomes, définissant le schéma de type associé aux combinateurs de base, qui sont :

$$K \in \alpha \rightarrow \beta \rightarrow \alpha$$
$$S \in (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$$

et une règle d'inférence unique définissant le type du CL-terme (XY) :

$$\frac{X \in \alpha \rightarrow \beta,\ Y \in \alpha}{(XY) \in \beta}$$

Nous dénotons dans la suite ces deux schémas d'axiomes respectivement par (→K) et (→S) et la règle d'inférence par (→E).

Exemple : montrons que le CL-terme SKK, qui est l'identité, est de type $\alpha \rightarrow \alpha$.

- Par instanciation des deux schémas d'axiomes nous avons :

$$S \in (\alpha \rightarrow (\beta \rightarrow \alpha) \rightarrow \alpha) \rightarrow (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$
$$K \in \alpha \rightarrow (\beta \rightarrow \alpha) \rightarrow \alpha$$

- Puis, par application de la règle (→E), nous déduisons $SK \in (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$.

- Et, en instanciant le schéma d'axiome (→K) par $K \in \alpha \rightarrow \beta \rightarrow \alpha$ nous obtenons $SKK \in \alpha \rightarrow \alpha$.

fin-exemple.

1.2. Assimilation des CL-termes à des preuves.

La correspondance entre Programme et Preuve proposée par Curry et Feys repose sur l'assimilation des types à des formules propositionnelles. En effet, par abstraction des sujets dans les formules d'affectation, le système TA_C devient un système de déduction de type. Les schémas d'axiomes sont :

1. $\alpha \rightarrow \beta \rightarrow \alpha$
2. $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

La règle d'inférence est :

$$\frac{\alpha \rightarrow \beta \ , \ \alpha}{\beta}$$

Curry et Feys remarquent qu'en interprétant la flèche comme l'implication intuitioniste, que nous notons $\Rightarrow$, les deux schémas d'axiomes correspondent à des schémas d'axiomes classiques de la logique intuitioniste et la règle d'inférence au Modus Ponens. Par exemple, la preuve de la formule d'affectation SKK $\in \alpha \rightarrow \alpha$ que nous avons donnée précédemment fournit, par abstraction des sujets, la démonstration de l'implication $\alpha \Rightarrow \alpha$ suivante :

$$\frac{\dfrac{(\alpha \Rightarrow (\beta \Rightarrow \alpha) \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha \qquad \alpha \Rightarrow (\beta \Rightarrow \alpha) \Rightarrow \alpha}{(\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha}\,(MP) \qquad \alpha \Rightarrow \beta \Rightarrow \alpha}{\alpha \Rightarrow \alpha}\,(MP)$$

Prouver le type $\alpha \rightarrow \alpha$ ou la formule $\alpha \Rightarrow \alpha$ revient donc au même. L'assimilation des preuves à des CL-termes repose sur le fait que le CL-terme sous-jacent à la preuve décrit cette preuve. Par exemple la construction des sujets sous-jacente à la preuve précédente est :

$$\frac{\dfrac{S \qquad K}{SK} \qquad K}{SKK}$$

– Dans le cadre du typage, SKK peut être vu comme une construction possible d'un CL-terme de type $\alpha \rightarrow \alpha$.

– En logique intuitioniste, prouver $\alpha \Rightarrow \beta$ revient à fournir un moyen de construire une preuve de β à partir d'une preuve de α. SKK, qui est la fonction identité $\lambda x \, . \, x$, permet bien d'obtenir une preuve de α à partir d'une preuve de

α : il suffit de retourner la preuve de α donnée en paramètre. SKK, dans le cadre de la logique intuitioniste, décrit donc la preuve de $\alpha \Rightarrow \alpha$.

Nous avons donc :

Logique Combinatoire	P ($\Rightarrow$)
TYPE	PROPOSITION
CL-TERME	PREUVE

Le fait d'assimiler les preuves à des CL-termes permet de calculer sur les preuves. Dans l'exemple choisi, nous aurions pu prouver le type $\alpha \rightarrow \alpha$ en construisant la preuve SKK(SKK) puisque SKK est l'identité. Les règles de réduction de la Logique Combinatoire relative à K et S sont :

KXY $\nabla 1$ X
SXYZ $\nabla 2$ XZ(YZ)

Elles permettent de réduire SKK(SKK) en SKK de la manière suivante :

SKK(SKK) $\nabla 2$ K(SKK)(K(SKK)) $\nabla 1$ SKK

On peut donc parler de forme normale de preuve, ce qui devient très intéressant.

1.3. Application à la Synthèse de Programmes.

L'application à la synthèse de programmes est immédiate : en considérant l'énoncé du problème comme un type, il " suffit " de prouver ce type dans un système de typage adéquat. Cette preuve fournit, par effet de bord, un terme qui est un programme possible ayant le type exigé. Pour pouvoir effectivement décrire des énoncés, la notion de type a été étendue. Ceci a donné lieu à différents systèmes. Parmi les plus connus citons AUTOMATH de N.G. De Bruijn [Bru 68], la théorie des types de Martin-Löf [Mar 75] et la théorie des Constructions de T. Coquand et G. Huet [CoH 84]. Les principales extensions, par rapport à la définition des types que nous avons utilisée jusqu'à maintenant, sont les suivantes :

- les types de base liste, entier, ...
- l'opération de produit cartésien,
- l'union disjointe,
- l'union disjointe d'une famille d'ensembles notée $\Sigma_{x \in A} B$,
- le type de l'ensemble des fonctions de A dans B, où B peut dépendre de x, noté $\prod_{x \in A} B$. C'est une généralisation du type $A \rightarrow B$.

Σ et $\prod$ permettent de décrire des types qui dépendents de la valeur d'une variable. Comme nous le verrons, ceci est nécessaire pour permettre la description d'énoncés de problème. Des analogies avec les formules de la logique intuitioniste ont aussi été établies pour ces nouveaux constructeurs de type. Nous nous contentons de les énoncer :

- Le produit cartésien de A et B correspond à $A \wedge B$.
- L'union disjointe de A et B correspond à $A \vee B$.
- $\Sigma_{x \in A} B(x)$ correspond à $(\exists x \in A)\, B(x)$.
- $\prod_{x \in A} B(x)$ correspond à $(\forall x \in A)\, B(x)$.

Nous donnons maintenant un exemple, emprunté à B. Nordström et K. Petersson, illustrant l'utilisation des types comme énoncé de programmes [NoP 83]. Leur travail est basé sur la théorie des types de Martin-Löf. Nordström et Petersson utilisent une notation ensembliste que nous avons conservée, celle-ci étant plus lisible.

Définition du type sous-ensemble :

Si A est un type, x une variable et B une formule de la logique des Prédicats qui peut dépendre de x alors $\{x \in A \mid B\}$ est un type. Les occurrences libres de x dans B deviennent liées dans cette expression. Notons que l'appartenance d'un élément à ce type peut être indécidable puisque B peut être un prédicat indécidable.

fin-définition.

Dans la notation des typés que nous avons donnée précédemment $\{x \in A \mid B\}$ correspond à $\Sigma_{x \in A} B$ où A et B sont des types. Nordström et Petersson introduisent une distinction entre type et formule logique puisque, dans l'expression $\{x \in A \mid B\}$, A est un type et B une formule logique. Cette distinction n'est en fait que conceptuelle puisque les formules logiques peuvent être interprétées comme des types en égard à la correspondance donnée précédemment. Nous conservons néanmoins cette séparation et nous utilisons les règles habituelles de déduction sur les formules logiques.

Exemple. Soit la fonction tri définie par les équations :

$$\text{tri(nil)} = \text{nil}$$
$$\text{tri(a . s)} = \text{insert(a, tri(s))}$$

Nous voulons montrer que tri est une fonction de l'ensemble des listes d'éléments de type A dans l'ensemble des listes d'éléments de type A qui ont la propriété d'être ordonnées et qui sont une permutation de la donnée. Le type du codomaine de la fonction tri est donc un type dépendant, puisqu'il varie suivant la valeur de la donnée. Ce type peut être exprimé par la formule :

$$(t \in \text{liste(A)}) \rightarrow \{y \in \text{liste(A)} \mid \text{ord}(y) \wedge \text{perm}(y, t)\}$$

Liste(A) désigne le constructeur du type liste d'objets de type A et **nil** et . sont les constructeurs d'objets de ce type. Nous donnons ci-dessous les règles d'inférence nécessaires à cette preuve :

• **Règle d'introduction des ensembles :**

$$\frac{a \in A \qquad B[a / x] \text{ est vrai}}{a \in \{x \in A \mid B\}} \text{ (R1)}$$

L'élément a appartient à l'ensemble des x de type A ayant la propriété B si et seulement si a est de type A et si la formule B[a / x], obtenue à partir de B en substituant toutes les occurrences libres de x par a, est vraie. Cette règle correspond à la règle d'introduction de l'opérateur d'union disjointe Σ du système de Martin-Löf.

• **Règles d'élimination des ensembles :**

$$\frac{a \in \{x \in A \mid B\}}{a \in A} \text{ (R2)}$$

$$\frac{a \in \{x \in A \mid B\}}{B[a / x] \text{ est vrai}} \text{ (R3)}$$

Si l'élément a appartient à l'ensemble des x de type A ayant la propriété B alors il appartient à A (règle R2) et la formule obtenue après substitution de x par a dans B est vraie (règle R3). R2 et R3 correspondent respectivement aux règles de projection à gauche et à droite de Σ.

• **Règle d'élimination de $\rightarrow$:**

$$\frac{p \in (x \in A) \rightarrow B \qquad a \in A}{p(a) \in B[a / x]} \text{ (R4)}$$

Cette règle est une extension, dans le cadre des types dépendants, de la règle relative au typage de l'application (XY) que nous avons utilisée pour la Logique Combinatoire. Nous supposons d'autre part disposer de propriétés sur les prédicats ord et perm et de l'axiome **nil** ∈ **liste(A)**.

Preuve :

Prouver que le programme précédent est bien de type $(t \in \text{liste}(A)) \rightarrow \{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(y, t)\}$ revient, en utilisant la règle R4, à montrer :

- $\text{nil} \in (\{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(y, t)\}\ [\text{nil} / t])$
- $\text{insert}(a, \text{tri}(s)) \in (\{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(y, t)\}\ [a.s / t])$

C'est à dire :

(a) $\text{nil} \in \{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(y, \text{nil})\}$

(b) $\text{insert}(a, \text{tri}(s)) \in \{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(y, a.s)\}$

• **Preuve de (a) :** en supposant que $\text{ord}(\text{nil}) \wedge \text{perm}(\text{nil}, \text{nil})$ est vrai, nous prouvons directement cette formule par l'application suivante de la règle R1 :

$$\frac{\text{nil} \in \text{liste}(A) \qquad \text{ord}(\text{nil}) \wedge \text{perm}(\text{nil}, \text{nil})}{\text{nil} \in \{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(y, \text{nil})\}}$$

• **Preuve de (b) :** pour mener à bien cette preuve nous utilisons une hypothèse d'induction et la formule définissant la fonction insert. Soient H_1 et H_2 les hypothèses suivantes :

- $\text{tri}(s) \in \{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(y, s)\}$
- $\text{insert} \in ((a \in A) \rightarrow (s \in \{x \in \text{liste}(A) \mid \text{ord}(x)\}) \rightarrow \{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(a.s, y)\})$

insert est une fonction ayant comme données un élément de type A et une liste ordonnée d'éléments de type A. Le type du codomaine est l'ensemble des listes de type A ayant la propriété d'être ordonnées et constituées des éléments de la liste initiale et de l'élément ajouté. Les fonctions sont manipulées sous leur forme curryfiée, c'est pour cela que le type de insert est une fonction du type A dans l'ensemble des fonctions sur les listes ordonnées. La propriété utilisée est $(D_1 \times D_2) \rightarrow D_3 = D_1 \rightarrow (D_2 \rightarrow D_3)$.

Par application de la règle R1, décrivant comment prouver des formules de la forme $a \in \{x \in A \mid B\}$, nous devons montrer :

(c) - $\text{insert}(a, \text{tri}(s)) \in \text{liste}(A)$

(d) - $\text{ord}(\text{insert}(a, \text{tri}(s))) \wedge \text{perm}(\text{insert}(a, \text{tri}(s)), a.s)$

• **Preuve de (c).**

$$\frac{H_2 \qquad a \in A}{\text{insert}(a) \in (s \in \{x \in \text{liste}(A) \mid \text{ord}(x)\}) \rightarrow \{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(a.s, y)\}} \text{(R4)} \quad (H_3)$$

Nous devons maintenant montrer que tri(s) ∈ {x ∈ liste(A) | ord(x)}. Ceci peut être déduit de l'hypothèse d'induction (H_1) de la manière suivante :

$$\frac{\dfrac{\text{tri}(s) \in \text{liste}(A) \quad \text{(R2)} \qquad \dfrac{\text{ord}(\text{tri}(s)) \wedge \text{perm}(\text{tri}(s), s)}{\text{ord}(\text{tri}(s))} \quad \text{(R3)}}{\text{tri}(s) \in \{x \in \text{liste}(A) \mid \text{ord}(x)\}} \text{(R1)} \quad (H_4)$$

Les déductions en pointillé décrivent des déductions classiques que nous ne précisons pas. **insert(a, tri(s)) ∈ liste(A)** se déduit donc de H_3 et H_4 par la déduction suivante :

$$\frac{H_3 \qquad H_4}{\text{insert}(a, \text{tri}(s)) \in \{y \in \text{liste}(A) \mid \text{ord}(y) \wedge \text{perm}(a.\text{tri}(s), y)\}} \text{(R4)} \quad (H_5)$$

$$\frac{}{\text{insert}(a, \text{tri}(s)) \in \text{liste}(A)} \text{(R2)}$$

• **Preuve de (d)** (ord(insert(a, tri(s))) ∧ perm(insert(a, tri(s)), a.s)).

Pour mener à bien cette preuve nous utilisons l'hypothèse d'induction H_1 et la formule H_5 déduite précédemment du type de la fonction insert.

$$\frac{\text{tri(s)} \in \{x \in \text{liste(A)} \mid \text{ord(x)} \wedge \text{perm(x, s)}\}}{\text{ord(tri(s))} \wedge \text{perm(tri(s), s)}}\ (R3) \qquad \frac{\text{insert(a, tri(s))} \in \{y \in \text{liste(A)} \mid \text{ord(y)} \wedge \text{perm(a.tri(s), y)}\}}{\text{ord(insert(a, tri(s)))} \wedge \text{perm(a.tri(s), insert(a, tri(s)))}}\ (R3)$$

$$\frac{\text{ord(tri(s))} \wedge \text{perm(tri(s), s)} \qquad \text{ord(insert(a, tri(s)))} \wedge \text{perm(a.tri(s), insert(a, tri(s)))}}{\text{ord(insert(a, tri(s)))} \wedge \text{perm(insert(a, tri(s)), a.s)}}$$

Pour ce dernier pas nous utilisons la propriété de perm **perm(x1, x2) ∧ perm(a.x1, y1) ⇒ perm(y1, a.x2)**. Ceci termine la preuve. Nous venons donc de prouver que la fonction tri définie par les deux équations **tri(nil) = nil** et **tri(a.s) = insert(a, tri(s))** est bien de type : **(t ∈ liste(A)) → {y ∈ liste(A) | ord(y) ∧ perm(y, t)}.**

Dans le cadre de la Synthèse de Programmes nous ne disposons que du type et nous voulons construire une fonction ayant le type demandé. Ceci revient à utiliser le même système que précédemment en ne connaissant que les parties droites des formules d'affectation, c'est à dire les types. La preuve effectuée est alors une preuve de type et celui-ci ne sera vrai que si il est possible d'exhiber un élément lui appartenant, c'est pour cela qque cette théorie est qualifiée d'intuitioniste. Un type vide est donc faux. Pour synthétiser la fonction tri par insertion à partir du type précédent nous devons donc supposer que les expressions suivantes sont des types :

A
liste(A)
(a ∈ A) → (s∈ {x ∈ liste(A) | ord(x)}) → {y ∈ liste(A) | ord(y) ∧ perm(a.s, y)}

Par exemple **nil** et **insert** sont respectivement des éléments des deux derniers types. Nous supposons d'autre part que le type A n'est pas vide et contient au moins un élément **elem**. Par exemple si A est le type des entiers alors **elem** peut être la constante 0. A partir de ces hypothèses nous pouvons donc construire une preuve du type de la fonction tri, issue de la preuve de correction que nous avons donnée précédemment. La construction du programme sous-jacente à cette preuve est :

- Cas t=nil :

$$\frac{\text{nil} \qquad \text{ord(nil)} \wedge \text{perm(nil, nil)}}{\text{nil}}\ (R1)$$

Cette construction correspond à la déduction nil ∈ {y ∈ liste(A) | ord(y) ∧ perm(y, nil)}. L'occurrence de nil dans la première prémisse désigne un objet de type liste(A) et l'occurrence de nil dans la conclusion désigne un objet de type {y ∈ liste(A) | ord(y) ∧ perm(y, nil)}.

- **Cas t= a . s :** nous faisons l'hypothèse que a . s est un élément de type liste(A) c'est à dire que a ∈ A et s ∈ liste(A). Nous avons alors :

$$\frac{s \qquad tri}{tri(s)}\ (R4)$$

Ceci correspond à la construction associée à la déduction de tri(s) ∈ {x ∈ liste(A) | ord(x)}. Pour ce faire nous avons utilisé l'hypothèse que tri est une fonction de type (s ∈ liste(A)) → {y ∈ liste(A) | ord(y) ∧ perm(y, s)} pour l'argument s. C'est l'hypothèse d'induction.

$$\frac{insert \qquad a}{insert(a)}\ (R4)$$

Ceci est la construction associée à la déduction de insert(a) ∈ (s ∈ {x ∈ liste(A) | ord(x)}) → {y ∈ liste(A) | ord(y) ∧ perm(a.s, y)}).

$$\frac{tri(s) \qquad insert(a)}{insert(a, tri(s))}\ (R4)$$

Ceci est la construction correspondant à la déduction de insert(a, tri(s)) ∈ {y ∈ liste(A) | ord(y) ∧ perm(y, a . s)}.

Nous pouvons donc déduire la définition si t=nil alors nil sinon (* t=a.s *) insert(a, tri(s)) qui permet de décharger

l'hypothèse d'induction qui consistait à supposer que tri était un élément du type de tri, ceci pour l'argument s. Pour obtenir l'objet tri nous devons appliquer la règle $\rightarrow$ introduction qui est :

$$\frac{\begin{array}{c}[x \in A] \\ b \in B\end{array}}{\lambda x \,.\, b \in A \rightarrow B}$$

[x ∈ A] indique que l'hypothèse x ∈ A faite au cours de la preuve peut être déchargée. Dans notre cas, ceci permet de décharger l'hypothèse s ∈ liste(A). Nous n'avons pas détaillé volontairement les règles relatives à l'introduction de conditionnelle et de récursion, celles-ci étant assez complexes, c'est pour cela que nous n'obtenons pas une définition de la fonction tri sous la forme d'un λ-terme.

2. LA LOGIQUE DE HOARE.

Hoare a proposé en 1969 une logique décrivant les preuves de correction de programmes [Hoa 69] vis à vis d'assertions sur ces programmes. Cette logique a ensuite été vue comme un moyen de décrire la sémantique des langages de programmation, sous forme d'assertions vérifiées par les objets du programme.

2.1. Rappels.

Cette logique est définie sur le langage des formules de correction que nous notons FC, constitué à partir du langage des assertions et du langage à définir. Le langage des assertions est un sous-ensemble du Calcul des Prédicats du premier Ordre.

définition.

Une formule de correction est :

– soit une assertion,

– soit une formule de la forme P {S} Q où P et Q sont deux assertions et S une construction du langage à définir. P et Q sont appelées respectivement **précondition** et **postcondition** de FC et (P, Q) forme une **spécification** du programme S.

fin-définition.

L'interprétation associée aux assertions est celle du CP1. Une formule P {S} Q est vraie, dans le cadre de la correction partielle, soit si P est vraie avant exécution de S, l'exécution de S termine et Q est vraie après exécution de S, soit si P est vraie et l'exécution de S ne termine pas ou soit si P est fausse. Si on s'intéresse à la correction totale l'exécution de S doit obligatoirement terminer. Pour un langage de programmation donné, les formules de correction valides sont définies à l'aide d'un système formel basé sur la syntaxe de ce langage. Il décrit les formules de correction attachées à chaque construction du langage en terme des formules de correction attachées aux paramètres de ces

constructions.

Exemples de définitions du langage PASCAL [HoW 73] :

• Axiome de l'affectation :

$$P[t / x] \ \{x := t\} \ P(x) \quad \textbf{(affec)}$$

• Définition de la conditionnelle :

$$\frac{P \wedge B \ \{S_1\} \ Q \qquad P \wedge \neg B \ \{S_2\} \ Q}{P \ \{\textbf{si } B \textbf{ alors } S_1 \textbf{ sinon } S_2\} \ Q} \quad \textbf{(cond)}$$

• Règle de la conséquence :

$$\frac{P \ \{S\} \ Q \qquad P_1 \Rightarrow P \qquad Q \Rightarrow Q_1}{P_1 \ \{S\} \ Q_1} \quad \textbf{(cons)}$$

Cette règle permet de garantir la complétude du système proposé. Elle fait intervenir des preuves de formules classiques, il est supposé que tous les théorèmes du CP1 sont des axiomes de ce système formel. La complétude n'est donc que relative à la complétude du CP1.

fin-exemples.

Décrire la sémantique des langages à l'aide de ce formalisme pose en pratique quelques problèmes, les règles d'inférence devenant vite très compliquées et d'application très restreinte. Ceci est plus lié aux langages de programmation modélisés qu'au formalisme proprement dit. En effet, la logique de Hoare a été principalement employée pour des langages classiques avec affectation et les effets de bord possibles à prendre en compte lui font perdre son caractère intuitif. Cette logique est habituellement employée pour prouver la correction d'un programme S vis à vis d'une spécification (P, Q) ce qui revient à montrer que P {S} Q est un théorème de cette logique. Nous donnons un exemple très simple de preuve de correction, pour illustrer ensuite sur ce même exemple, l'utilisation de la logique de Hoare en synthèse.

Exemple.

Soit le programme :

```
programme SUP ;
    var x, y, max : entier ;
    debut
        si x ≥ y alors max := x sinon max := y
    fin
```

Montrons qu'il vérifie bien la spécification (vrai, max ≥ x ∧ max ≥ y).

Preuve :

Nous utilisons l'instanciation suivante de la règle de la conditionnelle :

$$\frac{\text{vrai} \wedge x \geq y \ \{ \max := x \} \ \max \geq x \wedge \max \geq y \quad \textbf{(a)} \qquad \text{vrai} \wedge \neg(x \geq y) \ \{ \max := y \} \ \max \geq x \wedge \max \geq y \quad \textbf{(b)}}{\text{vrai} \ \{ \textbf{si } x \geq y \textbf{ alors } \max := x \textbf{ sinon } \max := y \} \ \max \geq x \wedge \max \geq y} \ \textbf{(cond)}$$

• **Preuve de (a) :**

$x \geq x \wedge x \geq y \ \{ \max := x \} \ \max \geq x \wedge \max \geq y$ est une instance de l'axiome de l'affectation, (a) peut donc être déduit par la règle de la conséquence par la déduction :

$$\frac{x \geq x \wedge x \geq y \ \{ \max := x \} \ \max \geq x \wedge \max \geq y \qquad x \geq y \Rightarrow x \geq x \wedge x \geq y}{\text{vrai} \wedge x \geq y \ \{ \max := x \} \ \max \geq x \wedge \max \geq y} \ \textbf{(cons)}$$

• **Preuve de (b) :**

De la même manière **(b)** peut être démontré par l'instance de la règle de la conséquence suivante :

$$y \geq x \wedge y \geq y \ \{ \ max := y \ \} \ max \geq x \wedge max \geq y$$

$$\neg(x \geq y) \Rightarrow y \geq x \ \wedge \ y \geq y$$

---------- **(cons)**

$$vrai \wedge \neg(x \geq y) \ \{ \ max := y \ \} \ max \geq x \wedge max \geq y$$

(a) et **(b)** sont donc des théorèmes et par la même **(vrai, max ≥ x ∧ max ≥ y)** est une spécification vérifiée par le programme SUP.

fin-exemple.

2.2. Spécifications = formules ? Programmes = Preuves ?

Le programme décrit une preuve de la spécification puisqu'à chaque construction du langage de programmation est associée une règle d'inférence. Comme ceci a été fait précédemment dans le cadre du typage, nous aimerions établir une correspondance entre spécification et formule de la logique et donc entre preuve en logique de Hoare et preuve de la logique classique. S. Cook [Coo 78] décrit l'interprétation **I** associée aux formules de correction en terme d'une formule de la logique classique, en faisant intervenir explicitement la fonction d'interprétation associée au langage. Sous une forme simplifiée, cette interprétation est :

P {S} Q est un théorème de la logique de Hoare

si et seulement si

$\forall$ s, s' (P(s) $\wedge$ s'= out(S, s) $\Rightarrow$ Q(s')) est un théorème de la logique classique

s et s' désignent des états mémoire, c'est à dire une association nom de variables et valeurs, et **out** est la fonction définissant la sémantique opérationnelle du langage. Cette traduction est correcte, en effet :

- si P(s) est faux alors **I** est vraie,

- si P(s) est vrai et si l'exécution de S termine, c'est à dire si il existe bien une valeur s' telle que s'=**out**(S, s), alors **I** n'est vraie que si Q(s') est vrai,

- si P(s) est vrai et que l'exécution de S ne termine pas alors s'=**out**(s) est faux quelque soit s' est donc **I** est vraie.

Cette interprétation établit bien une correspondance entre **spécification** et **formule du Calcul des Prédicats du Premier Ordre.** Nous pouvons alors chercher à caractériser les preuves décrites par les règles d'inférence. La logique de Hoare est plus souvent assimilée à une logique des programmes qu'à une logique des spécifications, puisqu'elle est généralement utilisée pour la preuve de programmes. Chercher à caractériser les preuves faites dans cette logique n'est pas habituel, caractériser cette correspondance pour l'instruction conditionnelle et l'affectation, celles-ci décrivant des modes de raisonnements utilisés en démonstration.

Exemple 1 :

Par l'interprétation choisie, la règle de la conditionnelle décrit donc comment déduire la formule :

$$\forall\, s, s'\ (P(s) \wedge s' = \mathbf{out}(\mathbf{si}\ B\ \mathbf{alors}\ S_1\ \mathbf{sinon}\ S_2, s) \Rightarrow Q(s'))$$

Les hypothèses nécessaires à cette démonstration sont les interprétations associées aux prémisses $P \wedge B\ \{S_1\}\ Q$ et $P \wedge \neg B\ \{S_2\}\ Q$ et la définition de la fonction **out** pour la conditionnelle qui peut être décrite par la formule :

$$(s' = \mathbf{out}(\mathbf{si}\ B\ \mathbf{alors}\ S_1\ \mathbf{sinon}\ S_2,\ s)) \equiv (B(s) \wedge s' = \mathbf{out}(S_1,\ s) \vee \neg B(s) \wedge s' = \mathbf{out}(S_2,\ s))$$

Cette règle décrit donc la déduction :

$$\frac{\begin{array}{l}\forall\, s, s'\ (P(s) \wedge B(s) \wedge s' = \mathbf{out}(S_1, s) \Rightarrow Q(s')) \\ \forall\, s, s'\ (P(s) \wedge \neg B(s) \wedge s' = \mathbf{out}(S_2, s) \Rightarrow Q(s'))\end{array}}{\forall\, s, s'\ (P(s) \wedge s' = \mathbf{out}(\mathbf{si}\ B\ \mathbf{alors}\ S_1\ \mathbf{sinon}\ S_2, s) \Rightarrow Q(s'))}$$

Elle peut donc être vue comme une description d'un raisonnement de type analyse par cas sur les formules d'interprétation. Et, si on s'intéresse uniquement au cas où l'exécution de S termine alors **out**(S, s) est nécessairement une fonction définie pour toute donnée vérifiant la précondition P. Donc pour chaque s ayant la propriété P il existe un et seul s' tel que s'=**out**(S, s). La formule d'interprétation peut se réécrire en :

$$\forall\, s\ \exists\, s'\ (P(s) \Rightarrow Q(s')) \text{ avec } s' = \mathbf{out}(S, s).$$

La règle précédente exprime maintenant comment prouver constructivement une telle formule. Comme dans la théorie des types, nous pouvons donc isoler la preuve des spécifications et la construction qui découle de cette preuve. La règle de la conditionnelle devient :

$$\frac{\begin{array}{l}\forall\, s\ \exists\, s'\ (P(s) \wedge B(s) \Rightarrow Q(s')) \\ \forall\, s\ \exists\, s'\ (P(s) \wedge \neg B(s) \Rightarrow Q(s'))\end{array}}{\forall\, s\ \exists\, s'\ (P(s) \Rightarrow Q(s'))}$$

La construction associée est :

$$\frac{\mathbf{out}(S_1, s) \quad \mathbf{out}(S_2, s)}{\mathbf{si}\ B(s)\ \mathbf{alors}\ \mathbf{out}(S_1, s)\ \mathbf{sinon}\ \mathbf{out}(S_2, s)}$$

Exemple 2 : La formule d'interprétation associée à l'axiome de l'affectation $P[t / x]\ \{x := t\}\ P(x)$ est :

$$\forall\ s,\ s'\ (P[t / x](s) \wedge s' = \mathbf{out}(x:=t, s) \Rightarrow P(s'))$$

Le résultat de **out(x:=t, s)** est l'état mémoire s dans lequel on associe à x la nouvelle valeur t. Nous notons cette opération $s \cup (x \leftarrow t)$. L'axiome de l'affectation peut donc s'interpréter comme l'axiome :

$$\forall\ s\ \exists\ s'\ (P\ [t / x]\ (s) \Rightarrow P(s'))$$

et la construction sous-jacente est :

$$s \cup (x \leftarrow t)$$

fin-exemples.

Ce parallèle est difficile à mettre en évidence, les langages de programmation classiques ne décrivant pas des méthodes de raisonnements habituels : itérations, structure de données... Les langages fonctionnels sont une classe particulière pour laquelle la correspondance entre preuve en logique de Hoare et preuve en logique classique semble facile à exprimer. C'est une des raisons pour laquelle l'analogie établie dans le cadre de la théorie des types est plus naturelle, le λ-calcul étant, par définition, fonctionnel. Par exemple, la règle de définition de fonction, proposée par Hoare et Wirth [HoW 73], s'énonce de la manière suivante :

> Soit f une fonction définie, pour une liste de paramètres formels x, par son corps S. Rappelons qu'en PASCAL le corps d'une fonction contient des affectations à une variable fictive du nom de la fonction, le rôle de cette variable étant de mémoriser le résultat de la fonction. Si f n'apparait pas libre dans P et que aucune des variables de x n'apparaisse libre dans Q, alors la règle est :

$$\frac{P\ \{S\}\ Q}{\forall\ x, y\ (P \Rightarrow Q[f / f(x, y)])}$$

y désigne l'ensemble des variables libres dans S qui sont considérées comme des paramètres

implicites. Comme nous le verrons dans la seconde partie, cette règle peut ne pas être correcte lorsque les fonctions sont partielles.

Cette règle lie donc directement spécification et formule logique sans faire intervenir la fonction d'évaluation du langage et, par la même, relie preuve de spécification et preuve classique. A une spécification est donc associée une implication qui, rappelons le est, sous sa forme intuitioniste, la flèche des types. Nordström établit d'ailleurs la correspondance suivante entre spécification et type :

Si {P(x), Q(x, y)} est une spécification où x est la donnée et y le résultat, alors tous les programmes vérifiant cette spécification ont le type $(x \in \{ z \in A \mid P(z) \}) \rightarrow \{ y \in B \mid Q(x, y) \}$ où A et B sont les types des objets x et y.

2.3. Application à la Synthèse de Programmes.

Comme nous l'avons fait précédemment pour les systèmes de typage, nous pouvons obtenir directement un système de démonstration de spécifications de programmes. Il suffit de s'abstraire des constructions {S} qui correspondent aux sujets des formules d'affectation de type. De manière analogue, la preuve effectuée produira, par l'intermédiaire des constructions successives sous-jacentes, un programme vérifiant la spécification. Une spécification (P, Q) sera donc valide si et seulement si on peut construire un programme S tel que P { S } Q est un théorème de la logique de Hoare.

Exemple : montrons que la spécification **(vrai, max ≥ x ∧ max ≥ y)** est vraie c'est à dire qu'il existe un programme, SUP, tel que vrai { SUP } max ≥ x ∧ max ≥ y est un théorème de la logique de Hoare. Ceci, revient donc à prouver :

$$\forall s \; \exists s' \; (vrai[s] \Rightarrow (max \geq x \wedge max \geq y)[s'])$$

s et s' sont des états mémoires de la forme {x ← v1, y ← v2, max ← v3) où x, y et max sont les variables du problème et v1, v2 et v3 les valeurs associées à ces variables. P[s] signifie que P est vrai pour les valeurs de s. Nous introduisons les crochets pour distinguer les arguments initiaux des assertions et celui rajouté par la conversion en une formule d'interprétation, c'est à dire l'argument représentant l'état mémoire. En utilisant les correspondances que nous avons données pour l'axiome de l'affectation et la règle de la conditionnelle, nous décrivons cette synthèse comme une preuve constructive de la formule ci-dessus.

Preuve : par l'instanciation de la règle de la conditionnelle suivante :

$$\forall s \; \exists s' \; ((x \geq y)[s] \Rightarrow (max \geq x \wedge max \geq y)[s']) \quad \textbf{(a)}$$
$$\forall s \; \exists s' \; (\neg(x \geq y)[s] \Rightarrow (max \geq x \wedge max \geq y)[s']) \quad \textbf{(b)}$$

---------- **(cond)**

$$\forall s \; \exists s' \; (vrai[s] \Rightarrow (max \geq x \wedge max \geq y)[s'])$$

$\forall$ s $\exists$ s' (((max $\geq$ x $\wedge$ max $\geq$ y) [x / max]) [s] $\Rightarrow$ (max $\geq$ x $\wedge$ max $\geq$ y)[s']) est une instance de l'axiome de l'affectation, (a) peut donc être déduit par la règle de la conséquence de la manière suivante :

$$\forall s \exists s' ((x \geq x \wedge x \geq y)\ [s] \Rightarrow (max \geq x \wedge max \geq y)[s'])\ .$$

$$x \geq y \ \Rightarrow x \geq x \ \wedge \ x \geq y$$

---------- (cons)

$$\forall s \exists s' ((x \geq y)[s] \Rightarrow (max \geq x \wedge max \geq y)[s'])$$

De la même manière (b) peut être démontré par l'instance de la règle de la conséquence suivante :

$$\forall s \exists s' ((y \geq x \wedge y \geq y)\ [s] \Rightarrow (max \geq x \wedge max \geq y)[s'])$$

$$\neg(x \geq y) \ \Rightarrow y \geq x \ \wedge \ y \geq y$$

---------- (cons)

$$\forall s \exists s' (\neg(x \geq y)[s] \Rightarrow (max \geq x \wedge max \geq y)[s'])$$

La construction sous-jacente à cette preuve est donc :

$$s \cup (max \leftarrow x) \qquad\qquad s \cup (max \leftarrow y)$$

---------- (cond)

$$\textbf{si } x \geq y \textbf{ alors } s \cup (max \leftarrow x) \textbf{ sinon } s \cup (max \leftarrow y)$$

Ceci correspond au programme **si** x $\geq$ y **alors** max := x **sinon** max := y.

fin-exemple.

La logique de Hoare fournit donc directement un système de démonstration pour la Synthèse de Programmes, la difficulté étant de caractériser, pour toute construction, les preuves effectivements exécutées.

3. CONCLUSION.

Si nous assimilons les spécifications aux types, les programmes aux λ-termes et la relation **P ∈ T à la relation la construction S vérifie la spécification (P, Q)**, alors ces théories décrivent toutes deux une logique de raisonnement sur des objets de type programmes, bien que la logique de Hoare présente à première vue un caractère plus hétéroclite. En effet, elle est définie en partie à partir du Calcul des Prédicats, en raison de la règle de la conséquence. Cette règle décrit une propriété générale de la relation **la construction S vérifie la spécification (P, Q)** qui est :

> si P_1 est une formule plus faible que P et si Q_1 est une formule plus forte que Q alors P_1 {S} Q_1 est vrai indépendamment de la forme de S.

En fait il peut en être de même dans la théorie des types. Par exemple Martin-Löf, pour décrire sa logique des types, utilise quatre formes de jugement qui sont :

- A est un ensemble (noté A ens),
- a est élément de A (noté a ∈ A),
- a et b éléments de A sont égaux (noté a=b ∈ A),
- A et B sont deux ensembles égaux (noté A=B).

Pour notre part, nous avons utilisé uniquement les deux premiers : le jugement **A ens** qui correspond à la définition des types et le jugement **a ∈ A** qui correspond à la construction d'un élément du type A. Comme pour les deux premières formes de jugement, l'égalité entre ensembles ou éléments d'un même ensemble est décrite par des règles. Par exemple le jugement **A=B** est défini par :

$$\frac{a \in A}{\;\;}$$

$$a \in B$$

La double barre signifie que la déduction peut s'effectuer dans les deux sens. Les règles générales décrivent le fait que = est une relation d'équivalence. Ce sont :

$$\frac{A \text{ ens}}{A = A} \qquad \frac{A = B}{B = A} \qquad \frac{A = B \quad B = C}{A = C}$$

$$\frac{a \in A \qquad A = B}{a \in B}$$

ainsi que toutes les règles relatives à l'égalité, propres à chaque constructeur d'ensembles.

Nous pouvons nous demander qu'elle est l'interprétation de la règle de la conséquence dans les théories des types, c'est à dire quelle propriété générale des types elle décrit. C'est l'inclusion. En effet si nous voulions rajouter un jugement de la forme $A \subseteq B$ défini par :

$$\frac{a \in A}{a \in B}$$

Nous aurions alors les règles suivantes :

$$\frac{a \in A \qquad A_1 \subseteq A}{a \in A_1} \qquad\qquad \frac{f \in A \to B \qquad A \subseteq A_1}{f \in A_1 \to B}$$

(si f est définie pour le domaine A alors, à fortiori, elle est définie pour tout sous-domaine de A)

En combinant ces deux règles nous obtenons :

$$\frac{f \in A \to B \qquad A_1 \subseteq A \qquad B \subseteq B_1}{f \in A_1 \to B_1}$$

Cette dernière règle devient l'équivalent de la règle de la conséquence en assimilant la spécification (P, Q) au type $P \to Q$ et la formule d'affectation $x \in A$ à la relation $A(x)$.

Nous retrouvons donc, dans la théorie des types comme en logique de Hoare, des prémisses ayant des significations différentes ; des jugements d'égalité, des jugements d'appartenance... La théorie des types offre cependant un cadre plus homogène. D'une part les jugements ont une signification propre en dehors des systèmes proposés et d'autre part toute formule peut être déduite dans ces logiques. En logique de Hoare, par contre, la signification des formules P {S} Q est entièrement dépendante de la notion d'exécution et a été définie pour les besoins de la cause. De plus, le Calcul des Prédicats du premier Ordre est un prérequis à la logique de Hoare. Mais ceci offre l'avantage, comme nous le verrons par la suite, de séparer les deux niveaux de preuves nécessaires au raisonnement sur les programmes.

Bien que ces deux approches modélisent le processus de construction de programmes, ceci ne suffit pas pour en envisager la mécanisation. En effet, pour prouver un type ou une spécification, nous disposons de plusieurs règles d'inférence ou axiomes qui peuvent, en plus, être " paramétrés " par des propriétés logiques entre types ou spécifications. La recherche d'une preuve est donc un processus très combinatoire et ces formalisations n'apportent pas de solution quant à la démarche à suivre. Nous allons maintenant présenter les principaux travaux en Synthèse à partir d'une spécification logique. Comme nous le verrons, la plupart ne s'appuient pas explicitement sur ces fondements : des méthodes de preuve ad-hoc sont généralement proposées et certaines intégrent des stratégies permettant de guider le déroulement d'une preuve.

CHAPITRE 3

SYNTHESE A PARTIR DE SPECIFICATION COMPLETE : ETAT DE L'ART.

1. LES PREMIERS TRAVAUX.

Les premiers travaux en Synthèse de Programmes ont été publiés en 1969 et sont dûs à C. Green [Gre 69] et Waldinger & Lee [WaL 69]. En plus de l'intérêt historique de les évoquer, il est a noter que ces travaux décrivent déjà clairement la problématique de la Synthèse de programmes. A cette époque, s'intéresser à un tel sujet est une conséquence naturelle de l'obtention de résultats importants dans différents domaines, notamment :

- La formalisation des techniques de preuve de programmes par Floyd en 1967 [Flo 67] et Hoare en 1969 [Hoa 69]. Une correspondance entre programme et preuve a ainsi été établie de manière rigoureuse et a ensuite été directement utilisée pour la construction de programmes.

- La découverte du Principe de Résolution dû à Robinson en 1965 [Rob 65]. Cette méthode de preuve a permis d'envisager sérieusement la mécanisation de preuve de théorèmes. Mener des preuves en utilisant le principe de Résolution offre deux avantages conséquents : l'unicité de la règle d'inférence n'introduit pas un niveau de combinatoire supplémentaire et la preuve effectuée a un caractère construtif. Par exemple, la preuve de formules de la forme $\exists x\ P(x)$ fournit, si il en existe, un terme t tel que P(t) est vrai. Ce mode de démonstration permet donc de résoudre des problèmes exprimables par des formules logique du premier ordre, relatifs à la recherche d'objets. Les programmes peuvent être vus comme des réponses à des énoncés de la forme $\forall x\ \exists y\ P(x, y)$ où x représente les données et y les résultats calculés par les programmes.

- Le développement de programmes dont le but est de résoudre des classes de problèmes. Citons, par exemple, le Compilateur Heuristique de H. A. Simon [Sim 63]. Ce programme permet de calculer une séquence d'opérateurs permettant de passer d'une situation donnée, décrite dans un langage de description d'états, à une situation désirée. Ce travail est quelquefois recensé comme le premier en synthèse de programmes pour la génération de plans.

Basés tout deux sur la Résolution, les travaux de Waldinger & Lee et de Greene différent dans la manière de l'utiliser. Waldinger & Lee prouvent l'énoncé initial par Résolution à partir de propriétés du domaine du problème. Un programme LISP est ensuite déduit de l'arbre de preuve décoré par les unifications successivement effectuées. Pour qu'une preuve puisse effectivement fournir un programme, Waldinger et Lee limitent l'utilisation de la Résolution afin de n'obtenir que des preuves qu'ils qualifient de primitives. La Résolution Primitive qu'ils utilisent est basée sur les symboles que l'utilisateur déclare comme primitifs (fonctions, prédicats ou variables), c'est à dire les objets qui peuvent apparaître dans le programme final. Par exemple une variable représentant un résultat, ne peut être substituée que par un terme ne contenant que des symboles primitifs. Ceci garantit que la solution obtenue pourra effectivement être évaluée. Greene, pour sa part, obtient directement le programme par Résolution à partir de trois types de clauses décrivant respectivement :

– les propriétés du domaine du problème ;

– une axiomatique des constructions du langage de programmation visé. Par exemple, la conditionnelle de LISP peut être définie par les deux implications suivantes :

$$x=nil \Rightarrow cond(x, y, z)=z$$
$$\neg(x=nil) \Rightarrow cond(x, y, z)=y$$

– le lien entre les fonctions primitives du langage cible et les domaines de problèmes. Par exemple, l'équivalence $ATOM(x) \equiv \neg(atom(x)=nil)$ établit le lien entre la fonction booléenne atom et la relation ATOM définie sur le domaine des listes.

Greene propose d'autre part des tactiques permettant de choisir à quel pas de la preuve utiliser ces différentes informations. Nous ne rentrerons pas dans les détails de ces travaux mais les illustrons par le développement d'un même exemple, emprunté à Waldinger et Lee [LeW 69]. Le problème consiste à construire une fonction qui renvoit 1 si la liste donnée est un atome et 0 sinon. La question peut être formulée par :

$$\forall x \; \exists y \; [(ATOM(x) \wedge y=1) \vee (\neg ATOM(x) \wedge y=0)]$$

Cette formule, une fois niée et mise sous forme clausale, se transforme en :

$$(Q_1) \qquad \neg ATOM(a) \vee \neg(y=1)$$
$$(Q_2) \qquad ATOM(a) \vee \neg(y=0)$$

a est un symbole de constante introduit par l'étape de Skolemisation de la mise sous forme clausale.

1.1. La méthode de Waldinger et Lee.

Nous supposons disposer de l'axiome $x=x$ sur le domaine des entiers, soit A cet axiome. D'autre part nous déclarons que le prédicat ATOM est primitif.

• **Preuve** :

L'application de la résolution sur Q1 et A d'une part et sur Q2 et A d'autre part permet d'engendrer les clauses intermédiaires $\neg ATOM(a)$ et $ATOM(a)$ et donc la clause vide. L'énoncé initial est donc prouvé.

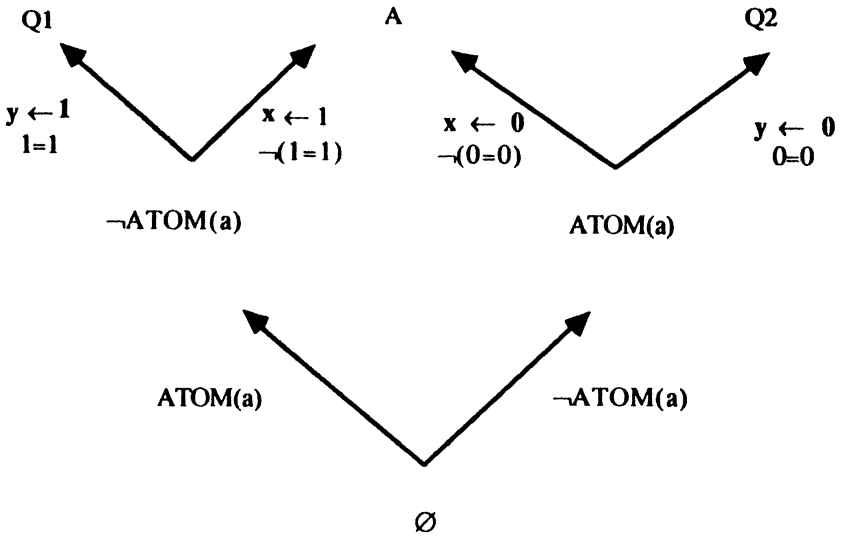
• **Arbre de preuve** : Le principe de construction en est le suivant :

- Un nœud représente une clause.

- Il existe un arc de A à B si et seulement si A est une clause résolvante obtenue directement à partir de B. Chaque arc (A, B) est étiqueté par :

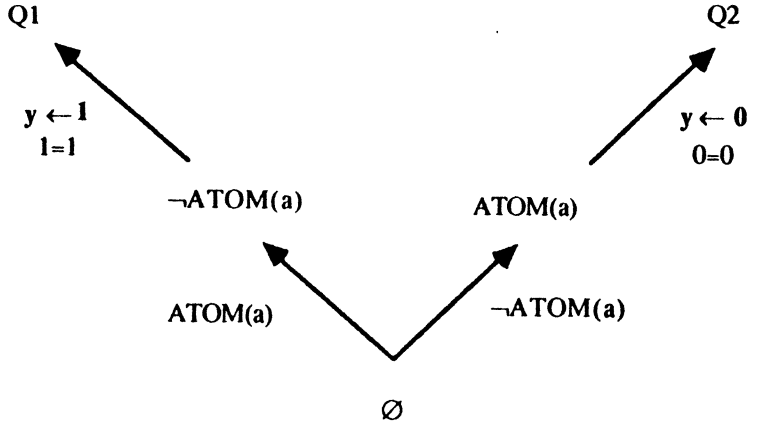
- les substitutions effectuées pour obtenir A à partir de B ;
- la négation du littéral supprimé de B.

Pour l'exemple traité, l'arbre produit est :

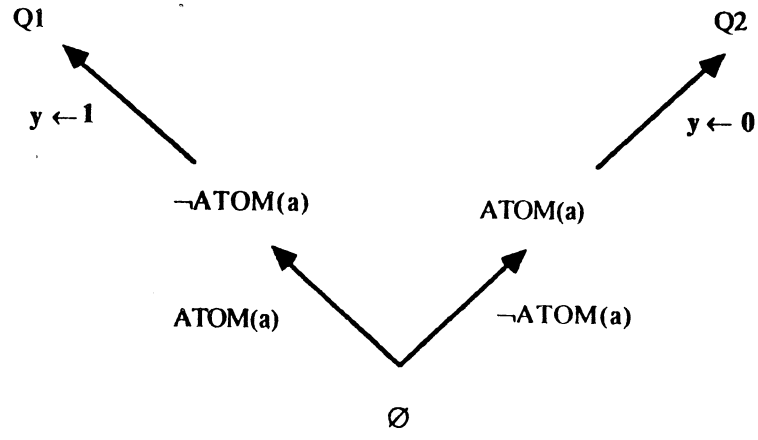


Les étiquettes des branches correspondant aux littéraux supprimés décrivent les conditions qui doivent être vérifiées pour que les instanciations étiquetant ces mêmes branches soient applicables. Le programme est donc issu directement de cet arbre, après que celui-ci ait été simplifié. Les simplifications applicables sur l'arbre précédent sont les suivantes :

- Les conditions ¬(1=1) et ¬(0=0) sont toujours fausses, le nœud A ne sera donc jamais atteint. L'arbre précédent est alors simplifié en :



- Il n'y a plus de choix possible à partir des nœuds intermédiaires ¬ATOM(a) et ATOM(a), les prédicats 1=1 et 0=0 ne sont donc plus nécessaires. L'arbre final est :



Le programme LISP obtenu est donc :

```
( LAMBDA (A)
( PROG (Y)
(COND ((ATOM A) (GO A1)))
(SETQ Y 0)
(RETURN Y)
A1 (SETQ Y 1)
(RETURN Y)))
```

fin-exemple.

1.2. La méthode de Greene.

Cet exemple, non traité par Greene, a été reconstitué à partir de ce qui est dit dans [Gre 69]. Les axiomes que nous utiliserons sont les suivants :

(A_1) $(x_1=\text{nil}) \vee \text{cond}(x_1, y_1, z_1)=y_1$

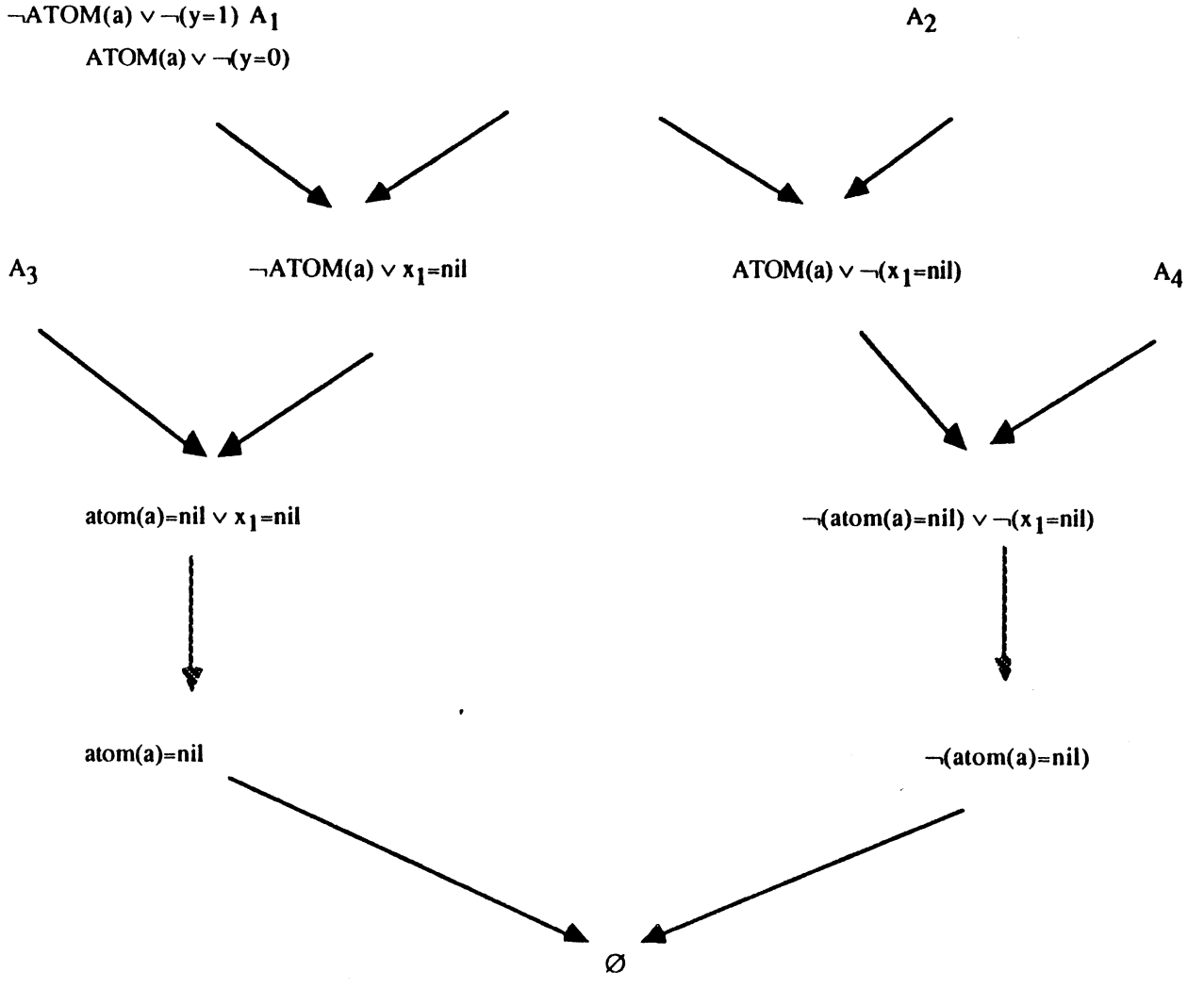
(A_2) $\neg(x_1=\text{nil}) \vee \text{cond}(x_1, y_1, z_1)=z_1$

(A_3) $\text{ATOM}(x) \vee (\text{atom}(x)=\text{nil})$

(A_4) $\neg\text{ATOM}(x) \vee \neg(\text{atom}(x)=\text{nil})$

Ce sont les formes clausales de la définition de la fonction **cond** et du lien entre la fonction **atom** de LISP et le prédicat ATOM du domaine des listes.

• **Preuve :**



Les flèches en pointillé indiquent des étapes de factorisation. En effet le principe de résolution utilisé étant la résolution binaire il est donc nécessaire, pour être complet, de s'autoriser la factorisation. Les différentes substitutions effectuées sont :

$$y \leftarrow cond(x_1, y_1, z_1)$$
$$y_1 \leftarrow 1$$
$$z_1 \leftarrow 0$$
$$x_1 \leftarrow atom(a)$$

Par composition la substitution obtenue est y ← **cond(atom(a), 1, 0)**, elle décrit un programme répondant au problème posé. Rappelons que **a** est une constante désignant une donnée quelconque du problème.

fin-exemple.

Cet exemple, vraiment très simple puisque le prédicat ATOM est considéré comme primitif, est suffisant pour montrer les utilisations possibles de la Résolution en Synthèse de programmes. Des programmes plus complexes ont été obtenus par ces méthodes, notamment des programmes récursifs. L'utilisateur doit alors fournir, dans la question, l'hypothèse d'induction nécessaire. Ces deux travaux montrent déja clairement la nécessité de mener la preuve de manière adéquate. Waldinger et Lee utilisent la notion de Preuve Primitive dans le but de limiter les preuves possibles à celles correspondant à un programme. Greene, pour sa part, classifie les connaissances nécessaires en trois catégories : 1) les propriétés du domaine du problème, 2) le lien entre primitives du langage cible et définition de ce domaine, 3) l'axiomatisation des constructions du langage cible. Ces trois types de connaissances interviennent alors à des étapes différentes de la preuve. Greene émet d'ailleurs l'idée qu'il serait plus efficace de séparer totalement le niveau de preuve pour la construction de programme et le niveau de preuve sur le domaine du problème.

2. SYNTHESE DE PROGRAMMES : LES PRINCIPAUX TRAVAUX.

Avant de présenter de manière détaillée les travaux illustrant l'état actuel de la synthèse déductive de programmes, nous présentons leurs principales caractéristiques.

• L'énoncé est classiquement une formule logique du premier ordre de la forme $\forall x \exists y\ Q(x, y)$ dans laquelle x représente les données, y les résultats et Q définit le lien attendu entre les données et les résultats. Pour un tel énoncé deux buts peuvent être poursuivis :

– Construire un programme qui calcule une valeur possible r de y, ceci pour chaque x, telle que Q(x, r) est vrai. Ce programme décrit donc une fonction f(x). Dans ce cas nous disons que *le programme est une solution possible de l'énoncé* et nous avons :

$$\forall x\ [Q(x, f(x)) \Rightarrow \exists y\ Q(x, y)]$$

– Construire un programme qui fournit tous les r possibles tels que Q(x, r) est vrai, ceci pour chaque x. Le programme engendré ne décrit plus le calcul d'une fonction mais d'un processus non déterministe a tel que :

$$\forall x\ \forall y\ [Q(x, y) \equiv y=a(x)]$$

Nous disons alors que le programme est équivalent à l'énoncé. Par rapport à la première solution, la complexité du processus de synthèse est accrue car la preuve menée doit préserver l'équivalence entre formules et non pas seulement l'existence d'une solution.

La majorité des travaux en Synthèse s'intéresse à la recherche d'une solution possible. D'autres cherchent à préserver l'équivalence entre programme synthétisé et énoncé initial mais les exemples traités portent toujours sur des énoncés définissant un résultat unique. Il est donc difficile de se rendre compte de l'intérêt et surtout des difficultés liées à ce but.

• Le langage cible, dans lequel les programmes sont exprimés, est généralement un langage applicatif, fonctionnel ou logique.

• Les différents travaux se classifient suivant la méthode de preuve adoptée. Nous les avons regroupés en trois approches : Transformationnelle, Déductive et Réécriture.

– **Approche Transformationnelle** (Manna et Waldinger, Burstall, Clark et Darlington, Bibel ...). Dans cette approche l'énoncé initial est modifié, par des règles de transformations, jusqu'à obtenir un nouvel énoncé qui est jugé être un programme. Une suite d'énoncés E_1, ..., E_n, où E_1 est l'énoncé initial et E_n est le programme, est donc successivement élaborée. Si on s'intéresse à la recherche d'une solution à l'énoncé initial alors les transformations effectuées devront être telles que :

$$\forall x \; \exists y \; (E_n \Rightarrow E_{n-1} \Rightarrow \; ... \Rightarrow E_1)$$

Et, si on s'intéresse à la recherche de toutes les solutions possibles alors ces transformations devront être telles que :

$$\forall x \; \forall y \; (E_n \equiv E_{n-1} \equiv \; ... \equiv E_1)$$

– **Approche Déductive** (Manna et Waldinger, Smith ...). L'énoncé initial, de la forme $\forall x \; \exists y \; Q(x, y)$, est prouvé de telle manière qu'il soit possible d'extraire un programme calculant la valeur de y pour chaque x. Par exemple, les travaux de Greene et de Waldinger & Lee, que nous venons de présenter, relèvent de cette approche. Dans cette approche, on cherche une preuve possible de l'énoncé est donc une valeur possible de y pour chaque x. Calculer toutes les réponses possibles reviendrait à chercher toutes les preuves de l'énoncé qui mènent à des résultats différents. Pour ce faire, il suffirait de prouver la formule :

$$\forall x \; \exists Y \; \forall y \; (y \in Y \equiv Q(x, y))$$

où Y désigne l'ensemble des résultats possibles.

– **Approche Réécriture** (Dershowitz, Kodratoff, Perdrix ...). La procédure de Complétion est appliquée sur un ensemble d'équations décrivant l'énoncé et les propriétés du domaine du problème. Le programme est obtenu sous forme d'un ensemble de règles de réécriture dérivées par complétion. Cette approche relève, comme nous le verrons, de l'approche transformationnelle mais est souvent recensée comme une approche en tant que telle en raison de l' outil très particulier qu'elle utilise.

A première vue, les approches transformationnelle et déductive sont très différentes. Dans la première l'existence du résultat n'est pas prouvé et donc il pourrait être possible d'obtenir un programme à partir d'un énoncé n'ayant pas de modèle valide, ce qui n'est bien sur pas le cas dans l'approche déductive. Mais, en pratique ceci ne semble pas être le cas car les transformations effectuées garantissent, au moins en partie, la construction du résultat cherché et donc son existence. Lorsque les énoncés définissent des résultats existants les traitements dans ces deux approches sont très proches.

• En dehors des stratégies inhérentes au mode de preuve adopté, des stratégies propres à la construction de programme sont utilisées pour maitriser la combinatoire de la synthèse. Elles sont en général assez similaires car elles découlent de résultats bien connus (lien entre récursion et définition inductive ...) ou elles décrivent des tactiques classiques de programmation. Leur rôle, essentiel, est de permettre de maitriser la complexité de la dérivation du programme. Dans

cette présentation générale nous ne ferons qu'évoquer les stratégies proposées, un chapitre leur étant consacré dans la dernière partie.

Nous présentons maintenant les principaux travaux illustrant ces différentes approches. Le but de cette présentation est de montrer les particularités de ces travaux et la similarité de la démarche mise en œuvre, ceci sur le développement d'un même exemple. L'exemple retenu consiste à construire un programme calculant le maximum d'un ensemble d'entiers. L'énoncé choisi est, pour toutes les approches :

$$\forall E \; \exists \max [E \neq \emptyset \Rightarrow \max \in E \wedge SUP(\max, E)]$$

E désigne l'ensemble initial, max le résultat à construire. SUP(max, E) est vrai si et seulement si max est supérieur ou égal à tous les éléments de l'ensemble E ou si E est vide. Nous supposons que les relations SUP et $\in$ sont connues, c'est-à-dire que nous disposons de définitions et de théorèmes sur ces relations. D'autre part, nous ne nous occupons pas ici des problèmes de représentation des objets, le programme résultat sera exprimé en fonction des primitives du type abstrait ensemble. Par la suite nous n'indiquerons plus les quantificateurs portant sur E et max.

2.1. Approche Transformationnelle.

Rappelons que dans cette approche l'énoncé initial est transformé jusqu'à obtenir un nouvel énoncé qui est jugé être un bon programme. Il faut donc caractériser les transformations nécessaires à la synthèse de programmes et proposer des stratégies pour choisir, pour un énoncé donné, la suite de transformations à effectuer pour " se rapprocher " d'un programme. Trois travaux principaux relèvent de cette approche : le système de transformation de programmes de Burstall et Darlington [BuD 77] étendu à la synthèse de programmes, le système DEDALUS de Manna et Waldinger [MaW 79] et le système LOPS de Bibel [Bib 78]. Ce dernier est conçu comme un outil d'aide à la construction de programmes, l'utilisateur guide, par l'intermédiaire de stratégies, les transformations à effectuer. Citons aussi C. Hogger [Hog 78], [Hog 81] qui étudie la dérivation de programmes sous forme de clauses de Horn à partir d'énoncés logiques. L'unicité de formalisme entre le langage d'énoncés et le langage cible lui permet de formaliser clairement la démarche sous-jacente à l'approche transformationnelle. Il utilise deux types de transformation, des règles de simplification qui permettent de simplifier un énoncé et une règle de substitution qui permet d'utiliser un théorème pour réécrire un énoncé. Les travaux que nous allons présenter se différencient par les règles de transformations et les stratégies utilisées.

Nous présentons d'abord le traitement de l'exemple du maximum dans le cadre de la logique pure, en dérivant une suite d'énoncés équivalents. Nous exposons ensuite les systèmes de Burstall et Darlington et de Manna et Waldinger. Nous décrivons enfin les principales stratégies du système LOPS en les illustrant sur l'exemple.

2.1.1. Présentation de l'exemple.

Nous définissons la fonction maximum par :

$$\forall E, \max (E \neq \emptyset \Rightarrow ((\text{maximum}(E)=\max) \equiv (\max \in E \wedge SUP(\max, E))))$$

La suite de transformations va porter sur l'énoncé initial $E \neq \emptyset \Rightarrow (\max \in E \wedge SUP(\max, E))$ et nous utiliserons les théorèmes suivants :

(1) $(E \neq \emptyset) \equiv (\exists\, m, E_1 (E = \{m\} \cup E_1))$

(2) $x \notin \emptyset$

(3) $(x \in \{m\} \cup E_1) \equiv (x = m \vee x \in E_1)$

(4) $SUP(n, \emptyset)$

(5) $SUP(n, \{m\} \cup E_1) \equiv (n \geq m \wedge SUP(n, E_1))$

(6) $m \geq m$

(7) $SUP(m_1, E) \equiv (E \neq \emptyset \wedge \exists\, m_2 (m_2 \in E \wedge SUP(m_2, E) \wedge m_1 \geq m_2) \vee E = \emptyset)$

L'énoncé initial est donc : $E \neq \emptyset \Rightarrow max \in E \wedge SUP(max, E)$.

Dérivation :

– **Pas 1.** En utilisant l'équivalence (1) et en appliquant la propriété de substitutitivé de l'égalité, on obtient $\exists\, m, E_1 (E = \{m\} \cup E_1 \Rightarrow max \in \{m\} \cup E_1 \wedge SUP(max, \{m\} \cup E_1))$, ce qui se simplifie en : $max \in \{m\} \cup E_1 \wedge SUP(max, \{m\} \cup E_1)$.

– **Pas 2.** Par (3) et (4) l'énoncé devient : $(max \in E_1 \vee max = m) \wedge max \geq m \wedge SUP(max, E_1)$.

– **Pas 3.** En distribuant le $\wedge$ sur le $\vee$:

$(max \in E_1 \wedge max \geq m \wedge SUP(max, E_1)) \vee (max = m \wedge max \geq m \wedge SUP(max, E_1))$.

– **Pas 4.** L'égalité max=m et le théorème (6) permettent de simplifier la seconde partie de la disjonction et d'obtenir : $(max \in E_1 \wedge max \geq m \wedge SUP(max, E_1)) \vee (max = m \wedge SUP(m, E_1))$.

– **Pas 5.** Nous introduisons le tiers-exclus $E_1 \neq \emptyset \vee E_1 = \emptyset$ en mettant la formule sous forme conjonctive :

$(max \geq m \wedge E_1 \neq \emptyset \wedge max \in E_1 \wedge SUP(max, E_1))$
$\vee (max \geq m \wedge E_1 = \emptyset \wedge max \in E_1 \wedge SUP(max, E_1))$
$\vee (max = m \wedge SUP(m, E_1))$

– **Pas 6.** En simplifiant la seconde partie de la disjonction par le théorème (2) et les lois $A \wedge faux \equiv faux$ et $A \vee faux \equiv A$, nous obtenons :

$(max \geq m \wedge E_1 \neq \emptyset \wedge max \in E_1 \wedge SUP(max, E_1)) \vee (max = m \wedge SUP(m, E_1))$.

– **Pas 7.** Nous utilisons maintenant l'équivalence définissant la fonction maximum pour la donnée E_1, c'est à dire $E_1 \neq \emptyset \Rightarrow ((\text{maximum}(E_1)=\text{max}) \equiv (\text{max} \in E_1 \wedge \text{SUP}(\text{max}, E_1)))$. Cette hypothèse permet de remplacer l'occurrence $\text{max} \in E_1 \wedge \text{SUP}(\text{max}, E_1)$ par $\text{maximum}(E_1)=\text{max}$. La formule obtenue est donc :

$$(\text{max} \geq m \wedge E_1 \neq \emptyset \wedge \text{maximum}(E_1)=\text{max}) \vee (\text{max}=m \wedge \text{SUP}(m, E_1))$$

– **Pas 8.** En utilisant le théorème (7) nous obtenons :

$$(\text{max} \geq m \wedge E_1 \neq \emptyset \wedge \text{maximum}(E_1)=\text{max}) \vee (\text{max}=m \wedge E_1=\emptyset)$$
$$\vee (\text{max}=m \wedge E_1 \neq \emptyset \wedge \exists\, m_2\, (m_2 \in E_1 \wedge \text{SUP}(m_2, E_1) \wedge m \geq m_2))$$

– **Pas 9.** Et en utilisant la définition de $\text{maximum}(E_1)$:

$$(\text{max} \geq m \wedge E_1 \neq \emptyset \wedge \text{maximum}(E_1)=\text{max}) \vee (\text{max}=m \wedge E_1=\emptyset)$$
$$\vee (\text{max}=m \wedge E_1 \neq \emptyset \wedge \exists\, m_2\, (\text{maximum}(E_1)=m_2 \wedge m \geq m_2))$$

– **Pas 10.** $\exists\, m_2\, (\text{maximum}(E_1)=m_2 \wedge m \geq m_2)$ se réécrit en $m \geq \text{maximum}(E_1)$ par la propriété de substitutivité de l'égalité. L'énoncé devient :

$$(\text{max} \geq m \wedge E_1 \neq \emptyset \wedge \text{maximum}(E_1)=\text{max})$$
$$\vee (\text{max}=m \wedge E_1=\emptyset)$$
$$\vee (\text{max}=m \wedge E_1 \neq \emptyset \wedge \text{maximum}(E_1) \geq m)$$

fin dérivation.

Nous avons donc obtenu une formule équivalente à l'énoncé initial puisque toutes les transformations effectuées préservent l'équivalence. Exprimé sous forme fonctionnelle, le programme sous-jacent à cette formule est :

$\text{maximum}(\{m\} \cup E_1) \leftarrow$ **si** $E_1=\emptyset$ **alors** m
sinon
si $\text{maximum}(E_1) \geq m$ **alors** $\text{maximum}(E_1)$
sinon m

fin-exemple.

2.1.2. Le système de transformation de Burstall et Darlington.

Darlington montre dans [Dar 75] comment le système de transformation de programmes proposé par Burstall et lui-même peut être utilisé pour la synthèse de fonctions, en étendant simplement le langage des équations récursives aux expressions ensemblistes. La même stratégie, appelée pliage-dépliage, que celle proposée pour la transformation

est utilisée. Les équations manipulées sont orientées et sont de la forme $f(e_1, ..., e_n) \leftarrow E$ où $e_1, ..., e_n$ sont des expressions ne contenant que des variables ou des constructeurs et E est une expression quelconque qui définit $f(e_1, ..., e_n)$. L'ensemble des équations est manipulable à l'aide de règles d'inférence qui décrivent des transformations valides sur les équations. Ces règles sont les suivantes :

– **Règle de Définition.** Cette règle permet d'introduire une définition de fonction c'est à dire une nouvelle équation dont la partie gauche n'est une instance d'aucune autre partie gauche de définition.

– **Règle d'Instanciation.** Cette règle permet d'introduire une instance d'une équation déja existante.

– **Dépliage.** Cette règle permet de remplacer une occurrence d'un terme fonctionnel apparaissant dans une partie droite par sa définition. Son application est la suivante :

Si E ← E' et F ← F' sont deux équations et s'il existe une occurrence G dans F' d'une instance de E alors la règle de dépliage permet d'ajouter l'équation F ← F" où F"est obtenu à partir de F' en remplaçant G par E' correctement instancié.

– **Pliage.** Cette règle permet de remplacer une occurrence de définition apparaissant dans une partie droite par le terme fonctionnel qu'elle définit :

S'il existe une occurrence G dans F' d'une instance de E' alors la règle de pliage permet d'ajouter l'équation F ← F" où F"est obtenu à partir de E en remplaçant G par E correctement instancié.

– **Application de lois** sur les parties droites des équations : commutativité, associativité ...

Burstall et Darlington proposent une tactique pour appliquer ces règles d'inférence, tactique qui a été prouvée totalement correcte par L. Kott sous certaines conditions (plus de dépliages que de pliages) [Kot 78], [Kot 82].

2.1.3. Le système DEDALUS de Manna et Waldinger.

Manna et Waldinger ont développé un système expérimental, DEDALUS (DEDuctive ALgorithm Ur-Synthesizer), basé sur l'approche transformationnelle. Les transformations qu'ils proposent permettent d'appliquer des théorèmes sur le domaine du problème ou d'introduire des primitives du langage cible. Ces transformations ne sont pas définies de manière formelle, ce qui rend la compréhension de ce système difficile. Néanmoins l'intérêt du travail de Manna et Waldinger réside dans le fait qu'ils recensent des principes de base de la programmation applicables en synthèse de programmes (comment introduire des récursions, des fonctions auxiliaires ...). D'autre part Manna et Waldinger [MaW 79] proposent également des tactiques permettant de guider le choix des théorèmes à utiliser dans les transformations. Ils leur associent d'une part des conditions d'applicabilité, ceci afin d'éviter les impasses (ne pas utiliser une équivalence dans un sens puis, juste après, dans l'autre sens ...) . Ils utilisent d'autre part des critères tels que l'application de ces théorèmes permettent rapidement l'introduction d'une primitive du langage cible. Les théorèmes permettant de résoudre le problème pour des valeurs donnés ont la priorité la plus forte, puis ceux permettant d'introduire directement une récursion, puis les autres ...

Les tactiques proposées par Manna et Waldinger et par Burstall et Darlington sont des solutions très ponctuelles. La nécessité de développer des stratégies plus appropriées, notamment en ce qui concerne la synthèse de fonctions récursives ou l'introduction de fonctions auxiliaires, a été rapidement montrée : technique du pliage forcé de Darlington [Dar 75], exemples de synthèse d'algorithmes de tri développées par Clark et Darlington [ClD 78] ...

2.1.4. Le système LOPS de Bibel.

Le système LOPS (LOgical Program Synthesis), proposé par Bibel, est basé sur un ensemble de stratégies qui décrivent une suite particulière de transformations à effectuer. Ces stratégies ont été élaborées à partir de l'étude d'exemples [Bib 78], [Bib 80]. La démarche générale, pour un énoncé donné, consiste à déterminer une suite de stratégies à appliquer jusqu'à obtenir un énoncé exécutable. Nous détaillons ci-dessous les stratégies GUESS, DOMAIN et GET-REC. Utilisées l'une à la suite de l'autre, eles permettent de synthétiser des programmes récursifs lorsque la donnée est une collection d'objets et lorsque le résultat est construit à partir de ces objets. C'est le cas de notre problème, puisque le résultat est un élément de l'ensemble initial.

• GUESS et DOMAIN doivent être considérées simultanément. Leur application revient à faire une conjecture sur le résultat. Si celle-ci s'avère convenir alors le résultat ou une partie de celui-ci est trouvé. Dans le cas contraire ces stratégies pourront permettre l'introduction de récursion. La conjecture sur la valeur du résultat se fait à partir des relations de l'énoncé liant les données et le résultat. Par des critères sémantiques, DOMAIN retient la conjonction de ces relations la plus efficace de la manière suivante :

Soit P l'ensemble des conjonctions possibles de ces relations et y la variable désignant le résultat cherché. Pour chaque conjonction C, soient q(C)=cardinal({y : C(y)}) et s(C) le nombre de pas nécessaires pour déterminer un y arbitraire tel que C(y) est vrai. L'expression q(i) * s(i) permet de définir un ordre total sur P. La conjonction retenue est le plus petit élément dans cet ordre.

• La stratégie GET-REC cherche à introduire un appel récursif de la fonction. La première étape consiste à sélectionner un schéma de récursion parmi l'ensemble des schémas disponibles dans le système. La seconde étape demande de reformuler, si possible, l'énoncé en faisant intervenir l'hypothèse d'induction découlant du schéma retenu. Cette étape nécessite l'aide d'un démonstrateur de théorèmes manipulant des théorèmes sur le domaine du problème.

Traitement de l'exemple (proposé dans [Bib 78] et [Bib 80]).

- Application des stratégies GUESS et DOMAIN. La conjecture se fait à partir des relations max $\in$ E et SUP(max, E). DOMAIN choisit la première car trouver un élément de E nécessite moins de pas que trouver une valeur supérieure à tous les éléments de E. L'application de la stratégie GUESS consiste alors à prendre un élément quelconque m de E et à considérer les deux cas max=m et max$\neq$m. GUESS transforme donc l'énoncé en :

$$\forall E, m\ \exists max\ (m \in E \Rightarrow (E \neq \emptyset \Rightarrow max \in E \wedge SUP(max, E) \wedge (max = m \vee max \neq m)))$$

Le $\vee$ désigne ici la disjonction exclusive. Remarquons que la nouvelle formulation obtenue n'est pas équivalente à l'énoncé initial.

- Application de la stratégie GET-DNF. Le but de cette stratégie est de réécrire l'énoncé sous forme normale disjonctive, ce qui donne :

$$\forall E, m\ \exists max\ (m \in E \Rightarrow (E \neq \emptyset \Rightarrow max \in E \wedge SUP(max, E) \wedge max \neq m) \vee (E \neq \emptyset \Rightarrow max \in E \wedge SUP(max, E) \wedge max = m))$$

Rappelons que $E \neq \emptyset \Rightarrow (max \in E \wedge SUP(max, E) \equiv maximum(E))$. Dans la seconde partie de la disjonction, le résultat est trouvé, en raison de l'égalité max=m, cette formule peut donc se simplifier en : $(E \neq \emptyset \Rightarrow maximum(E)=m)$.

- Application de la stratégie GET-REC. GET-REC s'applique sur la première partie de la disjonction (cas max≠m) et nécessite un schéma de récursion faisant intervenir max, m, E et la relation $\in$. Nous l'obtenons à partir du théorème :

$$m \in E \Rightarrow ((max \in E \wedge max \neq m) \equiv max \in E/m \wedge max \neq m))$$

E /m désigne l'ensemble des éléments de E autres que m. D'après l'induction sous-jacente à ce théorème, l'appel récursif introduit sera de la forme maximum(E /m). GET-REC nécessite de connaître un théorème donnant une formule équivalente à SUP(max, E) et faisant intervenir l'hypothèse SUP(max', E /m), par exemple :

$$m \in E \Rightarrow (SUP(max, E) \equiv (m \leq max \wedge SUP(max, E/m))$$

On obtient alors le nouvel énoncé :

$$m \in E \Rightarrow (max \in E/m \wedge SUP(max, E/m) \wedge m \leq max) \vee maximum(E)=m$$

L'appel récursif maximum(E /m) n'est défini que si E /m est différent de l'ensemble vide. Pour pouvoir introduire cet appel il faut réécrire l'énoncé en :

$$m \in E \Rightarrow ((E/m \neq \emptyset \wedge max \in E/m \wedge SUP(max, E/m) \wedge m \leq max)) \vee (E/m = \emptyset \wedge max \in E/m \wedge SUP(max, E/m) \wedge m \leq max) \vee maximum(E)=m)$$

Cette même étape était nécessaire dans la présentation précédente de l'exemple (pas 5). L'énoncé se simplifie donc en :

$$m \in E \Rightarrow ((E/m \neq \emptyset \wedge max \in E/m \wedge SUP(max, E/m) \wedge m \leq max) \vee maximum(E)=m)$$

Comme nous supposons par hypothèse d'induction que $max \in E/m \wedge SUP(max, E/m)$ définit maximum(E /m), nous obtenons finalement :

$$m \in E \Rightarrow ((max=maximum(E/m) \wedge m \leq max \wedge E/m \neq \emptyset) \vee (maximum(E)=m))$$

Rappelons que la disjonction est exclusive, maximum(E) est donc égal à m si et seulement si m est strictement supérieur à maximum(E/m) ou si E /m est l'ensemble vide.

- **Application de la stratégie GET-EP.** La valeur du résultat dépend donc de la condition $E/m \neq \emptyset \wedge m \leq max$. Cette condition ne peut pas être directement évaluée puisque max désigne le résultat dont la valeur dépend de cette condition. Le rôle de la stratégie GET-EP est de rendre cette formule évaluable. Dans ce cas ceci est trivial en raison de l'égalité max=maximum(E /m). L'énoncé final, et donc le programme, est décrit par la formule :

$$m \in E \Rightarrow (max=maximum(E/m) \wedge m \leq maximum(E/m) \wedge E/m \neq \emptyset) \vee (maximum(E)=m)$$

Remarquons que l'évaluation de ce programme peut poser problème puisque maximum(E /m) est indéfini lorsque l'ensemble E est réduit à l'élément m.

fin-exemple.

Les stratégies de LOPS sont donc des guides plus généraux que les tactiques dont nous avons parlés jusqu'à présent. En effet, elles ne décrivent non plus une transformation mais des suites de transformations à effectuer pour se rapprocher d'un programme. Notamment GET-REC, dont le rôle est d'introduire des appels récursifs, se décompose en plusieurs étapes nécessitant des théorèmes ayant des formes particulières (donnée d'une induction, puis théorèmes permettant de réécrire l'énoncé en faisant apparaître cette induction). Mais ces stratégies ne sont pas validées sémantiquement; elles ne sont pas décrites formellement comme des enchainements particuliers de transformations sur les énoncés mais sont plutôt assimilées à des tactiques de programmation. Il est donc difficile de raisonner sur ces stratégies. Il en résulte certaines imprécisions lors de leur application. La dérivation d'un programme à l'aide de ces stratégies comporte encore des pas qui doivent être effectués de manière aveugle, c'est à dire sans guide aucun. GUESS, DOMAIN et GET-REC sont les principales stratégies employées dans LOPS et permettent, comme nous l'avons déjà dit, de traiter assez naturellement des problèmes tels que le résultat est un sous-ensemble des données. En effet, dans ce cas, la conjecture faite par GUESS fournit directement une induction sur la donnée puisqu'elle revient à en extraire une partie. Mais ces stratégies peuvent s'appliquer à d'autres problèmes : dans [Fro 85], un exemple de synthèse d'un programme calculant le quotient et le reste de la division euclidienne de deux entiers est présenté. Trouver la conjecture à mettre en place et la décomposition des données qui en découle devient plus acrobatique.

Une nouvelle implantation du système LOPS est actuellement en cours de développement à l'université de Münich. Dans sa version finale les stratégies seront assistées d'une base de connaissances, d'un démonstrateur de théorèmes, d'un générateur et d'un explorateur d'exemples ainsi que de l'utilisateur [BiH 84]. Le démonstrateur sera capable, lorsqu'une formule s'avère ne pas être vraie, de fournir des contre-exemples, dans le but de particulariser cette formule jusqu'à ce qu'elle soit effectivement valide. Le générateur fournira des modèles de formules et l'explorateur sera chargé de les généraliser. Ces généralisations devront alors être prouvées ou réfutées par le démonstrateur. Le but de ces deux composants est de simuler ce que fait le programmeur lorsqu'il est amené à représenter une situation d'une façon ou d'une autre (dessin ...) pour maitriser un problème.

2.2. Approche Déductive.

Dans cette approche, l'énoncé initial est prouvé. Si cette formule est un théorème alors la preuve effectuée doit être " adéquate ", c'est à dire elle doit fournir un moyen effectif de construire les résultats. Peu de travaux en Synthèse de Programmes sont présentés comme mettant en œuvre une démonstration dans une telle logique. Le plus connu et le mieux formalisé est le système des tableaux déductifs de Manna et Waldinger que nous présentons de manière détaillée sur l'exemple.

2.2.1. Les tableaux déductifs de Manna et Waldinger.

Manna et Waldinger [MaW 80], [MaW 85] proposent un système de preuve utilisant des techniques d'unification, d'induction mathématique et de réécriture. Le déroulement de la preuve est mémorisé dans un tableau et des règles d'inférence permet de déduire de nouvelles lignes de ce tableau. Chaque ligne contient soit une assertion, soit un but auxquels peut être associé une expression qui décrit le programme élaboré lors de la déduction de ce but ou de cette assertion. Le découpage en assertions et buts n'est fait que pour une raison de lisibilité, une assertion pouvant être réécrite en un but en la niant et réciproquement.

Définition :

> **Un tableau est valide, pour une interprétation donnée si, lorsque toutes les instances des assertions sont vraies alors au moins une instance d'un but est vraie, ou si une au moins des assertions est fausse.**

fin-définition.

Le tableau initial sera donc vrai si le but **true** ou l'assertion **false** est déduite. Les principales règles de déduction sont :

– **Des règles de transformation** permettant de réécrire une assertion ou un but en une formule équivalente. Ces règles sont de la forme **r → s si P** où r et s sont des formules équivalentes lorsque P est vrai. Ces règles s'appliquent de la manière suivante :

> Si une assertion E contient un sous terme t unifiable avec r par l'unificateur θ, alors la nouvelle assertion **(if P then E [t / s])** θ est engendrée. Pθ désigne la formule obtenue par application de la substitution θ sur P et **E [t / s]** est obtenue en remplaçant t par s dans E. Si E est un but alors le nouveau but engendré est $P\theta \wedge E[t / s]\theta$.

– **Des règles de découpage** d'assertions et de buts.

– **Une règle d'induction** mathématique basée sur le principe d'induction bien fondée. Soit $P(x) \Rightarrow \exists y\, Q(x, y)$ un théorème à prouver et soit f la fonction calculant y à partir de x. Par hypothèse d'induction, on peut engendrer l'assertion :

$$\textbf{if } \inf(x_1, x) \textbf{ then if } P(x_1) \textbf{ then } Q(x_1, f(x_1))$$

inf est une relation d'ordre bien fondé tel que x_1 est inférieur à x dans cet ordre.

– **Des règles de résolution** s'appliquant sur des formules non obligatoirement sous forme clausale. Leur application correspond informellement à l'analyse par cas. Par exemple, la règle de résolution entre un but G et une assertion A (GA-résolution) est la suivante :

> Si A et G contiennent respectivement des sous-termes A1 et G1 unifiables par l'unificateur θ et qui ne sont pas sous la portée d'un quantificateur alors l'application de cette règle engendre le nouveau but $(G\ [G_1 / \textbf{true}] \wedge \neg A\ [A_1 / \textbf{false}])\ \theta$.

Les autres règles de Résolution sont la AA-résolution entre deux assertions, la GG-résolution entre deux buts et la AG-résolution entre une assertion et un but. Elles découlent de la règle précédente en raison de la dualité existant entre assertions et buts. Si t est le sous-terme commun et θ l'unificateur en résultant, alors :

- la AA-résolution entre A_1 et A_2 produit l'assertion $(A_1\ [t / \textbf{true}] \vee A_2\ [t / \textbf{false}])\ \theta$,
- la GG-résolution entre G_1 et G_2 produit le but $(G_1\ [t / \textbf{true}] \wedge G_2\ [t / \textbf{false}])\ \theta$,
- la AG-résolution le but $(G\ [t / \textbf{false}] \wedge \neg A\ [t\ / \textbf{true}])\ \theta$.

Pour toutes les règles de Résolution l'expression associée à la formule déduite est définie comme suit :

> si les deux formules ont une expression associée, respectivement t_1 et t_2, alors l'expression associée à la formule déduite est $(\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2)\ \theta$, sinon c'est soit $t_1\ \theta$ soit $t_2\ \theta$ ou soit vide.

Une stratégie permet de limiter l'application des règles de résolution : elle est basée sur la polarité, positive ou négative, associée à chaque formule [Mur 82]. Soit POL(A) la polarité de la formule A : Si A est une assertion alors POL(A) est négative et si A est un but alors POL(A) est positive. La polarité des sous-formules est définie à partir de la formule dans laquelle elles apparaissent. Par exemple :

- $POL(\textbf{if } A \textbf{ then } B) = p \Rightarrow POL(B) = p \wedge POL(A) = inv(p)$
- $POL(A \wedge B) = p \Rightarrow POL(A) = p \wedge POL(B) = p$

Nous désignons une polarité positive par + et une polarité négative par -. inv(p) désignela polarité inverse de p, c'est à dire inv(+) = - et inv(-) = +.

Une résolution dont l'effet est de remplacer une sous-formule par **true** n'est effectuée, par cette stratégie, que si la polarité associée à cette sous-formule est positive. Inversement, on ne remplacera une sous-formule par **false** que si sa polarité est négative.

Exemple : Considérons l'assertion **if A(y, d) then B(b, y)** et le but **A(c, x) ∧ B(x, a)**. Nous avons alors :

$POL(\textbf{if } A(y, d) \textbf{ then } B(b, y)) = -$

$POL(A(y, d)) = +$

POL(B(b, y)) = -

POL(A(c, x) ∧ B(x, a)) = +

POL(A(c, x)) = +

POL(B(x, a)) = +

L'application de la GA-résolution sur l'occurrence A produit le but :

((A(c, x) ∧ B(x, a)) [A(c, x) / **true**]) ∧ ¬(**if** A(y, d) **then** B(b, y)) [A(y, d) / **false**]) {y ← c, x ← d}

En appliquant la substitution {y ← c, x ← d} cette formule se simplifie en B(d, a) ∧ ¬(**if false then** B(b, c)). Or **if false then A** se réduit en **false**. Cette application engendre donc le but trivial **false**. En utilisant la stratégie de polarité cette déduction n'a pas lieue puisque A(y, d) a une polarité positive et que la GA-résolution sur cette occurrence a pour effet de la remplacer par **false**. Comme nous le voyons sur cet exemple, cette stratégie permet d'éviter les déductions qui engendrent des buts réduits à **false** ou des assertions réduites à **true**. Ces cas ne nous intéressent pas, puisque nous voulons obtenir le but **true** ou l'assertion **false**.

fin-exemple.

L'utilisation de ce système comme interpréteur d'un langage logique et fonctionnel, appelé TABLOG, est proposé dans [MMW 84]. Murray a montré [Mur 82] que les règles précédentes et les deux transformations ¬**false** → **true** et **if P then false** → ¬**P** constituent un système complet pour la logique du premier ordre, c'est à dire qu'il existe une dérivation dans ce système pour chaque formule valide du CP1.

Traitement du problème du maximum.

Le tableau initial est :

assertions	buts	programme
A1. $E \neq \emptyset$		
B2.	max ∈ E ∧ SUP(max, E)	max

La donnée E est considérée comme une constante et max comme une variable. Ce tableau est donc valide si :

- soit $E \neq \emptyset$ est faux ;

- soit $E \neq \emptyset$)est vrai et il existe une instance de max telle que $max \in E \wedge SUP(max, E)$ est vrai. L'expression associée obtenue lors de la dérivation du but true sera alors un programme vérifiant l'énoncé initial.

Dans la suite nous indiçons par B_i les lignes buts et par A_i les lignes assertions. Les lignes du tableau seront représentées par les couples {X, E} où X est soit une assertion soit un but et E est l'expression associée à X si nécessaire.

Preuve :

– En appliquant les règles de transformation :

– $x \in E \rightarrow x=choix(E) \vee x \in reste(E)$ **si** $E \neq \emptyset$

– $SUP(m, E) \rightarrow m \geq choix(E) \wedge SUP(m, reste(E))$ **si** $E \neq \emptyset$

sur B_2 nous obtenons :

$$\{ E \neq \emptyset \wedge (max=choix(E) \vee max \in reste(E)) \wedge max \geq choix(E) \wedge SUP(max, reste(E)),\ max\}\ (B_3)$$

Les fonctions choix et reste sont supposées prédéfinies et sélectionnent respectivement un élément quelconque de E et le reste.

– En appliquant la GA-résolution entre B_3 et A_1 :

$$\{true \wedge (max=choix(E) \vee max \in reste(E)) \wedge max \geq choix(E) \wedge SUP(max, reste(E)) \wedge \neg false,\ max\}\ (B_4)$$

Cette formule se réécrit, en distribuant le $\wedge$ sur le $\vee$ et en simplifiant à l'aide de règles de transformation logique que nous ne précisons pas, en :

$$\{(max=choix(E) \wedge max \geq choix(E) \wedge SUP(max, reste(E))) \vee (max \in reste(E) \wedge max \geq choix(E) \wedge SUP(max, reste(E))),\ max\}\ (B_5)$$

– En appliquant la règle de découpage de but relative à la disjonction sur B_5, nous obtenons les deux nouveaux buts :

$$\{max \in reste(E) \wedge max \geq choix(E) \wedge SUP(max, reste(E)),\ max\} \quad (B_6)$$

$$\{max=choix(E) \wedge max \geq choix(E) \wedge SUP(max, reste(E)),\ max\} \quad (B_7)$$

• **Traitement du but B_6.**

B_6 contient une instance du but initial B_2. Par application de la règle d'induction bien fondée on peut engendrer l'assertion :

$$\{\textbf{if inf}(E_1, E)\ \textbf{then if}\ E_1 \neq \emptyset\ \textbf{then maximum}(E_1) \in E_1 \wedge \textbf{SUP}(\textbf{maximum}(E_1), E_1),\}\ (A_8)$$

inf désigne une relation d'ordre bien fondé sur les ensembles, par exemple l'inclusion stricte.

– En appliquant la GA-résolution entre B_6 et cette assertion, le sous-terme commun étant max ∈ reste(E) ∧ SUP(max, reste(E)) nous obtenons :

$$\{\textbf{maximum(reste(E))} \geq \textbf{choix(E)} \wedge \neg(\textbf{if inf(reste(E), E) then if reste(E)} \neq \emptyset\ \textbf{then false}),\ \texttt{maximum(reste(E))}\}\ (B_9)$$

Ceci se simplifie en :

$$\{\textbf{maximum(reste(E))} \geq \textbf{choix(E)} \wedge \textbf{inf(reste(E), E)} \wedge \textbf{reste(E)} \neq \emptyset,\ \texttt{maximum(reste(E))}\}\ (B_{10})$$

Nous supposons que inf(reste(E), E) peut être démontré à partir de règles faisant intervenir la cardinalité des ensembles. B_{10} se réduit alors en {**maximum(reste(E))** ≥ **choix(E)** ∧ **reste(E)** ≠ ∅, `maximum(reste(E))`} (B_{11}).

• **Traitement du but B_7 {max=choix(E) ∧ max ≥ choix(E) ∧ SUP(max, reste(E)),** `max`**}.**

Nous supposons disposer des assertions supplémentaires suivantes :

(A_{12})	**m=m**
(A_{13})	**m ≥ m**
(A_{14})	**if SUP(m_1, E) ∧ $m_2 \geq m_1$ then SUP(m_2, E)**

– Par GA-résolution entre le but B_7 et successivement A_{12} et A_{13} nous obtenons :

$$\{\textbf{SUP(choix(E), reste(E))},\ \texttt{choix(E)}\}\ (B_{15})$$

– En appliquant la règle SUP(m, E) → true si E=∅.nous obtenons : {**reste(E)** =∅, `choix(E)`} (B_{16}).

– D'autre part, la GA-résolution entre B_{15} et A_{14} produit :

$$\{\neg(\text{if SUP}(m_1, \text{reste(E)}) \wedge \text{choix(E)} \geq m_1 \text{ then false}),\ \texttt{choix(E)}\}$$

C'est-à-dire $\{\text{SUP}(m_1, \text{reste(E)}) \wedge \text{choix(E)} \geq m_1,\ \texttt{choix(E)}\}$ (B_{17}).

– En utilisant la GA-résolution entre B_{17} et l'hypothèse d'induction A_8 :

$$\{\text{choix(E)} \geq \text{maximum(reste(E))} \wedge \neg(\text{if inf(reste(E), E) then if reste(E)} \neq \emptyset \text{ then maximum(reste(E))} \in \text{reste(E)} \wedge \text{false}),\ \texttt{choix(E)}\}$$

Nous supposons que inf(reste(E), E) peut être prouvé, cette formule se simplifie donc en :
$\{\text{choix(E)} \geq \text{maximum(reste(E))} \wedge \text{reste(E)} \neq \emptyset,\ \texttt{choix(E)}\}$ (B_{18}).

• **Nous avons donc obtenu :**

$\{\text{maximum(reste(E))} \geq \text{choix(E)} \wedge \text{reste(E)} \neq \emptyset,\ \texttt{maximum(reste(E))}\}$ (B_{11})

$\{\text{reste(E)} = \emptyset,\ \texttt{choix(E)}\}$ (B_{16})

$\{\text{choix(E)} \geq \text{maximum(reste(E))} \wedge \text{reste(E)} \neq \emptyset,\ \texttt{choix(E)}\}$ (B_{18})

– En utilisant alors la GA-Résolution entre B_{11} et l'assertion if $\neg(m2 \geq m1)$ then $m_1 \geq m_2$, nous dérivons :

$$\{\text{reste(E)} \neq \emptyset \wedge \neg(\text{if } \neg(\text{choix(E)} \geq \text{maximum(reste(E))}) \text{ then false}),\ \texttt{maximum(reste(E))}\}$$

C'est à dire $\{\text{reste(E)} \neq \emptyset \wedge \neg(\text{choix(E)} \geq \text{maximum(reste(E))}),\ \texttt{maximum(reste(E))}\}$ (B_{19}).

– Puis par GG-résolution entre B_{18} et B_{19} :

$$\{\text{vrai} \wedge \text{reste(E)} \neq \emptyset \wedge \neg(\text{false}),$$
$$\texttt{if choix(E)} \geq \text{maximum(reste(E))} \texttt{ then choix(E) else maximum(reste(E))}\}$$

C'est-à-dire :

$$\{\text{reste(E)} \neq \emptyset,$$
$$\texttt{if choix(E)} \geq \text{maximum(reste(E))} \texttt{ then choix(E) else maximum(reste(E))}\}\ (B_{20})$$

– Puis, par GG-Résolution avec B_{16} ({**reste(E)=∅**, `choix(E)` })nous déduisons le but **true** avec l'expression associée :

```
if reste(E) = ∅ then choix(E)
else  if choix(E) ≥ maximum(reste(E)) then choix(E)
      else maximum(reste(E))
```

Le tableau initial a donc pu être prouvé puisque nous avons obtenu le but **true** et l'expression ci-dessus est le programme synthètisé.

fin-exemple.

Dans le développement de cet exemple nous nous sommes rendus compte de l'importance de l'ordre dans lequel sont effectuées les déductions. En effet, soient les buts B_{11}, B_{16} et B_{18} obenus lors de la preuve :

{maximum(reste(E)) ≥ choix(E) ∧ reste(E) ≠ ∅ , `maximum(reste(E))`**}** (B_{11})

{reste(E) = ∅ , `choix(E)`**}** (B_{16})

{choix(E) ≥ maximum(reste(E)) ∧ **reste(E) ≠ ∅** , `choix(E)`**}** (B_{18})

Nous aurions pu choisir de faire d'abord disparaitre la condition reste(E) ≠ ∅ de B_{16} et B_{18} par GG-Résolution avec B_{11}. Nous aurions alors obtenu les deux buts :

– **{maximum(reste(E)) ≥ choix(E),**
`if reste(E)=∅ then choix(E) else maximum(reste(E))`**}**

– **{choix(E) ≥ maximum(reste((E)),**
`if reste(E)=∅ then choix(E) else choix(E)`**}**

Par application de la GG-Résolution entre ces deux buts, nous aurions dérivé le programme :

```
if choix(E) ≥ maximum(reste(E))
        then if reste(E)=∅ then choix(E) else choix(E)
        else if reste(E)=∅ then choix(E) else maximum(reste(E))
```

Ce programme n'est pas correct puisque maximum(reste(E)) est indéfini lorsque reste(E) est l'ensemble vide. L'évaluation de la codiition initiale n'est donc pas toujours possible. Ce problème provient du fait que les règles de Résolution sont basées sur le tiers exclus $A \vee \neg A$ et ne tiennent pas compte du fait que A peut ne pas être défini.

Dans le cadre de la Synthèse de Programmes, nous sommes amenés à manipuler des fonctions partielles (par exemple maximum n'est définie que pour des ensembles non vides) et il faut donc en tenir compte explicitement, ce qui n'est pas fait dans le système de démonstration utilisé par Manna et Waldinger.

2.2.2. D'autres travaux.

La synthèse déductive a aussi été étudiée comme une application de certains résultats obtenus en logique et en démonstration automatique. Citons d'abord les travaux issus de la logique intuitioniste, qui, comme nous l'avons vu, est un modèle adéquat pour la construction de programmes : citons Hsiang [Hsi 82], Goto [Got 79], Tyugu [Tyu 77] et le système NJL de Goto basé sur l'interprétation de Gödel de l'arithmétique intuitioniste de Heyting. Ces travaux ont un intérêt plus théorique que pratique dans le sens où ils ne se préoccupent pas des aspects stratégiques nécessaires à la Synthèse de programmes. D'autres travaux sont issus de l'étude de la mécanisation possible de preuves inductives de formules de la forme $\forall x\ \exists y\ Q(x, y)$. Prouver inductivement de telles formules revient à construire une fonction récursive calculant une valeur de y pour chaque valeur de x possible. Il est à noter que de telles preuves sont faciles à mettre en place, si tant est que l'on dispose de l'hypothèse d'induction appropriée. S'intéresser à ces preuves est un cas particulier des preuves nécessaires en synthèse de programmes qui nécessite notamment en plus des raisonnements de type analyse par cas et utilisation de lemmes intermédiaires. Le système PRECOMAS de M. Franova exécute par exemple des preuves inductives par filtrage constructif à partir des définitions des opérateurs apparaissant dans la formule [Fra 84], [Fra 85]. Dans ce système l'hypothèse d'induction utilisée découle directement des définitions disponibles de ces opérateurs. Citons aussi le système CYPRESS de D. Smith qui, bien qu'il ne soit pas présenté sous forme d'un système de preuve, relève de l'approche déductive. Les programmes sont construits par instanciation progressive de schémas de programmes génériques auquels sont associés des formules à vérifier, elles aussi génériques. Ces formules décrivent les conditions nécessaires pour qu'une instance d'un schéma soit correcte vis-à-vis d'un énoncé de problème. L'utilisation de schémas de programmes, justifiés par la preuve de correction, avait déjà été proposée par S. Gerhart qui, dans [Ger 75], décrit différents schémas itératifs. Smith propose des stratégies permettant de prouver les formules associées aux schémas de programmes, notamment pour des schémas de type *Divide and Conquer*. Le système de preuve que nous utiliserons et qui sera présenté dans la seconde partie peut être vu comme une formalisation de cette démarche, les stratégies proposées par Smith s'y greffant sans difficultés.

2.3. L'Approche Réécriture.

Dans cette approche, l'outil de démonstration utilisé est la procédure de Complétion [KnB 70]. Cette procédure a été proposée initialement par Knuth-Bendix dans le but d' ngendrer des systèmes de réécriture permettant de décider de la validité d'identités dans les théories équationnelles. Nous rappelons brièvement les différentes étapes de cette procédure.

Procédure de Complétion.

Soit $\mathfrak{E}$ un ensemble d'équations, $\mathfrak{R}$ le système de réécriture produit et $>$ un ordre bien fondé monotone sur les termes. Tant qu'il reste des équations non traitées dans $\mathfrak{E}$ faire :

- Choisir une équation M=N (ou N=M) orientable par $>$ en $M \rightarrow N$.
- Ajouter la règle $M \rightarrow N$ à $\mathfrak{R}$.
- Réduire les parties droites des règles déjà engendrées à l'aide de cette règle.
- Ajouter à $\mathfrak{E}$ toutes les paires critiques obtenues à partir de cette nouvelle règle et de $\mathfrak{R}$.
- Enlever les règles dont la partie gauche contient une instance de M.

– Utiliser $\mathfrak{R}$ pour réduire les deux parties des équations restantes dans $\mathfrak{E}$ et enlever celles qui se réduisent à des identités.

Si $\mathfrak{E}$ est vide après exécution de cette procédure alors il y a succès sinon il n'est pas possible de trouver un système de réécriture pour l'ordre considéré car il existe des équations non orientables. Rappelons que cette procédure est semi-décidable.

Rappel : Il existe une paire critique entre deux règles $L \rightarrow R$ et $L' \rightarrow R'$ n'ayant aucune variable en commun si L contient un sous-terme t, non réduit à une variable, unifiable avec L' ou réciproquement. La paire critique engendrée est l'équation $R\theta = (L[R' / t])\theta$ (réciproquement $R'\theta=(L'[R / t])\theta$) où θ est l'unificateur obtenu et L[R' / t] désigne la formule obtenue après substitution de t par R' dans L.

2.3.1.Utilisation de la procédure de Complétion.

Les équations portent sur des formules logiques, l'interprétation associée à l'égalité est donc l'équivalence de la logique. Le système de réécriture initial contient les propriétés nécessaires sur le domaine du problème ainsi que l'énoncé donné sous la forme $Q(x, y) \rightarrow F(x, y)$. F est le nom du prédicat à synthétiser ($F(x, y) \equiv f(x)=y$) et Q la formule décrivant le lien attendu entre la donnée x et le résultat y. En réécriture toutes les variables libres sont supposées quantifiées universellement, il en résulte que le programme obtenu permettra de calculer F(x, y) pour tout x et y. D'autre part l'ordre de réécriture choisi doit être tel que le terme définissant le prédicat à synthétiser est plus grand que ce dernier, qui, lui même doit être plus grand que les constantes **true** et **false**. Cet ordre guide la synthèse et influe donc directement sur le programme dérivé. La procédure de Complétion permet de déduire une nouvelle définition du prédicat F, équivalente à la première. On obtient un programme lorsqu'on dispose d'un ensemble de règles où F apparait en partie gauche et qui définissent F complétement. L'ordre de réécriture choisi est donc très important puisqu'il permet de décider si une équation produit ou non une définition du prédicat F suivant l'orientation donnée à une équation portant sur F. Notons que la procédure de Complétion se charge de mettre en place les appels récursifs sans traitement particulier, par superposition avec l'énoncé initial.

La synthèse par Réécriture est un cas particulier de l'approche transformationnelle dans le sens où elle transforme l'énoncé initial $Q(x, y) \equiv F(x, y)$ en un ensemble de nouvelles équivalences définissant F. L'existence d'un résultat vérifiant Q(x, y) pour chaque x n'est en aucune sorte prouvée, puisque F est un nouveau symbole que l'on définit. La procédure de Complétion complète le système initial mais ne démontre pas que l'énoncé est un théorème de la théorie initiale. Fronhöfer et Furbach comparent, sur des exemples, la synthèse par Complétion et le système de Burstall et Darlington. En effet ces approches permettent toutes deux de dériver de nouvelles équations à partir d'un ensemble initial [FrF 86]. Bien que le calcul de paires critiques puisse être assimilé aux règles de pliage et de dépliage du système de Burstall et Darlington, l'orientation utilisée diffère et les règles permettant de dériver de nouvelles équations ne sont pas les mêmes. Dans le système de Burstall et Darlington les équations sont orientées par la flèche signifiant *est définie par* . Dans l'approche réécriture, l'orientation des règles est fournie par l'ordre choisi et les nouvelles équations sont obtenues uniquement par superposition entre les parties gauches de règles. Il n'est par exemple pas possible de simuler le pliage et le dépliage pour une même règle. Fronhöfer et Furbach montrent néanmoins la similarité de ces deux approches bien que, pour certains exemples, les résultats obtenus ne sont pas les mêmes.

Cette approche de la Synthèse de Programmes n'a été que peu étudiée, si ce n'est par Dershowitz [Der 83], [Der 85], Kodratoff et Perdrix [KoP 83], [Per 84]. Il en résulte certaines imprécisions, notamment la notion de programme n'est pas clairement définie, comme nous le verrons sur l'exemple. Dershowitz implante actuellement un système de

synthèse basé sur la procédure de Complétion en utilisant les environnements REVE [Les 83] et RRL [KaS 83]. Les problèmes subsistant concernent le choix de l'ordre bien fondé, la possibilité de manipuler des équations conditionnelles, l'introduction de fonctions intermédiaires qui, pour le moment, doivent être fournies par l'utilisateur.

2.3.2. Traitement de l'exemple.

Nous ne dérivons que les pas de la Complétion nécessaires à la synthèse et nous ne précisons pas l'ordre employé.

- L'énoncé est : $\mathbf{max \in E \wedge SUP(max, E) \rightarrow MAX(E, max)}$

- Les règles employées sont :

(R_1) $m \in \emptyset \rightarrow$ false

(R_2) $m \in \{m\} \cup E_1 \rightarrow$ true

(R_3) $x \in \{m\} \cup E_1 \rightarrow x \in E_1$ si $x \neq m$

(R_4) $SUP(n, \emptyset) \rightarrow$ true

(R_5) $SUP(n, \{m\} \cup E_1) \rightarrow SUP(n, E_1) \wedge n \geq m$

(R_6) $m \geq m \rightarrow$ true

Ces règles décrivent les définitions des opérateurs SUP et $\in$. Nous supposons d'autre part disposer des règles de réécriture relatives à la simplification de formules booléennes. Nous avons employé une règle conditionnelle, R_3, ceci pour une question de clarté. Mais les mêmes règles pourraient être décrites dans une théorie purement équationnelle en fusionnant les règles R_2 et R_3 en $x \in \{m\} \cup E_1 \rightarrow (x=m) \vee (x \in E_1)$.

• **Déroulement de la Complétion :**

– En superposant l'énoncé avec $\mathbf{R_1}$ nous obtenons l'équation : false $\wedge$ SUP(max, $\emptyset$) = MAX($\emptyset$, max). Après simplification, cette égalité se réduit à false = MAX($\emptyset$, max) et s'oriente en :

$$\mathbf{MAX(\emptyset, max) \rightarrow false}$$

– En superposant l'énoncé avec $\mathbf{R_2}$ nous obtenons : $MAX(\{m\} \cup E_1, m) = SUP(m, \{m\} \cup E_1)$. Or $SUP(m, \{m\} \cup E_1)$ se réduit en $SUP(m, E_1) \wedge m \geq m$ par la règle $\mathbf{R_5}$. Cette formule se réduit à son

tour en SUP(m, E_1) par la règle R_6 et la simplification A $\wedge$ true $\rightarrow$ A. L'équation devient donc MAX({m} $\cup E_1$, m) = SUP(m, E_1) et nous l'orientons en :

$$\mathbf{MAX(\{m\} \cup E_1, m) \rightarrow SUP(m, E_1)}$$

Remarquons qu'en choisissant cette orientation nous admettons implicitement que la relation SUP puisse intervenir dans le programme final.

– En superposant l'énoncé avec R_3 nous obtenons l'équation :

$$MAX(\{m\} \cup E_1, max) = max \in E_1 \wedge \supseteq SUP(max, \{m\} \cup E_1) \quad \text{si } max \neq m$$

La partie droite de cette égalité se réduit en max $\in E_1 \wedge$ SUP(max, E_1) $\wedge$ max $\geq$ m par la règle R_5. Ce terme se réduit à son tour à l'aide de l'énoncé initial. Nous obtenons finalement l'équation MAX({m} $\cup$ E_1, max) = MAX(E_1, max) $\wedge$ max $\geq$ m sous la condition max$\neq$m. L'orientation choisie est :

$$\mathbf{MAX(\{m\} \cup E_1, max) \rightarrow MAX(E_1, max) \wedge max \geq m \quad si\ max \neq m}$$

fin complétion.

Nous avons donc dérivé les trois règles :

(a) $\mathbf{MAX(\emptyset, max) \rightarrow false}$

(b) $\mathbf{MAX(\{m\} \cup E_1, m) \rightarrow SUP(m, E_1)}$

(c) $\mathbf{MAX(\{m\} \cup E_1, max) \rightarrow MAX(E_1, max) \wedge max \geq m \quad si\ max \neq m}$

Cet ensemble de règles constitue un programme définissant le prédicat MAX. Il reste à montrer que la définition obtenue est complète, c'est à dire qu'elle couvre toutes les valeurs possibles du domaine d'entrée. Nous n'en ferons pas la démonstration, ceci étant évident pour cet exemple. Des méthodes permettant de vérifier cette propriété ont été décrites par Thiel [Thi 84], Dershowitz [Der 83], Kounalis et Zhang [KoZ 85] et Common [Com 86], ceci pour des formes particulières de définitions.

Dershowitz montre comment exécuter un tel programme toujours à l'aide de la procédure de Complétion [Der 83]. Si nous désirons, par exemple, trouver le maximum de l'ensemble {1, 3, 2} la question se formule par :

$$\mathbf{MAX(\{1, 3, 2\}, max) \rightarrow Réponse(max)}$$

{1, 3, 2} est une notation abrégée de {1} $\cup$ {3} $\cup$ {2} $\cup \emptyset$. Réponse est un prédicat supérieur aux constantes true et

false et inférieur à tout autre symbole dans l'ordre choisi. Le programme s'exécute en superposant successivement les différentes règles contenant le prédicat **Réponse** avec une règle du système de réécriture obtenu par synthèse jusqu'à obtenir la règle **Réponse(val) → true**. val est alors le résultat cherché. Cette démarche est similaire à l'exécution d'un programme PROLOG. Sur l'exemple précédent, l'exécution se déroule de la manière suivante :

– La superposition de la question avec (c) produit MAX({3, 2}, max) ∧ **max ≥ 1 → Réponse(max) si** max≠1.

– La superposition entre la cette règle et (b) engendre l'équation SUP(3, {2}) ∧ **3 ≥ 1 = Réponse(3)** si 3≠1. Cette équation se réduit, par les règles définissant l'égalité et le prédicat ≥ que nous ne précisons pas, en : SUP(3, {2}) = **Réponse(3)**. Le nouveau " but " devient : SUP(3, {2}) → **Réponse(3)**.

– La superposition de cette règle avec la règle R_5 produit l'équation SUP(3, ∅) ∧ **3 ≥ 2 = Réponse(3)** qui se réduit en **true = Réponse(3)** par la règle R_4. Cette équation s'oriente en **Réponse(3) → true** ce qui termine l'exécution.

L'algorithme ci-dessus, obtenu également par H. Perdrix [Per 84], ne correspond pas à celui que nous avons synthétisé jusqu'à présent puisque la règle (b), décrivant le cas où l'élément sélectionné est le maximum, ne fait pas intervenir d'appels récursifs. Nous avons cherché à obtenir cet algorithme mais ceci présente des difficultés que nous allons mettre en évidence. Pour obtenir cet algorithme il faut continuer la dérivation à partir de la règle (b) en la réorientant en :

$$\text{(d)} \quad SUP(m, E_1) \rightarrow MAX(\{m\} \cup E_1, m)$$

Ensuite, pour introduire un appel récursif, nous devons utiliser une propriété de la forme : $\exists\, m_1$ **(MAX(E,** m_1**)** $\wedge$ **m** $\geq$ m_1**) ≡ SUP(m, E) si E≠∅**. Mais cette formule ne peut être traduite en une règle de réécriture à cause du quantificateur existentiel. Nous utilisons alors la propriété $\forall\, m_1$ **(MAX(E,** m_1**)** $\Rightarrow$ **m** $\geq$ m_1**) ≡ SUP(m, E) si E≠∅**. Cette formule décrit la même chose que la formule ci-dessus puisqu'il existe toujours un maximum, lorsque E n'est pas vide, et que ce maximum est unique. Nous aimerions alors traduire cette formule par la règle :

$$SUP(m, E) \rightarrow (MAX(E, m_1) \Rightarrow m \geq m_1)$$

Cette règle serait alors superposable avec (d) et permettrait le déroulement de la synthèse. Mais cette orientation n'est pas possible ; l'ensemble des variables qui apparaissent en partie droite de règle doit être inclus dans l'ensemble des variables de la partie gauche. Dershowitz rencontre un problème analogue dans la synthèse d'un algorithme de tri par insertion [Der 85]. La solution qu'il adopte revient à utiliser, dans notre cas, la règle :

$$(R_7) \qquad SUP(m, E) \equiv (MAX(E, m_1) \Rightarrow m \geq m_1) \rightarrow \textbf{true} \quad \text{si } E \neq \emptyset$$

Déroulement de la synthèse :

- En superposant la règle (d) avec R_7 sur l'ocurrence SUP(m, E) nous obtenons :

$$MAX(\{m\} \cup E_1, m) \equiv (MAX(E_1, m_1) \Rightarrow m \geq m_1) \rightarrow \textbf{true} \quad \textbf{si } E_1 \neq \emptyset$$

- En superposant la règle (d) avec R_4 sur l'ocurrence SUP(m, E) nous obtenons :

$$MAX(\{m\}, m) \rightarrow \textbf{true}$$

fin-synthèse.

Le programme obtenu est finalement :

(a') $\textbf{MAX}(\emptyset, \textbf{max}) \rightarrow \textbf{false}$

(b') $\textbf{MAX}(\{\textbf{m}\}, \textbf{m}) \rightarrow \textbf{true}$

(c') $\textbf{MAX}(\{\textbf{m}\} \cup E_1, \textbf{m}) \equiv (\textbf{MAX}(E_1, m_1) \Rightarrow \textbf{m} \geq m_1) \rightarrow \textbf{true} \quad \textbf{si } E_1 \neq \emptyset$

(d') $\textbf{MAX}(\{\textbf{m}\} \cup E_1, \textbf{max}) \rightarrow \textbf{MAX}(E_1, \textbf{max}) \wedge \textbf{max} \geq \textbf{m} \quad \textbf{si max} \neq \textbf{m}$

La règle c' peut être interprétée comme décrivant le fait que m est le maximum de $\{m\} \cup E_1$ si et seulement si on peut montrer que m est supérieur au maximum de E_1. Cet ensemble de règles correspond au programme synthétisé à l'aide des approches précédentes. Ce programme peut être exécuté de la même manière que le précédent. Néanmoins, admettre comme partie d'un programme des règles de la forme c', c'est à dire où la partie gauche n'est pas réduite au seul prédicat à définir, semble délicat. D'une part il devient très difficile de prouver la complétude de telles définitions. D'autre part, lors de l'exécution, le calcul de paires critiques entre la règle contenant la réponse à trouver et une partie gauche de définition peut ne plus être possible. Illustrons ceci en utilisant ce programme dans le cas de multi-ensembles, l'énoncé initial et les règles utilisées étant toujours correctes. La règle c', correspondant au cas où l'élément selectionné est effectivement le maximum, devra alors être utilisée plusieurs fois lorsque le maximum est répété.

Exemple :

Cherchons le maximum du multi-ensemble {2, 2, 1}. La question est alors MAX({2, 2, 1}, max) → **Réponse(max)**. Mais l'exécution échoue. En effet, la règle utilisée en premier est (c') et la nouvelle règle engendrée est alors :

$$\textbf{Réponse(2)} \equiv (\textbf{MAX}(\{2, 1\}, m_1) \Rightarrow 2 \geq m_1) \rightarrow \textbf{true} \quad \textbf{si } \{2, 1\} \neq \emptyset$$

Pour arriver à la règle Réponse(2) → true nous devons de nouveau utiliser la règle (c') pour l'instance (MAX({2, 1} , 2) ≡ (MAX({1}, m'_1) ⇒ 2 ≥ m'_1)) → **true**. Mais il n'est pas possible de trouver de paires critiques entre les parties gauches de ces deux règles puisque aucune d'elles ne recouvre l'autre.

fin-exemple.

L'utilisation de la Complétion en synthèse de programmes est une approche assez nouvelle puisque les premiers travaux datent de 82 [Der 82] et 83 [Kod 83]. Elle semble prometteuse dans le sens où elle utilise un cadre unifiant entre calcul (Réécriture) et démonstration (Complétion). D'autre part, le domaine de la réécriture étant en pleine expansion, des résultats futurs pourront peut être permettre de développer cette approche. Néanmoins, une certaine rigidité inhérente aux systèmes de réécriture semble être un handicap à leur utilisation dans le cadre de la synthèse de programmes. En effet :

1) Les systèmes de réécriture décrivent des calculs préservant l'équivalence de formules. Ceci est peut être une contrainte trop forte pour la Synthèse de Programmes.

2) La manière d'exprimer les règles peut permettre ou non le calcul de paires critiques, comme nous l'avons vu lors du calcul du maximum pour les multi-ensembles, et influe donc directement sur le programme engendré.

3) L'ordre retenu contient, en fait, la structure de la dérivation qui sera effectuée : la manière dont les théorèmes interviennent, les primitives qui constituent le programme ... Son choix étant lié au résultat, il semble difficile de l'établir à priori !

D'autre part la procédure de Complétion est un processus aveugle. Son rôle initial est de produire un système de réécriture à partir d'un ensemble d'équations sans se préoccuper de la forme des règles engendrées. En synthèse, ceci est important puisque les règles produites décrivent un programme. Il est donc nécessaire de greffer sur cette procédure des stratégies propres à la synthèse de programmes. Kodratoff [KoP 83] et Perdrix [Per 84] proposent de telles stratégies, pour la synthèse de fonctions récursives. Ces stratégies, basées sur les types abstraits algébriques, sont très semblables à celles proposées par Smith. Dans le cadre de la Réécriture, ces stratégies reviennent à transformer les règles de réécriture en les éclatant suivant la forme de programme désiré : l'usage de la Complétion dans le processus de synthèse devient alors annexe.

Jacquet, dans [Jac 88], étend l'application de la procédure de Complétion en Synthèse de Programmes en ajoutant des informations de type dans les formules logiques. Les types utilisés sont des types dépendants de la valeur de certaines variables. En prenant en compte des relations entre les types, certaines connaissances nécessaires à la dérivation d'un programme n'ont plus besoin d'être données sous forme d'équations mais sont codées directement dans l'algorithme d'unification (utilisation de treillis de types). L'utilisation de types dépendants permet, dans certains cas, de décrire des connaissances conditionnelles. L'utilisation de logique multi-sortes en démonstration automatique a déjà fait ses preuves. Mais, dans cette approche, la notion de type se veut plus riche puisqu'elle permet de décrire des propriétés ou même des relations entre objets. La séparation entre type dépendant et relation entre objets exprimées par des formules logiques est actuellement assez arbitraire. Traiter des propriétés au niveau de l'algorithme d'unification typé est certes intéressant mais ne permet pas de prendre en considération toutes les propriétés nécessaires.

3. LES CONCLUSIONS ACTUELLES.

3.1. Synthèse Déductive et Synthèse Transformationnelle.

Nous avons présenté deux approches principales : l'approche Transformationnelle, qui procéde en réécrivant la formule initiale en une formule équivalente correspondant à un programme, et l'approche Déductive, qui consiste à construire une preuve de la formule initiale. Dans cette seconde approche les preuves effectuées doivent être constructives. Bien qu'apparaissant différentes d'un point de vue théorique, ces approches sont dans la pratique très

semblables, comme ceci est apparu sur l'exemple. En fait, dans l'approche Transformationnelle, les transformations effectuées doivent " rapprocher d'un programme " et sont aussi, à ce titre, constructives. La démarche générale de ces deux approches est donc la même. L'existence des résultats est prouvée par construction, si ce n'est que dans l'approche Transformationnelle cette existence n'est pas nécessairement prouvée pour toutes les données.

Dans l'exemple du maximum traité dans le cadre général de l'approche transformationnelle (page 49) l'énoncé initial S_1 utilisé est, rappelons le :

$$max \in \{m\} \cup E_1 \wedge SUP(max, \{m\} \cup E_1)$$

Nous avons remplacé la précondition $E \neq \emptyset$ par l'égalité $E = \{m\} \cup E_1$. L'énoncé final S_2, obtenu après transformation, est :

$$(max \geq m \wedge E_1 \neq \emptyset \wedge maximum(E_1) = max) \vee (max = m \wedge E_1 = \emptyset) \vee (max = m \wedge E_1 \neq \emptyset \wedge maximum(E_1) \leq max)$$

Les transformations appliquées étant valides nous avons $\forall E_1, m, max\ (S_1 \equiv S_2)$. Dans S_2 la variable max est construite à partie de E_1 et m sous condition que $maximum(E_1)$ soit définie, cette condition portant sur l'existence d'un ordre bien fondé. Montrons maintenant que $\forall E_1, m\ \exists\ max\ S_2$ est bien un théorème.

Preuve :

$\exists max$ ($(max \geq m \wedge E_1 \neq \emptyset \wedge maximum(E_1) = max)$

$\vee (max = m \wedge E_1 = \emptyset)$

$\vee (max = m \wedge E_1 \neq \emptyset \wedge maximum(E_1) \leq max))$

$\equiv (maximum(E_1) \geq m \wedge E_1 \neq \emptyset) \vee E_1 = \emptyset \vee (E_1 \neq \emptyset \wedge maximum(E_1) \leq m)$

$\equiv E_1 = \emptyset \vee (E_1 \neq \emptyset \wedge (maximum(E_1) \geq m \vee maximum(E_1) \leq m))$

$\equiv E_1 = \emptyset \vee E_1 \neq \emptyset$

$\equiv$ v r a i

fin-preuve.

Nous avons donc prouvé que $\forall E_1, m\ \exists\ max\ S_2$ est un théorème et donc, par l'équivalence liant S_1 et S_2, que $\forall E_1, m\ \exists\ max\ S_1$ est aussi un théorème. Par contre si l'énoncé initial ne prenait pas en compte le fait que l'ensemble E doit être différent de l'ensemble vide il s'exprimerait alors par la formule S_1':

$$max \in E \wedge SUP(max, E)$$

Nous aurions alors obtenu, par transformation, le même énoncé final S_2. En effet, en procédant par analyse par cas c'est à dire en rajoutant l'axiome $E=\emptyset \vee E\neq\emptyset$ nous pouvons transformer S_1' en :

$$(E=\emptyset \wedge \max \in E \wedge SUP(max, E)) \vee (E=\{m\} \cup E_1 \wedge S_1(E_1, m, max))$$

S_1' se réduit donc en S_1 puisque $\max \in \emptyset$ est faux et permet donc de dériver l'énoncé final S_2. Nous exhibons alors une construction possible du résultat mais l'existence de ce résultat n'est pas vraie pour toute donnée, puisque le cas $E=\emptyset$ a été éliminé.

L'approche transformationnelle peut donc être vue comme un processus de preuve, à condition de pouvoir prouver l'existence du résultat pour l'énoncé final, le caractère constructif de la preuve étant pris en charge par les transformations. Il n'existe pas de comparaison théorique entre ces deux approches, la difficulté provenant du fait qu'il n'y a pas de caractérisation formelle de l'approche transformationnelle. Pour notre part, nous considérons que ces deux approches mettent en œuvre toutes deux une preuve de l'énoncé initial, bien que cette preuve soit incomplète dans l'approche transformationnelle. En fait le processus de synthèse consiste à mettre en place des structures de contrôle permettant le calcul du résultat. Ces structures sont valides, pour un énoncé donné, si et seulement si cet énoncé peut être réécrit en un forme correspondant à chacune de ces structures. Par exemple la conditionnelle ou l'analyse par cas revient à transformer l'énoncé en une formule disjonctive faisant intervenir le tiers exclus, la récursion revient à réécrire l'énoncé en une formule exprimée en terme de cet énoncé. **L'approche Transformationnelle décrit la preuve en terme de réécriture de la formule initiale et l'approche Déductive décrit la même preuve en terme des structures de contrôle mises en place.** Cette dernière peut donc être vue comme une modélisation des transformations à effectuer pour " tendre vers un programme ". Nous ne parlons plus, par la suite, que de Synthèse Déductive, puisque ces deux approches décrivent pour nous le même processus.

3.2. Approches guidées par la connaissance et Approches guidées par le but.

Pour en revenir à l'exemple, en dehors des différentes manières de décrire la preuve effectuée, les mêmes propriétés sur le domaine du problème ont bien sûr été nécessaires. Elles consistaient en :

1– des définitions récursives des prédicats SUP et $\in$. Ces définitions reposent sur la décomposition d'un ensemble non vide en un élément distingué et le reste. Elles ont permis de réécrire l'énoncé en terme de cette décomposition et donc d'utiliser l'hypothèse d'induction appropriée.

2– la propriété $m_1 \in E \wedge SUP(m_1, E) \wedge m \geq m_1 \Rightarrow SUP(m, E)$. Ce théorème a permis d'obtenir la version classique de l'algorithme de calcul du maximum d'un ensemble n'effectuant qu'un seul parcours de la donnée.

3– Des théorèmes annexes : $m=m$, $m\leq m$ ...

En fait, ces propriétés induisait directement le programme que nous avons effectivement obtenu, puisqu'elles en décrivent l'idée algorithmique. Il y a deux manières d'aborder cette dépendance entre propriétés sur le domaine du problème et programmes synthétisés :

–Soit la preuve construite est induite d'un ensemble de propriétés (**systèmes guidés par la connaissance**). La majorité des travaux en synthèse de programmes se base sur une telle démarche. Parmi ceux que nous avons décrits, citons le travail de Dershowitz et les tableaux déductifs de Manna et Waldinger.

–Soit les propriétés nécessaires sont induites par la forme de preuve désirée (**systèmes guidés par le but**). Dans ces systèmes (Bibel, Smith et Perdrix, parmi ceux que nous avons présentés) la première étape du processus de synthèse consiste à retenir une manière particulière de prouver l'énoncé : choix des stratégies dans le système LOPS, choix des schémas de programmes de Smith. Le déroulement de la preuve sélectionnée exige alors une certaine forme de propriétés qui peut exister ou non pour le domaine considéré. Par exemple la stratégie GET-REC de LOPS, qui permet l'introduction de récursions, demande de pouvoir réécrire l'énoncé **E** en faisant intervenir l'hypothèse d'induction **I** découlant du schéma d'induction retenu. Pour ce faire, un théorème de la forme $\mathbf{H} \wedge \mathbf{I} \equiv \mathbf{E}$, où **H** est une condition additionnelle dépendant du problème, est donc nécessaire.

La première approche, sauf pour des domaines d'application restreints, n'est pas envisageable de manière générale. D'une part on ne peut supposer disposer de toutes les propriétés " intéressantes " pour la synthèse de tout programme et d'autre part il faudrait des critères permettant de sélectionner, parmi un grand nombre, celles qui sont appropriées. Or ces critères devraient prendre en compte le fait que ces propriétés permettent l'élaboration d'un programme acceptable, ce qui ne peut être détecté statiquement. La seconde approche est plus raisonnable dans le sens où elle ne demande pas, en théorie, de disposer à priori de toutes ces propriétés. Mais elle nécessite de pouvoir choisir la forme de preuve à effectuer. Des critères permettant de faire ces choix devraient prendre en compte le fait qu'il existe, pour le domaine du problème, des propriétés permettant le bon déroulement de ces preuves. Or ces propriétés peuvent ne pas être directement connues du système ! Le processus de synthèse nécessite donc une intervention explicite de l'utilisateur, soit au niveau des propriétés à utiliser, soit au niveau de la forme de preuve à effectuer. Dans le cadre de la réalisation d'un système d'aide à la programmation, cette solution est néanmoins préférable. Il semble plus naturel que l'utilisateur ait une idée *à priori* de la forme du programme à construire que de l'ensemble des théorèmes sur le domaine du problème permettant la déduction d'un programme.

3.3. Notre contribution.

En regard à l'état actuel des résultats en Synthèse de Programmes, nous avons cherché à caractériser de manière systématique les propriétés nécessaires à la synthèse d'une forme d'algorithme particulière. Ceci permet de disposer d'un guide permettant de les rechercher parmi un grand ensemble ou de les inférer à partir des connaissances explicites du système, si elles font défaut.

Pour raisonner sur les propriétés nécessaires à la construction de programmes, nous avons définit un système formel de preuve qui met en évidence deux niveaux de preuves interagissant : celui nécessaire à l'élaboration proprement dite du programme et celui nécessaire à la démonstration des propriétés sur le domaine du problème. Nous n'avons pas trouvé de formalisation qui nous convienne soit parce que ces deux niveaux de preuves ne sont pas explicitement mis en évidence, comme dans le Théorie des Types où tout est prouvé constructivement, soit parce que ces formalisations ne sont pas correctes pour la synthèse de programmes. C'est le cas par exemple du système des tableaux déductifs de Manna et Waldinger qui, comme nous l'avons vu, permet des déductions non valides lorsque les fonctions cherchées sont partielles. Notre système décrira la forme de preuve associée à la construction de chaque primitive du langage cible. Le terme **preuve** sera utilisé lorsque nous faisons référence à la logique de la synthèse de programme et le terme **programme** lorsque nous parlerons le langage du programmeur. C'est cette adéquation entre preuve, programme et propriétés qui nous permettra de maitriser l'interaction qu'il faut établir avec l'utilisateur.

Illustrons sur l'exemple du maximum comment nous voulons utiliser la correspondance entre preuve et propriété. Lors du déroulement de la synthèse, il semble raisonnable de disposer des définitions des relations $\in$ et SUP basées sur la décomposition d'un ensemble en un élément distingué et le reste. Or, comme nous l'avons vu lors du traitement de cet exemple dans l'approche Réécriture, ces définitions conduisent au programme :

maximum($\{m\} \cup E$) $\leftarrow$ **si** $E=\emptyset$ **alors** m
sinon si SUP(m, E) **alors** m
sinon maximum(E)

On peut vouloir calculer maximum(E) dans tous les cas afin de n'effectuer qu'un seul parcours de l'ensemble initial. Pour cela, nous devons exprimer la condition SUP(m, E) en une condition équivalente C faisant intervenir le résultat maximum(E). D'après la forme du programme, la formule logique associée à la condition **SUP(m, E)** est $E\neq\emptyset \wedge$ **SUP(m, E)**. Dans cet exemple, la correspondance entre preuve et formule logique est évidente, à la construction **si A alors ... sinon si B ...** nous associons la formule $A \wedge B$, mais ceci n'est pas toujours aussi simple. Nous savons d'autre part que, par hypothèse d'induction, la valeur maximum(E) est telle que :

$$\textbf{maximum(E)} \in \textbf{E} \wedge \textbf{SUP(maximum(E), E)}$$

Nous déduisons donc que la condition C devra être telle que :

$$C(\text{maximum(E)}, E, m) \wedge \text{maximum(E)} \in E \wedge \text{SUP(maximum(E)}, E) \Rightarrow \text{SUP(m, E)}$$

La forme de la propriété nécessaire à l'obtention du programme classique du calcul du maximum est donc déterminée. Par exemple l'instance $\textbf{maximum(E)} \leq \textbf{m} \wedge \textbf{SUP(maximum(E), E)} \Rightarrow \textbf{SUP(m, E)}$ utilisé lors du traitement de l'exemple convient. Supposons maintenant que ce théorème ne soit pas connu. Nous devons trouver une instance correcte de la formule ci-dessus, à partir des définitions des relations $\in$ et SUP. Nous nous restreignons à la recherche de la condition C telle que :

$$C(m_1, E, m) \wedge \text{SUP}(m_1, E) \Rightarrow \text{SUP(m, E)}$$

m_1 désigne le résultat maximum(E).

Recherche de C :

A partir de la définition du prédicat SUP nous obtenons les instanciations suivantes :

- SUP(m, E) $\leftarrow$ **si** $E=\emptyset$ **alors** vrai **sinon** $m \geq$ choix(E) **et** SUP(m, reste(E))

- SUP(m_1, E) $\leftarrow$ **si** $E=\emptyset$ **alors** vrai **sinon** $m_1 \geq$ choix(E) **et** SUP(m_1, reste(E))

Nous recherchons C par induction sur E en utilisant ces deux définitions.

- Si E=∅ alors nous devons trouver C telle que : $C(m_1, E, m) \wedge$ vrai $\Rightarrow$ vrai. En effet SUP(m, ∅) est vrai quelque soit m. Or A $\Rightarrow$ vrai est un théorème, la condition C est donc quelconque.

- Si E≠∅ alors nous devons trouver une condition C telle que :

$$C(m_1, E, m) \wedge m_1 \geq \text{choix}(E) \wedge \text{SUP}(m_1, \text{reste}(E)) \Rightarrow m \geq \text{choix}(E) \wedge \text{SUP}(m, \text{reste}(E)) \quad \textbf{(a)}$$

Par hypothèse d'induction nous pouvons supposer :

$$C(m_1, \text{reste}(E), m) \wedge \text{SUP}(m_1, \text{reste}(E)) \Rightarrow \text{SUP}(m, \text{reste}(E))$$

Pour démontrer la formule **(a)** une solution est donc de prouver les implications :

1- $C(m_1, E, m) \wedge m_1 \geq \text{choix}(E) \wedge \text{SUP}(m_1, \text{reste}(E)) \Rightarrow m \geq \text{choix}(E)$

2- $C(m_1, E, m) \wedge m_1 \geq \text{choix}(E) \wedge \text{SUP}(m_1, \text{reste}(E)) \Rightarrow C(m_1, \text{reste}(E), m)$

3- $C(m_1, E, m) \wedge m_1 \geq \text{choix}(E) \wedge \text{SUP}(m_1, \text{reste}(E)) \Rightarrow \text{SUP}(m_1, \text{reste}(E))$

En effet, si les formules 2 et 3 sont des théorèmes alors on peut déduire SUP(m, reste(E)) directement à partir de l'hypothèse d'induction. La formule 1 décrit comment prouver m ≥ choix(E) qui n'est pas déductible de l'hypothèse d'induction.

Preuve de 1 :

Nous supposons disposer du théorème $m_1 \geq x \wedge m \geq m_1 \Rightarrow m \geq x$. De cette formule nous déduisons alors que la relation $m \geq m_1$ est une solution pour $C(m_1, E, m)$.

Preuve de 2 et 3 :

La seconde formule est vraie pour cette instance puisque $C(m_1, E, m)$ et $C(m_1, \text{reste}(E), m)$ décrivent la même propriété, $m \geq m_1$. La formule 3 est trivialement vraie.

fin-recherche de C.

Nous avons donc trouvé une condition C telle que $C(m_1, E, m) \wedge \text{SUP}(m_1, E) \Rightarrow \text{SUP}(m, E)$ est un théorème. Ceci permet donc de réécrire la condition SUP(m, E), apparaissant dans le programme initial, en m ≥ maximum(E) et d'obtenir le nouveau programme :

maximum($\{m\} \cup E$) $\leftarrow$ si $E=\emptyset$ alors m
sinon si $m \geq$ maximum(E) **alors m**
sinon maximum(E)

Si le théorème qui nous a permis de trouver une instance de C n'avait pas été directement disponible, il aurait fallu réitérer la démarche décrite à partir de la définition de la relation $\geq$.

fin-exemple.

Pour mettre en œuvre une telle démarche le système que nous proposons s'organise en trois parties qui sont :

- 1) le système de preuve pour la synthèse. Son rôle est de mettre en œuvre le déroulement d'une preuve, une fois la forme de la preuve choisie. Pour son bon déroulement, la preuve nécessite des propriétés sur le domaine du problème dépendantes de l'énoncé traité et de la preuve choisie. Le système de preuve communique donc avec le module de traitement des demandes en lui adressant des demandes d'instances de propriétés adéquates.

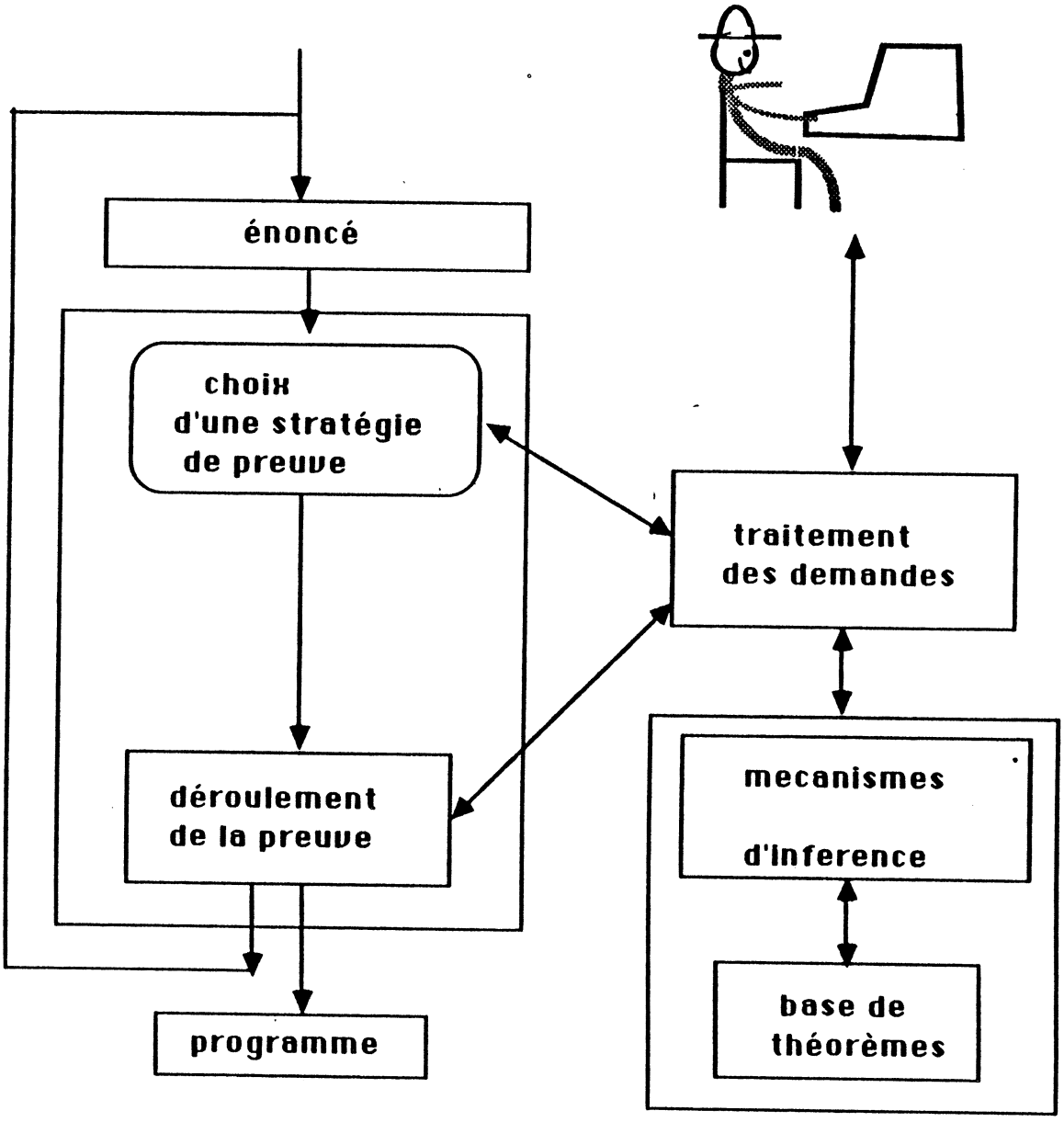
- 2) une base de connaissances déductive. Son rôle est de prouver ou inférer sur le domaine du problème des instances de propriétés, ceci à partir d'un ensemble de théorèmes connus et de mécanismes d'inférence du premier ordre.

- 3) un module de traitement des demandes. Ce module est chargé de fournir les réponses au système de preuve. Il interagit avec la base de connaissances déductive et avec l'utilisateur suivant le degré d'automatisation possible du traitement des demandes.

Dans cette thèse nous nous intéressons uniquement au système de preuve pour la synthèse. Notre but est d'exhiber les demandes nécessaires au bon déroulement de la preuve afin de caractériser en particulier ce qu'on attend des processus déductifs de la base de connaissances.

L'organisation générale du système peut donc se schématiser de la manière suivante :

(voir schéma page suivante)



Les demandes adressées au module de traitement des demandes sont de 2 niveaux : soit relatives au choix de la preuve à mettre en place, soit à l'existence de théorèmes sur le domaine du problème. L'intérêt du système variera selon la part du travail fait par l'utilisateur et par la base de connaissances :

– Dans l'idéal, l'utilisateur n'intervient à aucun niveau : le système est entièrement automatique et est donc capable de prendre en compte notamment des critères d'efficacité.

– L'utilisateur intervient au niveau des choix. Cela revient à supposer qu'il a une idée de la forme du programme qu'il souhaite obtenir : par exemple un programme récursif utilisant une certaine décomposition des données, une composition fonctionnelle faisant intervenir une certaine fonction ... Le rôle d'un tel système est d'aider l'utilisateur à préciser son idée initiale en lui fournissant les énoncés qu'il reste à résoudre et pour lesquels il devra continuer à faire des choix. Un tel système

assure alors entièrement la preuve de correction du programme construit. Le problème de l'efficacité est remis entre les mains de l'utilisateur.

– L'utilisateur intervient à tous les niveaux. Il choisit la forme du programme et fournit lui-même les théorèmes justifiant son algorithme. Bien que l'utilisateur fasse alors lui-même tout le travail, un tel système peut être vu comme un système d'aide à la programmation. Son rôle consiste à forcer l'utilisateur à préciser de manière descendante l'algorithme qu'il veut construire ainsi que sa preuve. Un tel système, bien que peu ambitieux, peut néanmoins aider l'utilisateur à décomposer son problème et à se poser les bonnes questions en temps voulu, ceci dépendant de la pertinence de l'interaction mise en place.

Trouver des critères permettant de choisir la preuve à mettre en place semble difficile. En effet, en dehors des problèmes d'efficacité, les choix effectués doivent permettre le bon déroulement de la preuve retenue. Or ceci dépend directement des propriétés du domaine que nous supposons ne pas connaitre *à priori* ! Ceci n'est pas complètement paradoxal : les connaissances permettant de faire les choix peuvent être d'un niveau plus général que celles nécessaires à l'obtention du programme. Cette hiérarchisation des connaissances correspond à ce que nous avons traduit vis à vis de l'utilisateur par " préciser de manière descendante une idée initiale ".

Pour mettre en évidence les propriétés nécessaires à chaque forme de programme nous présentons, dans un premier temps, le langage cible et sa sémantique axiomatique. Puis nous décrirons le langage d'énoncés et son système de preuve, issu de l'axiomatisation du langage cible. Enfin nous présentons des stratégies permettant de guider le déroulement d'une preuve. Dans ces stratégies, les demandes de propriétés adressées au module de traitement des demandes devront être suffisamment précises afin que celui-ci soit en mesure de fournir des réponses.

partie 2

SYSTEME DE PREUVE

ET

STRATEGIES

CHAPITRE 1

LE LANGAGE CIBLE ET SA SEMANTIQUE.

Dans cette seconde partie nous nous attachons à présenter de manière précise le système de démonstration pour la synthèse de programmes sur lequel nous nous basons. Ce premier chapitre est relatif au langage de programmation cible et à sa sémantique axiomatique.

1. LE LANGAGE CIBLE.

Comme dans la majorité des travaux en Synthèse de Programmes, nous nous intéressons à la recherche de fonctions vérifiant un énoncé donné et non pas à la recherche de toutes les fonctions possibles. Le langage cible que nous visons sera donc de type fonctionnel ; choisir un langage relationnel, style PROLOG, ne nous apporterait aucun avantage supplémentaire. D'autre part, ce langage sera sans effets de bords possibles, ceux-ci s'axiomatisant difficilement. De toute façon, ce qui importe ce sont les différentes constructions offertes par le langage cible. La preuve de l'énoncé sera faite en utilisant l'axiomatisation du langage, le programme obtenu n'étant qu'une représentation syntaxique de cette preuve. Par exemple la règle de la conditionnelle, est, rappelons le :

$$\frac{P \wedge B \{S_1\} Q \qquad P \wedge \neg B \{S_2\} Q}{P \{S\} Q}$$

Cette règle décrit la sémantique de la conditionnelle et axiomatise indifféremment les schémas de programmes **si B alors S_1 sinon S_2** de PASCAL, **cond((B S_1) (t S_2))** de LISP ou les deux clauses **B $\Rightarrow$ S_1** et **not(B) $\Rightarrow$ S_2** de PROLOG.

Le langage que nous avons choisi, et que nous nommons LF pour langage fonctionnel, s'apparente plus à un formalisme de définition de fonctions calculables qu'à un langage de programmation complet : il contient peu de primitives et peu de structures de données. En effet, nous nous intéressons principalement à la recherche de méthodes algorithmiques permettant de résoudre un problème, méthodes qui peuvent s'exprimer à l'aide d'un ensemble restreint de commandes. Nous laissons donc la production d'un programme mettant en œuvre efficacement ces méthodes pour une étape ultérieure. Cette séparation peut être jugée arbitraire : la manière d'aborder un problème est souvent très dépendante des possibilités offertes par le langage de programmation utilisé (primitives, structures de données et effets de bord possibles ...), mais, en raison de la complexité de la tâche visée, nous nous limitons volontairement. L'extension, dans le cadre d'un " vrai " langage de programmation ne pose pas, en général, de problèmes théoriques mais introduit une combinatoire supplémentaire.

Ce langage est proche d'un formalisme proposé par McCarthy [Car 63] et qui a servi de base au langage LISP. La classe de fonctions qu'il décrit est définie inductivement à partir d'un ensemble de fonctions de base et par application des fonctionnelles **conditionnelle, composition fonctionnelle** et **récursion**. Pour notre part, le langage sera typé. En effet, un énoncé de problème n'a de sens que dans un modèle particulier. Par exemple $\max(x) \in x \wedge \text{SUP}(\max(x), x)$, qui est un énoncé possible de la fonction maximum, ne peut être valide que si x est un objet de type collection d'éléments comparables. Nous introduirons donc une notion de type dans le langage d'énoncés, ceci permettant de sélectionner la théorie dans laquelle la preuve sera effectuée. Les programmes, obtenus par synthèse, sont donc implicitement typés puisque issus d'énoncés typés. D'autre part, nous désirons, dans la mesure du possible, garantir la correction totale du programme engendré, c'est à dire détecter les données pour lesquelles le programme a une valeur indéfinie. Toutes les fonctions seront rendues totales par l'ajout de la fonction constante " indéfinie " de type quelconque, que nous notons $\perp$.

1.1. Typage.

Dans le but de réutiliser les programmes synthétisés, il est intéressant de construire les programmes au bon niveau d'abstraction, c'est à dire pour tous les types pour lesquels ces programmes ont un sens. Par exemple, la fonction qui calcule le nombre d'éléments d'une liste est la même quelque soit les objets qui constituent la liste. Pour cela, le typage que nous utilisons est polymorphe. La preuve d'un énoncé ne se déroule donc plus dans un seul modèle mais dans une classe de modèles.

Définition des types :

Soit $\mathcal{V}$ un ensemble infini dénombrable de noms de variables de type et $\mathcal{O}$ un ensemble d'opérateurs sur les types (entier$_0$, booléen$_0$,..., liste$_1$, ..., $\times_2$, $\rightarrow_2$, ...), les indices indiquant l'arité de ces opérateurs. $\rightarrow_2$ est un opérateur infixé.

- tout élément de $\mathcal{V}$ est un type.
- si p_n est un élément de $\mathcal{O}$ et t_1, ..., t_n sont n types alors $p_n(t_1, ..., t_n)$ est un type.

Le type $t_1 \rightarrow t_2$ désigne le type des fonctions de t_1 dans t_2. Nous appelerons profil un tel type.

fin-définition.

Il existe une relation d'ordre partiel, $\leq$, sur les types définie par :

$t_1 \leq t_2$ si et seulement si il existe une substitution θ telle que $t_2\,\theta = t_1$.

Intuitivement ceci signifie que le type t_2 est plus général que le type t_1. Par exemple les relations **entier** $\leq$ t, **liste(liste(t_1))** $\leq$ **liste(t_2)** sont vérifiées. Par définition du polymorphisme, si un objet est de type t alors il possède également tous les types inférieurs à t dans l'ordre $\leq$ donné ci-dessus.

Rappel : une substitution est une fonction définie sur un domaine de noms de variables, v_1, ..., v_n, et dans un

domaine de termes, $t_1, ..., t_m$. Une substitution est généralement décrite par la donnée des couples (v_i, t_j), pour l'ensemble des variables v_i pour lesquelles le terme t_j n'est pas la variable v_i elle-même. Pour toutes les autres variables, la substitution est la fonction identité.

Cette forme de typage est proposée, par exemple, dans les langages ML [GMW 79] et RUSSEL [DeD 80]. Elle permet de décrire des programmes ayant un profil générique. Ces programmes peuvent alors être utilisables pour tous les profils inférieurs dans $\leq$. Cependant cette forme de généricité est limitée : par exemple le programme calculant le maximum d'une liste d'éléments comparables ne peut être décrit à l'aide d'une telle généricité. D'une part il ne peut s'appliquer que sur des listes d'objets pour lesquels il existe une relation d'ordre total, et, d'autre part l'opérateur de comparaison, propre à chaque type, intervient dans le programme. Une description générique de cet algorithme nécessite donc un paramètre supplémentaire, l'opérateur de comparaison. Il existe des langages proposant un tel typage et qui offrent un degré d'abstraction vraiment intéressant pour la paramétrisation de programmes. Citons par exemple CLEAR [BuG 80], OBJ2 [FGJ 85] et LPG [BeE 86] dans lesquels il est possible de caractériser des classes de types ou d'opérateurs ayant certaines propriétés comme par exemple l'existence d'un ordre total (notion de propriétés dans LPG et de théories dans OBJ2 et CLEAR).

Nous décrivons maintenant le langage LF dans le cadre d'un typage polymorphe à la ML.

1.2. Syntaxe.

Un *programme* est un ensemble de *définitions* décrites à l'aide de *termes*. Nous définissons ci-dessous ces objets. Nous supposons d'autre part disposer d'un ensemble $\mathfrak{B}$ de fonctions de base. Nous n'énumérons pas cet ensemble mais nous admettons qu'il contient au moins toutes les fonctions traditionnellement admises comme fonctions de base (les constructeurs, les sélecteurs...) pour les types permis ainsi que la fonction partout indéfinie, $\perp$.

• Les termes.

Soit $\mathfrak{V}$ un ensemble infini dénombrable de noms de variables typées.

- Toute variable de type t est un terme de type t.

- Terme **Conditionnelle** : Si C est un terme de type **booléen** et si E_1 et E_2 sont 2 termes de types respectifs t_1 et t_2 alors **si** C **alors** E_1 **sinon** E_2 est un terme dont le type est le plus grand type t tel que $t \leq t_1$ et $t \leq t_2$, dans l'ordre $\leq$ défini sur les termes de type. Si t est le type vide, le plus petit élément dans l'ordre sur les types, alors le terme **Conditionnelle** n'est pas défini. La partie sinon est obligatoire, bien qu'elle puisse être réduite à $\perp$, car nous imposons que les fonctions soient totales.

- Terme **Composition fonctionnelle** : Soient $E_1, ..., E_n$ n termes de types respectifs $t_1, ..., t_n$, tels que les ensembles des variables de type des t_i soient disjoints, et g une fonction de LF de profil $(t'_1 \times ... \times t'_n) \rightarrow t$. Si les couples $(t_i\theta_{i-1}, t_i'\theta_{i-1})$ sont unifiables, θ_{i-1} désignant la composition des

substitutions $\theta_0, ..., \theta_{i-2}$ obtenues après unification des couples $(t_1\theta_0, t_1'\theta_0), ..., (t_{i-1}\theta_{i-2}, t_{i-1}'\theta_{i-2})$ et θ_0 étant l'identité, alors $g(E_1, ..., E_n)$ est un terme de type $t\theta_n$.

- **Les définitions.**

Une définition d'une fonction f de profil $(t_1 \times ... \times t_k) \rightarrow t$ est de la forme $f(x) \leftarrow T$ où f est un nom de fonction non encore définie, x est un k-uplet de variables de type $(t_1 \times ... \times t_k)$ et T est un terme de type t dont toutes les variables appartiennent au k-uplet x.

- **Les programmes.**

Un programme $\mathcal{P}$ est un ensemble de définitions $f_i(x) \leftarrow T_i$ tel que :

– Il existe au plus une seule définition d'une fonction f_i dans $\mathcal{P}$;
– Toute fonction apparaissant dans les termes T_i sont soit des fonctions de base soit font l'objet d'une définition dans $\mathcal{P}$.

On définit une opération d'union sur les programmes. $\mathcal{P}_1 \cup \mathcal{P}_2$ est un programme contenant toutes les définitions de $\mathcal{P}_1$ et de $\mathcal{P}_2$, après renommage des fonctions, afin qu'il n'y ait pas de fonctions définies à la fois dans $\mathcal{P}_1$ et dans $\mathcal{P}_2$.

- **Les définitions récursives.**

- Une définition $f(x) \leftarrow T$ est récursive en h dans $\mathcal{P}$ si T contient un sous-terme de la forme h(t) ou si T contient une sous-terme de la forme g(t) où g fait l'objet d'une définition dans $\mathcal{P}$ et que celle-ci est récursive en h dans $\mathcal{P}$.

- Une définition d'une fonction f est récursive dans $\mathcal{P}$ si cette définition est récursive en f dans $\mathcal{P}$.

fin-définitions.

Un programme est récursif si il contient au moins une définition récursive. Dans ce cas nous chercherons à prouver l'arrêt du programme par l'utilisation d'un ordre bien fondé. En effet, l'existence d'un ordre bien fondé, tel que les arguments de toutes les occurrences récursives d'une fonction sont inférieurs à l'argument initial, est une condition suffisante pour prouver l'arrêt des programmes.

La classe de fonctions décrite par McCarthy ne peut pas être comparée dans l'absolu à d'autres formalismes puisqu'elle dépend de l'ensemble des fonctions de base. Cependant McCarthy a montré que C(succ, eq) contient toutes les fonctions récursives partielles au sens de Church et Kleene, succ étant la fonction successeur sur les entiers et eq la fonction égalité. Il constate que la conditionnelle ne rajoute rien vis à vis des autres formalismes ne l'utilisant pas si

ce n'est une facilité d'expression de certaines fonctions. Pour notre part, nous aimerions garantir l'arrêt des programmes synthétisés, c'est à dire construire des fonctions qui peuvent être rendues totales par l'ajout de la valeur indéfinie. Certaines sous-classes de fonctions récursives totales ont été caractérisées ; celles pour lesquelles on dispose de systèmes de preuve adaptés permettant d'en prouver l'arrêt ; par exemple basés sur l'axiomatique de Peano, sur les définitions constructives de domaines (listes, arbres, ensembles bien fondés...).

1.3. Fonctions multi-valuées.

Lors du processus de synthèse on peut être amené à construire un programme calculant plusieurs résultats. Dans ce cas il peut être intéressant de calculer ces résultats simultanément, dans le but soit de factoriser la structure du programme soit de faciliter le déroulement de la preuve, notamment dans le cas de récursions croisées. Il faut donc pouvoir décrire des définitions définissant plusieurs fonctions à la fois. Pour cela, il suffit d'étendre les définitions précédentes dans le cas de n-uplets de définitions et de termes, de la manière suivante :

- **Les n-termes.**

Nous ne précisons plus les contraintes de type, ce sont les mêmes que pour les définitions précédentes.

- Si t est un terme alors (t) est un 1-terme.

- Si (t_1) est un n-terme et si (t_2) est un k-terme alors (t_1, t_2) est un n+k-terme.

- Si (t_1) et (t_2) sont deux n-termes et si C est un terme alors **si** C **alors** (t_1) **sinon** (t_2) est un n-terme.

En ce qui concerne la composition fonctionnelle, nous n'avons pas besoin de l'étendre puisqu'elle s'applique déjà à des n-uplets de termes.

- **Les n-définitions.**

Si E est un n-terme dont les variables libres appartiennent au k-uplet x alors $(f_1(x), ..., f_n(x)) \leftarrow E$ est une n-définition.

fin-définitions.

Nous présentons l'axiomatique de LF pour des définitions simples, les règles étant déjà suffisamment compliquées! L'extension aux n-définitions ne pose évidemment pas de problème particulier.

1.4. Sémantique opérationnelle.

Nous décrivons maintenant la sémantique opérationnelle de LF, celle-ci servant à démontrer la correction de la sémantique axiomatique. Pour cela nous définissons la fonction **eval** qui, pour une fonction f, un programme $\mathcal{P}$ et un k-uplet bien typé quelconque A de constantes, délivre la valeur f(A) dans $\mathcal{P}$. Nous supposons que toutes les fonctions sont strictes, c'est à dire telles que $f(\ldots, \perp, \ldots)=\perp$. Cette restriction sera nécessaire pour démontrer la complétude de notre système.

• **Définition de eval.**

– Soit le k-uplet A contient la valeur $\perp$ alors **eval**(f, $\mathcal{P}$, A) = $\perp$.

– Soit f est une fonction de base, l'interpréteur peut donc la calculer pour l'argument A. Et donc **eval**(f, $\mathcal{P}$, A) = f(A).

– Soit $\mathcal{P}$ contient une définition de f de la forme f(x) $\leftarrow$ E alors :

eval(f, $\mathcal{P}$, A) = **evaluer**(E[x / A], $\mathcal{P}$)

E[x / A] désigne le terme instancié obtenu après substitution de A à x. **evaluer** est une fonction définie inductivement de la manière suivante :

1. **evaluer**(c, $\mathcal{P}$) = c si c est une constante.

2. **evaluer**(**si** C **alors** E_1 **sinon** E_2, $\mathcal{P}$)
   = **evaluer**(E_1, $\mathcal{P}$) si **evaluer**(C, $\mathcal{P}$) = **vrai**
   = **evaluer**(E_2, $\mathcal{P}$) si **evaluer**(C, $\mathcal{P}$) = **faux**
   = $\perp$ sinon

3. **evaluer**(g(E_1, ..., E_n), $\mathcal{P}$)= **eval**(g, $\mathcal{P}$, (**evaluer**(E_1, $\mathcal{P}$), ..., **evaluer**(E_n, $\mathcal{P}$)))

– Soit f est un symbole non défini dans $\mathcal{P}$ et **eval**(f, $\mathcal{P}$, A) = $\perp$.

fin-définition.

La sémantique de ce langage correspond à la sémantique du plus petit point fixe et les arguments sont évalués avant appel (passage de paramètres par valeur).

2. SEMANTIQUE AXIOMATIQUE.

Décrire la sémantique d'un langage de programmation sous forme axiomatique revient à énoncer les relations qui sont vérifiées par les objets des programmes. Le langage permettant d'exprimer ces relations est appelé *langage des assertions*. Les sémantiques axiomatiques sont classiquement décrites à l'aide d'un système formel défini sur un langage liant les programmes et les relations entre objets. Nous appelerons *langage des formules de correction* l'union de ces deux langages. Les formules valides du système formel défini sur cette union sont les formules telles que les objets du programme S vérifient bien les relations R. En logique de Hoare, formalisme le plus utilisé, le langage des assertions est un sous-ensemble du CP1 et les formules de correction, appelées aussi formules de Hoare, sont de la forme P { S } Q où S est un programme et P et Q des assertions. Si P { S } Q est un théorème nous dirons indifféremment que (P, Q) est une spécification du programme S, ou que f vérifie la spécification (P, Q).

2.1. Les formules de Correction.

Notre langage cible étant fonctionnel, nous avons cherché à nous inspirer de la description de la sémantique des définitions de fonctions dans les langages impératifs. Dans la partie 1 nous avons donné une règle, proposée par Hoare et Wirth [HoW 73], relative à la définition de fonction en PASCAL. Rappelons la :

Si f est une fonction définie, pour une liste de paramètres x, par son corps S, alors :

$$\frac{P(x, y)\ \{\ S\ \}\ Q(x, y, f)}{\forall\ x, y\ (P(x, y) \Rightarrow Q(x, y, f)\ [\ f / f(x, y)])}$$

y désigne l'ensemble des variables du programme non locales à S et f désigne la variable représentant le résultat de la fonction f. Nous ne rappelons pas les restrictions attachées à cette règle, celles-ci ne nous concernant pas.

Cette règle permet donc de déduire une formule $\mathcal{F}$ de la logique classique à partir d'une définition de fonction. $\mathcal{F}$ peut alors être utilisée pour prouver la correction de programme contenant une invocation de cette fonction. Puisque notre langage est fonctionnel nous pourrions donc choisir de représenter directement les formules de correction par des implications de la forme :

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

En effet, les seules variables qui peuvent apparaître dans la définition de f sont les paramètres x et donc P et Q sont des relations qui ne dépendent que de x et f(x). La sémantique axiomatique décrirait alors comment déduire de nouvelles implications de la logique classique, suivant les différentes formes possibles de définitions de fonctions. Mais la règle proposée par Hoare et Wirth peut mener à des inconsistances lorsque les fonctions sont partielles. Par exemple, M. O'Donnell [Don 82] montre que le paradoxe de Russel peut être simulé par l'utilisation de cette règle.

Paradoxe de RUSSEL :

Soit R l'ensemble de tous les ensembles qui ne se contiennent pas eux-mêmes. R se contient-il? Si R se contient alors, par définition, il n'appartient pas à R et donc il ne se contient pas. Réciproquement, si R ne se contient pas alors, toujours d'après la définition de R, il doit appartenir à R et donc se contenir!

O'Donnell choisit de représenter un ensemble E par sa fonction caractéristique g qui vaut 1 si son argument appartient à E et 0 sinon. Soit r la fonction définie par :

fonction r(g) : **debut** r := 1 - g(g) **fin**

Cette fonction, appliquée à un ensemble, c'est à dire à une fonction caractéristique, renvoie 1 si cet ensemble appartient à R et 0 sinon. L'expression g(g) calcule donc si E est élément de lui-même. Montrons alors que la règle précédente permet des déductions non valides.

Démonstration :

- Par l'axiome de l'affectation, le corps de la fonction r est tel que :

$$(1 - g(g) = 1 - g(g))\ \{r := 1 - g(g)\}\ r(= 1 - g(g))$$

- En utilisant les deux théorèmes vrai $\Rightarrow (1 - g(g) = 1 - g(g))$ et $(r(g) = 1 - g(g)) \Rightarrow r(g) \neq g(g)$ ainsi que la règle de la conséquence nous déduisons la formule de correction : vrai $\{r := 1 - g(g)\}\ r \neq g(g)$.

- Par la règle attachée à la définition de fonction nous avons alors $\forall$ g (vrai $\Rightarrow$ $r(g) \neq g(g)$). Ceci permet alors de déduire l'instance $r(r) \neq r(r)$ qui est inconsistante puisque les deux parties de l'égalité sont syntaxiquement identiques. La règle utilisée n'est donc pas correcte.

fin-démonstration.

Cet exemple est un cas extrême mais il illustre bien la difficulté liée à la manipulation en logique classique d'objets partiellement définis. En effet, la formule $\forall$ x (P(x) $\Rightarrow$ Q(x, f(x))) dérivée par la règle donnée ne décrit pas le fait que la relation Q doit être vérifiée par le **résultat** de l'évaluation de f, ceci pour les x vérifiant P. Or cette évaluation peut être indéfinie comme c'est le cas dans la dérivation ci-dessus puisque r(r) boucle. Rappelons que nous avons rencontré un problème analogue dans le système des tableaux de Manna et Waldinger, les fonctions partielles entrainant des déductions non valides. En synthèse, les fonctions manipulées peuvent être partielles, il est donc nécessaire de prendre en considération la valeur $\perp$. Il n'est donc pas possible de traduire la définition d'une fonction par une formule ne faisant pas intervenir l'évaluation explicite de la fonction. Nous avons donc choisi de représenter une formule de correction attachée à la définition d'une fonction par la formule :

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

eval est la fonction d'interprétation du langage cible, définie dans la première section de ce chapitre.

Nous noterons par la suite cette formule : $\mathcal{P}$: $\forall x\ (P(x) \rightarrow Q(x, f(x)))$. La flèche désigne donc le fait que la relation Q doit être vérifiée par le résultat de l'évaluation de f pour une donnée x, sous l'hypothèse que f est définie par le programme $\mathcal{P}$.

Définition :

$$\mathcal{P} : \forall x\ (P(x) \rightarrow Q(x, f(x)))$$
est un théorème de notre logique

si et seulement si

$$\forall x, y\ \ (P(x) \wedge\ y = \mathbf{eval}(f, \mathcal{P}, x) \Rightarrow Q(x, y) \wedge\ y \neq \perp)$$
est un théorème de la logique classique

fin-définition.

Cette interprétation n'est complète que si nous disposons d'une axiomatisation de la fonction **eval**. Cette axiomatisation découle de la définition des fonctions **eval** et **evaluer** donnée précédemment. Nous ne la donnerons que lorsque nécessaire, c'est à dire lors de la preuve de correction de l'axiomatisation du langage cible.

D'après cette interprétation, si $\mathcal{P}$: $P(x) \rightarrow Q(x, f(x))$ est un théorème alors $P(x) \Rightarrow Q(x, f(x))$ est un théorème de la logique classique. En effet, dans notre interprétation, nous prouvons $Q(x, f(x))$ en calculant le résultat $f(x)$, ce qui est une manière particulière de prouver $Q(x, f(x))$. L'interprétation que nous donnons à la flèche est donc un cas particulier de l'interprétation associée à l'implication classique. Il en résulte que tout théorème de notre logique, relatif à une fonction f, pourra être utilisé par la suite pour démontrer des propriétés de f ne nécessitant pas de prendre en compte la calculabilité de cette fonction.

2.2. Le système formel.

Décrire la sémantique de notre langage sous forme axiomatique ne pose pas de problème particulier puisque les constructions de ce langage sont simples. Mais notre but est d'exhiber les propriétés sur le domaine du problème attachées à chaque construction possible du langage cible. Les règles décriront donc les spécifications de chaque construction possible sans présumé de la forme des spécifications vérifiées par les paramètres de ces constructions. Illustrons brièvement cette démarche sur la règle de la conditionnelle énoncée en logique de Hoare. La règle habituellement rencontrée est :

$$\frac{P \wedge B\ \{S_1\}\ Q \qquad P \wedge \neg B\ \{S_2\}\ Q}{P\ \{\ \text{si } B \text{ alors } S_1 \text{ sinon } S_2\ \}\ Q}$$

Cette règle décrit une spécification (P, Q), vérifiée par la conditionnelle **si B alors S_1 sinon S_2**, en terme des spécifications ($P \wedge B$, Q) et ($P \wedge \neg B$, Q) vérifiées par les constructions S_1 et S_2. En supposant que les variables apparaissant dans la condition B ne sont modifiées ni par S_1, ni par S_2 nous présenterons cette règle sous la forme :

$$\frac{P_1 \{S_1\} Q_1 \quad P_2 \{S_2\} Q_2}{(P_1 \wedge B) \vee (P_2 \wedge \neg B) \{ \text{ si B alors } S_1 \text{ sinon } S_2 \} (Q_1 \wedge B) \vee (Q_2 \wedge \neg B)}$$

Cette règle est sémantiquement équivalente à la première mais est décrite pour une utilisation des prémisses vers le but et non l'inverse. Quel est, pour nous, l'intérêt d'une telle représentation ? Si nous voulons maintenant construire un programme conditionnel S, tel que P { S } Q, nous savons alors, par la règle de la conséquence, que nous devons trouver des relations P_1, P_2, Q_1 et Q_2 telles que :

$$P \Rightarrow (P_1 \wedge B) \vee (P_2 \wedge \neg B)$$
$$(Q_1 \wedge B) \vee (Q_2 \wedge \neg B) \Rightarrow Q$$

Ces deux implications mettent en évidence les propriétés que doivent vérifier les relations P et Q, P_1 et Q_1 et P_2 et Q_2 pour qu'il soit possible de construire S sous la forme d'un programme conditionnel. Cette présentation des règles va donc permettre d'établir le lien entre les différentes constructions du langage cible et les propriétés du domaine du problème sous-jacentes à ces constructions.

Les règles d'inférence que nous allons présenter décrivent les spécifications vérifiées par une fonction f faisant l'objet d'une définition de la forme $f(x) \leftarrow F$ où F est soit un terme conditionnel, soit une composition fonctionnelle soit un terme récursif en f. Les prémisses décriront les spécifications supposées être vérifiées par les fonctions f_i paramètres de F. Les règles auront donc la forme générale suivante :

$$\frac{\mathcal{P}_i : P_i(y_i) \rightarrow Q_i(y_i, f_i(y_i))}{(\cup_i \mathcal{P}_i) \cup \{ f(x) \leftarrow F \} : P(x) \rightarrow Q(x, f(x))}$$

Elles décrivent les spécifications (P, Q) obtenues pour chaque construction possible F du langage cible, en terme des spécifications (P_i, Q_i) des fonctions paramètres de F.

Le système proposé sera correct si et seulement si toute formule déductible sera valide vis à vis de l'interprétation que

nous avons donnée, c'est à dire sera telle que :

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

La correction est classiquement prouvée par induction sur la structure du programme : on montre d'une part que les axiomes sont corrects et d'autre part, pour chaque règle d'inférence, que la conclusion est correcte en supposant que c'est le cas pour les prémisses.

Nous montrerons ensuite, dans la section 2.5, que le système proposé est complet, c'est à dire que toute formule de correction valide dans l'interprétation donnée est déductible dans ce système. La complétude des logiques de Hoare est habituellement démontrée en prouvant que **pfpré**(S, Q) { S } Q est déductible, quelques soient le programme S et la précondition Q, **pfpré** désignant la plus faible précondition de S pour la postcondition Q. Rappelons que P est une plus faible précondition de S pour la postcondition Q si et seulement si P est une précondition de S pour Q et si toute autre précondition possible R est telle que $R \Rightarrow P$. La complétude est prouvée habituellement par induction sur la structure de S.

Exemple : pour prouver la complétude de la règle de la conditionnelle, nous devons montrer que la formule suivante est déductible dans notre système :

$$\mathbf{pfpré}(\mathbf{si}\ B\ \mathbf{alors}\ S_1\ \mathbf{sinon}\ S_2, Q)\ \{\ \mathbf{si}\ B\ \mathbf{alors}\ S_1\ \mathbf{sinon}\ S_2\ \}\ Q$$

D'autre part, par hypothèse d'induction sur la structure de la preuve, nous pouvons supposer que les deux formules $\mathbf{pfpré}(S_1, Q)\ \{ S_1 \}\ Q$ et $\mathbf{pfpré}(S_2, Q)\ \{ S_2 \}\ Q$ sont déductibles dans le système formel considéré.

Démonstration : par la règle de la conditionnelle, il suffit de prouver les prémisses :

$$\mathbf{pfpré}(\mathbf{si}\ B\ \mathbf{alors}\ S_1\ \mathbf{sinon}\ S_2, Q) \wedge B\ \{ S_1 \}\ Q$$
$$\mathbf{pfpré}(\mathbf{si}\ B\ \mathbf{alors}\ S_1\ \mathbf{sinon}\ S_2, Q) \wedge \neg B\ \{ S_2 \}\ Q$$

Or **pfpré**(**si** B **alors** S_1 **sinon** S_2, Q) est égale à $(\mathbf{pfpré}(S_1, Q) \wedge B) \vee (\mathbf{pfpré}(S_2, Q) \wedge \neg B)$, par définition de la plus faible précondition. Les deux formules précédentes sont donc respectivement équivalentes à :

$$\mathbf{pfpré}(S_1, Q) \wedge B\ \{ S_1 \}\ Q \text{ et}$$
$$\mathbf{pfpré}(S_2, Q) \wedge \neg B\ \{ S_2 \}\ Q$$

Or ces deux formules sont déductibles dans le système formel considéré par hypothèse d'induction. Nous pouvons donc en déduire que **pfpré**(**si** B **alors** S_1 **sinon** S_2, Q) { **si** B **alors** S_1 **sinon** S_2 } Q est déductible quelques soient les formules B, S_1, S_2 et Q.

fin-exemple.

Pour notre part, nous ne pouvons pas utiliser une telle méthode. En effet, d'une part les postconditions proposées par les règles d'inférence auront une forme particulière et donc les règles ne s'appliqueront donc pas nécessairement à une formule pfpré(S, Q) { S } Q quelconque. D'autre part les préconditions et les postconditions des prémisses et de la conclusion ne sont jamais les mêmes, or les propriétés des plus faibles préconditions ne sont valides que pour une même postcondition. Pour établir la complétude nous devrons alors utiliser non pas les propriétés d'une classe particulière de préconditions, mais les propriétés d'une classe particulière de spécifications. Nous qualifierons ces spécifications de *spécifications les plus générales*. Nous démontrerons donc que toute spécification vérifiée par un programme peut être déduite des spécifications *les plus générales* par la règle de la conséquence et que les spécifications apparaissant dans les conclusions des règles sont bien *les plus générales*.

Nous présentons maintenant notre système.

2.3. Les axiomes.

Les fonctions de base sont spécifiées par des axiomes propres au système, ils sont donc de la forme :

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

Ces axiomes sont vrais indépendamment de tout programme, par définition des fonctions de base. Nous décrivons cela on énonçant qu'il soit vrais pour le programme vide, noté $\emptyset$. En effet, nous avons la règle d'inclusion suivante :

$$\mathcal{P}_1 \supseteq \mathcal{P}_2 \ \wedge \ \mathcal{P}_2 : P(x) \rightarrow Q(x, f(x)) \ \Rightarrow \ \mathcal{P}_1 : P(x) \rightarrow Q(x, f(x))$$

Les fonctions de base du langage cible peuvent être séparées en deux ensembles : celles qui sont nécessaires et suffisantes, c'est à dire celles qui décrivent les objets des types manipulés par le langage (les constructeurs, les indicateurs ...) et celles qui sont en fait des facilités offertes par le langage, c'est-à-dire celles qui pourraient être décrites à l'aide d'un programme en termes des précédentes. La première classe de fonctions décrit des représentations particulières des objets de chaque type, représentations qui sont utilisées pour définir des opérateurs sur ces types. Si nous supposons que les noms de ces fonctions sont les mêmes dans le langage cible et dans les théories décrites dans la Base de Connaissances alors nous spécifions ces fonctions, dont l'existence est un postulat, par l'axiome $\emptyset$: P(x) $\rightarrow$ f(x)=f(x). Par exemple :

$\emptyset$: vrai $\rightarrow$ nil=nil

$\emptyset$: vrai $\rightarrow$ cons(a, b)=cons(a, b)

nil : $\rightarrow$ liste(t)

cons : (t $\times$ liste(t)) $\rightarrow$ liste(t)

Prouver la correction des axiomes ne nous intéresse que pour les " vraies " fonctions de base, c'est à dire les différents constructeurs de type. Les axiomes relatifs à la seconde catégorie de fonctions de base peuvent être prouvés corrects à partir des programmes permettant leur évaluation. Si nous supposons que les fonctions constructeurs sont des fonctions totales alors la preuve de correction est directe. En effet, leur précondition étant réduite à vrai, il faut montrer :

$$\text{vrai} \wedge y = \text{eval}(f, \emptyset, x) \Rightarrow y = f(x) \wedge y \neq \perp$$

Or eval(f, ∅, x) est égale à f(x) puisque f est une fonction de base. Il faut donc prouver $y=f(x) \Rightarrow y=f(x) \wedge f(x)\neq\perp$. Mais puisque f est totale, f(x) est nécessairement différent de ⊥ quelque soit x. Cette formule est donc vrai pour tout axiome relatif aux constructeurs.

2.4. Les règles d'inférence.

2.4.1. Règle de la Conséquence.

$$\mathcal{P} : P_1(x) \rightarrow Q_1(x, f(x))$$

$$\forall x \ (P(x) \Rightarrow P_1(x))$$

$$\forall x, y \ (P(x) \wedge Q_1(x, y) \Rightarrow Q(x, y))$$

cons _____

$$\mathcal{P} : P(x) \rightarrow Q(x, f(x))$$

Si f est une fonction définie pour un domaine D_1, décrit comme l'extension de la précondition P_1, et si P décrit un domaine D inclus dans D_1(condition décrite par la seconde prémisse) alors la fonction f est aussi définie pour le sous-domaine D. D'autre part, si le résultat y de f, pour un x donné, est tel que $Q_1(x, y)$ et si $Q_1(x, y)$ implique Q(x, y) pour tous les x appartenant à D (condition décrite par la troisième prémisse) alors f est aussi définie par la relation Q(x, f(x)). Nous devons considérer tous les résultats possibles de la fonction f, la spécification (P_1, Q_1) ne décrivant pas nécessairement une seule fonction.

Remarque : Dans la règle habituelle de la conséquence la troisième prémisse est $\forall x, y \ (Q_1(x, y) \Rightarrow Q(x, y))$. Dans notre cas nous pouvons préciser que cette implication doit être vraie sous l'hypothèse P(x), puisque la donnée x n'est pas modifiée par l'évaluation de f. Cette extension de la règle de la conséquence est importante, car elle permettra la preuve de complétude du système. Il sera possible de comparer des spécifications dont les préconditions et les postconditions sont différentes.

Exemple :

Supposons qu'il a été prouvé que la fonction maximum vérifie la spécification suivante :

$$(x\neq nil, maximum(x) \in x \wedge SUP(x, maximum(x)))$$

Nous pouvons alors, par exemple, en déduire la spécification $(x\neq nil \wedge ORD(x), maximum(x)=car(x))$ par la règle de la conséquence. ORD est la propriété des listes décrivant le fait que la liste est ordonnée par ordre décroissant. La dérivation est :

$$x \neq nil \rightarrow maximum(x) \in x \wedge SUP(x, maximum(x))$$

$$\forall x \ (x \neq nil \wedge ORD(x) \Rightarrow x \neq nil)$$

$$\forall x, y \ (x \neq nil \wedge ORD(x) \wedge y \in x \wedge SUP(x, y) \Rightarrow y = car(x))$$

cons _____

$$x \neq nil \wedge ORD(x) \rightarrow maximum(x) = car(x)$$

Preuve de correction : nous devons montrer :

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

L'hypothèse découlant de la première prémisse est : $P_1(x) \wedge y = eval(f, \mathcal{P}, x) \Rightarrow Q_1(x, y) \wedge y \neq \perp$ (a).

Preuve :

$P(x) \wedge y = eval(f, \mathcal{P}, x) \Rightarrow$ $P(x) \wedge P_1(x) \wedge y = eval(f, \mathcal{P}, x)$ par la seconde prémisse

$\Rightarrow$ $P(x) \wedge Q_1(x, y) \wedge y \neq \perp$ par (a)

$\Rightarrow$ $Q(x, y) \wedge y \neq \perp$ par la troisième prémisse.

fin-preuve.

2.4.2. Règle de l'extension des arguments.

$$\mathcal{P} \cup \{ f(x) \leftarrow F \} : P(x) \rightarrow Q(x, f(x))$$

arg _____

$$\mathcal{P} \cup \{ f(x, y) \leftarrow F \} : P(x) \rightarrow Q(x, f(x, y))$$

Cette règle permet d'ajouter des arguments fictifs aux fonctions et ainsi de simplifier la présentation des règles en supposant que certaines fonctions sont définies pour un même ensemble de variables.

2.4.3. Règle de la Conditionnelle.

$$\mathcal{P}_0 : P_0(x) \rightarrow (C(x)=\text{vrai}) \equiv Q_0(x)$$

$$\mathcal{P}_1 : P_1(x) \rightarrow Q_1(x, f_1(x))$$

$$\mathcal{P}_2 : P_2(x) \rightarrow Q_2(x, f_2(x))$$

cond _____

$$\mathcal{P}_0 \cup \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{ f(x) \leftarrow \text{si } C(x) \text{ alors } f_1(x) \text{ sinon } f_2(x) \} :$$

$$P_0(x) \wedge (P_1(x) \wedge Q_0(x) \vee P_2(x) \wedge \neg Q_0(x))$$

$$\rightarrow Q_0(x) \wedge Q_1(x, f(x)) \vee \neg Q_0(x) \wedge Q_2(x, f(x))$$

C est une fonction booléenne décrivant, pour les x vérifiant la précondition P_0, une propriété Q_0 de x. Nous supposons que toutes les postconditions caractérisant les fonctions booléennes sont de la forme $(f(x)=\text{vrai}) \equiv Q_0(x)$. Ce choix sera justifié par la suite, lors de la présentation du langage d'énoncés. f_1 et f_2 sont deux fonctions vérifiant respectivement les spécifications (P_1, Q_1) et (P_2, Q_2). La conditionnelle **si** C **alors** f_1 **sinon** f_2 est alors définie pour les x tels que C(x) est définie et tels que f_1 est définie lorsque C(x) est vrai ou tels que f_2 est définie lorsque C(x) est faux. Par définition de la conditionnelle, le résultat de f est celui de f_1 si C(x) est vrai et celui de f_2 sinon.

Exemple. soit f définie par le programme suivant :

$$f(x1, x2) \leftarrow \textbf{si } \text{inf}(x1, x2) \textbf{ alors } -(x2, x1) \textbf{ sinon } \text{div}(x2, x1)$$

Nous supposons que les fonctions **inf**, - et **div** sont des fonctions prédéfinies décrites par les axiomes :

- $\emptyset$: vrai $\rightarrow$ $(\text{inf}(x1, x2)=\text{vrai}) \equiv x1 < x2$
- $\emptyset$: $x2 \geq x1$ $\rightarrow$ $-(x2, x1) + x1 = x2$
- $\emptyset$: $x1 \neq 0$ $\rightarrow$ $x2=\text{div}(x2, x1)x1 + \text{res}(x2, x1) \wedge \text{res}(x2, x1) < x1$

res(x2, x1) est le reste de la division euclidienne de x2 par x1.

La règle ci-dessus permet donc de déduire une spécification de la fonction f.

- La précondition est : $\text{vrai} \wedge [x2 \geq x1 \wedge x1 < x2 \vee x1 \neq 0 \wedge \neg(x1 < x2)]$. C'est-à-dire $x1 \neq 0 \vee x2 \neq 0$.

- La postcondition est :

$$[f(x1, x2) + x1 = x2 \wedge x1 < x2]$$
$$\vee$$
$$[\, x2 = f(x1, x2) * x1 + res(x2, x1) \wedge res(x2, x1) < x1 \wedge \neg(x1 < x2)]$$

fin-exemple.

Preuve de correction. Il faut montrer :

$$P_0(x) \wedge (P_1(x) \wedge Q_0(x) \vee P_2(x) \wedge \neg Q_0(x)) \wedge y = \mathbf{eval}(f, \mathcal{P}, x)$$
$$\Rightarrow (Q_0(x) \wedge Q_1(x, y) \vee \neg Q_0(x) \wedge Q_2(x, y)) \wedge y \neq \perp$$

Le programme $\mathcal{P}$ dans lequel prouver cette formule est, d'après la règle d'inférence, $\mathcal{P}_0 \cup \mathcal{P}_1 \cup \mathcal{P}_2 \cup$ { f(x) ← si C(x) alors $f_1(x)$ sinon $f_2(x)$ }. f étant définie par une forme conditionnelle, l'axiomatisation de la fonction eval utilisée est :

$$y=\mathbf{eval}(f, \mathcal{P}, x)$$
$$\equiv$$
$$(\mathbf{eval}(C, \mathcal{P}, x)=\mathbf{vrai} \wedge y=\mathbf{eval}(f_1, \mathcal{P}, x))$$
$$\vee (\mathbf{eval}(C, \mathcal{P}, x)=\mathbf{faux} \wedge y=\mathbf{eval}(f_2, \mathcal{P}, x))$$
$$\vee (\mathbf{eval}(C, \mathcal{P}, x)=\perp \wedge y=\perp)$$

Les interprétations associées aux prémisses sont :

(a) $P_0(x) \wedge y=\mathbf{eval}(C, \mathcal{P}_0, x) \Rightarrow (y=\mathbf{vrai}) \equiv Q_0(x) \wedge y \neq \perp$

(b) $P_1(x) \wedge y=\mathbf{eval}(f_1, \mathcal{P}_1, x) \Rightarrow Q_1(x, y) \wedge y \neq \perp$

(c) $P_2(x) \wedge y=\mathbf{eval}(f_2, \mathcal{P}_2, x) \Rightarrow Q_2(x, y) \wedge y \neq \perp$

Puisque $\mathcal{P}$ contient $\mathcal{P}_0$, $\mathcal{P}_1$ et $\mathcal{P}_2$ ces formules sont aussi vraies pour le programme $\mathcal{P}$. De (a) nous déduisons les formules suivantes :

(d) $P_0(x) \wedge \mathbf{vrai}=\mathbf{eval}(C, \mathcal{P}, x) \Rightarrow Q_0(x)$

(e) $P_0(x) \wedge \mathbf{faux}=\mathbf{eval}(C, \mathcal{P}, x) \Rightarrow \neg Q_0(x)$

(f) $P_0(x) \Rightarrow \mathbf{eval}(C, \mathcal{P}, x) \neq \perp$

La justification de la dernière assertion est la suivante :

lorsque eval(C, $\mathcal{P}$, x) = $\perp$ nous avons $P_0(x)$ $\wedge$ ($\perp$ = eval(C, $\mathcal{P}$, x) $\Rightarrow$ ($\perp$=vrai)$\equiv Q_0(x)$ $\wedge$ $\perp \neq \perp$. $\perp$ est un terme quelconque, auquel nous n'associons pas de sens particulier, donc $\perp \neq \perp$ et $\perp$=vrai s'évaluent à faux. L'implication précédente se réécrit en $\neg(P_0(x)$ $\wedge$ $\perp$=eval(C, $\mathcal{P}$, x)), c'est à dire en $P_0(x)$ $\Rightarrow$ eval(C, $\mathcal{P}$, x)$\neq\perp$.

Preuve :

$P_0(x) \wedge ((P(x) \wedge Q_0(x)) \vee (P_2(x) \wedge \neg Q_0(x))) \wedge y$=eval(f, $\mathcal{P}$, x)

$\Rightarrow P_0(x) \wedge ((P_1(x) \wedge Q_0(x)) \vee (P_2(x) \wedge \neg Q_0(x))) \wedge$ ((eval(C, $\mathcal{P}$, x)=vrai $\wedge$ y=eval(f_1, $\mathcal{P}$, x))
$\vee$ (eval(C, $\mathcal{P}$, x)=faux $\wedge$ y=eval(f_2, $\mathcal{P}$, x)) $\vee$ (eval(C, $\mathcal{P}$, x)=$\perp$ $\wedge$ y=$\perp$)) $\wedge$ eval(C, $\mathcal{P}$, x)$\neq\perp$
(par définition de **eval** et par la formule **f**).

$\Rightarrow ((P_1(x) \wedge Q_0(x)) \vee (P_2(x) \wedge \neg Q_0(x))) \wedge ((Q_0(x)$ $\wedge$ y=eval(f_1, $\mathcal{P}$, x))
$\vee$ ($\neg Q_0(x)$ $\wedge$ y=eval(f_2, $\mathcal{P}$, x))) (par **d** et **e**).

$\Rightarrow (P_1(x) \wedge Q_0(x) \wedge$ y=eval(f_1, $\mathcal{P}$, x)) $\vee$ $(P_2(x) \wedge \neg Q_0(x) \wedge$ y=eval(f_2, $\mathcal{P}$, x)) (simplifications).

$\Rightarrow (Q_1(x, y) \wedge Q_0(x)) \vee (Q_2(x, y) \wedge \neg Q_0(x)) \wedge y \neq \perp$ (par **b** et **c**).

fin-preuve de correction.

2.4.4. Règle de la Composition Fonctionnelle.

$$\mathcal{P}_g : P_g(y) \rightarrow Q_g(y, g(y))$$

$$\mathcal{P}_i : P_i(x) \rightarrow Q_i(x, f_i(x))$$

comp$_____$

$$\mathcal{P}_g \cup (\cup_{(i=1, n)} \mathcal{P}_i) \cup \{ f(x) \leftarrow g(f_1(x), ..., f_n(x)) \} :$$

$$\Pi_{(i=1, n)} P_i(x) \wedge \forall y (\Pi_{(i=1, n)} Q_i(x, y_i) \Rightarrow P_g(y))$$

$$\rightarrow \exists y (\Pi_{(i=1, n)} Q_i(x, y_i) \wedge Q_g(y, f(x)))$$

y_i désigne le ième argument du n-uplet y, $\Pi_{(i=1, n)} P_i(x)$ la conjonction des formules $P_i(x)$ pour i égal 1 à n. Soient n fonctions f_i définies sur n domaines D_i et une fonction g définie pour n arguments d'un domaine D. La fonction f est alors définie pour les éléments appartenant à l'intersection des domaines D_i ($\Pi_{(i=1, n)} P_i(x)$) et pour lesquels ($f_1(x)$, ..., $f_n(x)$) appartient à D. Les résultats de ces fonctions sont caractérisés par les postconditions $Q_i(x, y_i)$, nous devons donc avoir :

$$\forall y_1, \ldots, y_n \ (\Pi_{(i=1, n)} (P_i(x) \wedge Q_i(x, y_i)) \Rightarrow P_g(y_1, \ldots, y_n))$$

Ceci garantit que la fonction g est bien définie pour les arguments ($f_1(x)$, ..., $f_n(x)$). Les résultats de la fonction f sont les z qui peuvent être obtenus à partir d'un n-uplet (y_1, ..., y_n) par la relation $Q_g(y_1, \ldots, y_n, z)$, ce n-uplet étant lui-même obtenu à partir de x par les relations $Q_i(x, y_i)$.

Exemple. Soit f la fonction qui délivre la plus grande différence d'une liste d'entiers définie par :

$$f(x) \leftarrow - (\max(x), \min(x))$$

Avec :

- $x \neq nil \rightarrow dans(\min(x), x) \wedge \forall w \ [dans(w, x) \Rightarrow \min(x) \leq w]$
- $x \neq nil \rightarrow dans(\max(x), x) \wedge \forall w \ [dans(w, x) \Rightarrow \max(x) \geq w]$
- $y1 \geq y2 \rightarrow -(y1, y2) + y2 = y1$

- D'après la règle ci-dessus, f est donc définie pour tous les x tels que :

$$x \neq nil \wedge \forall y1, y2 \ (dans(y1, x) \wedge \forall w \ (dans(w, x) \Rightarrow y1 \geq w)$$
$$\wedge dans(y2, x) \wedge \forall w \ (dans(w, x) \Rightarrow y2 \leq w) \Rightarrow y1 \geq y2)$$

De dans(y2, x) et $\forall w$ (dans(w, x) $\Rightarrow$ y1$\geq$ w) on peut déduire $y1 \geq y2$ pour tout y1 et y2. Le second terme de la disjonction se réduit donc à vrai. f est alors définie pour tout x non nul.

- La postcondition associée à f est :

$$\exists y1, y2 \ [dans(y1, x) \wedge \forall w \ [dans(w, x) \Rightarrow y1 \geq w] \wedge dans(y2, x) \wedge$$
$$\forall w \ [dans(w, x) \Rightarrow y2 \leq w] \wedge f(x) + y2 = y1]$$

fin-exemple.

Preuve de correction. Nous devons montrer :

$$\Pi_{(i=1,\,n)}\, P_i(x) \wedge \forall\, w\, (\Pi_{(i=1,\,n)}\, Q_i(x, w_i) \Rightarrow P_g(w)) \wedge z=\mathbf{eval}(f, \mathcal{P}, x)$$

$$\Rightarrow \exists\, y\, (\Pi_{(i=1,\,n)}\, Q_i(x, y_i) \wedge Q_g(y, z)) \wedge z \neq \perp$$

Pour $\mathcal{P}$ défini par $\mathcal{P}_g \cup (\cup\, \mathcal{P}_i) \cup \{\, f(x) \leftarrow g(f_1(x), ..., f_n(x)) \,\}$. La définition de la fonction **eval** pour la composition fonctionnelle, est, rappelons le :

$$(z=\mathbf{eval}(f, \mathcal{P}, x)) \equiv (\exists\, y_1, ..., y_n\, (\Pi_{(i=1,\,n)}\, y_i=\mathbf{eval}(f_i, \mathcal{P}, x) \wedge z=\mathbf{eval}(g, \mathcal{P}, (y_1, ..., y_n))))$$

Les interprétations associées aux prémisses 1 et 2 sont :

(a) $\quad P_i(x) \wedge w_i=\mathbf{eval}(f_i, \mathcal{P}_i, x) \quad \Rightarrow \quad Q_i(x, w_i) \wedge w_i \neq \perp$

(b) $\quad P_g(w) \wedge z=\mathbf{eval}(g, \mathcal{P}_g, w) \quad \Rightarrow \quad Q_g(w, z) \wedge z \neq \perp$

Puisque $\mathcal{P}_i$ et $\mathcal{P}_g$ sont inclus dans $\mathcal{P}$ les formules ci-dessus sont à fortiori vraies pour $\mathcal{P}$.

Preuve :

$\Pi_{(i=1,\,n)}\, P_i(x) \wedge \forall\, w\, (\Pi_{(i=1,\,n)}\, Q_i(x, w_i) \Rightarrow P_g(w)) \wedge z=\mathbf{eval}(f, \mathcal{P}, x)$

$\Rightarrow \quad \Pi_{(i=1,\,n)}\, P_i(x) \wedge \forall\, w\, (\Pi_{(i=1,\,n)}\, Q_i(x, w_i) \Rightarrow P_g(w))$
$\wedge\, \exists\, y_1, ..., y_n\, (\Pi_{(i=1,\,n)}\, y_i=\mathbf{eval}(f_i, \mathcal{P}, x) \wedge z=\mathbf{eval}(g, \mathcal{P}, y))$ (par définition de eval).

$\Rightarrow \quad \exists\, y\, (\Pi_{(i=1,\,n)}\, Q_i(x, y_i) \wedge z=\mathbf{eval}(g, \mathcal{P}, y)) \wedge \forall\, w\, (\Pi_{(i=1,\,n)}\, Q_i(x, w_i) \Rightarrow P_g(w))$ (par a).

$\Rightarrow \quad \exists\, y\, (\Pi_{(i=1,\,n)}\, Q_i(x, y_i) \wedge P_g(y) \wedge z=\mathbf{eval}(g, \mathcal{P}, y))$

$\Rightarrow \quad \exists\, y\, (\Pi_{(i=1,\,n)}\, Q_i(x, y_i) \wedge Q_g(y, z)) \wedge z \neq \perp$ (par b).

fin-preuve.

2.4.5. Règle de la Récursion.

La construction récursive que nous avons choisi d'axiomatiser est de la forme :

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

Nous nous intéressons principalement aux définitions directement récursives. Pour qu'une telle définition soit correcte il doit exister un cas d'arrêt de la récursion, c'est pour cela que nous imposons la forme conditionnelle si C(x) alors $f_1(x)$ sinon au premier niveau. Pour une telle construction, la règle se déduit des 2 précédentes puisque la récursion est une combinaison particulière d'une conditionnelle et de compositions fonctionnelles. Par définition de la récursion, nous ne pouvons caractériser l'énoncé (P(x), Q(x, f(x))) de la définition récursive qu'en fonction des énoncés des appels récursifs ($P(y_i)$, $Q_i(y_i, f(y_i))$). La précondition P et la postcondition Q seront donc déterminées par des équations récursives. La règle est alors :

$$\mathcal{P}_0 : \; P_0(x) \rightarrow (C(x)=\text{vrai}) \equiv Q_0(x)$$

$$\mathcal{P}_1 : \; P_1(x) \rightarrow Q_1(x, f_1(x))$$

$$\mathcal{P}_{i,j} : \; P_{i,j}(x) \rightarrow Q_{i,j}(x, h_{i,j}(x))$$

$$\mathcal{P}_g : \; P_g(z, x) \rightarrow Q_g(z, x, g(z, x))$$

rec_____

$$\mathcal{P}_0 \cup \mathcal{P}_1 \cup (\cup_{(i=1,\, n;\, j=1,\, k)} \mathcal{P}_{i,j}) \cup \mathcal{P}_g$$

$$\cup \{ f(x) \leftarrow \text{si } C(x) \text{ alors } f_1(x) \text{ sinon } g(f(h_{1,1}(x), ..., h_{1,k}(x)), ..., f(h_{n,1}(x), ..., h_{n,k}(x)), x) \} :$$

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

Soit f une fonction d'arité k dont la définition contient n appels récursifs. Il existe alors n * k fonctions $h_{i,j}$ qui calculent les paramètres des n appels récursifs. L'indice i désigne l'appel dans lequel la fonction $h_{i,j}$ intervient et j la place de cet argument dans le n-uplet. La fonction de recomposition g est définie pour les données z et x où z représente les n résultats des appels.

Nous allons maintenant établir les équations récursives définissant P et Q en utilisant les règles de la conditionnelle et de la composition fonctionnelle. Nous nommons respectivement $z_1, ..., z_n$ les termes $f(h_{1,1}(x), ..., h_{1,k}(x)), ..., f(h_{n,1}(x), ..., h_{n,k}(x))$ et w le terme $g(z_1, ..., z_n, x)$.

• **Recherche de P :**

1– D'après la règle de la composition fonctionnelle, le terme z_i, c'est à dire le résultat du ième appel récursif, vérifie :

- la précondition $A_i(x)$: $\Pi_{(j=1,\, k)} P_{i,j}(x) \wedge \forall y_i \, (\Pi_{(j=1,\, k)} Q_{i,j}(x, y_{i,j}) \Rightarrow P(y_i))$

- la postcondition $B_i(x, z_i)$: $\exists y_i \, (\Pi_{(j=1,\, k)} Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i))$

Nous avons supposé, par hypothèse d'induction, que (P, Q) est une spécification de la fonction f pour les arguments $(h_{i,1}(x), ..., h_{i,k}(x))$.

2- Par la règle de la composition fonctionnelle, le terme w, représentant le résultat de la construction récursive, vérifie la précondition suivante :

$$\Pi_{(i=1,\,n)}\, A_i(x) \wedge \forall z\, (\Pi_{(i=1,\,n)}\, B_i(x, z_i) \Rightarrow P_g(z, x))$$

C'est à dire :

$$\Pi_{(i=1,\,n\,;\,j=1,\,k)}\, (P_{i,j}(x) \wedge \forall\, y_i\, (\Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}) \Rightarrow P(y_i)))$$

$$\wedge\ \forall\, z\, (\Pi_{(i=1,\,n)}\ \exists\, y_i\, (\Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i)) \Rightarrow P_g(z, x))$$

$\Pi_{(i=1,\,n\,;\,j=1,\,k)}\, A_{i,j}$ est une notation simplifiée de $\Pi_{(i=1,\,n)}\, (\Pi_{(j=1,\,k)}\, A_{i,j})$.

3 - Or $\exists\, y\, R(x, y, z) \Rightarrow P_g(z, x)$ est une formule équivalente à $\forall y\, (R(x, y, z) \Rightarrow P_g(z, x))$ puisque $P_g(z, x)$ ne dépend pas de y. En factorisant les deux implications, nous obtenons la formule A(x) suivante :

$$\Pi_{(i=1,\,n\,;\,j=1,\,k)}\, P_{i,j}(x) \wedge \forall\, z\, (\Pi_{(i=1,\,n)}\, \forall y_i\, (\Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}) \Rightarrow P(y_i) \wedge (Q(y_i, z_i) \Rightarrow Pg(z, x))))$$

4 - D'après la règle de la conditionnelle, la précondition P est $P_0(x) \wedge (P_1(x) \wedge Q_0(x) \vee \neg Q_0(x) \wedge A(x))$. P est donc définie par l'équation récursive :

$$P(x)$$

$$\equiv$$

$$(P_0(x) \wedge P_1(x) \wedge Q_0(x))$$

$$\vee$$

$$(P_0(x) \wedge \neg Q_0(x) \wedge \Pi_{(i=1,\,n;\,j=1,\,k)}\, P_{i,j}(x)$$

$$\wedge\ \forall z\ (\Pi_{(i=1,\,n)}\, \forall y_i\, (\Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}) \Rightarrow P(y_i) \wedge (Q(y_i, z_i) \Rightarrow P_g(z, x)))))$$

• **Recherche de la postcondition Q :**

1- Rappelons que le terme z_i vérifie la postcondition $B_i(x, z_i)\ \exists y_i\ (\Pi_{(j=1,\ k)}\ Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i))$. Et donc, la postcondition B(x, w) du terme w, désignant le résultat de la construction récursive, est : $\exists z\ (\Pi_{(i=1,\ n)}\ B_i(x, z_i) \wedge Q_g(z, x, w))$. C'est à dire :

$$\exists\, z\ (\Pi_{(i=1,\ n)}\ (\exists y_i\ (\Pi_{(j=1,\ k)}\ Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i))) \wedge\ Q_g(z, x, w))$$

2 - D'après la règle de la conditionnelle, une postcondition de f est : $Q_1(x, f(x)) \wedge Q_0(x) \vee\ B(x, f(x)) \wedge \neg Q_0(x)$. Q est donc définie par l'équation récursive suivante :

$$Q(x, f(x))$$

$$\equiv$$

$$(Q_1(x, f(x)) \wedge Q_0(x))$$

$$\vee$$

$$(\exists\, z\ (\Pi_{(l=1,\ n)}\ \exists y_i\ (\Pi_{(j=1,\ k)}\ Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i))) \wedge\ Q_g(z, x, f(x))) \wedge \neg Q_0(x))$$

Nous avons donc obtenu deux équations récursives caractérisant la précondition et la postcondition associées à la construction récursive que nous considérons. Il reste à démontrer que ces équations définissent bien quelque chose, c'est à dire qu'il existe effectivement des relations les vérifiant. Cette preuve est nécessaire pour garantir la complétude de notre système, nous la présenterons donc dans la section suivante. En ce qui concerne la correction, il suffit de prouver :

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

La preuve de cette formule ne pose pas de problème particulier. Il suffit de remplacer P et Q à l'aide des deux équivalences obtenues ci-dessus et de mettre en place les démonstrations attachées à la conditionnelle et aux compositions fonctionnelles qui constituent la définition récursive considérée. Cette preuve nécessite néanmoins l'hypothèse d'induction que $P(y_i) \wedge\ z_i = \mathbf{eval}(f, \mathcal{P}, y_i) \Rightarrow Q(y_i, z_i)\ \wedge z_i \neq \perp$ est valide pour les arguments des appels récursifs. Nous ne détaillons pas plus cette démonstration.

Exemple. Soit la fonction tri définie par tri(x) ← **si** nul(x) **alors** nil **sinon** insert(car(x), tri(cdr(x))), avec :

• vrai	→	(nul(x)=vrai) ≡ (x=nil)	(fonction C du schéma de récursion)
• vrai	→	nil=nil	(fonction f1 délivrant le résultat pour le cas de base)
• x≠nil	→	cdr(x)=cdr(x)	(fonction $h_{i,j}$ calculant l'argument de la récursion)
ord(y2)	→	perm(y1 . y2, insert(y1, y2)) ∧ ord(insert(y1, y2))	(fonction de recomposition g)

tri est une fonction bien définie puisqu'il existe un ordre bien fondé tel que cdr(x) est inférieur à x, lorsque x n'est pas nul.

1 – La précondition **P** doit donc être telle que :

$$P(x) \equiv (x=nil \vee x\neq nil \wedge \forall z, y\ (y=cdr(x) \Rightarrow P(y) \wedge (Q(y, z) \Rightarrow ord(z))))$$

Cette formule se simplifie en $P(x) \equiv x=nil \vee (x\neq nil \wedge \forall z\ (P(cdr(x)) \wedge (Q(cdr(x), z) \Rightarrow ord(z))))$.

2 – La postcondition **Q(x, w)** doit être telle que :

$$Q(x,w) \equiv ((w=nil \wedge x=nil) \vee (\exists z\ (\exists y\ (y=cdr(x) \wedge Q(y, z)) \wedge perm(car(x) . z, w) \wedge ord(w)) \wedge x\neq nil))$$

Q(x, w) doit donc être telle que $Q(x, w) \equiv (w=nil) \wedge x=nil) \vee (\exists z\ (Q(cdr(x), z) \wedge perm(car(x) . z, w) \wedge ord(w)) \wedge x\neq nil)$.

Montrons que (vrai, perm(x, w) ∧ ord(w)) est une spécification possible, c'est à dire que :

(a) vrai ≡ $x=nil \vee (x\neq nil \wedge \forall z\ (perm(y, z) \wedge ord(z) \Rightarrow ord(z)))$

(b) perm(x, w) ∧ ord(w) ≡ $(w=nil \wedge x=nil) \vee (\exists z\ (perm(cdr(x), z) \wedge ord(z) \wedge perm(car(x) . z, w) \wedge ord(w)) \wedge x\neq nil)$

Preuve de (a) :

vrai ≡ $x=nil \vee (x\neq nil \wedge \forall z\ (perm(cdr(x), z) \wedge ord(z) \Rightarrow ord(z)))$

≡ x=nil ∨ x≠nil

≡ vrai.

Preuve de (b) :

– Cas x=nil :

perm(x, w) ∧ ord(w) ∧ x=nil ≡ perm(nil, w) ∧ ord(w) ∧ x=nil

≡ w=nil ∧ x=nil (en utilisant les théorèmes ord(nil) et perm(nil, nil))

D'autre part la partie droite de l'équivalence **b** se réduit trivialement en w=nil ∧ x=nil lorsque x est la liste vide. **b** est donc bien une équivalence valide pour le cas considéré.

– Cas x≠nil. Soient les théorèmes suivants :

(1) ∃ b (perm(a . b, c) ∧ perm(d, b)) ≡ perm(a . d, c) (cette formule découle de la propriété de transitivité de la relation perm)

(2) ∀ x ∃ z (perm(z, w) ∧ ord(z)) (pour toute liste w il existe une permutation ordonnée)

perm(x, w) ∧ ord(w) ∧ x≠nil ≡ perm(car(x) . cdr(x), w) ∧ ord(w) ∧ x≠nil

≡ ∃ b (perm(car(x) . b, w) ∧ perm(cdr(x), b)) ∧ ord(w) ∧ x≠nil
(par le théorème (1))

≡ ∃ z (perm(car(x) . z, w) ∧ perm(cdr(x), z) ∧ ord(z)) ∧ ord(w) ∧ x≠nil
(par (2) et la transitivité de la relation perm).

D'autre part, la formule obtenue ci-dessus correspond bien à la partie droite de l'équivalence **b**, lorsque x n'est pas la liste vide. **b** est donc une équivalence valide.

fin-exemple.

2.5. Complétude.

Rappelons que nous voulons montrer que toute formule $\mathcal{P}$: P(x) → Q(x, f(x)) valide dans notre interprétation peut être démontrée. Pour cela, comme nous l'avons dit précédemment, nous devons définir une classe particulière de spécifications, *les plus générales*, desquelles il sera possible de déduire, pour un programme donné, toute autre spécification valide.

2.5.1. Spécifications les plus générales.

définition :

(P(x), Q(x, f(x))) est une spécification *la plus générale* si l'extension de P décrit le plus grand domaine de définition de f tel que f(x) est définie, et si, pour ce domaine, Q est la plus forte postcondition de f. Ceci s'exprime par les équivalences :

$$\forall x \ (\mathbf{eval}(f, \mathcal{P}, x) \neq \perp \equiv P(x)) \qquad \textbf{(a)}$$

$$\forall x, y \ (Q(x, y) \equiv (y = \mathbf{eval}(f, \mathcal{P}, x)) \wedge P(x)) \qquad \textbf{(b)}$$

fin-définition.

Remarque : les programmes étant fonctionnels, la plus forte postcondition détermine de manière unique le résultat associé à une valeur donnée de x puisque **eval**(f, $\mathcal{P}$, x) ne peut prendre qu'une seule valeur.

théorème :

Soit f une fonction du langage cible. Si S est une spécification *la plus générale*, c'est à dire telle que les propriétés **(a)** et **(b)** sont vraies, alors toute autre spécification vérifiée par f est déductible de S par la règle de la conséquence.

fin-théorème.

Preuve :

Si (P(x), Q(x, f(x))) est une spécification *la plus générale* et si ($P_1(x)$, $Q_1(x, f(x))$) est une spécification quelconque vérifiée par f alors on a :

(a) $\mathbf{eval}(f, \mathcal{P}, x) \neq \perp \equiv P(x)$

(b) $Q(x, y) \equiv (y = \mathbf{eval}(f, \mathcal{P}, x)) \wedge P(x)$

(c) $P_1(x) \Rightarrow Q_1(x, \mathbf{eval}(f, \mathcal{P}, x)) \wedge \mathbf{eval}(f, \mathcal{P}, x) \neq \perp$ ("f vérifie ($P_1(x)$, $Q_1(x, f(x))$)")

Montrons alors que (P_1, Q_1) peut être obtenue à partir de (P, Q) par la règle de la conséquence, qui est, rappelons le :

$$\mathcal{P} : P_1(x) \rightarrow Q_1(x, f(x)) \quad (1)$$

$$P(x) \Rightarrow P_1(x) \quad (2)$$

$$P(x) \wedge Q_1(x, y) \Rightarrow Q(x, y) \quad (3)$$

cons ----------

$$\mathcal{P} : P(x) \rightarrow Q(x, f(x))$$

Cela revient à montrer que les prémisses 2 et 3 sont nécessairement vraies.

$$\begin{aligned} -P_1(x) &\Rightarrow \mathbf{eval}(f, \mathcal{P}, x)\neq\perp \quad \text{(par c)} \\ &\Rightarrow P(x) \quad \text{(par a)} \end{aligned}$$

$$\begin{aligned} -P_1(x) \wedge Q(x, y) &\Rightarrow P_1(x) \wedge y=\mathbf{eval}(f, \mathcal{P}, x) \quad \text{(par b)} \\ &\Rightarrow Q_1(x, y) \quad \text{(par c).} \end{aligned}$$

fin-preuve.

La règle de la conséquence induit donc un ordre partiel sur les spécifications d'une même fonction : (P_1, Q_1) est *plus générale que* (P_2, Q_2) si et seulement si (P_2, Q_2) peut être déduite de (P_1, Q_1) par la règle de la conséquence. Cet ordre possède une borne inférieure unique qui est la classe des spécifications *les plus générales*. La règle habituelle de la conséquence se décompose en deux règles, la règle de la précondition et la règle de la postcondition, qui sont :

$$\frac{P\,\{S\}\,Q \qquad P_1 \Rightarrow P}{P_1\,\{S\}\,Q} \qquad\qquad \frac{P\,\{S\}\,Q \qquad Q \Rightarrow Q_1}{P\,\{S\}\,Q_1}$$

Ces règles permettent respectivement de comparer les spécifications ayant même postcondition, la borne minimale étant (**pfpré**(S, Q), Q) ou les spécifications ayant même précondition, la borne minimale étant (P, **pfpost**(P, S)). Des spécifications n'ayant pas même précondition ou même postcondition ne sont pas comparables. Pour pouvoir comparer les spécifications en général, nous devons donc d'une part comparer les préconditions et d'autre part les postconditions pour une précondition donnée. Comme dans notre langage il n'y a pas modification de la valeur des données, ceci se décrit aisément puisque la précondition est toujours vraie une fois la fonction évaluée. Si ce n'était pas le cas, il faudrait déterminer quelle est la propriété vérifiée par x après exécution, par exemple en utilisant l'axiome de l'affectation pour la plus forte postcondition.

La preuve de complétude est faite par induction sur la structure de f. Nous allons montrer, pour chaque règle d'inférence, que la spécification proposée dans la conséquence est *la plus générale*, c'est à dire vérifie les propriétés (a) et (b), en supposant, par hypothèse d'induction, que les spécifications apparaissant dans les prémisses sont bien *les plus générales*. Si nous montrons d'autre part que les axiomes décrivent les spécifications *les plus générales* des fonctions de base, nous prouvons par induction que toutes les spécifications proposées sont *les plus générales*. Notre système est alors complet puisque, par la règle de la conséquence, il est effectivement possible de prouver toute spécification valide, comme il vient d'être montré.

Pour toute conclusion de la forme $\mathcal{P}$: $P(x) \rightarrow Q(x, f(x))$, nous devons donc montrer :

$$\text{(a)} \quad \mathbf{eval}(f, \mathcal{P}, x)\neq\perp \quad \equiv \quad P(x)$$

(b) $Q(x, y) \equiv (y=eval(f, \mathcal{P}, x)) \wedge P(x)$

Nous avons déja démontré que ces conclusions sont correctes, c'est à dire que $P(x) \Rightarrow Q(x, eval(f, \mathcal{P}, x)) \wedge eval(f, \mathcal{P}, x) \neq \perp$ est vrai pour chaque spécification déductible par les règles d'inférence. Pour prouver la complétude il suffit donc de démontrer :

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

2.5.2. Complétude de la règle de la conditionnelle.

La spécification proposée par cette règle est, rappelons le :

$$((P_0(x) \wedge P_1(x) \wedge Q_0(x)) \vee (P_0(x) \wedge P_2(x) \wedge \neg Q_0(x)),\ (Q_0(x) \wedge Q_1(x, y)) \vee (\neg Q_0(x) \wedge Q_2(x, y)))$$

y désigne le résultat f(x) et (P_0, Q_0), (P_1, Q_1) et (P_2, Q_2) sont les spécifications des fonctions C, f_1 et f_2. Par hypothèse d'induction nous supposons que ces spécifications sont les plus générales. Nous avons donc :

(1) $eval(C, \mathcal{P}, x) \neq \perp \equiv P_0(x)$

(2) $eval(f_1, \mathcal{P}, x) \neq \perp \equiv P_1(x)$

(3) $eval(f_2, \mathcal{P}, x) \neq \perp \equiv P_2(x)$

(4) $((y=vrai) \equiv Q_0(x)) \equiv y=eval(C, \mathcal{P}, x) \wedge P_0(x)$

(5) $Q_1(x, y) \equiv y=eval(f_1, \mathcal{P}, x) \wedge P_1(x)$

(6) $Q_2(x, y) \equiv y=eval(f_2, \mathcal{P}, x) \wedge P_2(x)$

De (4) nous déduisons :

(7) $Q_0(x) \equiv (vrai=eval(C, \mathcal{P}, x)) \wedge P_0(x)$

(8) $\neg Q_0(x) \equiv (faux=eval(C, \mathcal{P}, x)) \wedge P_0(x)$

• Montrons que la précondition proposée vérifie bien la propriété (a'). Ceci revient à montrer :

$$\textbf{eval}(f, \mathcal{P}, x)\neq\perp \Rightarrow P_0(x) \wedge P_1(x) \wedge Q_0(x) \vee P_0(x) \wedge P_2(x) \wedge \neg Q_0(x)$$

Pour f définie par f(x) ← **si** C(x) **alors** $f_1(x)$ **sinon** $f_2(x)$.

Preuve :

eval(f, $\mathcal{P}$, x)≠⊥

$\Rightarrow$ (**eval**(C, $\mathcal{P}$, x)=vrai $\wedge$ **eval**(f_1, $\mathcal{P}$, x)≠⊥) $\vee$ (**eval**(C, $\mathcal{P}$, x)=faux $\wedge$ **eval**(f_2, $\mathcal{P}$, x)≠⊥)

$\vee$ (**eval**(C, $\mathcal{P}$, x)=⊥ $\wedge$ ⊥≠⊥)

$\Rightarrow$ (**eval**(C, $\mathcal{P}$, x)=vrai $\wedge$ $P_0(x) \wedge P_1(x)$) $\vee$ (**eval**(C, $\mathcal{P}$, x)=faux $\wedge$ $P_0(x) \wedge P_2(x)$) (par 1, 2 et 3).

$\Rightarrow$ $(P_0(x) \wedge P_1(x) \wedge Q_0(x)) \vee (P_0(x) \wedge P_2(x) \wedge \neg Q_0(x))$ (par 7 et 8).

fin-preuve.

• Montrons que la postcondition proposée vérifie la propriété (b'), c'est à dire est telle que :

$$(Q_0(x) \wedge Q_1(x, y)) \vee (\neg Q_0(x) \wedge Q_2(x, y)) \Rightarrow (y=\textbf{eval}(f, \mathcal{P}, x)) \wedge y\neq\perp$$

Preuve :

$Q_0(x) \wedge Q_1(x, y) \vee \neg Q_0(x) \wedge Q_2(x, y)$

$\Rightarrow$ (**eval**(C, $\mathcal{P}$, x)=vrai $\wedge$ y=**eval**(f_1, $\mathcal{P}$, x)) $\vee$ (**eval**(C, $\mathcal{P}$, x)=faux $\wedge$ y=**eval**(f_2, $\mathcal{P}$, x)) (par 5, 6, 7 et 8).

$\Rightarrow$ y=**eval**(f, $\mathcal{P}$, x) $\wedge$ y≠⊥

fin-preuve.

2.5.3. Complétude de la règle de la Composition Fonctionnelle.

La spécification déduite par cette règle est, rappelons le :

$$(\Pi_{(i=1, n)} P_i(x) \wedge \forall y_1, \ldots, y_n (\Pi_{(i=1, n)} Q_i(x, y_i) \Rightarrow P_g(y_1, \ldots, y_n)),$$

$$\exists y (\Pi_{(i=1, n)} Q_i(x, y_i) \wedge Q_g(y_1, \ldots, y_n, z)))$$

z désigne le résultat f(x) et (P_g, Q_g) et (P_i, Q_i) sont les spécifications des fonctions g et f_i. Nous avons donc les hypothèses :

$$(1)\quad \text{eval}(g, \mathcal{P}, w)\neq\perp \equiv P_g(w)$$

$$(2)\quad \text{eval}(f_i, \mathcal{P}, x)\neq\perp \equiv P_i(x)$$

$$(3)\quad Q_g(w, z) \equiv P_g(w) \wedge z=\text{eval}(g, \mathcal{P}, w)$$

$$(4)\quad Q_i(x, w_i) \equiv P_i(x) \wedge w_i=\text{eval}(f_i, \mathcal{P}, x)$$

• Montrons que la précondition proposée vérifie la propriété (a'), c'est à dire est telle que :

$$\text{eval}(f, \mathcal{P}, x)\neq\perp \Rightarrow (\Pi_{(i=1, n)} P_i(x) \wedge \forall y_1, \ldots, y_n (\Pi_{(i=1, n)} Q_i(x, y_i) \Rightarrow P_g(y_1, \ldots, y_n))$$

Pour f définie par $f(x) \leftarrow g(f_1(x), \ldots, f_n(x))$.

Preuve :

eval(f, x)≠⊥

$\Rightarrow$ $\exists w_1, \ldots, w_n (\Pi_{(i=1, n)} w_i=\text{eval}(f_i, \mathcal{P}, x) \wedge \text{eval}(g, \mathcal{P}, (w_1, \ldots, w_n))\neq\perp)$ (par la définition de eval).

$\Rightarrow$ $\exists w_1, \ldots, w_n (\Pi_{(i=1, n)} (w_i=\text{eval}(f_i, \mathcal{P}, x) \wedge w_i\neq\perp) \wedge \text{eval}(g, \mathcal{P}, (w_1, \ldots, w_n))\neq\perp)$.
En effet, comme nous nous intéressons uniquement aux fonctions strictes nous avons $\text{eval}(f, \mathcal{P}, x)\neq\perp \Rightarrow x\neq\perp$.

$\Rightarrow$ $\Pi_{(i=1, n)} P_i(x) \wedge \exists w_1, \ldots, w_n (\Pi_{(i=1, n)} w_i=\text{eval}(f_i, \mathcal{P}, x) \wedge P_g(w_1, \ldots, w_n))$ (par 1 et 2).

$\Rightarrow$ $\Pi_{(i=1, n)} P_i(x) \wedge \exists w_1, \ldots, w_n (\Pi_{(i=1, n)} Q_i(x, w_i) \wedge P_g(w_1, \ldots, w_n))$ (par 3).

D'autre part puisque Q_i est la plus forte postcondition nous avons :

$$Q_i(x, w_i) \Rightarrow \forall y_i (Q_i(x, y_i) \Rightarrow y_i=w_i)$$

Cette propriété décrit le fait que les plus fortes postconditions déterminent le résultat de manière unique. Et donc $\Pi_{(i=1, n)} Q_i(x, w_i) \wedge P_g(w_1, \ldots, w_n) \Rightarrow \forall y_i (\Pi_{(i=1, n)} Q_i(x, y_i) \Rightarrow P_g(y_1, \ldots, y_n))$. On déduit donc :

$\Rightarrow$ $\Pi_{(i=1, n)} P_i(x) \wedge \forall y \Pi_{(i=1, n)} (Q_i(x, y_i) \Rightarrow P_g(y_1, \ldots, y_n))$

fin-preuve.

• Montrons maintenant que la postcondition proposée est la plus forte pour la précondition précédente, c'est à dire que :

$$\exists y_1, ..., y_n \, (\Pi_{(i=1, n)} \, Q_i(x, y_i) \wedge Q_g(y_1, ..., y_n, z)) \Rightarrow (z=\mathbf{eval}(f, \mathcal{P}, x)) \wedge z \neq \perp$$

Preuve :

$\exists y_1, ..., y_n \, (\Pi_{(i=1, n)} \, Q_i(x, y_i) \wedge Q_g(y_1, ..., y_n, z))$

$\Rightarrow$ $\exists y_1, ..., y_n \, (\Pi_{(i=1, n)} \, y_i=\mathbf{eval}(f_i, \mathcal{P}, x) \wedge z=\mathbf{eval}(g, \mathcal{P}, (y_1, ..., y_n)) \wedge z \neq \perp)$ (par 3, 4 et 1).

$\Rightarrow$ $z=\mathbf{eval}(f, \mathcal{P}, x) \wedge z \neq \perp$

fin-preuve.

2.5.4. Complétude de la règle de la Récursion.

En ce qui concerne la règle de la récursion rappelons que nous n'avons pas donné de manière explicite une spécification mais que nous avons caractérisé la précondition et la postcondition à l'aide des deux équations récursives élaborées en section 2.2.6, qui sont, rappelons le :

$$\mathbf{P}(x) \equiv$$

$$(P_0(x) \wedge P_1(x) \wedge Q_0(x))$$

$$\vee \, (P_0(x) \wedge \neg Q_0(x) \wedge \Pi_{(i=1, n; j=1, k)} \, P_{i,j}(x)$$

$$\wedge \, \forall z \, (\Pi_{(i=1, n)} \, \forall y_i \, (\Pi_{(j=1, k)} \, Q_{i,j}(x, y_{i,j}) \Rightarrow P(y_i) \wedge (Q(y_i, z_i) \Rightarrow P_g(z, x)))))$$

et

$$\mathbf{Q}(x, f(x)) \equiv$$

$$(Q_1(x, f(x)) \wedge Q_0(x))$$

$$\vee \, (\exists z \, (\Pi_{(i=1, n)} \, (\exists y_i \, (\Pi_{(j=1, k)} \, Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i))) \wedge Q_g(z, x, f(x))) \wedge \neg Q_0(x))$$

Soient (1) et (2) ces deux équivalences.

Nous devons montrer d'une part qu'il existe bien une précondition et une postcondition vérifiant ces équations et d'autre part que ces solutions sont bien *les plus générales*. Cette dernière démonstration ne pose pas de problème et découle de la complétude de la règle de la composition fonctionnelle et de la conditionnelle. Prouvons maintenant l'existence de P et Q. Rappelons que toute précondition et postcondition proposées dans les règles d'inférence doivent,

pour garantir la correction et la complétude, vérifier les équivalences :

(a)	$\mathbf{eval}(f, \mathcal{P}, x) \neq \perp$	$\equiv$	$P(x)$
(b)	$(y = \mathbf{eval}(f, \mathcal{P}, x)) \wedge P(x)$	$\equiv$	$Q(x, y)$

La formule **a** énonce le fait que la précondition de la spécification la plus générale doit décrire l'ensemble exact des données pour lesquelles la fonction f est définie. La fonction f est, par définition de la fonction **eval**, le plus petit point fixe de l'équation $f(x) = \mathbf{si}\ C(x)\ \mathbf{alors}\ f_1(x)\ \mathbf{sinon}\ g(f(h_{1,1}(x), ..., h_{1,k}(x)), ..., f(h_{n,1}(x), ..., h_{n,k}(x)), x)$. La solution attendue pour P est donc le plus petit point fixe de l'équation **(1)**.

La formule **b** énonce le fait que, pour une précondition P quelconque, Q doit être la plus forte postcondition. Rappelons que cette contrainte permet de garantir que tout autre postcondition possible Q_1 peut être déduite de Q par la règle de la conséquence, en prouvant l'implication $P(x) \wedge Q(x, y) \Rightarrow Q_1(x, y)$. La solution attendue doit donc être la plus petite solution dans l'ordre partiel sur les formules, défini par l'implication. Manna et Pnueli [MaP 69] caractérise cette solution comme le prédicat valide minimum. Or l'ordre défini par $\Rightarrow$ correspond à l'ordre sur les points fixes. Q doit donc être le plus petit point fixe de l'équation **(2)**.

Les solutions attendues pour les équations **(1)** et **(2)** doivent donc être les plus petits points fixes. Montrons maintenant que ceux-ci existent. Les théorèmes et définitions ci-dessous sont issus de *Fixpoint Induction and Proofs of Program Properties* [Par 69].

théorème 1 :

Soit l'équivalence $\forall x\ (\mathcal{C}[X](x) \equiv X(x))$, où X est une variable relationnelle et la notation $\mathcal{C}[X]$ désigne le fait que la formule $\mathcal{C}$ dépend de la relation X. Si $\mathcal{C}$ est une formule monotone alors il existe nécessairement un plus petit point fixe.

fin-théorème.

Définition 1 :

$\mathcal{C}$ est une formule monotone en X si et seulement si la formule suivante est valide :

$$\forall x\ (X(x) \Rightarrow Y(x)) \Rightarrow \forall x\ (\mathcal{C}[X](x) \Rightarrow \mathcal{C}[Y](x))$$

fin-définition.

Théorème 2 :

Si $\mathcal{C}$ est une formule ne contenant que les symboles $\forall$, $\exists$, $\wedge$, $\vee$ et $\neg$ et telle que toutes les occurrences de X sont positives dans $\mathcal{C}$ alors $\mathcal{C}$ est monotone en X.

fin-théorème.

Définition 2 :

Une occurrence de X est positive dans $\mathfrak{C}$ si et seulement si elle n'apparait dans aucune sous-formule de la forme $\neg F$.

fin-définition.

Il suffit maintenant d'appliquer le théorème 1. La formule définissant Q est monotone en P et Q et donc, par le théorème 2, il existe un plus petit point fixe Q_0 vérifiant l'équation (2). D'autre part, en reportant Q_0 dans l'équivalence (1) et en utilisant l'équivalence entre les formules $A \Rightarrow B$ et $\neg A \vee B$, la formule définissant P est aussi monotone en P. Par le théorème 1 il existe donc un plus petit point fixe P_0 à l'équation (1). Ceci termine la preuve de l'existence d'une spécification associée aux définitions récursives.

2.5.5. Complétude des axiomes.

En ce qui concerne les axiomes il suffit de prouver la complétude de ceux associés aux fonctions constructeurs qui sont, rappelons le, de la forme $\emptyset$: vrai $\rightarrow$ f(x)=f(x). Il faut donc montrer :

(a') $\quad$ **eval**(f, $\emptyset$, x)$\neq\perp \Rightarrow$ vrai

(b') $\quad$ y=f(x) $\Rightarrow$ y=**eval**(f, $\emptyset$, x) $\wedge$ y$\neq\perp$

La formule (a') est trivialement vraie. D'autre part **eval**(f, $\emptyset$, x)=f(x) lorsque f est une fonction de base et **eval**(f, $\emptyset$, x) est toujours définie, les constructeurs étant supposés être des fonctions totales, la formule (b') est donc aussi un théorème. Ceci termine la preuve de complétude de notre système qui est relative à celle du CP1 en raison de la règle de la conséquence.

Nous avons proposé un système formel décrivant la sémantique du langage cible, ceci en terme des spécifications que vérifient les programmes. Le fait que le langage utilisé soit sans effet de bord a permis de décrire ces spécifications sans trop de difficulté. Un tel système formel peut être utilisé en tant que :

- **système de démonstration** : à partir d'une programme $\mathcal{P}$ et d'une spécification (P, Q) chercher à prouver que $\mathcal{P}$: P(x) $\rightarrow$ Q(x, f(x)) est un théorème. Ceci revient à faire de la preuve de programmes.

- **système d'inférence** :

  - A partir d'un programme $\mathcal{P}$ trouver une spécification (P, Q) telle que $\mathcal{P}$: P(x) $\rightarrow$ Q(x, f(x)) est un théorème, f étant une fonction définie dans $\mathcal{P}$. C'est ce que nous avons fait sur les exemples en décrivant la sémantique d'un programme en terme d'une spécification qu'il vérifie.

  – A partir d'une spécification (P, Q) trouver un programme $\mathcal{P}$ contenant une définition de f telle que

$\mathcal{P}$: $P(x) \rightarrow Q(x, f(x))$ est un théorème. Ceci est le processus nécessaire à la synthèse de programme. Le système peut alors être vu comme un système de démonstration de spécifications, la preuve élaborée pouvant être décrite par une fonction.

Bien que basés sur la même description de la sémantique du langage, les systèmes sous-jacents à ces différents utilisations peuvent être présentés différemment si on prend en compte la démarche propre au but visé. Avant de donner la présentation utilisée pour la synthèse, nous définissons, dans un premier temps, le langage de spécifications que nous retenons. Ce sera un sous-ensemble du Calcul des Prédicats : en effet, nous nous intéressons qu'aux formules qui définissent effectivement des objets à construire et donc qui nécessitent d'être prouvées dans un tel système.

CHAPITRE 2

LE LANGAGE ET LES PREUVES DES ENONCES

"Spécifier ou comment matérialiser l'abstrait"

J. Abrial [Abr 84].

1. LE LANGAGE D'ENONCES.

La programmation automatique vise la recherche d'un programme répondant à un problème donné. Cette " définition " est très vague car la classe des problèmes susceptibles d'être traités par informatique est difficilement caractérisable autrement que par le fait qu'on dispose d'un programme permettant de les résoudre! En effet, ceci évolue constamment : l'informatique était vue à ses débuts comme un moyen de décrire des traitements uniquement numériques, puis le développement de langages tels que LISP a permis de décrire des raisonnements sur des données symboliques. Et maintenant nous envisageons de doter les ordinateurs de facultés de raisonnements sur leurs propres facultés de raisonnements. Il semble vraiment difficile de chercher à caractériser les problèmes auquels on peut être confrontés de manière générale et encore plus difficile de préciser comment les exprimer ... Nous ne nous attardons pas plus sur un sujet par trop mouvant et cherchons plutôt à préciser, avec les restrictions que nous imposons, la forme des problèmes que nous serons amenés à traiter, ceci dans le but de caractériser les formules que nous considérerons comme des énoncés.

Pour notre part nous assimilons les programmes à des fonctions : un programme décrit un moyen de calculer des objets, les résultats, à partir d'autres objets, les données. Cettte vue des programmes est bien sûr limitative puisque, par exemple, il semble difficile de faire entrer dans cette catégorie les programmes de simulation. En effet, ceux-ci ne calculent à proprement dit rien mais modélisent des comportements (processus industriels, comportements humains tel l'apprentissage ...) dans lesquels nous ne pouvons pas caractériser les résultats attendus en fonction des données. En assimilant les programmes à des fonctions on peut alors voir l'énoncé d'un problème comme la donnée d'une relation liant les valeurs d'entrée possibles et les résultats attendus. Dans ce cadre la problématique de la synthèse revient à trouver des fonctions telles que les résultats de ces fonctions vérifient cette relation. Le but devient donc de transformer cette relation en une dépendance fonctionnelle. Un grand nombre de problèmes, pour lesquels on dispose ou non de programmes, peuvent s'exprimer sous la forme d'une dépendance entre données et résultats escomptés.

Exemples.

– La fonction **max** d'une liste d'entiers, de profil **liste(entier) → entier**, vérifie l'énoncé :

$$\forall x \ ((max(x) \in x) \wedge SUP(max(x), x))$$

comme nous l'avons déja vu. Le lien établi ici est fonctionnel, il n'existe qu'une fonction possible, c'est

la fonction maximum.

– Un compilateur peut être vu comme une fonction **compil** de l'ensemble $\mathcal{P}(L_1)$ des programmes d'un langage L_1 dans l'ensemble $\mathcal{P}(L_2)$ des programmes du langage L_2. Une relation vérifiée par une fonction **compil**, de profil $\mathcal{P}(L_1) \rightarrow \mathcal{P}(L_2)$, peut alors être :

$$\forall x \ (\text{compil}(x) \equiv x)$$

$\equiv$ désigne la relation d'équivalence de programmes qu'il reste à définir ... Le lien défini ici n'est pas fonctionnel, différents compilateurs pouvant produire des programmes différents mais équivalents.

– Un générateur de compilateur peut être vu comme une fonction **géner-compil** prenant en entrée deux langages L_1 et L_2, le langage cible et le langage source, et produisant comme résultat une fonction **compil**. Cette fonction a pour profil (L_1 : langage, L_2 : langage) $\rightarrow$ ($\mathcal{P}(L_2) \rightarrow \mathcal{P}(L_1)$) et vérifie la relation :

$$\forall L_1, L_2 \ (\forall x \ (\text{géner-compil}(L_1, L_2)(x) \equiv x))$$

géner-compil est une fonction générique, dépendante des langages source et cible.

– Le problème de la synthèse peut également être exprimé sous cette forme. Pour un énoncé E, décrivant une relation R entre les données x de type T_d et les résultats de type T_r, la synthèse doit construire une fonction f de profil $T_d \rightarrow T_r$ telle que R(x, f(x)) est vraie pour tout x. Les données sont donc des énoncés et les résultats des fonctions. Le problème de la synthèse est donc de construire une fonction **synth** qui pour tout énoncé renvoie, si elle existe, une fonction vérifiant R. Le lien entre donnée et résultat peut s'exprimer par :

$$\forall E \ (\forall x \ (R(x, \text{synth}(E)(x))))$$

Cette relation décrit le fait que le résultat de l'exécution du programme synth(E) est tel que la relation R est vraie.

fin-exemples.

L'approche déductive de la Synthèse de Programmes nécessite de pouvoir raisonner sur les domaines de problème, afin de pouvoir construire une preuve de l'énoncé. Si nous voulons synthétiser les fonctions ci-dessus il faudrait alors disposer d'une théorie des listes, des langages ou des programmes pour pouvoir raisonner sur l'appartenance d'un élément à une liste, sur l'équivalence de programmes ... En ce qui concerne le problème de la Synthèse de Programmes, il faudrait disposer d'une théorie des énoncés et de la manière de les prouver pour obtenir des programmes. C'est justement à cette formalisation que nous nous intéressons dans ce travail. En dehors des limitations déjà données, les problèmes ne pourront donc porter que sur des domaines pour lesquels nous disposons d'une théorie sous-jacente. Ceci est limitatif car certains domaines se prêtent très difficilement à cela. Par exemple, utiliser une approche déductive pour la synthèse de programmes en robotique pose problème car il n'est pas évident de décrire formellement l'effet des actions exécutables par un robot : obstacles physiques, incertitude sur le résultat des actions ...

Nous allons maintenant présenter le langage des énoncés que nous utiliserons. Dans la plupart des travaux en synthèse de programmes le langage utilisé est le CP1 et un énoncé est classiquement de la forme :

$$\forall x \, \exists y \, (P(x) \Rightarrow Q(x, y))$$

x est la donnée, y le résultat, P(x) décrit la classe des données pour laquelle y est cherché et Q(x, y) est la relation définissant y en fonction de x. Il est supposé que les opérateurs, fonctionnels ou relationnels, apparaissant dans les formules P et Q sont connus du système, c'est à dire qu'il en posséde une définition. La preuve d'un tel énoncé, dans une logique adéquate, permet d'exhiber une fonction f permettant de calculer une valeur possible de y, pour chaque x ayant la propriété P. En fait, le problème de la synthèse est un problème du second ordre, puisque nous nous intéressons à la recherche d'une fonction. Un énoncé devrait donc être de la forme :

$$\exists f \, \forall x \; (P(x) \Rightarrow Q(x, f(x)))$$

Se ramener à des preuves constructives de la formule $\forall x \, \exists y \, (P(x) \Rightarrow Q(x, y))$ offre l'avantage conséquent de se limiter à des preuves du premier ordre, mais est plus limitatif. En effet, il n'est alors pas possible de définir récursivement la ou les fonctions cherchées puisque les noms de ces fonctions ne peuvent pas apparaître dans la relation entre données et résultats. Cette restriction est rapidement trop contraignante, certains problèmes se décrivant très facilement par une définition récursive et très difficilement, voir pas du tout, autrement. C'est le cas par exemple des fonctions primitives sur des domaines définis inductivement, c'est à dire des fonctions définies directement sur les constructeurs. Il faut donc considérer la possibilité de décrire la relation liant les données et les résultats par des programmes, notamment récursifs. Par exemple, la fonction **nbre-élem**, qui calcule le nombre d'élements d'une liste, vérifie la relation :

$$\exists f \, \forall x \; (f(x) = \textbf{si null}(x) \textbf{ alors } 0 \textbf{ sinon } 1 + f(\text{cdr}(x)))$$

Pour prendre en compte cette extension souhaitable, une solution est donc de considérer le prédicat d'égalité comme jouant le rôle de la flèche " est définie par " du langage cible et d'inclure les termes du langage cible en tant que termes du CP1. En fait, par rapport à la définition classique, il suffit d'ajouter les termes conditionnels. D'autre part, il faut autoriser, dans l'énoncé, des références au nom des fonctions cherchées. Cette extension nous fera quitter les preuves du premier ordre. Par exemple lors de la preuve de l'existence de la fonction **f** ci-dessus nous devrons utiliser l'hypothèse que **f**(cdr(x)) existe bien et est la fonction **nbre-élem**. Cette hypothèse d'induction est du second ordre puisqu'elle porte sur l'existence d'une fonction. Nous reviendrons sur ce problème dans la section 2.4.

D'autre part, comme nous l'avons déja dit, un énoncé n'a de sens que pour un domaine d'application particulier et donc la preuve doit s'effectuer dans la théorie propre à ce domaine. Pour prendre en compte cela, nous avons choisi d'introduire explicitement des contraintes de type dans les énoncés. Les énoncés sont alors définis dans le CP1 typé. Le typage que nous utilisons est le même que celui du langage cible, c'est à dire polymorphe. La preuve ne s'effectuera donc pas dans une seule théorie mais dans un ensemble de théories.

1.1. Calcul des prédicats typé avec égalité.

Le typage a été introduit en logique afin de pouvoir décrire aisément des formules portant sur des objets de nature différente. L'introduction explicite du typage dans le CP1 n'augmente en rien sa puissance d'expression puisque les contraintes de type peuvent être décrites par les propriétés attachées aux objets. Mais il permet d'une part de simplifier les formules et d'autre part de prendre en compte les contraintes de type dans les mécanismes d'inférence, ce qui est souvent plus efficace. Ceci permet, par exemple, d'éviter de faire des déductions qui mènent à des impasses. Par

exemple, lors de preuve par Résolution dans une logique typée, l'algorithme d'unification ne permet d'unifier deux termes que si bien sûr ceux-ci sont unifiables au sens classique mais aussi si ils ont le même type ou si ils possèdent un sous-type commun, lorsque des relations entre types sont prises en considération. De tels algorithmes d'unification restent encore efficaces, suivant bien entendu les relations entre type permises [Wal 83], [Sch 86].

1.1.1. Syntaxe.

Nous rappelons ci-dessous la définition des termes typés, cette définition étant la même que celle des termes du langage cible. Soient $\mathcal{V}$ un ensemble de variables typées, $\mathcal{P}$ un ensemble de nom de prédicats définis sur un n-uplet de types et $\mathcal{F}$ un ensemble de nom de fonctions ayant un profil de la forme $(s_1, \ldots, s_n) \rightarrow s$, où $s_1, \ldots, s_n$ et s sont des noms de type.

Définition des termes typés :

- Toute variable de type s est un terme de type s.

- Soient f un symbole de fonction de profil $(s_1, \ldots, s_n) \rightarrow s$ et $t_1, \ldots, t_n$ n termes respectivement de type $s'_1, \ldots, s'_n$, tels ques les ensembles de variables de type de ces expresions soient disjoints deux à deux. $f(t_1, \ldots, t_n)$ est alors un terme dont le type est $s\theta_n$ où θ_n est la composition des unificateurs des couples $(s_i\theta_{i-1}, s'_i\theta_{i-1})$, pour i=1 à n. Rappelons que θ_0 est le fonction identité.

- Si t_1 et t_2 sont deux termes de type $s_1$et s_2 et C un terme de type booléen alors si C alors t_1 sinon t_2 est un terme dont le type est la plus grande borne inférieure du couple (s_1, s_2) dans l'ordre défini sur les termes de type (voir chapitre précédent).

- $\perp$ est un terme de type quelconque.

Définition des formules atomiques :

- Si P appartient à $\mathcal{P}$ et est défini sur $(s_1, \ldots, s_n)$ et si $t_1, \ldots, t_n$ sont n termes respectivement de type $s'_1, \ldots, s'_n$, alors $P(t_1, \ldots, t_n)$ est une formule atomique, à condition que les couples $(s_i\theta_{i-1}, s'_i\theta_{i-1})$ soient unifiables. De la même manière nous supposons que les ensembles de variables de type des s'_i sont disjoints deux à deux.

- Si t_1 et t_2 sont deux termes de types respectifs s_1 et s_2 alors $t_1 = t_2$ est une formule atomique, à condition que $s_1$et s_2 soient unifiables.

Définition des formules bien formées :

- Toute formule atomique est une formule bien formée.

- Si P et Q sont deux formules bien formées alors $P \Rightarrow Q$, $P \vee Q$, $P \equiv Q$, $P \wedge Q$ et $\neg P$ sont des formules bien formées.

– Si P est une formule bien formée et x une variable de type s alors $\forall$ x : s P et $\exists$ x : s P sont des formules bien formées.

fin-définitions.

Nous avons introduit explicitement le terme $\perp$ car les fonctions que nous sommes amenées à manipuler peuvent être partielles. La sémantique des formules contenant $\perp$ est donnée ci-après en section 1.1.3.

1.1.2. Sémantique des termes conditionnels.

Les termes conditionnels que nous introduisons auront la même sémantique que la conditionnelle du langage cible. Ils sont donc définis de la manière suivante :

– **si vrai alors** t_1 **sinon** t_2 = t_1
– **si faux alors** t_1 **sinon** t_2 = t_2
– **si** $\perp$ **alors** t_1 **sinon** t_2 = $\perp$

Nous assimilons les termes à des expressions calculables. En effet, à chaque constructeur de termes est associé une procédure d'évaluation, par exemple la fonction **eval** décrivant la sémantique opérationnelle du langage cible. Il est donc possible d'évaluer tout terme complétement instancié, en supposant disposer d'algorithmes permettant de calculer les fonctions de $\mathcal{F}$. C'est pour cette raison que nous avons défini les termes conditionnels comme une construction de terme × terme × terme et non pas, comme le propose par exemple Manna [Man 74], de formule-bien-formée × terme × terme. En effet, en ce qui concerne les formules bien formées, nous ne disposons pas de procédure d'évaluation : même si nous pouvons décider de la validité des prédicats pour des arguments constants, il n'y a pas de procédure de décision associée à chaque constructeur de formules bien formées. C'est le cas des quantificateurs. La définition des termes conditionnels que nous adoptons introduit donc une distinction explicite entre **terme booléen et formule logique**. Les termes booléens sont des formules logiques pour lesquelles nous disposons d'une procédure de décision adaptée. Ils sont "inclus " dans les formules bien formées dans le sens où un terme booléen **t** décrit la propriété **t = vrai**. C'est pour cela que nous avons représenté les postconditions relatives aux fonctions booléennes par une relation de la forme (f(x)=vrai) ≡ Q(x). En effet, dans le cadre de la synthèse, nous partons de formules logiques et nous voulons en déduire des programmes. Une propriété Q d'une donnée x n'aura donc le statut d'un programme que si nous avons établi une procédure permettant d'évaluer cette propriété, pour toutes les valeurs possibles de x et cette procédure sera la fonction f cherchée.

L'introduction des termes conditionnels n'augmente en rien la puissance du CP1, nous ne les avons rajoutés que pour offrir une certaine souplesse d'expression à l'utilisateur. En effet nous avons les propriétés :

- f(..., **si** C **alors** t_1 **sinon** t_2, ...) = **si** C **alors** f(..., t_1, ...) **sinon** f(..., t_2, ...) lorsque f est stricte.

- P(..., **si** C **alors** t_1 **sinon** t_2, ...) ≡ C=vrai ∧ P(..., t_1, ...) ∨ C=faux ∧ P(..., t_2, ...) ∨ C=$\perp$ ∧ P(..., $\perp$, ...)

Ces propriétés permettent donc de réécrire toute formule contenant des termes conditionnels en des formules n'en

contenant pas. Par contre, au niveau des termes, la conditionnelle permet de décrire certaines fonctions de manière finie, ce qui ne serait pas le cas en son absence. En effet les définitions par " pattern " ne permettent pas toujours de décrire syntaxiquement la forme des objets pour lesquelles elles s'appliquent : les nombres premiers par exemple ... Rappelons qu'une fonction est stricte si et seuelement si $f(..., \perp, ...) = \perp$.

1.1.3. Sémantique du terme $\perp$

Les fonctions que nous allons construire doivent être strictes. Cette restriction a été utile pour démontrer la complétude de la règle de la composition fonctionnelle. Prendre en compte les fonctions non strictes ne nous intéresse pas puisque la valeur indéfinie désigne, entre autres, le fait que les calculs ne s'arrêtent pas et que nous cherchons à garantir la correction totale des programmes engendrés. **Toute formule contenant $\perp$ doit donc être fausse** puisque $\perp$ ne sera jamais une solution admise. Cette interprétation peut sembler abusive puisque par exemple $P(\perp)$ et $\neg P(\perp)$ sont toutes deux des formules fausses mais elle évite les raisonnements sur l'indéfini. Cette interprétation est d'ailleurs adoptée aussi en preuves de programmes [DeB 80]. Au niveau des termes nous ne pouvons pas être aussi catégorique, puisque des termes conditionnels contenant $\perp$ peuvent être bien définis. En effet, d'après leur sémantique, les expressions **si** vrai **alors** t_1 **sinon** $\perp$ et **si** faux **alors** $\perp$ **sinon** t_2 s'évaluent respectivement à t_1 et t_2. Nous sommes donc obligés de considérer à part l'interprétation des formules contenant $\perp$ dans une conditionnelle.

Définition.

Soit F une formule bien formée, alors :

- **Si F ne contient pas $\perp$ son interprétation est l'interprétation classique.**

- **Si F contient $\perp$ dans un terme conditionnel son interprétation est définie par les équivalences :**

$$F(..., \textbf{si } \perp \textbf{ alors } t_1 \textbf{ sinon } t_2, ...) \equiv \text{faux}$$
$$F(..., \textbf{si } C \textbf{ alors } \perp \textbf{ sinon } t_2, ...) \equiv C=\text{faux} \wedge F(..., t_2, ...)$$
$$F(..., \textbf{si } C \textbf{ alors } t_1 \textbf{ sinon } \perp, ...) \equiv C=\text{vrai} \wedge F(..., t_1, ...)$$

- **Sinon F est une formule non valide.**

fin-définition.

1.2. Les énoncés.

Les énoncés se présentent sous la forme d'un triplet de la forme :

profil : $var_1 = f_1(x : t_x) \rightarrow t_a, ..., var_n = f_n(x : t_x) \rightarrow t_b$
préc : $P(x)$
post : $Q(x, var_1, ..., var_n)$

La donnée du problème est le k-uplet x. Les résultats des n fonctions sont nommées par des variables, var_1, ..., var_n. Plusieurs fonctions peuvent être décrites par un même énoncé, lorsqu'elles s'appliquent à des arguments vérifiant une même précondition.

• Le **profil** permet de préciser les noms des fonctions à construire et les arguments sur lesquels elles portent. Nous imposons de nommer l'occurrence des fonctions f_i à construire ceci permettant, si la postcondition définit récursivement ces fonctions, de différencier les occurrences d'utilisation et les occurrences de définition. Les indications de type permettent, comme nous l'avons dit, de sélectionner les théories dans lesquelles la preuve se déroulera.

• La **précondition** (section **prec**) décrit la classe des objets pour laquelle les fonctions seront synthétisées. Elle est optionnelle : le processus de synthèse peut être amené à restreindre, en cours de synthèse, la classe des objets sur laquelle porte le problème soit parce que certains opérateurs ne sont pas définis pour tous ces objets ou soit par manque de capacité du système.

• La **postcondition** est une formule décrivant le lien attendu entre données et résultats. Ce lien n'est pas nécessaire de type fonctionnel ; l'énoncé peut être vrai pour différentes fonctions non égales. Dans ce cas nous disons que l'énoncé est non déterministe. C'est pour cette raison que nous préférons parler d'énoncés de problèmes plutôt que d'énoncés de fonctions. Imposer aux énoncés de caractériser des résultats uniques serait d'une part trop contraignant et, d'autre part, non cohérent. En effet le système peut être amené à engendrer des énoncés non déterministes à partir d'un énoncé initial qui est, lui, déterministe.

1.2.1. Les préconditions.

Les préconditions doivent décrire une propriété du k-uplet de données. Cette propriété doit être indépendante de la fonction à construire, puiqu'elle décrit son domaine de définition. **Une précondition est donc une formule dans laquelle les seules variables libres appartiennent au k-uplet x et telle qu'il n'y ait pas d'occurrence des symboles fonctionnels à construire.** Nous imposons aussi que certains éléments du k-uplet x apparaissent effectivement en tant que variables libres, ceci garantissant, de manière syntaxique, que cette propriété dépend bien des données. Les préconditions sont généralement des formules simples mais elles peuvent, parfois, être aussi compliquées à prouver que la postcondition. Par exemple si nous voulons décrire une fonction calculant la racine carrée d'un entier, telle que cette racine est aussi entière, il est difficile de caractériser le sous-ensemble des entiers auquel elle s'applique autrement que par le fait que la racine carrée est entière ! Un énoncé pour ce problème serait donc :

profil : rac = f(x : entier) → entier
préc : $\exists$ y (y^2 = x)
post : rac^2 = x

Un programme vérifiant cet énoncé est :

rac(x) ← si h(0, x) < x+1 alors h(0, x) sinon ⊥

h(n, x) ← si n^2=x alors n sinon si n = x+1 alors n sinon g(n+1, x)

Si le lien attendu entre x et la fonction f est Q(x, f(x)) alors la précondition attachée à f est, en toute rigueur, l'ensemble des x pour lesquels il est possible de construire f; ceci peut s'exprimer par la formule du premier ordre $\exists y\ Q(x, y)$. Cette traduction n'est en fait pas entièrement exacte puisque $\exists y\ Q(x, y)$ peut être prouvé sans exhiber une solution pour y, alors qu'il est peut être impossible de trouver une telle solution. Mais nous ne pouvons pas trouver de formulation plus adéquate puisque les préconditions sont des formules du CP1 qui ne sont donc pas nécessairement prouvées de manière constructive.

1.2.2. Les postconditions.

Les postconditions sont plus difficiles à caractériser : nous aimerions pouvoir établir des critères syntaxiques permettant de détecter les formules qui caractérisent effectivement des résultats. Par exemple les postconditions **vrai**, **faux**, x=x ne nous intéresse pas puisque les programmes solutions sont quelconques. D'autre part, ce qui est plus important, nous aimerions pouvoir décomposer la postcondition afin de faciliter le processus de synthèse. Cette décomposition devrait permettre de retrouver les sous-formules de la postcondition qui établissent un lien direct entre un sous-ensemble des données et un sous-ensemble des résultats et qui devront donc être traitées de manière indissociables. Par exemple si une postcondition est de la forme $A(x_1, var) \wedge B(x_2, var)$, x_1 et x_2 étant les données et var le résultat à construire, nous pouvons chercher à caractériser les var vérifiant la relation $A(x_1, var)$, puis les var vérifiant la relation $B(x_2, var)$ et ensuite prendre l'intersection de ces 2 ensembles de solutions. Par contre si la postcondition est de la forme $A(x_1, var) \Rightarrow B(x_2, var)$ nous ne voudrons pas séparer cette formule, l'implication décrivant généralement un lien de cause à effet entre l'antécédent et le conséquent. La relation entre données et résultats peut être décrite par trois composants : une propriété des données, une propriété des résultats et une formule établissant ce que nous appelons *un lien effectif* entre données et résultats. Une postcondition aura donc, au plus haut niveau, la forme générale :

$$\vee_{(i=1, j)}\ (C_i(x) \wedge L_i(x, var) \wedge R_i(var))$$

$\vee_{(i=1, j)}$ désigne la disjonction de j formules. C_i est une propriété du k-uplet x de données, R_i est une propriété du n-uplet var de résultats et L_i est une relation liant x et var. Définir au plus haut niveau une postcondition comme une disjonction de telles relations permet de définir le lien attendu entre données et résultats par cas suivant les propriétés C_i des données.

Les contraintes sur les postconditions sont les suivantes :

1 - La postcondition ne peut pas être vide et doit contenir au moins une occurrence de chaque variable var_i représentant un résultat. Ceci signifie qu'elle ne peut pas être réduite uniquement à des propriétés de x.

2 - C_i peut être vide.

3 - R_i peut être vide.

4 - L_i peut être vide. Si c'est le cas les résultats var_i ne dépendent pas de x. A cause de la contrainte 1, il existe alors nécessairement une propriété R_i des var_i et donc toute fonction vérifiant cette propriété convient, entre autres les fonctions constantes. Par exemple, la postcondition réduite à var > 2 peut être prouvée par la fonction f(x) ← 3. Ce cas, qui ne semble pas très intéressant, est à considérer car il se rencontre dans les énoncés engendrés par le système, notamment en fin de synthèse, lorsque les

sous-problèmes qu'il reste à résoudre sont triviaux.

5 - Toutes les variables, autres que les données et les résultats, apparaissant dans les postconditions doivent être liées, c'est à dire doivent se trouver sous la portée d'un quantificateur, universel ou existentiel.

Les propriétés C_i et R_i sont respectivement des propriétés des données ou des résultats. Ce sont donc des formules dont les seules variables libres appartiennent soit au k-uplet de données soit au n-uplet de résultats. Il reste à décrire les formules établissant *un lien effectif* entre les données et les résultats. Pour cela nous définissons une fonction, appelée **relié**, qui décrit l'existence d'une causalité entre des objets, suivant la structure syntaxique des formules. Cette fonction est basée sur la signification intuitive associée aux constructeurs de formules du premier ordre. Deux formes F_1 et F_2 différentes syntaxiquement mais équivalentes sémantiquement pourront être telles que **relié**(F_1) ≠ **relié**(F_2).

Définition de la fonction relié :

Nous notons var(F) l'ensemble des variables libres d'un terme ou d'une formule bien formée F. Nous définissons la fonction **relié**(F), pour une formule bien formée F, qui calcule l'ensemble des variables entre lesquelles une causalité est établie dans F. La fonction **relié**(F) retourne donc un ensemble de couples (e_1, e_2) où e_1 et e_2 sont des ensembles de variables. **relié** est définie inductivement sur la structure des formules bien formées de la manière suivante :

– Si F est de la forme $t_1 = t_2$ alors **relié**(F) = { (var(t_1), var (t_2)) }

– Si F est de la forme $P(t_1, \ldots, t_n)$ où P est un symbole de prédicat alors **relié**(F) = {(e, e)} où $e = \cup_{(i=1, n)}$ **var** (t_i)

– Si F est de la forme $\neg A$, $\forall x$ A ou $\exists x$ A alors **relié**(F) = **relié**(A)

– Si F est de la forme $A \wedge B$ alors **relié**(F) = **relié**(A) $\cup$ **relié**(B)

– Si F est de la forme $A \vee B$ alors **relié**(F) = **relié**(A) $\cap$ **relié**(B)

– Si F est de la forme $A \Rightarrow B$ ou $A \equiv B$ alors **relié**(F) = **relié**(A) $\cup$ **relié**(B) $\cup$ { (var(A), var (B))}

On définit la fermeture symétrique et transitive de ces ensembles. Pour toute formule F :

- $(e_1, e_2) \in$ **relié**(F) $\Rightarrow (e_2, e_1) \in$ **relié**(F)

- $(e_1, e_2) \in$ **relié**(F) $\wedge (e_3, e_4) \in$ **relié**(F) $\wedge \exists$ var (var $\in e_2 \wedge$ var $\in e_3) \Rightarrow (e_1, e_4) \in$ **relié**(F).

Une formule F établit un lien entre la donnée x et le résultat var si et seulement si il existe (e_1, e_2) élément de **relié**(F) tel que x est élément de e_1 et y est élément de e_2.

fin-définitions.

L'analyse du langage des énoncés a été implantée dans le cadre d'un projet ENSIMAG [DoV 87]. Dans cet analyseur, le profil le plus général des fonctions à construire est inféré à partir des définitions des opérateurs apparaissant dans l'énoncé. Se placer dans un typage polymorphe permet, comme nous l'avons dit, de traiter les énoncés et les preuves au meilleur niveau d'abstraction. Le type inféré doit donc être le plus général dans l'ordre défini sur les types. L'algorithme utilisé est proche de l'agorithme W proposé par Milner et utilisé dans le langage ML [Mil 78]. L'utilisateur a néanmoins la possibilité de préciser les types pour lesquels il attend un programme. Ceux-ci doivent être pris en compte, le programme construit pour des types particuliers pouvant être différent de celui construit pour les types les plus généraux si des propriétés spécifiques à ces théories particulières existent et mènent à des preuves différentes.

Dans cette partie nous avons décrit la forme des énoncés attendus. Néanmoins il est difficile d'écrire de tels énoncés, le CP1 étant un formalisme très puissant mais très dense. Il serait donc nécessaire de prévoir des outils d'aide à la mise au point d'énoncés. Par exemple il serait intéressant de pouvoir les tester pour des valeurs particulières de données et de résultats. D'autre part, une des difficultés principales est d'arriver à préciser suffisamment les énoncés pour qu'ils caractérisent effectivement et uniquement les fonctions voulues. En effet, lorsque l'énoncé n'est pas déterministe, le choix du résultat construit sera fait par le système de preuve. Il serait alors intéressant d'admettre la possibilité de compléter les énoncés par des couples d'exemples qui décriraient, pour des valeurs des données, les valeurs attendues des résultats. Lorsque le système serait amené à choisir parmi l'ensemble des solutions possibles, il s'aiderait de ces exemples. Ceci ne permettrait pas de lever entièrement les ambiguités mais serait, à notre avis, une extension intéressante. Dans le cadre d'une implantation il est nécessaire de pouvoir proposer de telles aides.

2. PREUVE DES ENONCES.

Soit E un énoncé quelconque de la forme :

préc : $P(x)$
post : $Q(x, var_1, \ldots, var_n)$

Nous n'indiquons plus les indications de type, celles-ci n'étant pas directement nécessaire pour le niveau de preuve auquel nous nous intéressons. Les variables var_i représentent les résultats des n fonctions f_i. Nous dirons qu'un énoncé est vrai si il existe effectivement des fonctions calculables vérifiant la postcondition de cet énoncé, pour toute donnée vérifiant la précondition. Le but est donc de construire un programme $\mathcal{P}$, contenant des définitions des fonctions $f_1, \ldots, f_n$, tel que $\mathcal{P} : P(x) \rightarrow Q(x, f_1(x), \ldots, f_n(x))$ est un théorème de notre logique. Nous avons donc :

E est vrai

si et seulement si

$\exists\ \mathcal{P}$ **tel que** $(P(x) \rightarrow Q(x, f_1(x), \ldots, f_n(x)))$
est un théorème de notre logique

Le système décrivant la sémantique axiomatique de LF fournit directement un cadre pour la preuve constructive d'énoncés, en ne s'intéressant qu'à la partie droite des formules de correction. Montrons, en effet, que si la formule $P(x) \rightarrow Q(x, f(x))$ est démontrable alors on a nécessairement construit un programme $\mathcal{P}$ contenant une définition de f et tel que $\mathcal{P} : P(x) \rightarrow Q(x, f(x))$ est un théorème.

- Soit $P(x) \rightarrow Q(x, B(x))$ est un axiome associé à la fonction de base B. $\{ f(x) \leftarrow B(x) \}$ est alors un programme contenant une définition de la fonction f.

- Soit $P(x) \rightarrow Q(x, f(x))$ correspond à la conclusion d'une des règles relatives à la conditionnelle, à la composition fonctionnelle ou à la récursion. Une définition de f est alors de la forme $f(x) \leftarrow F(f_1(x), \ldots, f_n(x))$, F étant une de ces 3 constructions. On suppose, par hypothèse d'induction, que la preuve des prémisses de la règle utilisée permet d'exhiber des programmes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ définissant les fonctions $f_1, \ldots, f_n$. $\mathcal{P}_1 \cup \ldots \cup \mathcal{P}_n \cup \{ f(x) \leftarrow F(f_1(x), \ldots, f_n(x)) \}$ est bien un programme contenant une définition de f, après renommage si nécessaire.

- Soit $P(x) \rightarrow Q(x, f(x))$ est déductible de $P_1(x) \rightarrow Q_1(x, f_1(x))$ par la règle de la conséquence. Par hypothèse d'induction, on peut supposer que la preuve de cette formule fournit un programme $\mathcal{P}$ définissant f_1. $\mathcal{P} \cup \{ f(x) \leftarrow f_1(x) \}$ est bien un programme définissant complètement f.

La correction et la complétude, pour la preuve d'énoncés, découle de la correction et de la complétude du système définissant la sémantique de LF. En effet, le système sera correct si la preuve d'un énoncé fournit bien un programme vérifiant cet énoncé. Cette contrainte est garantie par la preuve de correction faite précédemment. Il sera complet si et seulement si, lorsqu'il existe un programme $\mathcal{P}$ vérifiant un énoncé E, ce programme peut être trouvé. Or, si $\mathcal{P}$ existe, on peut trouver une preuve que E est vérifié par $\mathcal{P}$ en raison de la complétude de la définition de la sémantique. E peut donc être prouvé par cette preuve et le programme produit sera $\mathcal{P}$. Cependant l'existence d'un programme est un problème semi-décidable. La semi-décidabilité découle d'une part de la semi-décidabilité du CP1 et d'autre part de la semi-décidabilité inhérente au système de preuve : il n'existe pas de processus fini permettant de prouver ou de réfuter un énoncé.

2.1. Le système de preuves.

Dans le cadre de la synthèse de programmes, on dispose d'une formule à prouver, l'énoncé E, et on désire construire une preuve de cette formule. Ceci signifie généralement que E peut être déduit d'un axiome ou d'une conclusion d'une des règles relatives aux constructions de LF en utilisant la règle de la conséquence. En effet, l'énoncé initial ne correspond pas nécessairement à une conclusion de ces règles, ces conclusions ayant une forme syntaxique bien particulière. Pour obtenir le système de preuve des énoncés nous incluons systématiquement la règle de la conséquence à chaque axiome et à chaque règle. Les règles deviennent de la forme :

$$\frac{\begin{array}{c} P_i(x_i) \rightarrow Q_i(x_i, f_i(x_i)) \\ P(x) \Rightarrow PREC(x) \\ P(x) \wedge POST(x, y) \Rightarrow Q(x, y) \end{array}}{P(x) \rightarrow Q(x, f(x))}$$

PREC et POST sont les formules décrivant respectivement la précondition et la postcondition associées à chaque construction. Ces formules sont constituées à partir des préconditions P_i et des postconditions Q_i des énoncés des sous-problèmes, qui, lors de la preuve d'énoncés, ne sont pas connues. Les P_i et les Q_i doivent donc être déterminées à partir des deux implications précédentes. La difficulté de la mise en place de la preuve est donc double :

- Il faut choisir, pour un énoncé donné, la preuve à mettre en place, c'est à dire la règle d'inférence à utiliser à chaque pas. En effet nous ne voulons pas, et ne pouvons d'ailleurs pas toujours, utiliser des critères syntaxiques pour déterminer la règle à appliquer.

- Une fois cette règle choisie, il faut trouver les énoncés caractérisant les fonctions paramètres de la construction retenue, ceci à partir des propriétés :

(a) $P(x) \Rightarrow PREC(x)$

(b) $P(x) \wedge POST(x, y) \Rightarrow Q(x, y)$

La recherche des énoncés des sous-problèmes est un problème du second ordre. De manière générale, nous devons trouver des relations $B_1, \ldots, B_n$ telles que $A[B_1, \ldots, B_n]$ est un théorème, $A[B_1, \ldots, B_n]$ désigne une formule quelconque dépendante des relations $B_1, \ldots, B_n$. Cela revient à prouver constructivement une formule de la forme :

$$\exists X_1, \ldots, X_n \; A[X_1, \ldots, X_n]$$

$X_1, \ldots, X_n$ sont des variables de prédicats. Si $B_1, \ldots, B_n$ sont des formules telles que $A[B_1, \ldots, B_n]$ est vraie, nous disons que $B_1, \ldots, B_n$ sont des solutions. Il peut exister plusieurs solutions et nous allons chercher à caractériser celles qui nous intéressent le plus dans le cadre de la synthèse de programmes.

• Caractérisation des meilleures solutions de (a) :

La formule (a) décrit le fait que le domaine de définition de la fonction f à construire doit être inclus dans le domaine de définition de la construction $F(f_1(x), \ldots, f_n(x))$ choisie, ce dernier domaine étant caractérisé par la formule PREC(x). La construction des f_i ne nous intéresse en fait que pour les plus petits domaines tels que cette inclusion est vraie. Les meilleures préconditions $P_1, \ldots, P_n$ sont donc des solutions de la formule (a) telles que toute autre solution à cette implication est impliquée par $(P_1, \ldots, P_n)$.

Propriété 1 :

Les meilleures solutions à l'implication $P(x) \Rightarrow PREC(x)$ sont les $P_1, \ldots, P_n$ telles que :

- $P(x) \Rightarrow PREC[P_1, \ldots, P_n](x)$

- $\forall X_1, \ldots, X_n \; (((x)P \Rightarrow PREC[X_1, \ldots, X_n](x)) \Rightarrow (P_1(x) \Rightarrow X_1(x) \wedge \ldots \wedge P_n(x) \Rightarrow X_n(x)))$

fin-propriété.

Ces solutions pourront être calculées de manière systématique à partir de la précondition initiale P, ceci pour chaque règle.

• Caractérisation des meilleures solutions de (b) :

La formule (b) décrit le fait que, pour tout x ayant la propriété P, les résultats calculés par la construction $F(f_1(x), \ldots, f_n(x))$ doivent aussi être des résultats pour le problème initial. Il est en fait très délicat de décrire les meilleures postconditions des sous-problème. Smith [Smi 82] propose les critères intuitifs suivants : les meilleures solutions doivent être

- *facilement calculables* ;
- *aussi simples que possible* ;
- *aussi faibles que possible* (Smith définit les solutions les plus faibles par celles préservant l'équivalence avec la formule B à prouver, sous l'hypothèse A).

Ce qui est difficile à traduire c'est le qualificatif aussi ... que possible ! En fait la postcondition initiale Q(x, y) peut être non déterministe, c'est à dire qu'il peut exister, pour un x donné, plusieurs valeurs de y vérifiant la relation Q(x, y). Puisque nous nous intéressons à la recherche d'une de ces solutions, nous ne cherchons pas nécessairement à préserver l'équivalence entre les formules POST(x, y) et Q(x, y). Par exemple, le fait de retenir une règle d'inférence particulière peut limiter les solutions possibles. C'est notamment le cas lorsque on choisit un axiome qui ne définit qu'une seule fonction. Le dernier critère proposé par Smith ne se justifie donc pas nécessairement.

Exemple :

Supposons que nous voulons construire une fonction f vérifiant la spécification (x≠nil, dans(f(x), x)), c'est à dire une fonction qui extrait un élément quelconque d'une liste x. f peut être la fonction car(x) ou bien la fonction qui renvoie le dernier élément de x. Dans ce dernier cas le programme construit sera de la forme :

$$f(x) \leftarrow \textbf{si } cdr(x)=nil \textbf{ alors } car(x) \textbf{ sinon } g(f(cdr(x)))$$

Pour déterminer une postcondition de la fonction g nous devrons alors, par la règle relative à la récursion et la règle de la conséquence, trouver une relation Q_g telle que :

$$x \neq nil \wedge cdr(x) \neq nil \wedge dans(y_1, cdr(x)) \wedge Q_g(y_1, x, y) \Rightarrow dans(y, x)$$

Nous sommes intéressés par la solution $y_1=y$ mais non pas par la solution $y_1=y \vee y=car(x)$, qui elle préserverait l'équivalence. Cette dernière solution ne nous convient pas puisque la relation y=car(x) n'est pas liée à un calcul récursif du résultat. Nous n'imposons donc pas nécessairement l'équivalence entre les formules POST(x, y) et Q(x, y). Les solutions obtenues dépendront de la propriété qu'elles doivent vérifiées et des connaissances du système qui sont supposées être simplificatrices.

fin-exemple.

Néanmoins, si plusieurs solutions sont possibles, on désire choisir celle qui restreint le moins l'ensemble des solutions. Ceci permet de garantir qu'on ne perd pas d'informations pertinentes et donc qu'on a des " chances " de pouvoir construire une preuve, si il en existe une, à partir de la règle d'inférence utilisée.

Propriété 2 :

Si $A_1, \ldots, A_n$ et $B_1, \ldots, B_n$ sont deux ensembles de solutions acceptables et si $POST[B_1, \ldots, B_n] \Rightarrow POST[A_1, \ldots, A_n]$ les solutions $A_1, \ldots, A_n$ sont préférées.

fin-propriété.

Cette contrainte exclut, dans la mesure du possible, les postconditions réduites à la constante **faux** qui est une solution toujours possible. Imposer que les solutions ne préservent pas nécessairement l'équivalence mais vérifient la propriété ci-dessus revient à prendre en compte les deux critères de Smith *P doit être la plus faible possible* mais aussi *facilement calculable*. Smith impose aussi que l'hypothèse obtenue soit *aussi simple que possible* : une telle condition est difficilement caractérisable. Pour notre part, nous imposerons simplement que les hypothèses découlant du choix d'une preuve soient effectivement utilisées dans la recherche de solutions. Par cela nous cherchons à garantir que la preuve retenue est pertinente vis à vis du domaine du problème. Ceci repose sur le fait que les théorèmes connus ou déduits sont supposés simplificateurs. Illustrons ceci sur un exemple.

Exemple.

Soit l'énoncé :

profil : $sup = f(x, y : entier) \rightarrow entier$
préc : vrai
post : $(sup=x \vee sup=y) \wedge sup \geq x \wedge sup \geq y$

Supposons que nous voulons construire un programme conditionnel de la forme :

$f(x, y) \leftarrow$ **si** $x \geq y$ **alors** $f_1(x, y)$ **sinon** $f_2(x, y)$

Nous devons alors trouver les préconditions P_1 et P_2 et les postconditions Q_1 et Q_2 des énoncés des fonctions f_1 et f_2. D'après la règle de la conditionnelle et la règle de la conséquence, ces formules doivent être telles que :

(a) $vrai \Rightarrow (P_1(x, y) \wedge x \geq y) \vee (P_2(x, y) \wedge y > x)$

(b) $Q_1(x, y, sup) \wedge x \geq y \Rightarrow (sup=x \vee sup=y) \wedge sup \geq x \wedge sup \geq y$

(c) $Q_2(x, y, sup) \wedge \neg(x \geq y) \Rightarrow (sup=x \vee sup=y) \wedge sup \geq x \wedge sup \geq y$

• Recherche des préconditions :

D'après la manière dont nous les avons caractérisées, les meilleures solutions pour P_1 et P_2 sont respectivement $x \geq y$ et $y > x$. En effet, $(x \geq y) \vee (y > x)$ est toujours vrai et donc ces deux formules sont bien des solutions de (a). D'autre part, quelques soient P_1 et P_2 vérifiant la formule (a) on a nécessairement $x \geq y \Rightarrow P_1(x, y)$ et $y > x \Rightarrow P_2(x, y)$ puisque la disjonction de la formule (a) est exclusive.

• **Recherche des postconditions :**

En ce qui concerne les postconditions Q_1 et Q_2, nous pouvons par exemple choisir la postcondition du problème initial. Mais cette solution n'est pas intéressante puisque les fonctions f_1 et f_2 seraient alors aussi compliquées à construire que la fonction initiale f. Le choix d'une construction conditionnelle présuppose intuitivement que la condition choisie permette de simplifier le problème. Nous désirons donc que l'hypothèse $x \geq y$ intervienne effectivement lors de la recherche de Q_1 et Q_2. Par exemple pour trouver Q_1 nous pouvons procéder de la manière suivante :

– La postcondition initiale est équivalente, en appliquant la substitutivité de l'égalité, à la disjonction $(sup=x \wedge x \geq x \wedge x \geq y) \vee (sup=y \wedge y \geq x \wedge y \geq y)$.

- Nous pouvons alors simplifier cette formule à l'aide de l'hypothèse $x \geq y$ et obtenir la solution :

$$(sup=x \wedge x \geq x) \vee (sup=y \wedge y \geq x \wedge y \geq y)$$

Cette solution est acceptable puisque l'hypothèse $x \geq y$ a été nécessaire. Mais nous pouvons faire mieux. En effet $y \geq x$ peut se simplifier, sous l'hypothèse $x \geq y$, en $y=x$. Une autre solution est donc :

$$(sup=x \wedge x \geq x) \vee (sup=y \wedge y = x \wedge y \geq y)$$

Et même, en simplifiant par les axiomes $x \geq x$ et $y \geq y$ nous obtenons $(sup=x) \vee (sup=y \wedge y = x)$, ce qui se simplifie encore en $sup = x$.

Nous pouvons procéder de même pour trouver Q_2 en utilisant l'hypothèse $\neg(x \geq y)$. Vis à vis de l'exemple, nous préférerons donc les instances de Q_1 et Q_2 $(sup = x)$ et $(sup = y)$.

fin-exemple.

Nous pouvons donc guider le processus de recherche d'hypothèses en imposant que certaines hypothèses découlant de la preuve soient utilisées. Par contre, en ce qui concerne le problème de la simplification, nous ne disposons pas de guide pour décider si une formule est sous la forme la plus simple possible ou si il est possible de la simplifier encore.

Nous présentons maintenant les règles d'inférence relatives à la preuve du langage d'énoncés. Les préconditions des sous-problèmes vont être détectées statiquement car elles ne dépendent que de la précondition initiale, par contre les postconditions ne pourront être élaborées que lors de la construction de chaque preuve, en fonction de l'énoncé initial et des propriétés particulières sur le domaine du problème.

2.1.1. Règle d'introduction d'une fonction de base.

$$P_1(x) \rightarrow Q_1(x, f0(x))$$

$$P(x) \Rightarrow P_1(x)$$

$$P(x) \wedge Q1(x, y) \Rightarrow Q(x, y)$$

cons _____

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

Une définition possible de la fonction f est alors $f(x) \leftarrow f_0(x)$.

2.1.2. Règle de simplification des arguments.

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

arg _____

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

Lors de la construction des énoncés des sous-problèmes, les données de ces sous-problèmes dépendent des données initiales et sont déduites de manière systématique, suivant la construction retenue. Cette règle permet de simplifier les données en enlevant les variables qui n'apparaissent pas dans la postcondition, c'est à dire les données dont les résultats ne dépendent pas.

2.1.3. Règle d'introduction d'une Conditionnelle.

Par définition de la sémantique de la conditionnelle $f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$ **sinon** $f_2(x)$ nous savons que les énoncés (P_0, Q_0), (P_1, Q_1) et (P_2, Q_2) doivent être tels que :

(a) $P(x) \Rightarrow P_0(x) \wedge ((P_1(x) \wedge Q_0(x) \vee P_2(x) \wedge \neg Q_0(x)))$

(b) $P(x) \wedge ((Q_0(x) \wedge Q_1(x, y) \vee \neg Q_0(x) \wedge Q_2(x, y))) \Rightarrow Q(x, y)$

• Recherche des préconditions :

$P_0(x)$ est la précondition initiale $P(x)$, $P_1(x)$ est la formule $P(x) \wedge Q_0(x)$ et $P_0(x)$ et $P_2(x)$ sont instanciées par

$P(x) \wedge \neg Q_0(x)$. Ces solutions sont les meilleures possibles, c'est à dire qu'elles vérifient la propriété que nous avons établie. En effet quelques soient P_0, P_1, Q_0 et P_2 vérifiant (a) nous avons bien :

$$P(x) \Rightarrow P_0(x)$$
$$P(x) \wedge Q_0(x) \Rightarrow P_1(x)$$
$$P(x) \wedge \neg Q_0(x) \Rightarrow P_2(x)$$

- La règle relative à l'introduction d'une conditionnelle est donc :

$$P(x) \rightarrow (C(x)=\text{vrai}) \equiv Q_0(x)$$

$$P(x) \wedge Q_0(x) \rightarrow Q_1(x, f_1(x))$$

$$P(x) \wedge \neg Q_0(x) \rightarrow Q_2(x, f_2(x))$$

$$P(x) \wedge Q_0(x) \wedge Q_1(x, y) \Rightarrow Q(x, y)$$

$$P(x) \wedge \neg Q_0(x) \wedge Q_2(x, y) \Rightarrow Q(x, y)$$

cond _____

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

La définition engendrée est $f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$ **sinon** $f_2(x)$.

2.1.4. Règle d'introduction d'une Composition Fonctionnelle.

Par la règle décrivant la sémantique d'une composition fonctionnelle de la forme $f(x) \leftarrow g(f_1(x), ..., f_n(x))$ nous devons avoir :

$$\text{(a)}\ P(x) \Rightarrow (\Pi_{(i=1, n)} P_i(x) \wedge \forall y_1, ..., y_n (\Pi_{(i=1, n)} Q_i(x, y_i) \Rightarrow P_g(y_1, ..., y_n)))$$

$$\text{(b)}\ P(x) \wedge \exists y_1, ..., y_n (\Pi_{(i=1, n)} Q_i(x, y_i) \wedge Q_g(y_1, ..., y_n, z)) \Rightarrow Q(x, z)$$

(P_i, Q_i) sont les énoncés des n fonctions f_i et (P_g, Q_g) est l'énoncé de la fonction de recomposition g. Nous simplifierons la seconde formule par l'équivalence $(\exists x\ A(x) \Rightarrow B) \equiv (\forall x\ (A(x) \Rightarrow B))$ qui est valide lorsque B ne dépend pas de x, ce qui est le cas.

• **Recherche des préconditions :**

Les fonctions f_i doivent être définies pour tous les objets x vérifiant P(x), une solution pour les préconditions P_i est donc la précondition initiale P. La fonction g doit être définie pour tous les n-uplets y qui représente les résultats $(f_1(x), \ldots f_n(x))$ lorsque P(x) est vrai. La précondition de g est donc l'ensemble des n-uplets $(y_1, \ldots, y_n)$ tel que :

$$\exists x \, (P(x) \wedge \Pi_{(i=1, n)} \, Q_i(x, y_i))$$

Ces solutions sont les meilleures possibles. Elles vérifient bien la formule **(a)** et nous avons, quelques soient des solutions P_i et P_g vérifiant **(a)**, :

$$P(x) \Rightarrow P_i(x)$$
$$\exists x \, (P(x) \wedge \Pi_{(i=1, n)} \, Q_i(x, y_i)) \Rightarrow P_g(y_1, \ldots, y_n)$$

• La règle d'introduction d'une composition fonctionnelle est donc :

$$P(x) \rightarrow Q_i(x, f_i(x))$$

$$\exists x \, (P(x) \wedge \Pi_{(i=1, n)} \, Q_i(x, y_i)) \rightarrow Q_g(y_1, \ldots, y_n, g(y_1, \ldots, y_n))$$

$$P(x) \wedge \Pi_{(i=1, n)} \, Q_i(x, y_i) \wedge Q_g(y_1, \ldots, y_n, z) \Rightarrow Q(x, z)$$

comp

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

La définition de f sous-jacente à cette règle est $f(x) \leftarrow g(f_1(x), \ldots, f_n(x))$.

2.1.5. Règle d'introduction d'une Récursion.

Rappelons que la règle de la récursion permet de construire une définition de la forme :

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

• **Recherche des préconditions :**

- D'après la règle d'introduction d'une conditionnelle, la précondition de C est la précondition initiale P(x), celle de f_1 est $P(x) \wedge Q_0(x)$ et celles des fonctions $h_{i,j}$ sont $P(x) \wedge \neg Q_0(x)$. Rappelons que $Q_0(x)$ désigne la postcondition de la

fonction C. D'autre part, d'après la règle de la composition fonctionnelle, les préconditions des appels récursifs sont :

$$\exists x \ (P(x) \wedge \neg Q_0(x) \wedge \Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}))$$

Or la précondition des appels récursifs est, par définition, $P(y_{i,1}, \ldots, y_{i,k})$. Par la règle de la conséquence, nous devons donc avoir :

$$\exists x \ (P(x) \wedge \neg Q_0(x) \wedge \Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j})) \Rightarrow P(y_{i,1}, \ldots, y_{i,k})$$

Cette formule se simplifie en $P(x) \wedge \neg Q_0(x) \wedge \Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}) \Rightarrow P(y_{i,1}, \ldots, y_{i,k})$. Ce sera donc une prémisse de la règle de la récursion dont la preuve nécessitera l'utilisation de propriétés sur le domaine du problème.

- La précondition de g est l'ensemble des couples (x, z) tels que x vérifie $P(x) \wedge \neg Q_0(x)$ et tels que z peut être le résultat des $f(h_{i,1}(x), \ldots, h_{n,k}(x))$, c'est à dire :

$$P(x) \wedge \neg Q_0(x) \wedge \exists x, y\, (P(x) \wedge \neg Q_0(x) \wedge \Pi_{(i=1,\,n\,;\,j=1,\,k)}\, (Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i)))$$

y_i désigne le k-uplet $y_{i,1}, \ldots, y_{i,k}$ et y le n-uplet $(y_1, \ldots, y_n)$. Cette formule se simplifie en :

$$P(x) \wedge \neg Q_0(x) \wedge \exists y\, (\Pi_{(i=1,\,n\,;\,j=1,\,k)}\, (Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i)))$$

• La règle de l'introduction d'une récursion est donc :

$$P(x) \rightarrow (C(x)=\text{vrai}) \equiv Q_0(x)$$

$$P(x) \wedge Q_0(x) \rightarrow Q_1(x, f_1(x))$$

$$P(x) \wedge \neg Q_0(x) \rightarrow Q_{i,j}(x, h_{i,j}(x))$$

$$P(x) \wedge \neg Q_0(x) \wedge \exists y\, (\Pi_{(i=1,\,n\,;\,j=1,\,k)}\, (Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i))) \rightarrow Q_g(z, x, g(z, x))$$

$$P(x) \wedge \neg Q_0(x) \wedge \Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}) \Rightarrow P(y_i)$$

$$P(x) \wedge Q_0(x) \wedge Q_1(x, w) \Rightarrow Q(x, w)$$

$$P(x) \wedge \neg Q_0(x) \wedge \Pi_{(i=1,\,n)}\, (\Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}) \wedge Q(y_i, z_i)) \wedge Q_g(z, x, w) \Rightarrow Q(x, w)$$

rec ———————————————————————————————————

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

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

Rappelons qu'une définition récursive n'est correcte que si il existe un ordre bien fondé tel que les arguments des appels récursifs soit inférieurs aux arguments initiaux. Pour garantir l'arrêt il faudra donc trouver un ordre bien fondé **inf** tel que :

$$P(x) \wedge \neg Q_0(x) \wedge \Pi_{(j=1,\,k)}\, Q_{i,j}(x, y_{i,j}) \Rightarrow \mathbf{inf}((y_{i,1}, \ldots, y_{i,k}), x)$$

En dehors du fait que la relation **inf** doit être trouvée, cette formule est du second ordre puisque **inf** est une formule qui doit vérifier la propriété d'être un ordre bien fondé.

2.2. Synthèse de la fonction maximum.

Nous présentons maintenant la synthèse d'un algorithme calculant le maximum d'une liste d'entiers. L'énoncé retenu est le même que celui que nous avons toujours utilisé, c'est à dire :

profil : max = **max**(x : liste(entier)) → entier
préc : x≠nil
post : max ∈ x ∧ SUP(max, x)

Nous devons donc construire une preuve de la formule x≠nil → max ∈ x ∧ SUP(max, x) dans notre système. Nous ne nous préoccupons pas ici du problème du choix de la preuve mais nous donnons néanmoins une idée intuitive des stratégies qui peuvent être employées.

• **Etape 1** : Synthèse de la fonction maximum.

Nous choisissons de construire un programme récursif basé sur la décomposition des listes en terme de nil et cons. La règle utilisée est la règle de la récursion dans le cas d'un appel unique. Par le choix de la décomposition la condition C(x) du cas de base est instanciée par cdr(x)=nil et la fonction h(x), calculant l'argument de la récursion, par cdr(x). Ces deux fonctions sont des fonctions de base, l'existence de C et h est donc prouvée trivialement. La définition engendrée est alors :

max(x) ← **si** cdr(x)=nil **alors** f_1(x) **sinon** g(max(cdr(x)), x)

D'après la règle de la récursion, les énoncés des sous-problèmes correspondant aux fonctions f_1 et g sont respectivement :

profil : max = f_1(x : liste(entier)) → entier
préc : x≠nil ∧ cdr(x)=nil
post : Q_1(x, max)

profil : max = g(z : entier, x : liste(entier)) → liste(entier)
préc : x≠nil ∧ cdr(x)≠nil ∧ z∈cdr(x) ∧ SUP(z, cdr(x))
post : Q_g(z, x, max)

Et les relations Q_1 et Q_g doivent vérifier les propriétés suivantes :

1 - $x \neq nil \wedge cdr(x) \neq nil \Rightarrow cdr(x) \neq nil$

2 - $x \neq nil \wedge cdr(x) = nil \wedge Q_1(x, max) \Rightarrow max \in x \wedge SUP(max, x)$

3 - $x \neq nil \wedge cdr(x) \neq nil \wedge z \in cdr(x) \wedge SUP(z, cdr(x)) \wedge Q_g(z, x, max) \Rightarrow max \in x \wedge SUP(max, x)$

1 décrit le fait que la précondition de l'appel récursif est vérifiée dans la partie sinon. Cette formule est trivialement vraie. Pour obtenir des instances de Q_1 et Q_g, vérifiant les formules 2 et 3, il faut disposer de théorèmes permettant d'une part de simplifier la formule $max \in x \wedge SUP(max, x)$ dans le cas $cdr(x)=nil$ et d'autre part d'un théorème permettant de prouver $max \in x \wedge SUP(max, x)$ en sachant qu'il existe z tel que $z \in cdr(x) \wedge SUP(z, cdr(x))$. Si nous disposons des définitions récursives des prédicats $\in$ et SUP suivantes :

$max \in x \quad \leftarrow$ **si** $x=nil$ **alors faux sinon** $max=car(x) \vee max \in cdr(x)$
$SUP(max, x) \leftarrow$ **si** $x=nil$ **alors vrai sinon** $max \geq car(x) \wedge SUP(max, cdr(x))$

nous avons alors les deux théorèmes :

- $cdr(x)=nil \wedge max=car(x) \Rightarrow max \in x \wedge SUP(max, x)$

- $(max=car(x) \wedge SUP(max, cdr(x))) \vee (max \in cdr(x) \wedge max \geq car(x) \wedge SUP(max, cdr(x))) \Rightarrow max \in x \wedge SUP(max, x)$

Le premier permet de prouver la formule 2 en instanciant $Q_1(x, max)$ par la relation **max=car(x)**. Le second permet de prouver la formule 3. L'hypothèse d'induction découlant de la forme du programme à construire est ($z \in$ **cdr(x)** $\wedge$ **SUP(z, cdr(x))**) et celle du théorème est (**max** $\in$ **cdr(x)** $\wedge$ **SUP(max, cdr(x))**). Nous assimilons ces deux hypothèses qui portent sur la même induction en terme de x. Ceci entraine l'instanciation de z par **max** qui se traduit, en terme relationnel, par l'égalité **max=z**. La solution obtenue pour Q_g est :

(max=car(x) $\wedge$ SUP(max, cdr(x))) $\vee$ (max=z $\wedge$ max $\geq$ car(x))

Et les énoncés de f_1 et g sont donc :

profil : max = f_1(x : liste(entier)) $\rightarrow$ entier
préc : $x \neq nil \wedge cdr(x)=nil$
post : $max=car(x)$

profil : max = g(z : entier, x : liste(entier)) $\rightarrow$ liste(entier)
préc : $x \neq nil \wedge cdr(x) \neq nil \wedge z \in cdr(x) \wedge SUP(z, cdr(x))$
post : $(max=car(x) \wedge SUP(max, cdr(x))) \vee (max=z \wedge max \geq car(x))$

L'énoncé initial pourra donc être prouvé par la contruction max(x) ← si cdr(x)=nil alors f_1(x) sinon g(max(cdr(x)), x) à condition de pouvoir construire f_1 et g à partir de ces deux énoncés.

• **Etape 2 :** Synthèse de f_1.

La fonction f_1 peut être construite à partir de la fonction de base **car**, dont l'axiome associé est x≠nil → car(x)=car(x). D'après la règle relativement à la reconnaissance d'une fonction de base il suffit de montrer :

1 - $x \neq nil \wedge cdr(x)=nil \Rightarrow x \neq nil$

2 - $x \neq nil \wedge cdr(x)=nil \wedge max=car(x) \Rightarrow max=car(x)$

La première formule garantit que car est bien définie pour la précondition de f_1 et la seconde formule garantit que car est bien une solution possible pour f_1. Ces deux formules sont vraies, l'existence de f_1 est donc prouvée par la construction f_1(x) ← car(x).

• **Etape 3 :** Synthèse de g.

Nous choisissons de construire g sous forme d'une conditionnelle. Ce choix peut être induit par le fait que la postcondition est une disjonction, ce qui correspond à la forme générale de la postcondition d'une conditionnelle. g sera donc définie par :

g(z, x) ← si C(z, x) alors g_1(z, x) sinon g_2(z, x)

D'après la règle de la conditionnelle, les énoncés de C, g_1 et g_2 sont respectivement :

profil	:	cond = C(z : entier, x : liste(entier)) → booléen
préc	:	$x \neq nil \wedge cdr(x) \neq nil \wedge z \in cdr(x) \wedge SUP(z, cdr(x))$
post	:	(cond=vrai) ≡ Q_0(z, x)
profil	:	max = g_1(z : entier, x : liste(entier)) → liste(entier)
préc	:	$x \neq nil \wedge cdr(x) \neq nil \wedge z \in cdr(x) \wedge SUP(z, cdr(x)) \wedge Q_0(z, x)$
post	:	Q_1(z, x , max)
profil	:	max = g_2(z : entier, x : liste(entier)) → liste(entier)
préc	:	$x \neq nil \wedge cdr(x) \neq nil \wedge z \in cdr(x) \wedge SUP(z, cdr(x)) \wedge \neg Q_0(z, x)$
post	:	Q_2(z, x, max)

Les propriétés que doivent vérifiées les relations Q_0, Q_1 et Q_2 sont :

1 - $x \neq nil \wedge cdr(x) \neq nil \wedge z \in cdr(x) \wedge SUP(z, cdr(x)) \wedge Q_0(z, x) \wedge Q_1(z, x, max)$

$\Rightarrow$ (max=car(x) $\wedge$ SUP(max, cdr(x))) $\vee$ (max=z $\wedge$ max $\geq$ car(x))

2 - x$\neq$nil $\wedge$ cdr(x)$\neq$nil $\wedge$ z $\in$ cdr(x) $\wedge$ SUP(z, cdr(x)) $\wedge \neg Q_0(z, x) \wedge Q_2(z, x, max)$
$\Rightarrow$ (max=car(x) $\wedge$ SUP(max, cdr(x))) $\vee$ (max=z $\wedge$ max $\geq$ car(x))

Nous choisissons d'instancier Q_1 et Q_2 par des critères syntaxiques, c'est à dire respectivement par la première et la seconde partie de la disjonction de la postcondition de la fonction g. Ce choix donne :

$Q_1(z, x, max)$ est défini par max=car(x) $\wedge$ SUP(max, cdr(x))
$Q_2(z, x, max)$ est défini par max=z $\wedge$ max $\geq$ car(x)

Pour ces instances les formules 1 et 2 sont trivialement vraies, ce sont donc des postconditions possibles pour g_1 et g_2. Les préconditions des énoncés relatifs à g_1 et g_2 ne sont pas connues en entier puisque Q_0 est une relation non encore déterminée. La démarche adoptée est alors la suivante : nous construisons d'abord g_1 et g_2 en calculant pour quelle précondition leur synthèse aboutit. Ceci permettra de déduire Q_0 et donc de construire la fonction C.

• **Etape 4** : Synthèse de g_1.

Comme précédemment nous pouvons définir g_1 par la fonction de base car. Nous devons donc prouver les propriétés :

1 - x$\neq$nil $\wedge$ cdr(x)$\neq$nil $\wedge$ z $\in$ cdr(x) $\wedge$ SUP(z, cdr(x)) $\wedge Q_0(z, x) \Rightarrow$ x$\neq$nil

2 - x$\neq$nil $\wedge$ cdr(x)$\neq$nil $\wedge$ z $\in$ cdr(x) $\wedge$ SUP(z, cdr(x)) $\wedge Q_0(z, x) \wedge$ max=car(x)
$\Rightarrow$ max=car(x) $\wedge$ SUP(max, cdr(x))

Preuve de 1 : cette formule est vraie indépendamment de la relation Q_0 puisque A $\Rightarrow$ A est toujours vrai.

Preuve de 2 : en utilisant la substitutivité de l'égalité, cela revient à montrer :

x$\neq$nil $\wedge$ cdr(x)$\neq$nil $\wedge$ z $\in$ cdr(x) $\wedge$ SUP(z, cdr(x)) $\wedge Q_0(z, x) \Rightarrow$ SUP(car(x), cdr(x))

Une hypothèse possible pour prouver SUP(car(x), cdr(x)) est la solution triviale SUP(car(x), cdr(x)). Mais, nous choisissons ici de ramener cette condition en terme de la variable z. En effet, z désigne le résultat de l'appel récursif max(cdr(x)) et son calcul nécessite déjà un parcours de la liste initiale x et il en est de même de SUP(car(x), cdr(x)). Chercher à exprimer cette dernière formule en fonction de z peut permettre d'optimiser ces deux parcours. Pour trouver Q_0 il faut donc disposer du théorème :

SUP(z, cdr(x)) $\wedge$ car(x) $\geq$ z $\Rightarrow$ SUP(car(x), cdr(x))

Par ce théorème la formule 2 est alors vraie pour l'instance car(x) $\geq$ z de $Q_0(z, x)$. Nous avons donc $g_1(z, x) \leftarrow$ car(x)

si et seulement si z et x sont tels que car(x) ≥ z. Les énoncés de C et g_2 sont alors :

profil : cond = C (z : entier, x : liste(entier)) → entier
préc : x≠nil ∧ cdr(x)≠nil ∧ z ∈ cdr(x) ∧ SUP(z, cdr(x))
post : (cond=vrai) ≡ (car(x) ≥ z)

profil : max = g_2 (z : entier, x : liste(entier)) → entier
préc : x≠nil ∧ cdr(x)≠nil ∧ z ∈ cdr(x) ∧ SUP(z, cdr(x)) ∧ z > car(x)
post : max=z ∧ max ≥ car(x)

• **Etape 5** : synthèse de C.

C(z, x) peut être définie par C(z, x) ← car(x) ≥ z, les fonctions car et ≥ étant des fonctions de base. Pour vérifier cela, il faut montrer que ces deux fonctions sont bien définies pour la précondition de C. En appliquant la règle relative à la composition fonctionnelle cela revient à prouver :

1 - x≠nil ∧ cdr(x)≠nil ∧ z ∈ cdr(x) ∧ SUP(z, cdr(x)) ⇒ x≠nil

2 - x≠nil ∧ cdr(x)≠nil ∧ z ∈ cdr(x) ∧ SUP(z, cdr(x)) ∧ w=car(x) ⇒ vrai

La première formule décrit le fait que la précondition de la fonction car doit être vérifiée pour l'argument x et la seconde que la précondition de la fonction ≥ doit être vérifiée pour les arguments car(x) et z. Ces 2 formules sont bien des théorèmes.

•**Etape 6** : synthèse de g_2.

g_2(z, x) peut être définie par g_2(z, x) ← z si et seulement si on peut montrer :

– x≠nil ∧ cdr(x)≠nil ∧ z ∈ cdr(x) ∧ SUP(z, cdr(x)) ∧ z > car(x) ∧ max=z ⇒ max=z ∧ z ≥ car(x)

Cette formule est vraie et donc g_2, définie par g_2(z, x) ← z, est une solution correcte. La synthèse est terminée puisqu'il n'y a plus de sous-problème à résoudre. Le programme final est :

max(x) ← si cdr(x)=nil alors f_1(x) sinon g(max(cdr(x)), x)
f_1(x) ← car(x)
g(z, x) ← si C(z, x) alors g_1(z, x) sinon g_2(z, n)
C(z, x) ← car(x) ≥ z
g_1(z, x) ← car(x)
g_2(z, x) ← z

Il existe un ordre bien fondé sur lequel est fondé la définition récursive, c'est l'ordre classique sur les listes.

Pour traiter cet exemple nous avons utilisé une aproche guidée par le but, c'est à dire que nous avons choisi, à chaque pas, la forme du programme à construire. A chacun de ces choix correspond un ensemble de formules du CP1. Soit celles-ci doivent être prouvées, soit on doit en trouver des instances valides. La recherche d'instances revient à trouver une hypothèse sous laquelle une formule est vraie. Pour cette recherche, nous avons imposé que certaines hypothèses connues soient utilisées. Par exemple lorsque nous avons cherché les hypothèses nécessaires à la preuve de :

$$z \in cdr(x) \wedge SUP(z, cdr(x)) \Rightarrow max \in cdr(x) \wedge SUP(max, cdr(x))$$

nous avons imposé que l'hypothèse $z \in cdr(x) \wedge SUP(z, cdr(x))$, correspondant à l'introduction de l'appel récursif max(cdr(x)), serve effectivement à la preuve de la formule $max \in cdr(x) \wedge SUP(max, cdr(x))$. Ceci garantit que le programme construit calcule bien le résultat à partir du résultat de l'appel récursif. Si cela n'avait pas été le cas, la construction d'un programme récursif de cette forme n'aurait pas été justifiée.

2.3. Synthèses à partir d'énoncés récursifs.

Par rapport aux énoncés classiquement admis, nous avons dit qu'il serait nécessaire d'admettre des énoncés caractérisant récursivement les fonctions cherchées. Ceci permet d'accroître la puissance d'expression du langage des énoncés mais rend le processus de synthèse intrinséquement du second ordre. En effet, cela revient à construire un plus petit point fixe et nécessite de raisonner sur les programmes au cours de leur construction. Illustrons ceci sur 2 exemples relatifs à la synthèse du pgcd.

Exemple 1. Soit l'énoncé :

profil	:	pg = f(x, y : entier) → entier
préc	:	$x \neq 0 \vee y \neq 0$
post	:	pg = **si** x=0 **alors** y **sinon** f(rem(y, x), x)

rem(y, x) désigne le reste de la division euclidienne de y par x. Cette fonction peut être définie par l'énoncé :

profil	:	reste = rem(a, b : entier) → entier
préc	:	$b \neq 0$
post	:	reste = **si** a < b **alors** a **sinon** rem(a - b, b)

Nous voulons construire un programme récursif de la forme :

$$pgcd(x, y) \leftarrow \textbf{si } x=0 \textbf{ alors } f_1(x, y) \textbf{ sinon si } y \geq x \textbf{ alors } g_1(x, y, pgcd(x, y-x)) \textbf{ sinon } g_2(x, y, pgcd(y, x))$$

Par les règles de la récursion et de la conditionnelle, les énoncés de f_1, g_1 et g_2 sont de la forme :

profil	:	pg = f_1(x, y : entier) → entier
préc	:	$(x \neq 0 \vee y \neq 0) \wedge x=0$
post	:	$Q_1(x, y, pg)$

profil : $pg = g_1(x, y, pg_1 : entier) \rightarrow entier$
préc : $(x \neq 0 \vee y \neq 0) \wedge x \neq 0 \wedge y \geq x$
post : $Q_2(x, y, pg_1, pg)$

profil : $pg = g_2(x, y, pg2 : entier) \rightarrow entier$
préc : $(x \neq 0 \vee y \neq 0) \wedge x \neq 0 \wedge x > y$
post : $Q_3(x, y, pg_2, pg)$

pg1 et pg2 désignent respectivement les résultats pgcd(x, y-x) et pgcd(y, x). Les propriétés que doivent vérifier Q_1, Q_2 et Q_3 sont :

1- $y \neq 0 \wedge x = 0 \wedge Q_1(x, y, pg) \Rightarrow pg =$ **si** $x=0$ **alors** y **sinon** f(rem(y, x), x)

2- $x \neq 0 \wedge y \neq 0 \wedge y \geq x \wedge pg_1 = f(x, y-x) \wedge Q_2(x, y, pg_1, pg) \Rightarrow pg =$ **si** $x=0$ **alors** y **sinon** f(rem(y, x), x)

3- $x \neq 0 \wedge y \neq 0 \wedge y \geq x \Rightarrow (x \neq 0 \vee y-x \neq 0)$

4- $x \neq 0 \wedge y \neq 0 \wedge x > y \wedge pg_2 = f(y, x) \wedge Q_3(x, y, pg_2, pg) \Rightarrow pg =$ **si** $x=0$ **alors** y **sinon** f(rem(y, x), x)

5- $x \neq 0 \wedge y \neq 0 \wedge x > y \Rightarrow (y \neq 0 \vee x \neq 0)$

Par hypothèse d'induction nous avons supposé que pg_1 et pg_2 vérifient bien l'énoncé initial c'est à dire sont tels que $pg_1 = f(x, y-x)$ et $pg_2 = f(y, x)$. Les formule 3 et 4, qui décrivent le fait que la fonction pgcd peut bien s'appliquer sur les arguments (x, y-x) et (y, x), sont trivialement vraies.

Preuve de 1 :

Lorsque x est nul l'égalité (pg = **si** $x=0$ **alors** y **sinon** f(rem(y, x), x)) se réduit à pg = y. Une solution pour $Q_1(x, y, pg)$ est donc la relation **pg = y**.

Preuve de 2 : Après simplification, nous recherchons une hypothèse Q_2 telle que :

$$x \neq 0 \wedge y \neq 0 \wedge y \geq x \wedge pg_1 = f(x, y-x) \wedge Q_2(x, y, pg_1, pg) \Rightarrow pg = f(rem(y, x), x)$$

Par définition de la fonction f, f(x, y-x) est égal à f(rem(y-x, x), x) puisque x est non nul. D'autre part, en utilisant la définition de la fonction **rem**, f(rem(y, x), x) est égal à f(rem(y-x, x), x) puisque x est supérieur à y. Nous devons donc trouver Q_2 tel que :

$$pg_1 = f(rem(y-x, x), x) \wedge Q_2(x, y, pg_1, pg) \Rightarrow pg = f(rem(y-x, x), x)$$

$pg = pg_1$ est une solution possible pour cette relation.

Preuve de 4 : Après simplification, nous recherchons une hypothèse Q_3 telle que :

$$y \neq 0 \wedge x \neq 0 \wedge x > y \wedge pg_2 = f(y, x) \wedge Q_3(x, y, pg_2, pg) \Rightarrow pg = f(rem(y, x), x)$$

Or f(rem(y, x), x) est égal à f(y, x) par définition de la fonction **rem**, puisque y est strictement inférieur à x. Nous devons donc trouver une relation Q_3 telle que : $pg_2 = f(y, x) \wedge Q_3(x, y, pg_2, pg) \Rightarrow pg = f(y, x)$.

Une solution est $pg = pg_2$.

La fonction $f_1(x, y)$ est donc la fonction qui retourne son second argument, les fonctions $g_1(x, y, pg_1)$ et $g_2(x, y, pg_2)$ celles qui retournent leur dernier argument. Le programme final est alors :

pgcd(x, y) ← **si** x=0 **alors** y **sinon si** y ≥ x **alors** pgcd(x, y-x) **sinon** pgcd(y, x)

fin-exemple.

Dans cet exemple le fait que l'énoncé initial caractérise la fonction cherchée de manière récursive n'a pas posé de problème particulier. Les hypothèses d'induction utilisées sont celles nécessaires à la construction d'un programme récursif : il a seulement été supposé que les appels récursifs vérifient l'énoncé initial. En effet l'égalité entre la fonction f, caractérisée par l'énoncé, et le programme pgcd, donné ci-dessus, peut être démontré en utilisant les hypothèses :

f(y, x) = pgcd(y, x) lorsque y est strictement inférieur à x
f(x, y - x) = pgcd(x, y - x) lorsque y est supérieur à x

Ces hypothèses sont valides si on considère l'ordre lexicographique sur les couples d'entiers. En effet, on a alors (y, x) < (x, y) lorsque y est strictement inférieur à x et (x, y-x) < (x, y) lorsque y est supérieur à x et x non nul.

Nous avons ainsi pu établir l'égalité de la fonction pgcd avec la fonction f grâce à la définition de la fonction **rem**. L'induction nécessaire à la preuve d'égalité de ces deux fonctions est en fait celle donnée dans l'énoncé. Montrons maintenant comment construire le programme basé sur la division euclidienne à partir du programme procédant par soustractions successives. Nous rencontrons ici une difficulté particulière puisque l'induction nécessaire est celle du programme et que celui-ci n'est que partiellement connu.

Exemple 2. Soit l'énoncé :

profil : pg = f(x, y : entier) → entier
préc : x≠0 ∨ y≠0
post : pg = **si** x=0 **alors** y **sinon si** y ≥ x **alors** f(x, y - x) **sinon** f(y, x)

On veut construire un programme de la forme :

pgcd(x, y) ← **si** x=0 **alors** $f_1(x, y)$ **sinon** g(x, y, pgcd(rem(y, x), x))

Par la règle de la récursion les énoncés de f_1 et g sont :

profil : $pg = f_1(x, y : entier) \rightarrow entier$
préc : $y \neq 0 \wedge x = 0$
post : $Q_1(x, y, pg)$

profil : $pg = g(x, y, pg_1 : entier) \rightarrow entier$
préc : $x \neq 0$
post : $Q_2(pg_1, x, y, pg)$

pg_1 désigne le résultat pgcd(rem(y, x), x). Les relations Q_1 et Q_2 doivent être telles que :

1- $y \neq 0 \wedge x = 0 \wedge Q_1(x, y, pg) \Rightarrow pg =$ **si** $x=0$ **alors** y **sinon si** $y \geq x$ **alors** $f(x, y - x)$ **sinon** $f(y, x)$

2- $x \neq 0 \wedge pg_1 = f(rem(y, x), x) \wedge Q_2(pg1, x, y, pg) \Rightarrow pg =$ **si** $x=0$ **alors** y **sinon**
si $y \geq x$ **alors** $f(x, y - x)$ **sinon** $f(y, x)$

3- $x \neq 0 \Rightarrow (x \neq 0 \vee y \neq 0)$

On a supposé, par hypothèse d'induction, que pgcd(rem(y, x), x) vérifie bien l'énoncé initial. La formule 3 décrit le fait qu'il faut pouvoir appliquer la fonction rem sur l'argument (y, x) et la fonction pgcd sur l'argument (rem(y, x), x). Elle est trivialement vraie.

Preuve de 1 : Par simplification on doit trouver Q_1 telle que $x=0 \wedge Q_1(x, y, pg) \Rightarrow pg = y$. Une solution possible pour Q_1 est donc la relation **pg=y**.

Preuve de 2 : Après simplification on veut trouver Q_2 tel que :

$x \neq 0 \wedge pg_1 = f(rem(y, x), x) \wedge Q_2(pg_1, x, y, pg) \Rightarrow pg =$ **si** $y \geq x$ **alors** $f(x, y - x)$ **sinon** $f(y, x)$

Nous procédons par cas suivant que x est strictement supérieur à y ou non.

- **Cas x > y** : Il faut donc trouver Q_2 tel que $x \neq 0 \wedge x > y \wedge pg_1 = f(rem(y, x), x) \wedge Q_2(pg_1, x, y, pg) \Rightarrow pg = f(y, x)$. Or f(rem(y, x), x) est égal à f(y, x) par définition de **rem**. Nous devons donc avoir :

$$pg_1 = f(y, x) \wedge Q_2(pg_1, x, y, pg) \Rightarrow pg = f(y, x)$$

Une solution pour $Q_2(pg_1, x, y, pg)$ est $pg = pg_1$.

- **Cas y ≥ x** : Il faut donc trouver Q_2 tel que $x \neq 0 \wedge y \geq x \wedge pg_1 = f(rem(y, x), x) \wedge Q_2(pg_1, x, y, pg) \Rightarrow pg = f(x, y - x)$.Or f(rem(y, x), x) est égal à f(rem(y-x, x), x) par définition de la fonction **rem**. Cela revient donc à trouver Q_2 tel que :

$$x \neq 0 \wedge y \geq x \wedge pg_1 = f(rem(y-x, x), x) \wedge Q_2(pg_1, x, y, pg) \Rightarrow pg = f(x, y - x) \qquad (a)$$

Il faut maintenant supposer, par hypothèse d'induction, que f(x, y-x) est obtenu par le programme pgcd, c'est à dire est tel que f(x, y -x) = g(f(rem(y -x), x), x), g étant la fonction qu'on cherche à caractériser. Cette hypothèse se traduit par l'implication :

$$pg_2 = f(rem(y-x, x), x) \wedge Q_2(pg_2, x, y-x, pg) \Rightarrow pg = f(x, y-x)$$

Nous utilisons cette hypothèse pour prouver (a). Il faut donc montrer :

$$pg_1 = f(rem(y-x, x), x) \wedge Q_2(pg_1, x, y, pg) \Rightarrow pg_2 = f(rem(y-x, x), x) \wedge Q_2(pg_2, x, y-x, pg)$$

Une solution triviale pour obtenir $pg_2 = f(rem(y-x, x), x)$ est de choisir l'hypothèse additionnelle $pg_1 = pg_2$. La relation Q_2 doit alors être telle que $Q_2(pg_1, x, y, pg) \Rightarrow Q_2(pg_1, x, y-x, pg)$. La solution précédemment trouvée, $Q_2(pg_1, x, y, pg)$ définie par $pg=pg_1$, convient puisque $pg=pg_1 \Rightarrow pg=pg_1$.

Les fonctions f_1 et g sont donc caractérisées par les postconditions pg = y et $pg = pg_1$. Une définition de ces fonctions est donc $f_1(x, y) \leftarrow y$ et $g(x, y, pg_1) \leftarrow pg_1$. Le programme final est :

pgcd(x, y) ← **si** x=0 **alors** y **sinon** pgcd(rem(y, x), x)

La difficulté lors de la synthèse à partir d'énoncés récursifs provient du fait qu'on peut être amené à supposé que les occurrences récursives de l'énoncé sont bien vérifiées par le programme alors que celui-ci n'est que partiellement connu. Ceci nécessite de représenter la preuve en cours de développement afin de pouvoir constituer l'hypothèse d'induction relative au programme en cours de construction.

fin-exemple.

Nous disposons maintenant d'un système de preuves des énoncés. Son intérêt est de mettre en évidence le lien entre schémas de programmes, schémas de preuves et schémas de théorèmes validant les programmes construits. La difficulté est maintenant de guider le processus de recherche d'une preuve ainsi que la recherche de théorèmes permettant le déroulement de cette preuve. Nous présentons maintenant des stratégiesde preuves, basées sur le système formel proposé. Ces stratégies auront deux interlocuteurs, l'un que nous appelons Utilisateur et qui est responsable des choix de la preuve à mettre en place et le second, la Base de Connaissances, auquel est demandé des instances appropriées des propriétés nécessaires. Dans ces stratégies, nous cherchons à proposer des critères permettant de juger si le programme est pertinent, vis à vis des informations contenues dans la base de théorèmes. D'autre part nous mettrons en évidence comment gérer le déroulement complet d'une preuve et les remises en cause possibles lorsque certains énoncés ne peuvent pas être prouvés complètement. Les stratégies que nous proposons reprennent en partie les stratégies utilisées dans les différents travaux, notamment celles du système LOPS. Les décrire dans un cadre formel offre l'avantage d'une part de les justifier et d'autre part de détecter les pas de la preuve qu'elles ne prennent pas en compte.

CHAPITRE 3

GESTION DE LA PREUVE ET STRATEGIES.

L'élaboration d'une preuve, pour un énoncé donné, se fera par l'intermédiaire de stratégies. Les stratégies que nous proposons permettent soit de transformer les énoncés soit de les prouver en engendrant une construction primitive du langage cible (fonctions de base, conditionnelle, composition fonctionnelle et récursion). Ces stratégies sont basées sur le choix de paramètres qui peuvent être vus comme les indications qui ont permis de choisir une preuve plutôt qu'une autre. C'est l'utilisateur qui fixe les stratégies à employer et leurs paramètres.

Les stratégies décrivent un enchainement particulier de la preuve des prémisses, à partir des paramètres fournis par l'utilisateur. Rappelons que les prémisses sont soit des schémas de formules à prouver dans notre logique, soit des schémas de propriétés qui doivent être vrais dans le domaine du problème. Pour que ces stratégies apportent une aide effective dans le processus de construction d'une preuve nous nous sommes attachés aux trois points suivants :

- Décomposer le plus précisément possible les demandes de propriétés afin qu'il soit effectivement envisageable d'en chercher des instances.

- Pour chaque stratégie, mettre en évidence des critères permettant de juger si cette stratégie est pertinente ou non vis à vis du problème traité. En effet nous ne sommes pas seulement intéressés par trouver une preuve d'un énoncé mais une preuve "efficace". Par exemple introduire une conditionnelle est intéressant uniquement si la condition choisie permet de simplifier le problème initial, sinon les sous-problèmes introduits sont aussi complexes que ce problème. L'évaluation de ces critères dépendra des propriétés existantes sur le domaine du problème et donc des réponses fournies aux demandes d'instances de propriétés faites par le système de preuve.

- Calculer la classe des données pour laquelle les stratégiess'appliquent avec succés. La précondition de l'énoncé, donnée par l'utilisateur ou établie par le système, décrit la classe des données pour laquelle une preuve est recherchée. Néanmoins, les choix de preuve faits peuvent ne permettre la preuve que pour une sous-classe de ces données. Lors de l'application d'une stratégie nous calculerons donc la *précondition dérivée* qui caractérise cette sous-classe. Ceci est nécessaire car certaines stratégies que nous proposons sont basées sur le calcul des préconditions dérivées pour la preuve des sous-problèmes engendrés.

Le déroulement standart d'une stratégie est le suivant:

début

1 - Choix des fonctions de l'énoncé auxquelles elle s'applique et des paramètres de la stratégie.

2 - Suite de demandes relatives à des propriétés sur le domaine du problème. Si les réponses obtenues sont jugées satisfaisantes, vis à vis des critères relatifs à la pertinence de la stratégie, alors le système constitue, s'il y a lieu, les énoncés des sous-problèmes. Si ces réponses ne sont pas satisfaisantes la

stratégie échoue.

3 - Itération du processus de synthèse sur les énoncés des sous-problèmes et traitement des préconditions dérivées lors de ces synthèses. L'ordre dans lequel les sous-problèmes sont traités dépend de la stratégie.

4 - Elaboration de la précondition dérivée pour laquelle la preuve de l'énoncé a abouti. Si un échec a été détecté au cours de la preuve cette précondition dérivée est **faux**.

- si cette précondition se réduit à **vrai** alors l'énoncé est un théorème et on dispose d'un programme qui décrit la preuve effectuée ;

- sinon, si l'existence d'une précondition dérivée découle d'une stratégie en cours d'application, la preuve faite n'est pas remise en cause ;

- sinon il y a traitement d'un échec (voir section 1.2 de ce chapitre).

5 - Si l'énoncé traité est l'énoncé initial il faut construire un programme calculant si une donnée appartient effectivement au domaine de définition des fonctions obtenues. L'énoncé de ce sous-problème est :

profil	:	cond = C(x : t) → booléen
préc	:	vrai
post	:	(cond=vrai) ≡ (P(x) ∧ H(x))

P(x) est la précondition initiale et H(x) la précondition dérivée lors de la preuve complète de l'énoncé. Le programme engendré est de la forme:

$$(f_1(x), \ldots, f_n(x)) \leftarrow \textbf{si } C(x) \textbf{ alors } (f_1'(x), \ldots, f_n'(x)) \textbf{ sinon } \perp$$

Les fonctions f_i' sont décrites par l'énoncé initial. Remarquons que si C(x) est indéfinie pour certaines valeurs de x ce programme renverra bien la valeur indéfinie par définition de la conditionnelle.

fin.

Lorsqu'un énoncé caractérise plusieurs fonctions et que la stratégie ne s'applique que sur un sous-ensemble de ces fonctions, les sous-problèmes engendrés caractérisent d'une part les fonctions introduites par la stratégie retenue et d'autre part les fonctions qui n'ont pas encore été traitées. En effet un énoncé peut décrire différentes fonctions qui ne doivent pas nécessairement être synthétisées par la même structure de preuve.

Il y a donc deux causes possibles d'échec. Soit la stratégie choisie n'est pas pertinente vis à vis du problème traité soit les sous-problèmes engendrés ne peuvent pas être démontrés. Le premier cas d'échec est détecté au pas 2, le second ne peut être détecté qu'après avoir cherché à prouver les énoncés des sous-problèmes. Ces échecs sont détectés à partir de la valeur de la précondition dérivée suivant que c'est la constante **vrai**, la constante **faux** ou une formule non réduite à l'une de ces deux valeurs. Lors de l'élaboration des préconditions dérivées il sera donc nécessaire de simplifier la formule obtenue pour la réduire à **vrai** ou **faux** dès que possible.

1. GESTION DE LA PREUVE ET DES ECHECS.

Nous choisissons de construire un arbre de preuve décrivant la démonstration en cours. Cet arbre permettra de mémoriser la preuve effectuée et de gérer la remise en cause de cette preuve lorsque des énoncés ne peuvent pas être prouvés.

1.1. Arbre de preuve.

Définition :

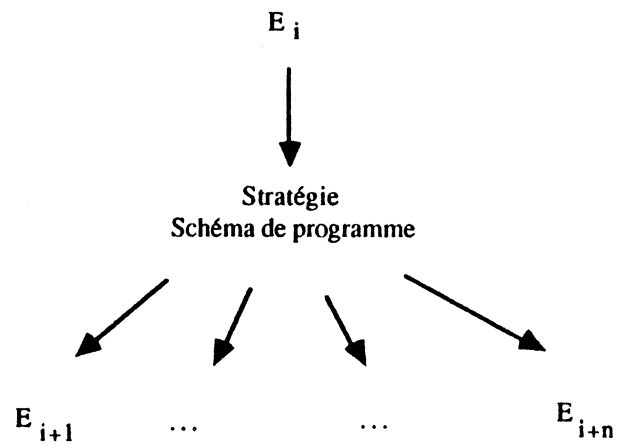
- la racine est étiquetée par l'énoncé à prouver ;

- un nœud **énoncé** qui n'a pas encore été prouvé, ou qui n'est pas en cours de preuve, est une feuille de l'arbre ;

- le fils d'un nœud **énoncé** qui a été traité est un arbre dont la racine contient les choix effectués et leur conséquence (stratégie utilisée, schéma de programme associé, ...) et dont les fils sont les énoncés des sous-problèmes engendrés par l'application de la stratégie retenue.

fin-définition.

Un niveau de l'arbre est donc de la forme :



Le nombre de fils dépend du nombre de sous-problèmes engendrés par la stratégie. La seule stratégie qui n'engendre pas de sous-problème est la stratégie **Reconnaissance de fonction de base** lorsqu'elle s'applique sur l'ensemble des fonctions caractérisées par l'énoncé traité. Un énoncé E est donc démontré si et seulement si toutes les feuilles du sous-arbre issu de E sont de la forme :

E_i

Reconnaissance de fonction de base
$f_i(x) \leftarrow b_i(x)$

Nous disons alors que le sous-arbre est complet. L'ensemble des schémas de programmes apparaissant dans les nœuds de ce sous-arbre constitue un programme vérifiant E. L'énoncé initial est donc un théorème si l'arbre issu de la racine est complet.

L'arbre de preuve est construit en profondeur d'abord puisque la stratégie globale de preuve que nous avons adoptée consiste à prouver complètement un énoncé avant de chercher à prouver les énoncés des sous-problèmes restants. Il sera élaboré dans la phase descendante, c'est à dire avant la preuve des sous-problèmes, afin de mémoriser les différents choix dès qu'ils sont effectués. Les énoncés E_{i+1}, ..., E_{i+n} des sous-problèmes sont alors traités l'un après l'autre. La preuve de chaque E_{i+k}, si il n'y a pas d'échec complet, produit un arbre complet issu de E_{i+k}. Lorsque tous les énoncés ont été traités et qu'il n'y a pas eu d'échec l'arbre total est donc complet.

1.2. Traitement des échecs.

Un cas d'échec est détecté si et seulement si la précondition dérivée lors d'une preuve n'est pas réduite à vrai et si l'existence d'une telle précondition ne découle pas de l'application d'une stratégie à un niveau supérieur dans l'arbre. Cette seconde condition provient du fait que certaines stratégies utilisent les préconditions dérivées lors de la preuve des sous-problèmes qu'elles engendrent. L'échec est complet si cette précondition est toujours fausse et partiel sinon.

Si un énoncé E n'a pu être démontré à l'aide d'une stratégie alors soit il peut être démontré par une autre stratégie soit il faut remettre en cause la preuve d'un énoncé de laquelle E est issu. Il n'y a pas de règle générale pour choisir, c'est donc à l'utilisateur de guider la remise en cause. Le système peut néanmoins lui apporter des informations. Il peut indiquer si il y a eu perte d'information entre l'énoncé E, pour lequel l'échec est détecté, et l'énoncé dont E est issu. En effet nous n'imposons pas que l'ensemble des solutions possibles d'un énoncé et des sous-problèmes issus de sa preuve soit nécessairement préservé (voir chapitre précédent, section 2.1.). L'échec peut donc provenir de cette perte d'information et il peut alors être préférable de remettre en cause la preuve qui a engendrée l'énoncé E.

Un énoncé peut être jugé non démontrable soit parce que toutes les stratégies choisies par l'utilisateur ont conduit à des échecs et que celui-ci n'en propose plus soit parce que cet énoncé n'a effectivement pas de solution. Prouver qu'un énoncé est faux, c'est à dire qu'il n'est pas possible de construire une fonction f le vérifiant, est indécidable dans notre système. En effet nous avons décrit comment prouver des formules de la forme $\exists f \forall x (P(x) \Rightarrow Q(x, f(x))$ mais nous ne nous sommes pas intéressés à la négation de ces formules. Par définition de la négation, $\neg A$ est vrai si et seulement si supposer A produit une contradiction. Il faudrait donc montrer que toute définition de f produit une contradiction, c'est à dire que toute preuve possible dans notre système conduit à un échec. Ceci est indécidable puisque l'ensemble des preuves est en général infini. Il existe néanmoins une condition suffisante.

Propriété :

Si $\forall$ x, y $\neg(P(x) \Rightarrow Q(x, y))$ est un théorème de la logique classique alors l'énoncé dont la précondition est P(x) et dont la postcondition est Q(x, f(x)) est faux.

fin-propriété.

En effet si pour tout y, Q(x, y) est faux lorsque x vérifie la précondition P alors aucune fonction f ne peut exister. Un exemple trivial est lorsque la postcondition est réduite à faux et la précondition décrit un ensemble non vide de données. Cette propriété n'est que suffisante puisque y peut exister mais on peut être incapable de le calculer. Il ne sera donc possible de démontrer qu'un énoncé n'a pas de solution que dans des cas particuliers, soit lorsque la propriété donnée ci-dessus est vraie soit lorsqu'on peut énumérer l'ensemble des preuves possibles de l'énoncé et montrer qu'elles ne sont pas correctes.

1.2.1. Traitement des échecs complets.

Lorsque la précondition dérivée se réduit à **faux** alors l'utilisateur choisit, parmi les énoncés ascendants de E_i, l'énoncé E à partir duquel reprendre la preuve. Le sous-arbre de preuve issu de E est coupé et l'utilisateur doit choisir une nouvelle preuve de cet énoncé.

1.2.2. Traitement des échecs partiels.

Lorsque la preuve de E n'a aboutie que pour un sous-ensemble non vide des données alors c'est à l'utilisateur de juger si il est pertinent de conserver cette preuve.

- Si la réponse est négative la preuve est rejetée et le traitement est le même que dans le cas d'un échec complet.

- Si la réponse est positive l'utilisateur doit alors décider si la preuve, pour les données acceptables ne vérifiant pas la précondition dérivée, doit être menée à partir de l'énoncé E ou d'un énoncé dont E est issu. En effet, pour les mêmes raisons que précédemment, certaines informations ont pu être perdues et il peut être préférable de compléter la preuve à un niveau supérieur.

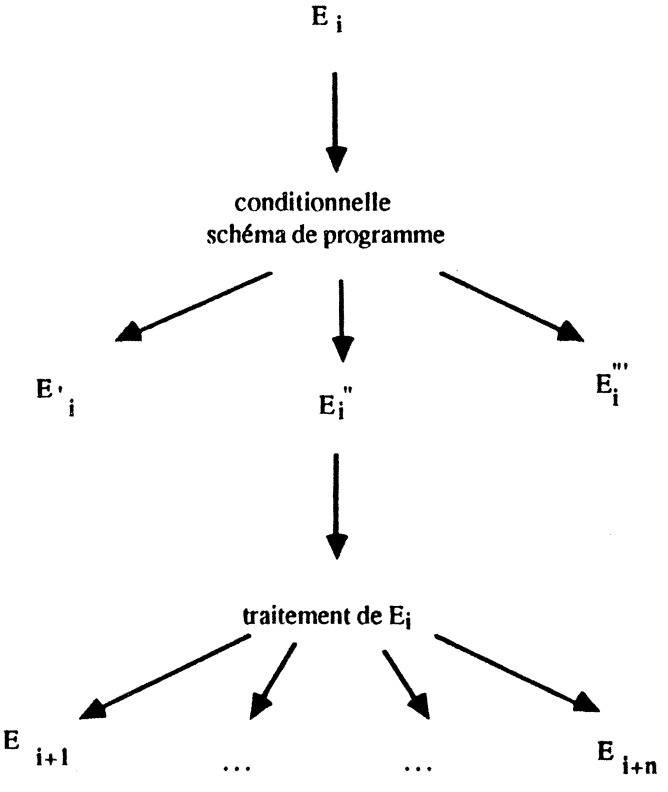
- si E n'est pas choisi alors la précondition dérivée est retournée à la stratégie appelante et l'échec partiel sera traité au niveau appelant. L'obtention d'une précondition dérivée différente de vrai ne remet généralement pas en cause la preuve ayant produit cette précondition sauf lorsque cette stratégie est relative à l'introduction d'une récursion. Dans ce cas il faut que la précondition dérivée soit aussi vérifiée par les appels récursifs. Cette étape est décrite dans lors de la présentation des stratégies d'introduction de récursion.

- si la remise en cause à lieu pour l'énoncé E_i alors il y a introduction d'une conditionnelle dont la condition décrit la précondition dérivée. Cela revient à appliquer une des stratégies relatives à l'introduction d'une conditionnelle (stratégie Instanciation d'une postcondition). Nous ne la détaillons pas ici mais présentons uniquement la modification de l'arbre qui en résulte.

Supposons que la précondition dérivée lors de la preuve de E_i soit H(x) et que l'énoncé E_i soit :

profils	:	$var_1 = f_1(x : t) \rightarrow t_1, \ldots, var_n = f_n(x : t) \rightarrow t_n$
préc	:	P(x)
post	:	$Q(x, var_1, \ldots, var_n)$

Le système construit l'arborescence suivante :



- Le schéma de programme engendré est :

$(f_1(x), \ldots, f_n(x)) \leftarrow$ **si** C(x) **alors** $(f_1"(x), \ldots, f_n"(x))$ **sinon** $(f_1"'(x), \ldots, f_n"'(x))$

- L'énoncé E_i' est :

profil	:	cond = C(x : t) → booléen
préc	:	P(x)
post	:	(cond = vrai) ≡ H(x)

L'énoncé $E_i"$ et l'arbre de preuve issu de cet énoncé sont obtenus à partir de l'arbre de preuve construit initialement lors de la preuve de E_i en renommant les fonctions $f_1, \ldots, f_n$ en $f_1", \ldots, f_n"$. L'énoncé $E_i"'$ est :

profils	:	$var_1 = f_1'''(x : t) \rightarrow t_1, \ldots, var_n = f_n'''(x : t) \rightarrow t_n$
préc	:	$P(x) \wedge \neg H(x)$
post	:	$Q'''(x, var_1, \ldots, var_n)$

$Q'''(x, var_1, \ldots, var_n)$ est une relation fournie par la base de connaissances par simplification de la postcondition initiale Q à l'aide de la propriété $\neg H(x)$.

Exemple.

Supposons que nous synthétisons la fonction **max** sans restreindre l'ensemble des données. L'énoncé initial est donc :

profil	:	max = **max**(x : liste(entier)) → entier
préc	:	vrai
post	:	max ∈ x ∧ SUP(max, x)

En choisissant la même preuve que celle développée au chapitre précédent, l'arbre obtenu au premier niveau est :

profil	:	max = **max**(x : liste(entier)) → entier
préc	:	vrai
post	:	max ∈ x ∧ SUP(max, x)

Récursion

max(x) ← **si** $C_0(x)$ **alors** $f_0(x)$ **sinon** g(max(cdr(x)), x)

La fonction **cdr** ne peut s'appliquer que pour x différent de nil, les énoncés de C_0, f_0 et g sont donc :

E_0 :

profil	:	cond = C_0(x : liste(entier)) → booléen
préc	:	vrai
post	:	(cond=vrai) ≡ x=nil

E_1 :

profil	:	max = f_0(x : liste(entier)) → entier
préc	:	x=nil
post	:	max ∈ nil ∧ SUP(max, nil)

E2 :

profil	:	max = g(z : entier, x : liste(entier)) → entier
préc	:	x≠nil
post	:	(max=car(x) ∧ SUP(max, cdr(x))) ∨ (max=z ∧ max ≥ car(x))

E_1 ne peut être prouvé puisque sa postcondition se réduit à **faux** en raison de l'équivalence max ∈ nil ≡ **faux**. Aucune preuve de cet énoncé ne peut donc être élaborée. Par contre la preuve des énoncés E_0 et E_2 ne pose pas de problème et la précondition dérivée est, dans les deux cas, **vrai**. D'après la règle relative à l'introduction de récursion l'énoncé caractérisant la fonction **max** n'est prouvé que pour la précondition dérivée **x≠nil**. Il y a donc échec partiel. Si nous choisissons de conserver cette preuve, l'argument de la récursion, cdr(x), doit donc vérifier la propriété cdr(x)≠nil. La construction g(max(cdr(x)), x) n'est donc valide que pour la précondition cdr(x)≠nil. Le cas cdr(x)=nil doit alors être traité par la fonction f_0 et le système modifie les énoncés E_0 et E_1 en :

E'_0 :	**profil**	:	cond = C_0(x : liste(entier)) → booléen
	préc	:	x≠nil
	post	:	cond=vrai ≡ cdr(x)=nil
E'_1 :	**profil**	:	max = f_0(x : liste(entier)) → entier
	préc	:	x≠nil ∧ cdr(x)=nil
	post	:	max=car(x)

La synthèse de ces deux fonctions aboutira maintenant pour la précondition dérivée **vrai** et donc l'introduction de récursion n'aboutira que pour la précondition dérivée **x≠nil**. Il reste à traiter l'échec de l'introduction d'une récursion pour la valeur nil. La seule remise en cause possible est sur l'énoncé initial. L'arbre serait alors modifié en :

profil : max = max(x : liste(entier)) → entier
préc : vrai
post : max ∈ x ∧ SUP(max, x)

↓

max(x) ← si C_1(x) alors max"(x) sinon max"'(x)

profil : cond = C_1(x)	**profil** : max = max"(x)	**profil** : max = max"'(x)
préc : vrai	**préc** : x≠nil	**préc** : x=nil
post : (cond=vrai) ≡ x≠nil	**post**: max ∈ x ∧ SUP(max, x)	**post**: faux

↓

Récursion

max"(x) ← si C(x) alors f_1(x) sinon g(max"(cdr(x)), x)

La synthèse de la fonction **max"'** renvoie la précondition dérivée **faux** puisque sa postcondition est réduite à **faux**.

Il n'y a plus de remise en cause possible, l'énoncé initial ne peut donc être prouvé que pour la précondition x≠nil. Le programme finalement synthétisé est :

max(x) ← **si** x≠nil **alors** max"(x) **sinon** ⊥
max"(x) ← **si** cdr(x)=nil **alors** car(x) **sinon** g(max"(cdr(x)), x)
g(z, x) ← **si** car(x) ≥ z **alors** car(x) **sinon** z

Dans cet exemple la remise en cause de l'énoncé initial n'a en fait pas lieue d'être. En effet l'équivalence a été préservée entre l'énoncé initial et E_1 pour le cas x=nil puisque le théorème utilisé était max ∈ nil ≡ **faux**. Il était donc possible de déduire directement que la fonction **max** n'est pas définie pour la valeur nil.

fin-exemple.

1.3. Demande de propriétés.

Les prémisses des règles de notre système relatives à des propriétés sur le domaine du problème découlent de l'implication :

$$P(x) \wedge POST[Q_1, \ldots, Q_n](x, y) \Rightarrow Q(x, y)$$

Q est la postcondition de l'énoncé à prouver, P sa précondition et $Q_1, \ldots, Q_n$ les postconditions des sous-problèmes. Les Q_i doivent être déterminés à partir des propriétés particulières du domaine du problème. Or, d'après les règles que nous avons proposées, $POST[Q_1, \ldots, Q_n]$ est toujours une forme sous forme conjonctive. La demande d'instances de propriétés peut alors se ramener à la recherche d'une hypothèse A, en terme des variables de Z, permettant de prouver la formule $B(Y, X) \Rightarrow C(Y)$ où Z est un sous-ensemble de $Y \cup X$. Le problème de la recherche d'instances de propriétés, qui est un problème du second ordre, est alors simplifié en le problème de la recherche d'hypothèses sous lesquelles une formule est un théorème.

Smith propose un système formel permettant de trouver de telles hypothèses, appelé Calcul de préconditions [Smi 82a]. Les hypothèses B connues sont assimilées aux théorèmes disponibles sur le domaine du problème. Dans le but à prouver, la formule C, toutes les variables quantifiées universellement sont skolémisées en fonction des variables quantifiées existentiellement. Le système calcule alors d'une part les préconditions nécessaires à la preuve de la formule C et d'autre part les instanciations portant sur les variables quantifiées existentiellement. Ce système est basé sur la syntaxe du but à prouver. Les formules obtenues sont simplifiées à l'aide d'un ensemble de règles de simplification qui sont soit basées sur des équivalences classiques du calcul propositionnel soit basées sur des simplifications liées à certains opérateurs succeptibles d'apparaître souvent (=, < , ...).

Dans le cadre de la synthèse de programmes l'une des difficultés principales de la recherche d'hypothèses est de pouvoir juger lorsqu'une solution est satisfaisante vis à vis de ce qu'on veut en faire. En effet plusieurs solutions sont possibles et il n'est pas possible de toutes les construire. L'ensemble des variables qui doivent apparaître dans l'hypothèse est une indication mais qui n'est pas toujours suffisante. En effet, la formule obtenue n'est pas nécessairement sous la forme la plus simple possible et, d'autre part, certaines demandes sont telles que les variables apparaissant dans le but à prouver et celles attendues dans les hypothèses sont les mêmes. Une autre difficulté, propre aussi à la Synthèse de Programmes, est relative à l'utilisation des hypothèses disponibles lors de la recherche des hypothèses manquantes. En effet, nous voulons pouvoir porter un jugement sur la pertinence de la preuve choisie, vis

à vis du problème traité. Or ceci dépend directement des propriétés propres au domaine du problème existantes. Nous désirons donc que les hypothèses disponibles découlant du choix d'une preuve servent effectivement lors de l'élaboration de cette preuve, ceci reviendra à imposer que ces hypothèses interviennent nécessairement lors de la recherche des hypothèses manquantes.

Exemple. Supposons que nous voulons construire un programme calculant la liste des facteurs premiers d'un entier. Un énoncé possible est :

profil : r=**fact**(n : entier) → liste(entier)
prec : $n \neq 0$
post : $prod(r) = n \wedge \forall w\ (dans(w, r) \Rightarrow premier(w))$

Avec :

prod(r) ← **si** r=nil **alors** 1 **sinon** car(r) * prod(cdr(r))

$premier(w) \equiv \forall u\ (u / w \Rightarrow (u=1 \vee u=w))$

u / w signifie que u est un diviseur de w. Supposons d'autre part que nous voulons construire une définition récursive de la fonction **fact**, basée sur la décomposition classique des entiers en 0 et succ. Ce programme sera de la forme :

fact(n) ← **si** n=1 **alors** $f_0(n)$ **sinon** g(fact(n-1), n)

Pour constituer l'énoncé de la fonction g nous devons trouver une hypothèse en terme de n, r_1 et r telle que la formule suivante soit vraie :

$$(n>1 \wedge prod(r_1) = n-1 \wedge \forall w\ (dans(w, r_1) \Rightarrow premier(w)))$$
$$\Rightarrow (prod(r) = n \wedge \forall w\ (dans(w, r) \Rightarrow premier(w)))$$

L'hypothèse obtenue sera la postcondition caractérisant la fonction g ayant comme données n et r_1. L'hypothèse d'induction fact(n-1)=r_1, décrite par la formule $prod(r_1) = n-1 \wedge \forall w\ (dans(w, r_1) \Rightarrow premier(w))$, doit intervenir lors de la recherche d'une telle hypothèse. En effet, si ce n'est pas le cas, le calcul de la décomposition en facteurs premiers de n-1 ne participe pas à l'élaboration du résultat pour la valeur n. et donc l'appel fact(n-1) est superflu.

• Si nous n'imposons pas l'utilisation de cette hypothèse d'induction alors nous pouvons accepter la solution :

$$prod(r) = prod(r_1) + 1 \wedge \forall w\ (dans(w, r) \Rightarrow premier(w))$$

En effet cette formule permet bien de prouver l'implication demandée. L'énoncé de la fonction g serait alors :

prec : $r=g(r_1$: liste(entier)) → liste(entier)
post : $\forall w\ (dans(w, r_1) \Rightarrow premier(w))$
post : $prod(r) = prod(r_1) + 1 \wedge \forall w\ (dans(w, r) \Rightarrow premier(w))$

La synthèse de la fonction g, à partir de cette postcondition, reste aussi complexe que celle de l'énoncé initial.
• Par contre si nous imposons l'utilisation de l'hypothèse d'induction alors la solution obtenue est **faux**. En effet, intuitivement, la preuve de $\forall w$ (**dans(w, r)** $\Rightarrow$ **premier(w)**) à partir de l'hypothèse $\forall w$ (**dans(w, r_1)** $\Rightarrow$ **premier(w)**) nécessite de trouver une relation entre les éléments de r et ceux de r_1 et d'autre part de montrer que lorsque les éléments de r ont la propriété d'être premier il en est de même des éléments de r_1. Or il n'existe pas de lien entre les différents nombres premiers, si ce n'est qu'ils sont tous premiers. La seule solution serait donc de montrer :

$$\forall w \, (dans(w, r) \Rightarrow dans(w, r_1))$$

La postcondition de la fonction de recomposition g serait alors la formule :

$$prod(r) = prod(r_1) + 1 \ \wedge \forall w \, (dans(w, r) \Rightarrow dans(w, r_1)) \ \wedge \ \forall w \, (dans(w, r) \Rightarrow premier(w)))$$

Or, de $\forall w$ (**dans(w, r)** $\Rightarrow$ **dans(w, r_1)**) on peut déduire que prod(r_1) est un diviseur de prod(r). Ceci est compatible avec la contrainte **prod(r) = prod(r_1) + 1**, si ce n'est pour le cas prod(r_1) = 1 et prod(r) = 2. De manière générale il n'est donc pas possible d'utiliser l'hypothèse d'induction fact(n-1) pour construire fact(n). Détecter ce cas permet donc de ne pas développer une preuve qui sera, de toute façon, inintéressante.

Cet exemple est traité au complet dans [BCF 84].

fin-exemple.

Une demande d'hypothèse aura donc les paramètres suivants :

- le but à prouver ;
- les hypothèses disponibles ;
- les hypothèses qui doivent être utilisées ;
- les variables désignant les données du problème qui, si possible, ne doivent pas être instanciées ;
- les variables désignant les résultats qui elles, par contre, peuvent être instanciées ;
- l'ensemble V des variables sur lesquelles l'hypothèse doit porter ;
- le type de la demande.

Il y aura échec de la recherche d'hypothèses soit lorsqu'il n'est pas possible de trouver une hypothèse portant uniquement sur les variables de V soit lorsque les hypothèses qui doivent servir ne sont pas utilisées. Par la suite, nous noterons en gras les variables représentant des résultats et en standart celles représentant les données du problème. Nous distinguons différents types de demandes d'hypothèses qui sont :

- **Trouver une hypothèse.** Ces demandes concernent les cas où nous sommes intéressés par une hypothèse faisant intervenir un ensemble V de variables différent de celui du but à prouver ou lorsque certaines hypothèses doivent servir. Dans ce cas toute formule répondant à ces conditions suffit, même si elle n'est pas sous une forme la plus simple possible. Nous supposons que les seules simplifications effectuées automatiquement reposent sur les lois classiques du calcul propositionnel (a $\vee$ **vrai** $\equiv$ **vrai**, a $\vee$ **faux** $\equiv$ a, ...).

- **Simplifier si possible.** Dans ce cas l'ensemble V est le même que celui du but à prouver et il y a aucune hypothèse à utiliser. **Simplifier si possible** vise la recherche de formules les plus simples possibles afin d'alléger le travail du système. Dans ce cas nous pouvons envisager d'utiliser un ensemble de règles de simplification usuelles.

- **Simplifier.** Dans certains cas nous désirons la solution la plus simple possible en sachant que **vrai** et **faux** sont les formules les plus simples. Ce dernier type de demande semble, à priori, le plus difficile à traiter car il n'existe pas de critère permettant de juger si une formule peut ou non encore être simplifiée. **Simplifier** correspond donc à chercher à prouver ou a réfuter une formule.

1.4. Gestion des préconditions.

La précondition d'un énoncé décrit la classe des données pour laquelle une preuve est cherchée. Lors de l'élaboration de cette preuve nous supposons implicitement que cette précondition est pertinente c'est à dire que les caractéristiques particulières des données permettent d'obtenir une preuve adéquate qui ne conviendrait pas nécessairement pour une classe plus générale. Cette supposition intervient principalement lors de la preuve par construction d'une définition récursive : nous imposons que les appels récursifs introduits vérifient bien la précondition sans savoir à priori si cette précondition est absolument nécessaire. Cette restriction permet de rendre plus " efficace " le processus de recherche de preuve. En effet, si nous voulions généraliser le plus possible les preuves effectuées, il faudrait évaluer les parties de la précondition qui contribuent effectivement à la preuve élaborée. La recherche d'instances de propriétés serait alors plus complexe puisque non restreinte aux données vérifiant la précondition. Nous partons donc du principe que la précondition donnée par l'utilisateur est pertinente, c'est à dire qu'il a effectivement une raison de réduire le domaine d'application des fonctions cherchées : soit parce que l'énoncé ne pourrait pas être prouvé pour d'autres données, les fonctions cherchées n'étant pas totale, soit parce que, en se limitant à cette classe de données, le programme obtenu peut être plus efficace que dans un cadre plus général.

En ce qui concerne les préconditions dérivées par le système, les règles décrivent le plus petit ensemble pour lequel les fonctions auxiliaires doivent être construites. Nous avons qualifié ces préconditions comme les meilleures (chapitre 2, section 2.1) dans le sens où il est possible d'en déduire toute autre précondition adéquate, par la règle de la conséquence.

Les deux règles de construction de préconditions sont, rappelons le, les suivantes :

- Si P(x) est la précondition initiale et si la définition engendrée est f(x) ← **si** C(x) **alors** $f_1(x)$ **sinon** $f_2(x)$ alors C doit être définie par la précondition P(x), f_1 pour la précondition $P(x) \wedge Q_0(x)$ et f_2 pour la précondition $P(x) \wedge \neg Q_0(x)$. Q_0 est la propriété des données décrite par la fonction booléenne C.

- Si P(x) est la précondition initiale et si la définition engendrée est $f(x) \leftarrow g(f_1(x), \ldots, f_n(x))$ alors les fonctions f_i doivent être définies pour la précondition initiale P(x) et g pour la précondition $\exists x\ (P(x) \wedge Q(x, y_1, \ldots y_n))$ où Q est la postcondition caractérisant l'ensemble des fonctions f_i.

Dans le cas de la composition fonctionnelle, la précondition associée à la fonction de recomposition décrit le fait que cette fonction doit être définie au moins pour tous les n-uplets $(y_1, \ldots, y_n)$ qui peuvent être obtenus à partir des x par application des fonctions f_i. Cette précondition ne décrit donc pas seulement une propriété des arguments de la fonction g mais aussi l'historique de la preuve, c'est à dire d'où proviennent les nouvelles données y_i introduites.

Conserver ces dernières informations est utile puisqu'elles peuvent permettre de déduire des propriétés nécessaires à la suite de l'élaboration de la preuve. Néanmoins nous ne désirons pas les conserver dans la précondition, d'une part car elles ont peu de chances d'intervenir directement dans la preuve et d'autre part afin de pouvoir réutiliser ces preuves dans un autre contexte. Nous généralisons donc la précondition associée à la fonction g en ne conservant que les relations qui ne décrivent pas les liens entre les différentes variables introduites au cours de la preuve. Ces relations, quant à elles, sont mémorisées dans une rubrique supplémentaire associée aux énoncés, que nous appelons *trace*. La trace est donc propre à la preuve en cours de développement. Cette séparation est nécessaire en raison de l'hypothèse que nous avons faite consistant à supposer que les propriétés des données présentes dans la précondition sont pertinentes. Nous jugeons donc, pour chaque sous-problème engendré, les propriétés disponibles qui sont pertinentes.

Si nous désirions évaluer les propriétés de la préconditions qui interviennent effectivement dans la preuve, sans présumer à priori qu'elles sont toutes nécessaires, la précondition de chaque énoncé serait initialement **vrai** et toutes les hypothèses disponibles seraient dans la trace. Lorsque certaines de ces propriétés s'avéreraient nécessaires elles seraient alors ajoutées dans la précondition. La solution que nous avons choisie est un compromis entre garantir le plus possible la généralité des programmes engendrés ou ne s'intéresser qu'à la construction d'un programme pour le problème traité. Ce compromis permet de simplifier le processus de recherche des propriétés nécessaires au déroulement d'une preuve particulière.

La trace d'un énoncé E est héritée par tous les énoncés issus de E. Elle est mise à jour lors de l'introduction de composition fonctionnelle, de récursion ou lorsque certaines données des énoncés peuvent être simplifiées (stratégie **Simplifier**). Elle contient donc les liens existants entre les objets intermédiaires introduits par la preuve et les propriétés des données qui n'interviennent plus dans l'énoncé. Nous choisissons de représenter cette trace sous la forme d'une liste d'hypothèses à laquelle nous associons un graphe décrivant la dépendance existant entre les différentes variables introduites au cours de la preuve. Par exemple l'énoncé engendré pour la fonction de recomposition g, lors de la construction d'une composition fonctionnelle, est :

profil : $g(y_1 : t_1, \ldots, y_n : t_n) \rightarrow t$
préc : $R(y_1, \ldots, y_n)$
post : ...
trace : $T, P(x), L(x, y_1, \ldots, y_n)$

Rappelons qu'une postcondition est constituée d'une partie L liant les données et les résultats et d'une partie R décrivant les propriétés attendues des résultats. T est la trace associée à l'énoncé prouvé par construction d'une composition fonctionnelle. Le graphe de dépendance des variables est complété par l'arc :

$\{ y_1, \ldots, y_n \}$

$\{ x \}$

Il décrit le fait que les variables $y_1, \ldots, y_n$ sont calculées à partir du n-uplet x. Les nœuds du graphe sont donc les ensembles de variables successivement introduits au cours de la preuve. Il existe un arc de e_1 à e_2 si et seulement si les variables de e_1 sont obtenues à partir des variables de e_2. Représenter les liens entre les variables sous forme d'un graphe permettra de décomposer certaines demandes de recherche de propriétés en différentes étapes et donc de faciliter

leur recherche.
Exemple.

Lors de la synthèse du tri par insertion, présentée dans l'annexe 1, le système de preuve est amené à se demander si la formule suivante est un théorème :

$$\neg(n \leq car(z)) \wedge trace \Rightarrow car(z) \leq car(w)$$

Cette demande est relative à la construction de la conditionnelle de la fonction insertion. La trace disponible est alors : { perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w), dans(k, w), k=n, ord(z), z≠nil }.

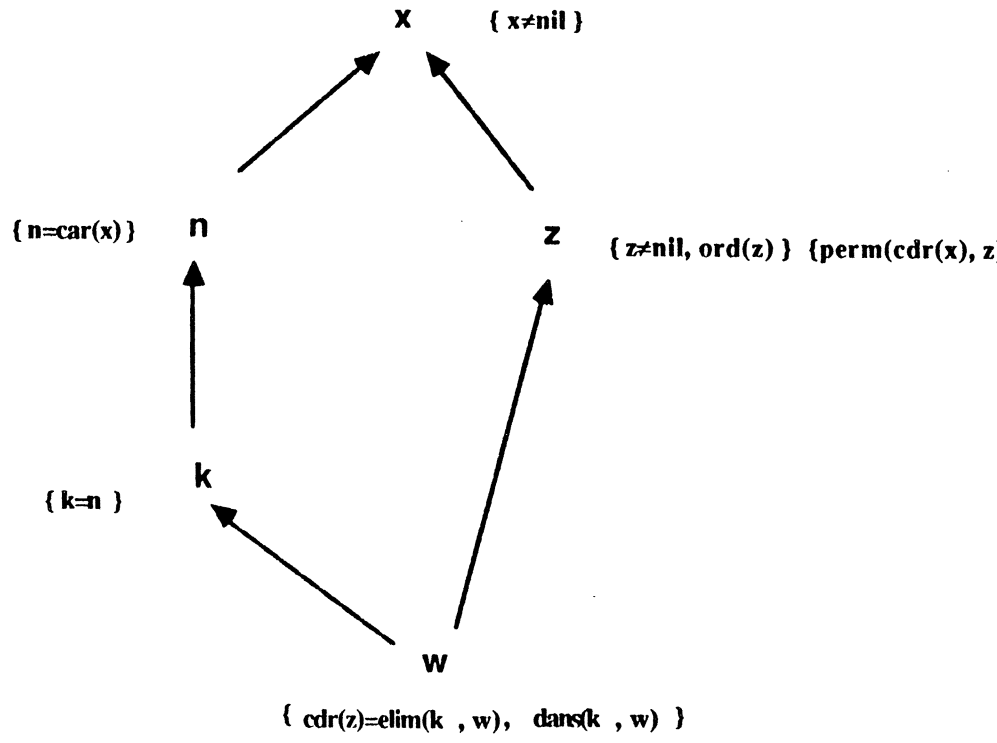
- x est la donnée initiale.

- z représente le résultat tri(cdr(x)) et est donc caractérisé par la relation perm(cdr(x), z) ∧ ord(z).

- w représente le résultat de l'insertion de car(x) dans z. Cette variable vérifie donc la relation cdr(z)=elim(k, w) ∧ ord(z).

- n et k sont des variables auxiliaires introduites au cours de la preuve, elles représentent toutes deux car(x).

Le graphe de dépendance sous-jacent à la preuve est :



Pour utiliser l'hypothèse n > car(z) il suffit de procéder de la manière suivante :

1 - ramener le but à prouver car(z) ≤ car(w) en terme des variables z et k en utilisant le lien existant entre w, z et k ;

2 - simplifier la formule obtenue en 1 à partir des propriétés de k et z ;

3 - ramener la formule obtenue en 2 en terme de n et z en utilisant le lien entre k et n ;

4 - simplifier la formule obtenue en 3 à l'aide de l'hypothèse **n > car(z)**.

Cette décomposition permet de prouver la formule donnée sans difficulté (voir annexe 1 page 13).

fin-exemple.

La preuve sera donc effectuée relativement à la précondition de l'énoncé afin de pouvoir réutiliser cette preuve. Chaque précondition dérivée devra ensuite être simplifiée dans le cadre du problème traité, c'est à dire en utilisant les hypothèses disponibles dans la **trace**. Si nous désirons pouvoir réutiliser les preuves dans un autre contexte nous devons alors gérer deux préconditions : une propre à l'application développée et l'autre ne prenant en compte que les hypothèses de la précondition.

2. LES STRATEGIES DE TRANSFORMATIONS.

Nous présentons maintenant les différentes stratégies que nous proposons. La première classe permet de transformer les énoncés en les simplifiant. En effet, un énoncé déduit par le système de synthèse caractérise souvent plusieurs fonctions. Lorsque ceci est possible, il est intéressant de décomposer cet énoncé en plusieurs énoncés caractérisant chacun un sous-ensemble de ces fonctions. Le processus de synthèse en est simplifié. Ces stratégies ne s'appuient pas sur les règles d'inférence données précédemment mais sur des lois classiques de déduction. La seconde classe de stratégies permet de construire une preuve des énoncés et repose sur le choix d'un schéma de programme à construire. Ce choix est fait par l'utilisateur, à partir des schémas primitifs, (fonctions de base, conditionnelle, composition fonctionnelle et récursion) qui doit en outre fixer certains paramètres de ces schémas de base. Ces paramètres peuvent correspondre aux indices qui permettent de choisir un schéma de programme plutôt qu'un autre, et leur choix revient à fixer certaines fonctions de ces schémas. Les stratégies que nous proposons sont donc attachées à des schémas de programmes simples mais il sera ensuite possible de définir des stratégies de preuve relatives à des schémas plus complexes en combinant ces stratégies de base. La récursion est déja un exemple puisqu'elle est construite à partir de la conditionnelle et de composition fonctionnelle.

Lors de la présentation des stratégies nous supposons que l'énoncé E à traité est :

profils	:	$var_1 = f_1(x : t_x) \rightarrow t_1, \ldots, var_n = f_n(x : t_x) \rightarrow t_n$
préc	:	P(x)
post	:	Q(x, var)
trace	:	T

var désigne par la suite l'ensemble des résultats $var_1, \ldots, var_n$.

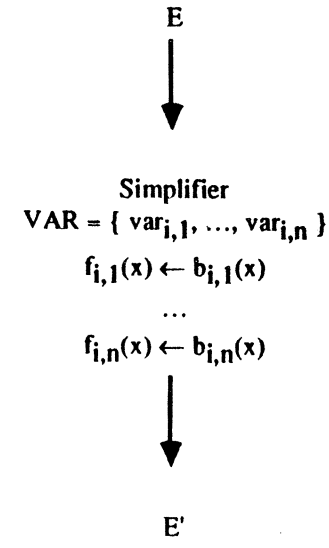
2.1. Stratégie Simplifier.

Cette stratégie permet de simplifier les énoncés et sera appliquée systématiquement à tout énoncé engendré par le système. En effet, ces énoncés sont déduits de manière systématique à partir de l'énoncé traité, de la stratégie appliquée et des réponses fournies par le module de traitement des demandes. Cette stratégie est basée sur la règle de simplification des arguments et permet de supprimer les objets qui n'interviennent pas dans la postcondition de l'énoncé.

Déroulement :

- Si une variable désignant une donnée n'apparait pas dans la postcondition alors d'une part elle est otée de la liste d'arguments des fonctions f_i et, d'autre part, toutes les relations de la précondition portant sur cette variable sont otées de la précondition et conservées dans la partie trace.

- Si une variable var_i désignant le résultat de la fonction f_i n'apparait pas dans la postcondition alors cette fonction est retirée du profil. Dans ce cas, n'importe quelle définition de f_i convient. Il est néanmoins nécessaire d'engendrer une définition, cette fonction ayant pu être introduite dans des définitions déjà élaborées. Ce cas dégénéré peut se produire lorsque le module de traitement des demandes renvoie des hypothèses permettant de résoudre le problème qui ne correspondent pas nécessairement à la preuve choisie par l'utilisateur. Certains sous-problèmes engendrés automatiquement n'auront alors pas besoin d'être résolus. Si de telles simplifications sont possibles, elles sont signalées à l'utilisateur et mémorisées dans l'arbre de la manière suivante :



VAR est l'ensemble des variables résultats qui n'apparaissent pas dans la postcondition et les fonctions $b_{i,j}$ correspondent aux définitions quelconques choisies. L'énoncé E' est obtenu à partir de E en otant du profil les fonctions qui n'ont plus de raison d'y apparaître.

fin-déroulement.

Cette stratégie permet d'alléger les énoncés manipulés par le système et d'obtenir des programmes propres c'est à dire tels que tous les arguments des fonctions sont nécessaires à leur calcul.

2.2. Stratégie Découper.

Cette stratégie permet de décomposer un énoncé suivant deux sous-ensembles de fonctions A_1 et A_2 qui peuvent être construits indépendamment l'un de l'autre.

Déroulement :

1. choix des fonctions de A_1. A_2 est alors l'ensemble des fonctions caractérisées dans l'énoncé qui n'appartiennent pas à A_1. Soient $\mathbf{var_a}$ l'ensemble des variables désignant les résultats des fonctions de A_1 et $\mathbf{var_b}$ l'ensemble des variables désignant les résultats des fonctions de A_2. Nous notons f_{a1}, ... f_{ak} les fonctions de A_1 et f_{b1}, ..., f_{bm} les autres.

2 . Demande de 2 hypothèses H_1 et H_2 telles que : $P(x) \wedge \mathbf{H_1}(x, \mathbf{var_a}) \wedge \mathbf{H_2}(x, \mathbf{var_b}) \Rightarrow Q(x, var)$.

Rappelons que les variables notées en gras désignent des résultats, elles peuvent donc être instanciées au cours de la recherche d'hypothèses. Dans cette requête les seules indications fournies pour la recherche d'hypothèses sont les variables que doivent lier les relations H_1 et H_2.

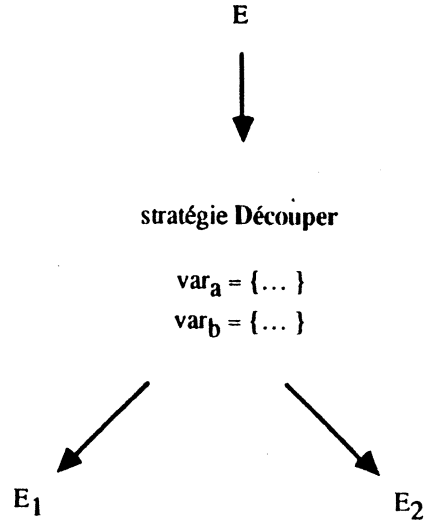
3 . Traitement de la réponse.

a). Si on ne peut trouver ni H_1 ni H_2 alors cette stratégie conduit à un échec.

b). Si H_1 a été trouvée mais que H_2 est une relation portant aussi sur des variables de var_a alors les fonctions f_{bi} dépendront des résultats des fonctions f_{ai}. Il y a alors échec de la stratégie **Découper**. Mais ceci est une indication pour appliquer la prochaine stratégie qui permet de décomposer un énoncé lorsqu'il existe un ordre de calcul sur les fonctions. Les fonctions à isoler seront f_{a1}, ..., f_{ak}.

c). Réciproquement si H_2 a été trouvée mais pas H_1 ceci est une indication pour d'appliquer la prochaine stratégie pour l'ensemble de fonctions f_{b1}, ..., f_{bm}.

d). Si H_1 et H_2 ont été trouvés le système construit l'arborescence suivante :



L'énoncé E_1 est :

profils : $var_{a1} = f_{a1}(x : t_x) \rightarrow t_{a1}$, ..., $var_{ak} = f_{ak}(x : t_x) \rightarrow t_{ak}$
préc : P(x)
post : $H_1(x, var_a)$
trace : T

L'énoncé E_2 est :

profils : $var_{b1} = f_{b1}(x : t_x) \rightarrow t_{b1}$,..., $var_{bm} = f_{bm}(x : t_x) \rightarrow t_{bm}$
préc : P(x)
post : $H_2(x, var_b)$
trace : T

4 . Ces deux énoncés peuvent être traités dans un ordre quelconque. L'utilisateur choisit par lequel commencer.Par exemple il peut être intéressant de prouver en premier le plus complexe afin d'être sûr de pouvoir le démontrer. Soit $C_1(x)$ la précondition dérivée lors de la preuve du premier énoncé choisi. Dans le cas où l'existence d'une précondition dérivée ne découle pas d'une stratégie en cours d'application alors si $C_1(x)$ se réduit à **faux** il y a échec total, sinon, si elle ne se réduit pas à **vrai**, l'utilisateur décide si il est quand même intéressant de continuer la preuve retenue.

5 . Preuve du second énoncé. Soit $C_2(x)$ la précondition dérivée.

6 . Elaboration de la précondition dérivée. Cette précondition est obtenue à partir de la formule $C_1(x) \wedge C_2(x)$. $C_1(x)$ et $C_2(x)$ sont supposées être les plus simplifiées possibles, vis à vis de la précondition et de la trace des énoncés E_1 et E_2. Il faut chercher à simplifier leur conjonction. Pour cela nous recherchons dans un premier temps les parties de C_2 qui ne sont pas impliquées par C_1.

- Demande d'une hypothèse simplificatrice H_1 telle que $\mathbf{H_1}(x) \wedge P(x) \wedge T \wedge C_1(x) \Rightarrow C_2(x)$

Par exemple si C_2 est une conséquence logique de C_1 l'hypothèse H_1 sera réduite à **vrai**. De la même manière nous cherchons à simplifier C_1.

- Demande d'une hypothèse simplificatrice H_2 telle que $\mathbf{H_2}(x) \wedge P(x) \wedge T \wedge H_1(x) \Rightarrow C_1(x)$

La précondition élaborée est $H_1(x) \wedge H_2(x)$. En effet nous avons bien $H_1(x) \wedge H_2(x) \Rightarrow C_1(x) \wedge C_2(x)$ en raison des deux implications ci-dessus et $H_1(x) \wedge H_2(x)$ est plus simple puisqu'elle ne contient plus de redondance.

fin-stratégie.

2.3. Stratégie Découper et Ordonner.

Cette stratégie permet d'isoler les fonctions qui se calculent directement à partir des données de celles qui se calculent à partir des résultats de ces premières fonctions. Cette stratégie est un cas particulier de construction de composition fonctionnelle, les fonctions intermédiaires étant choisies parmi l'ensemble des fonctions caractérisées dans l'énoncé.

déroulement :

1 . Choix des fonctions à isoler. Soient var_a l'ensemble des variables désignant les résultats de ces fonctions et var_b les variables désignant les résultats des autres fonctions.

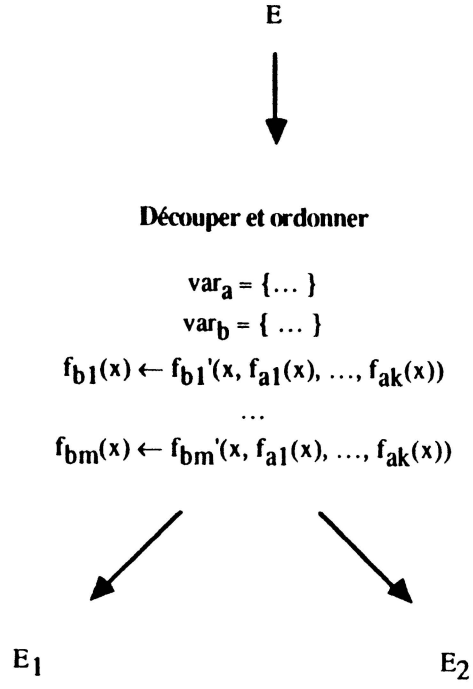
2 . Demande de 2 hypothèses H_1 et H_2 telles que $H_1(x, var_a) \wedge H_2(x, var) \Rightarrow Q(x, var)$.

Dans cette demande nous nous intéressons à la relation H_1, H_2 peut être la postcondition initiale Q. Il peut exister différentes solutions possibles pour H_1 qui caractérisent plus ou moins précisément les résultats de l'ensemble var_a. Nous désirons celle qui caractérise le mieux possible ces résultats. Si H_1 et H_1' sont deux solutions telles que $H_1(x, var_a) \Rightarrow H_1'(x, var_a)$ nous préférerons H_1.

3 . Traitement de la réponse.

a). Si H_1 n'a pas été trouvée la stratégie **Découper et Ordonner** échoue.

b). Si H_1 a été trouvé le système engendre l'arborescence suivante :



L'énoncé E_1 est :

profils : $var_{a1} = f_{a1}(x : t_x) \rightarrow t_{a1}$, ..., $var_{ak} = f_{ak}(x : t_x) \rightarrow t_{ak}$
préc : $P(x)$
post : $H_1(x, var_a)$
trace : T

L'énoncé E_2 est :

profils : $var_{b1} = f_{b1}'(x : t_x, var_a : t_a) \rightarrow t_{b1}$, ..., $var_{bm} = f_{bm}'(x : t_x, var_a : t_a) \rightarrow t_{bm}$
préc : $P(x)$
post : $H_2(x, var)$
trace : $T, H_1(x, var_a)$

L'application de la stratégie **Simplifier** permettra d'éliminer du profil de E_2 les données de var_a qui ne sont pas nécessaires. Le graphe de dépendance des variables est complété par l'introduction du nœud { var_a } et par l'arc :

Le lien entre ces deux ensembles de variables est $H_1(x, var_a)$.

4 . L'énoncé E_2 est prouvé en premier. Soit $C_2(x, var_a)$ la précondition dérivée lors de cette preuve. Si elle se réduit à **faux** il y a échec. Si elle se réduit à **vrai** il n'y a rien à faire. Sinon, cette contrainte doit être ajoutée à la postcondition de l'énoncé E_1 afin que les résultats des fonctions f_{ai} vérifient bien cette propriété. L'énoncé E_1 devient :

profils : $var_{a1} = f_{a1}(x : t_x) \rightarrow t_{a1}$, ..., $var_{ak} = f_{ak}(x : t_x) \rightarrow t_{ak}$
préc : $P(x)$
post : $H_1(x, var_a) \wedge C_2(x, var_a)$
trace : T

Le fait que la précondition dérivée lors de la preuve de E_2 soit différente de **vrai** ne provoque pas d'échec. Le calcul d'une précondition fait donc partie de la stratégie **Découper et Ordonner**.

5 . Preuve de E_1. La précondition dérivée lors de cette preuve est la précondition dérivée pour l'application de la stratégie **Découper et Ordonner**.

fin-stratégie.

La stratégie **Découper** est un cas particulier de la stratégie **Découper et Ordonner**, la relation H_2 cherchée ne devant pas faire intervenir les résultats des fonctions f_{ai}.

Exemple.

Soit l'énoncé :

profils : min = **min**(x : liste(entier)) → entier ; reste = **reste**(x : liste(entier)) → liste(entier)
préc : x≠nil
post : perm(min . reste, x) ∧ inf(min, reste)

Il décrit les fonctions **min** et **reste** nécessaires à la construction du tri par sélection. Supposons que nous choisissons de calculer séparément ces deux fonctions, la fonction **reste** étant construite à partir de la fonction **min**. On applique alors la stratégie **Découper et Ordonner** avec var_a = {min} et var_b = {reste}.

Déroulement :

- Demande de 2 hypothèses H_1 et H_2 telles que :

$$H_1(x, \mathbf{min}) \wedge H_2(x, \mathbf{min}, \mathbf{reste}) \Rightarrow \mathbf{perm}(\mathbf{min} . \mathbf{reste}, x) \wedge \mathbf{inf}(\mathbf{min}, \mathbf{reste})$$

On suppose disposer des 2 théorèmes :

(1)- dans(min, x) ∧ perm(reste, elim(min, x)) ≡ perm(min . reste, x)
(2)- inf(min, x) ∧ perm(min.reste, x) ⇒ inf(min, reste)

Une réponse possible pour H_1 est **dans(min, x) ∧ inf(min, x)** et une réponse pour H_2 est **perm(reste, elim(min, x))**. Les deux nouveaux énoncés sont alors :

E_1 : **profil** : min = **min**(x: liste(entier)) → entier
préc : x≠nil
post : dans(min, x) ∧ inf(min, x)

E_2 : **profil** : reste = **reste'**(x: liste(entier), min: entier) → liste-entier
préc : x≠nil
post : perm(reste, elim(min, x))
trace : dans(min, x) ∧ inf(min, x)

reste(x) est définie par reste(x) ← reste'(x, min(x)). La fonction reste' peut être synthétisée par la définition reste'(x) ← elim(min, x). Or elim n'est définie que si **min** est effectivement un élément de x, il faut donc vérifier que cette condition est vraie. Ceci fait partie des hypothèses conservées dans la trace, l'énoncé E_1 n'est donc pas affecté par la synthèse de la fonction **reste'**.

fin-déroulement.

Si nous n'avions pas utilisé le théorème (2) nous aurions obtenu les deux solutions **dans(min, x)** et **perm(reste, elim(min, x))** $\wedge$ **inf(min, reste)**. La synthèse de la fonction reste' n'aurait alors aboutit que pour la précondition dérivée **inf(min, elim(min, x))** qui n'est pas déductible de la trace. Cette condition aurait été ajoutée à la postcondition de E_1 et nous aurions obtenu un énoncé équivalent à celui obtenu précédemment.

3. LES STRATEGIES DE PREUVE.

Les deux stratégies **Découper** et **Découper et Ordonner** permettent de décomposer un énoncé caractérisant plusieurs fonctions. En effet les stratégies de preuve que nous allons maintenant décrire constituent assez souvent un seul énoncé, relatif à un ensemble de fonctions qui ne dépendent pas nécessairement les unes des autres. Les stratégies de preuve permettent la synthèse par reconnaissance d'une fonction de base, par construction d'une composition fonctionnelle, d'une conditionnelle ou d'une récursion. Certaines de ces stratégies ne peuvent s'appliquer que sur l'ensemble complet des fonctions caractérisées par l'énoncé. C'est le cas par exemple des stratégies relatives à l'introduction d'une conditionnelle : soit les fonctions peuvent être construites séparément et l'énoncé a pu être décomposé à l'aide d'une des deux stratégies précédentes soit ces fonctions dépendent les unes des autres et l'analyse par cas porte nécessairement sur toutes ces fonctions.

Nous avons développé une synthèse complète du tri par insertion (Annexe 1) et du tri par sélection (Annexe 2). Toutes les stratégies que nous exposons sont mises en œuvre dans ces exemples. Ces exemples sont très classiques, nous les avons cependant choisis pour deux raisons. D'une part la synthèse d'un algorithme de tri engendre de nombreux sous-problèmes et nous nous sommes particulièrement intéressés à la gestion d'une preuve complète. D'autre part la synthèse complète de ces algorithmes est loin d'être simple et il nous a semblé intéressant de les détailler complètement.

Nous présentons maintenant les stratégies de preuve dans le cas de leur application à une seule fonction de l'énoncé, lorsqu'elles ne s'appliquent pas nécessairement sur l'ensemble complet des fonctions décrites par l'énoncé. La généralisation ne pose pas de difficulté. Dans la suite, la fonction sur laquelle la stratégie porte sera la fonction f_i dont le résultat est désigné par la variable var_i. **var_autre** désignera l'ensemble des résultats des fonctions sur lesquelles la stratégie choisie ne s'applique pas.

3.1. Stratégie d'introduction de fonction de base.

Rapelons que la formule $P(x) \rightarrow Q(x, f(x))$ est prouvée par la construction $f(x) \leftarrow f_0(x)$, où f_0 est une fonction de base, si et seulement si on peut prouver :

$$\mathbf{P_0}(x) \rightarrow \mathbf{Q_0}(x, f_0(x))$$
$$P(x) \Rightarrow \mathbf{P_0}(x)$$
$$P(x) \wedge \mathbf{Q_0}(x, y) \Rightarrow Q(x, y)$$

L'utilisation de cette stratégie nécessite de fixer la fonction de base qui sera reconnue. Il suffit de la nommer, le système en possédant une définition. Il reste alors seulement à trouver les hypothèses telles que les deux implications ci-dessus soient des théorèmes, les relations P_0 et Q_0 étant connues.

Déroulement :

1 . choix de la fonction de base f_0 et de ses arguments. Son énoncé est :

profil : $r = f_0(x_0 : t_0) \rightarrow t$
préc : $P_0(x_0)$
post : $Q_0(x_0, r)$

Soit X la liste d'arguments de f_0 établie à partir du n-uplet initial x. Nous notons t_X le type de cet argument, déduit à partir du type t_x de x. Le système vérifie si la fonction f_0 peut être une solution possible pour f_i en comparant leur profil. Puisque les fonctions peuvent être génériques, cela revient à calculer si $t_X \rightarrow t_i$ est une instance du profil $t_0 \rightarrow t$. Si c'est le cas alors f_0 peut convenir, sinon il y a échec total de la stratégie.

2 . Si P_0 ne se réduit pas à **vrai** alors il y a demande d'une hypothèse H_1 telle que $\mathbf{H_1}(x) \wedge P(x) \Rightarrow P_0(X)$.

La propriété P(x) doit être utilisée lors de la preuve de $P_0(X)$. Ceci garantit que les domaines de définition de f_i et de f_0 ont une intersection non vide.

3 . Traitement de la réponse.

a). Si H_1 n'a pas été trouvé il y a échec. Ce cas correspond à la réponse $P_0(X)$, c'est à dire lorsque l'hypothèse initiale du problème à traiter ne permet en aucune sorte de démontrer la propriété P_0 attendue pour la reconnaissance de la fonction f_0.

b). Si H_1 a été trouvé, f_0 ne peut s'appliquer qu'aux données vérifiant cette propriété. Si elle n'est pas réduite à **vrai** et si l'énoncé traité ne provient pas du choix d'une stratégie basée sur le calcul d'une précondition dérivée il y a échec partiel. Le système demande alors à l'utilisateur si cette restriction lui convient. En cas de réponse négative il y a échec.

4 . Demande d'une hypothèse H_2 telle que $\mathbf{H_2}(x, var) \wedge H_1(x) \wedge P(x) \wedge Q_0(X, var_i) \Rightarrow Q(x, var)$.

Dans le cas de la reconnaissance d'une fonction de base nous admettons que la précondition dérivée dépende de la fonction de base reconnue. L'hypothèse H_2 cherchée peut donc porter sur toutes les variables résultats du n-uplet **var**. Nous imposons néanmoins que la postcondition $Q_0(X, var_i)$ de la fonction f_0 intervienne lors de la recherche de l'hypothèse H_2.

5 . Traitement de la réponse.

a). Si H_2 n'a pas été trouvée il y a échec.

b). Si H_2 est trouvée alors la fonction $f_0(X)$ n'est solution que pour les x tels que $H_1(x) \wedge H_2(x, var)$ $[var_i / f_0(x)]$.

6.1. Si il y a eu échec alors on peut modifier la postcondition de l'énoncé E en $Q(x, var) \wedge var_i \neq f_0(X)$.

- Demande d'une hypothèse simplificatrice Q_1 telle que $P(x) \wedge Q_1(x, var) \wedge var_i \neq f_0(X) \Rightarrow Q(x, var)$

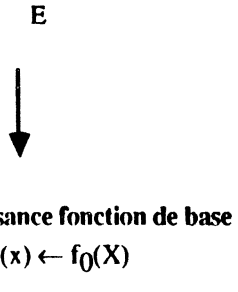
Si il y a eu simplification, c'est à dire si l'hypothèse $var_i \neq f_0(X)$ a été utilisée, l'arbre construit est :

E' est obtenu en remplaçant la postcondition initiale Q par Q_1.

6.2. Si il n'y a pas eu d'échec et si l'énoncé traité ne définit que la fonction f_i alors il n'y a pas de sous-problème engendré. Dans ce cas la précondition obtenue doit être simplifiée le plus possible.

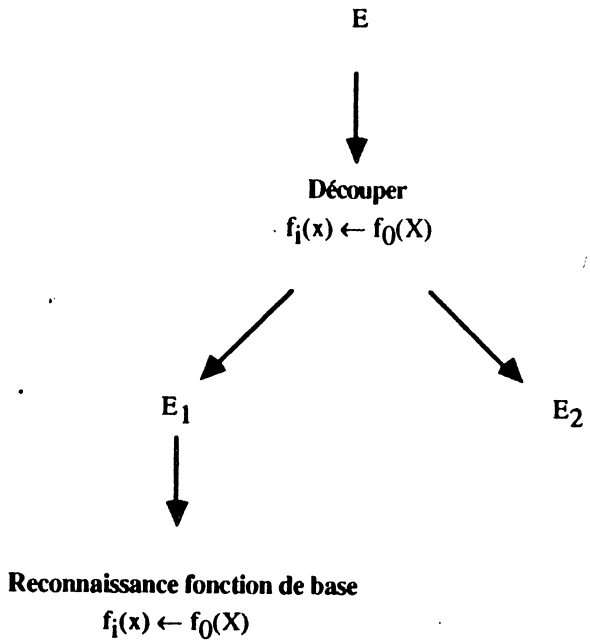
- Demande d'une hypothèse simplificatrice H_3 telle que $H_3(x) \wedge T \wedge P(x) \Rightarrow H_1(x) \wedge H_2(x, f_0(x))$

Rappelons que T est l'ensemble des hypothèses disponibles dans la trace de l'énoncé traité. L'arborescence construite est alors :



La précondition dérivée est $H_3(x)$.

6.3. Si E caractérise d'autres fonctions que f_i l'arbre engendré est :



L'énoncé E_1 relatif à la fonction f_i est l'énoncé de la fonction de base f_0 dans lequel f_0 est remplacé par f_i. L'énoncé E_2 est :

profils	:	$var\_autre_1 = f_1(x : t_x) \rightarrow t_1, \ldots,$
		$var\_autre_n = f_n(x : t_x) \rightarrow t_n$
préc	:	$P(x) \wedge H_1(x)$
post	:	$H_3(x, var)\ [var_i / f_0(x)]$
trace	:	T

Rappelons que var_autre désigne l'ensemble var \ $\{var_i\}$.

Il y a alors appel de la stratégie **Découper** vue précédemment, la fonction à isoler étant f_i. Il ne reste donc qu'à exécuter la dernière étape de cette stratégie, c'est à dire à prouver E_2.

Preuve de E_2. Soit $H_4(x)$ la précondition dérivée lors de cette preuve. La précondition dérivée pour l'appel de la stratégie **Découper** introduit est alors $H_1(x) \wedge H_4(x)$. Cette conjonction n'a pas besoin d'être simplifiée puisque H_4 a été élaborée en connaissant H_1.

fin-stratégie.

Appliquer cette stratégie sur plusieurs fonctions à la fois n'a d'intérêt que si les fonctions de base reconnues sont définies par un même énoncé. Il faut alors donner, pour chacune des fonctions sur lesquelles appliquer cette stratégie, les fonctions de base correspondantes et leurs arguments. Le déroulement de la stratégie est exactement le même que celui décrit ci-dessus.

Smith [Smi 82b] propose une stratégie pour la reconnaissance d'une fonction de base, fondée aussi sur le choix de la fonction à reconnaître. Il distingue deux cas : la reconnaissance a lieu en utilisant soit la postcondition de la fonction reconnue soit directement son nom. Dans ce second cas le pas 4 de la stratégie que nous proposons deviendrait être modifié en :

- Demande d'une hypothèse H_2 telle que $H_2(x, var) \wedge H_1(x) \wedge P(x) \wedge var_i=f_0(X) \Rightarrow Q(x, var)$.

Smith distingue ces deux cas parce qu'il différencie les fonctions de base qui peuvent être spécifiées par une postcondition en terme d'opérateurs plus primitifs et celles pour lesquelles ceci n'est pas possible (par exemple les constructeurs). Pour notre part nous n'avons pas ce problème. Toute fonction est caractérisée par une postcondition qui peut être soit une relation quelconque décrivant le lien entre les données et les résultats, soit une égalité de la forme $f_0(x_0) = f_0(x_0)$ lorsqu'il n'est pas possible de faire mieux. Néanmoins le problème soulevé par Smith se pose de manière plus générale lorsque la base de connaissances doit trouver des hypothèses nécessaires à la preuve de formules contenant une occurrence d'une fonction connue du système. En effet elles sont connues par le système ceci signifie que la base de théorèmes peut contenir, en plus d'un énoncé caractérisant ces fonctions, des théorèmes portant sur ces fonctions, par exemple ceux nécessaires au problème traité. Il peut donc ne pas être nécessaire d'utiliser les définitions de ces fonctions. C'est le module de traitement des demandes qui devra décider, à partir des connaissances explicites contenues dans la base de théorèmes, si il faut ou non utiliser les énoncés de ces fonctions. Au niveau du système de preuve la postcondition retenue pour les fonctions connue du sytème sera toujours réduite à l'égalité $f_0(x_0) = f_0(x_0)$.

Exemple. Supposons que nous devons construire une fonction vérifiant l'énoncé :

profil	:	var=**f**(x : liste(t)) → t
prec	:	x≠nil
post	:	dans(var, x)

La fonction de base **car** est une solution possible. Pour démontrer cela il y a demande d'une hypothèse H en terme de x telle que la formule **x≠nil ∧ var=car(x) ⇒ dans(var, x)** est vraie. **car** étant une fonction prédéfinie, la base de théorèmes peut contenir l'équivalence **x≠nil ≡ dans(car(x), x)**, qui permet de déduire directement l'hypothèse **vrai**. Par contre si ce théorème n'est pas connu explicitement le module de traitement des demandes devra chercher à l'inférer à partir des définitions de la fonction **car** et du prédicat **dans**, qui sont nécessairement connues. Supposons que ces fonctions sont caractérisées par les énoncés :

profils	:	r_1 = **car**(x : liste(t)) → t ; r_2 = **cdr**(x : liste(t)) → liste(t)
prec	:	x≠nil
post	:	$x = r_1 . r_2$
profil	:	r = **dans**(n : t , x : liste(t)) → booléen
prec	:	vrai
post	:	$(r=vrai) \equiv \exists y_1, y_2 (x = y_1 * (n . y_2))$

$*$ désigne la fonction **append** définie par exemple par :

$$y_1 * y_2 \leftarrow \text{si } y_1=nil \text{ alors } y_2 \text{ sinon } car(y_1) . (cdr(y_1) * y_2)$$

Trouver H permettant de prouver l'implication **x≠nil ∧ var=car(x) ⇒ dans(var, x)** revient a trouver les hypothèses nécessaires à la preuve de :

$$x \neq nil \wedge x = var \,.\, r_2 \Rightarrow \exists\, y_1, y_2\, (x = y_1 * (var \,.\, y_2))$$

nil et r_2 sont des instances possibles de y_1 et y_2 et donc H(x) se réduit à **vrai**. Cet exemple simple illustre bien la complexité de la recherche d'hypothèses lorsqu'il n'est pas supposé que tous les théorèmes nécessaires sont directement disponibles. Il faudra alors raisonner sur la connaissance explicite pour savoir comment trouver les hypothèses manquantes.

fin-exemple.

3.2. Stratégies d'introduction d'une Composition Fonctionnelle.

La formule $P(x) \rightarrow Q(x, f(x))$ est prouvée par une construction de la forme $f(x) \leftarrow g(h_1(x), ..., h_k(x))$ si et seulement si on peut prouver :

$$P(x) \rightarrow Q_i(x, h_i(x))$$
$$\exists x\, (P(x) \wedge \Pi_{(i=1,\,k)}\, Q_i(x, y_i)) \rightarrow Q_g(y_1, ..., y_k, g(y_1, ..., y_k))$$
$$P(x) \wedge \Pi_{(i=1,\,k)}\, Q_i(x, y_i) \wedge Q_g(y_1, ..., y_k, z) \Rightarrow Q(x, z)$$

Les stratégies proposées consiste soit à fixer la fonction de recomposition g (stratégie Instanciation de la fonction de recomposition), soit les fonctions intermédiaires h_i (stratégie Instanciation des fonctions intermédiaires). La stratégie **Découper et Ordonner** est un cas particulier de cette seconde stratégie : les fonctions h_i sont choisies parmi l'ensemble des fonctions caractérisées par l'énoncé en cours de traitement. Il en est de même de la stratégie **Reconnaissance d'une fonction de base.**

3.2.1. Stratégie Instanciation de la fonction de recomposition.

Déroulement :

1 . choix de la fonction g. Ce choix peut se faire parmi l'ensemble des fonctions de base ou à partir d'un énoncé donné par l'utilisateur et qui a déjà été prouvé. Supposons que l'énoncé de g soit :

profil : $z = g(y_1 : r_1, ..., y_k : r_k) \rightarrow t_g$
préc : $P_g(y_1, ..., y_k)$
post : $Q_g(y_1, ..., y_k, z)$

Ce choix fixe le nombre de fonctions auxiliaires h_i à k. Le système vérifie que le type de la fonction cherchée f_i est bien un cas particulier du type de la fonction g. Si ce n'est pas le cas il y a échec, sinon soit σ la substitution obtenue.

2 . Demande d'une hypothèse H_1 telle que :

$$P(x) \wedge H_1(x, y_1, \ldots, y_k, \text{var\_autre}) \wedge Q_g(y_1, \ldots, y_k, var_i) \Rightarrow Q(x, var)$$

L'hypothèse H_1 ne doit plus dépendre de la variable résultat var_i, celle-ci étant calculée directement à partir de $y_1, \ldots, y_k$ par la fonction g. Un guide pour la recherche de cette hypothèse est donc de faire disparaître la variable var_i en utilisant le lien Q_g donné. Il y a maintenant n-1+k résultats à calculer qui peuvent être instanciés lors de la recherche de H_1, les n-1 fonctions de l'énoncé initial non touchées par la stratégie et les k fonctions introduites par la décomposition.

3 . Traitement de la réponse.

a)- Si H_1 n'a pas été trouvée il y a échec.

b)- Si H_1 a été trouvée il reste à montrer que les résultats des fonctions h_i sont bien tels que P_g est vrai. Cette formule ne dépend que des variables y_i, elle peut donc être incluse directement dans la postcondition relative aux fonctions h_i. Nous essayons néanmoins de la simplifier.

- Demande d'une hypothèse simplificatrice H_2 telle que :

$$H_2(x, y_1, \ldots, y_k, \text{var\_autre}) \wedge P(x) \wedge H_1(x, y_1, \ldots, y_k, \text{var\_autre}) \Rightarrow P_g(y_1, \ldots, y_k)$$

Il n'y a pas de traitement de la réponse car $P_g(y_1, \ldots, y_k)$ est toujours une solution possible. L'arbre construit est alors :

Composition fonctionnelle
$f_i(x) \leftarrow g(h_1(x), \ldots, h_k(x))$

E_1

Avec E_1 : **profils** : $y_i = h_i(x: t_x) \rightarrow r_i\sigma$,
$\text{var\_autre}_i = f_i(x : t_x) \rightarrow t_i$
préc : $P(x)$
post : $H_1(x, y_1, \ldots, y_k, var_1, \ldots, \text{var\_autre}) \wedge H_2(x, y_1, \ldots, y_k, \text{var\_autre})$
trace : T

Les fonctions qui restent à construire sont décrites par un seul énoncé. Rappelons que σ est la substitution

obtenue après filtrage du type de la fonction de recomposition choisie et du type de la fonction f_i à construire.

4 . Preuve de E_1. La précondition dérivée lors de cette preuve est la précondition dérivée par l'application de la stratégie **Instanciation de la fonction de recomposition.**

fin-déroulement.

Nous avons décrit l'application de cette stratégie sur une seule fonction de l'énoncé, la fonction f_i. Son application à plusieurs fonctions est la même. Notons que l'application de cette stratégie sur plusieurs fonctions ne se justifie que si les fonctions intermédiaires engendrées peuvent être factorisées, ceci dans le but de simplifier et la synthèse et le nombre d'appels d'une même fonction pour les mêmes arguments.

3.2.2. Stratégie Instanciation des fonctions intermédiaires.

L'utilisateur choisit k fonctions qui permettront le calcul du résultat par recomposition. La première étape de cette stratégie est analogue à la reconnaissance d'une fonction de base, qui en est un cas particulier. En effet lorsque k est égal à 1 et lorsque la fonction de recomposition est l'identité alors la composition fonctionnelle se ramène à la reconnaissance de la fonction choisie.

Déroulement :

1 . Choix des k fonctions h_i et de leurs arguments. Soient $x_1, \ldots, x_k$ ces arguments choisis parmi les arguments initiaux x. Soient $\sigma_1, \ldots, \sigma_k$ les k substitutions obtenues lors de la vérification de la compatibilité de type entre les profils des fonctions h_i et les n-uplets x_i choisis. Supposons d'autre part que l'énoncé caractérisant les fonctions retenues soit :

profil	:	$y_1 = h_1(x_1 : t_{x1}) \rightarrow r_1, \ldots,$
		$y_k = h_k(x_k : t_{xk}) \rightarrow r_k$
préc	:	$P_1(x)$
post	:	$Q_1(x, y_1, \ldots, y_k)$

Nous supposons ici que les k fonctions sélectionnées sont décrites par un seul et même énoncé. Cette supposition n'est pas contraignante puisque la constitution d'un énoncé unique, à partir de plusieurs énoncés caractérisant différentes fonctions, ne pose pas de problème. Une fois les arguments communs fixés, la précondition est la simplification de la conjonction de toutes les préconditions et la postcondition est la conjonction des postconditions de chaque énoncé.

2 . Demande d'une hypothèse H_1 telle que $H_1(x) \wedge P(x) \Rightarrow P_1(x)$ lorsque P_1 ne se réduit pas à vrai.

Comme lors de la reconnaissance d'une fonction de base, la précondition initiale P doit intervenir au cours

de la recherche de H_1. Si H_1 n'a pas été trouvée il y a échec. Sinon, si elle ne se réduit pas à vrai et que l'existence d'une précondition dérivée ne découle pas d'une stratégie en cours de développement, le système demande à l'utilisateur si il faut continuer le traitement. En cas de réponse négative il y a échec.

3 . Demande d'une hypothèse H_2 telle que :

$$P(x) \wedge H_1(x) \wedge Q_1(x, y_1, \ldots, y_k) \wedge H_2(x, y_1, \ldots, y_k, var) \Rightarrow Q(x, var)$$

Ici il faut faire apparaître les variables $y_1, \ldots, y_k$ dans la formule cherchée, à l'aide de la relation Q_1. La donnée initiale x peut encore être une donnée de la fonction de recomposition, les fonctions h_i choisies pouvant ne pas constituer l'ensemble complet des fonctions intermédiaires nécessaires.

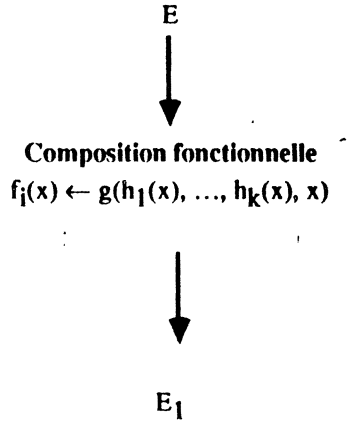
4 . Traitement de la réponse.

a) Si H_2 n'est pas trouvé il y a un échec.

b) Si H_2 est trouvé alors il y a 2 cas à distinguer. Soit la stratégie s'applique sur toutes les fonctions caractérisées par l'énoncé soit sur un sous-ensemble. Dans ce dernier cas il n'est pas possible de construire un seul énoncé décrivant la fonction de recomposition engendrée et les fonctions f_j qu'il reste à construire. En effet ces dernières ont comme seul argument le n-uplet initial x alors que la fonction de recomposition introduite a comme données $y_1, \ldots, y_k$ et éventuellement x. Or les données y_i ne sont pas accessibles lors du calcul des fonctions f_j, puisque f_j ne fait pas partie des fonctions pour lesquelles une définition de la forme $f_i(x) \leftarrow g(h_1(x), \ldots, h_k(x), x)$ est engendrée. Il faut donc constituer deux énoncés distincts. Pour cela nous faisons appel à la stratégie **Découper et Ordonner**.

5 . Cas où la stratégie s'applique sur toutes les fonctions décrites par l'énoncé.

5.1. L'arbre engendré est :

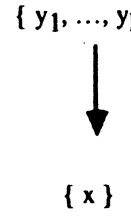


E_1 :

profil	:	$var_i = g(y_1 : r_1\sigma_1, \ldots, y_k : r_k\sigma_k, x : t_x) \rightarrow t_i$
préc	:	$P(x) \wedge H_1(x) \wedge R_1(y_1, \ldots, y_k)$
post	:	$H_2(x, y_1, \ldots, y_k, var_i)$
trace	:	$T, L_1(x, y_1, \ldots, y_k)$

L_1 et R_1 désignent respectivement les liens et les propriétés des résultats y_i de la postcondition Q_1 choisie au pas 1. Rappelons que nous décrivons les stratégies pour leur application à une seule fonction. Dans ce cas il n'y a donc plus qu'une seule fonction à construire, la fonction de recomposition g.

Le graphe de dépendance des variables est complété par le nœud $\{ y_1, ..., y_k \}$ et par l'arc :



5.2. Preuve de E_1. Soit $C_1(x, y_1, ..., y_k)$ la précondition dérivée au cours de cette preuve. Si elle est réduite à faux il y a échec. Sinon, pour établir la précondition dérivée par l'application de la stratégie **Instanciation des fonctions intermédiaires**, il faut exprimer C_1 uniquement en terme du n-uplet x.

- Demande d'une hypothèse C telle que $C(x) \wedge P(x) \wedge H_1(x) \wedge Q_1(x, y_1, ..., y_k) \Rightarrow C_1(x, y_1, ..., y_k)$.

Si C(x) a été trouvé la précondition retournée est C(x), sinon c'est $C_1(x, y_1, ..., y_k)$ $[y_1 / h_1(x), ..., y_k / h_k(x)]$. Cette solution n'est pas très satisfaisante puisqu'elle peut porter sur les fonctions intermédiaires du programme synthétisé, mais dans ce cas nous ne pouvons pas faire mieux, à cause du manque de connaissances du système.

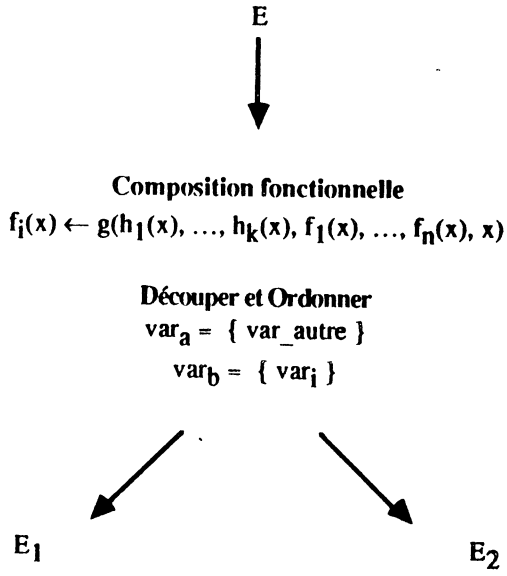
6 . Cas où la stratégie ne s'applique pas sur toutes les fonctions.

Pour obtenir deux énoncés distincts caractérisant d'une part la fonction de recomposition et d'autre part les fonctions f_j nous utilisons la stratégie **Découper et Ordonner** pour les instances var_a = { var_autre } et var_b = { var_i }. Rappelons que var_a désigne l'ensemble des fonctions qui auront comme seul argument la donnée initiale et que var_b désigne les fonctions qui peuvent utiliser les résultats de ces premières fonctions. g peut utiliser les résultats calculés par les fonctions f_j mais pas l'inverse puisque, nous l'avons déja dit, les arguments de g ne sont pas accessibles aux fonctions f_j. Il suffit donc d'exécuter la stratégie **Découper et Ordonner** pour les instances de var_a et var_b données ci-dessus, la différence étant que les données associées aux fonctions f_j et g ne sont pas les mêmes. Nous reprenons donc le déroulement de la stratégie **Découper et Ordonner** avec cette contrainte supplémentaire.

6.1 - Demande de deux hypothèses H_3 et H_4 telles que :

$$P(x) \wedge H_1(x) \wedge \mathbf{H_3}(x, \mathbf{var\_autre}) \wedge \mathbf{H_4}(x, y_1, ..., y_k, var\_autre, \mathbf{var_i}) \Rightarrow H_2(x, y_1, ..., y_k, var)$$

Comme précédemment, si H_3 n'est pas trouvé, il y a échec. Dans le cas contraire l'arbre construit est :



E_1 :

profils	:	$var\_autre_1 = f_1(x: t_x) \rightarrow t_1, \ldots,$
		$var\_autre_n = f_n(x: t_x) \rightarrow t_n,$
préc	:	$P(x) \wedge H_1(x)$
post	:	$H_3(x, var\_autre)$
trace	:	T

E_2 :

profil	:	$var_i = g(y_1: r_1\sigma_1, \ldots, y_k : r_k\sigma_k, x : t_x, var\_autre : t_{autre}) \rightarrow t_i$
préc	:	$P(x) \wedge H_1(x) \wedge R_1(y_1, \ldots, y_k)$
post	:	$H_4(x, y_1, \ldots, y_k, var)$
trace	:	$T, L_1(x, y_1, \ldots, y_k), H_3(x, var\_autre)$

Nous exécutons maintenant exactemement les mêmes étapes que celles de la stratégie **Découper et Ordonner.**

6. 2 . Preuve de E_2. Soit $C_2(y_1, \ldots, y_k, x)$ la précondition dérivée. Si elle n'est pas réduite à **faux**, ce qui provoque un échec, elle est ajoutée à la postcondition de E_1. L'existence d'une telle précondition découle donc de la stratégie utilisée.

6. 3 - Preuve de E_1. La précondition dérivée lors de cette preuvve est celle retournée par la stratégie **Instanciation des fonctions intermédiaires.**

fin-stratégie.

Si cette stratégie est appliquée sur n fonctions, les fonctions intermédiaires choisies doivent être les mêmes pour toutes ces fonctions. L'exécution de cette stratégie engendre alors n fonctions de recomposition qui sont toutes caractérisées dans le même énoncé E_2.

3.3. Stratégies d'introduction d'une Conditionnelle.

Rappelons que $P(x) \rightarrow Q(x, f(x))$ est démontré par la construction $f(x) \leftarrow$ **si** $C(x)$ **alors** $h_1(x)$ **sinon** $h_2(x)$ si et seulement si :

$$P(x) \rightarrow C(x)=vrai \equiv Q_0(x)$$
$$P(x) \wedge Q_0(x) \rightarrow Q_1(x, h_1(x))$$
$$P(x) \wedge \neg Q_0(x) \rightarrow Q_2(x, h_2(x))$$
$$P(x) \wedge Q_0(x) \wedge Q_1(x, y) \Rightarrow Q(x, y)$$
$$P(x) \wedge \neg Q_0(x) \wedge Q_2(x, y) \Rightarrow Q(x, y)$$

Les stratégies attachées à l'introduction d'une conditionnelle consiste soit à fixer la condition Q_0, soit à fixer l'une des postconditions des deux sous-problèmes caractérisant h_1 et h_2 soit les deux. Par exemple lors d'un échec partiel de la preuve d'un énoncé E nous avons introduit une conditionnelle en connaissant la condition : c'est la précondition dérivée lors de la preuve de E. Ce sera un cas particulier de la seconde stratégie que nous proposons. La dernière stratégie découle, par exemple, de l'utilisation de théorèmes définissant des relations par cas. La formule obtenue est sous forme disjonctive et peut être traitée comme la postcondition d'une conditionnelle. Ces stratégies ne s'appliquent que sur l'ensemble complet des fonctions décrites par l'énoncé. En effet, soit les fonctions peuvent être calculées séparément et dans ce cas il est possible d'appliquer une des stratégies **Découper** ou **Découper et Ordonner**, soit elles sont construites par la même analyse par cas. Nous décrivons donc ces stratégies pour l'ensemble complet des fonctions définies dans l'énoncé.

3.3.1. Stratégie Instanciation de la condition.

déroulement :

1 . Choix de la condition. L'utilisateur propose une fonction booléenne et ses arguments. Supposons que l'énoncé caractérisant cette fonction soit :

profil : cond=C(X : t_X) → booléen
prec : $P_0(X)$
post : (cond=vrai) ≡ $Q_0(X)$

2 . Si $P_0(x)$ ne se réduit pas à **vrai** alors il y a demande d'une hypothèse H_0 telle que $H_0(x) \wedge P(x) \Rightarrow P_0(X)$, sinon $H_0(x)$ est initialisé à **vrai**. Si H_0 n'a pas été trouvé il y a échec.

3 . Preuve de l'énoncé E_0 suivant :

profil : cond=C(X : t_X) → booléen
prec : $P(x) \wedge H_0(x)$
post : (cond=vrai) ≡ $Q_0(X)$

Soit $H_1(x)$ la précondition dérivée. Cette précondition fait partie de la précondition dérivée pour l'application de la stratégie **Instanciation de la condition**. Si $H_1(x)$ ne se réduit pas à vrai et si l'énoncé traité ne découle pas de l'application d'une stratégie basée sur le calcul d'une précondition dérivée, alors l'utilisateur décide si il faut continuer d'appliquer ou non cette stratégie. Si la réponse est négative il y a échec. Remarquons que si la condition choisie est connue du système cette étape n'est pas nécessaire, la précondition dérivée $H_1(x)$ est alors vrai.

$H_0(x) \wedge H_1(x)$ est la précondition nécessaire à la preuve de la fonction C pour le problème traité. Soit H(x) cette formule.

4 . Demande d'une hypothèse Q_1 telle que $H(x) \wedge P(x) \wedge Q_0(X) \wedge Q_1(x, var) \Rightarrow Q(x, var)$.

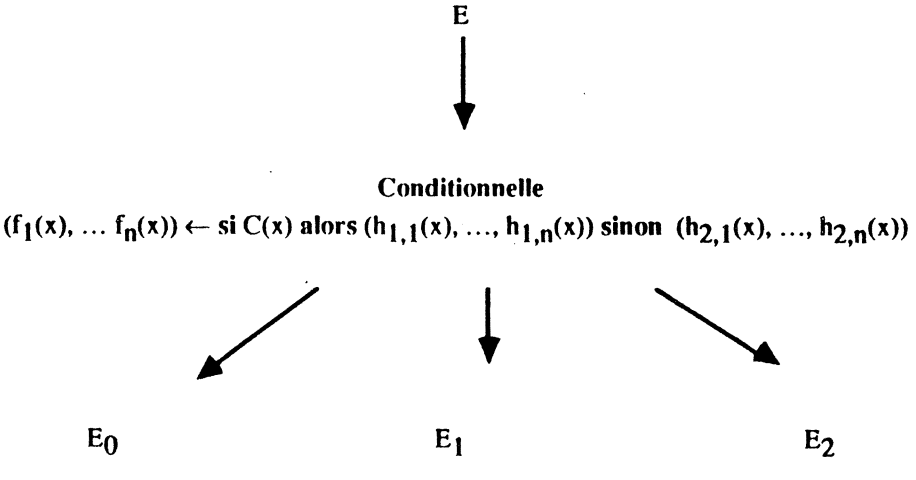
On impose ici que la propriété $Q_0(X)$ intervienne lors de la recherche de $Q_1(x, var)$. Ceci garantit que la condition choisie permet de transformer le problème initial, c'est à dire qu'il est effectivement intéressant d'introduire une telle conditionnelle.

5 . Si Q_1 n'a pas été trouvé il y a échec, sinon il y a demande d'une hypothèse Q_2 telle que :

$$H(x) \wedge P(x) \wedge \neg Q_0(X) \wedge Q_2(x, var) \Rightarrow Q(x, var)$$

De la même manière nous imposons que l'hypothèse $\neg Q_0(X)$ soit utilisée.

6. Si Q_2 n'a pas été trouvée il y a échec, sinon l'arborescence construite est :



E_0 est l'énoncé prouvé au pas 3. L'énoncé E_1 est :

profils : $var_1=h_{1,1}(x : t_x) \rightarrow t_1, \ldots, var_n=h_{1,n}(x : t_x) \rightarrow t_n$
préc : $P(x) \wedge H(x) \wedge Q_0(X)$
post : $Q_1(x, var)$
trace : T

L'énoncé E_2 est :

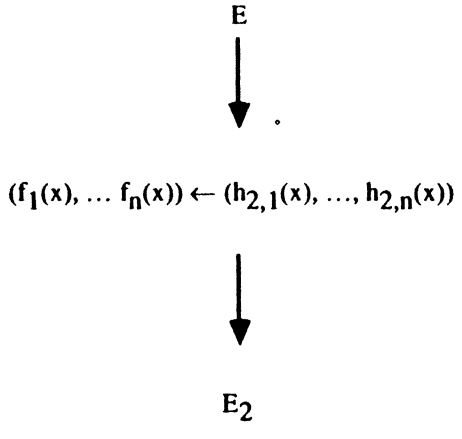
profil : $var_1=h_{2,1}(x : t_x) \rightarrow t_1, \ldots, var_1=h_{2,n}(x : t_x) \rightarrow t_n$
préc : $P(x) \wedge H(x) \wedge \neg Q_0(X)$
post : $Q_2(x, var)$
trace : T

7 . Preuve des énoncés E_1 et E_2. L'ordre de traitement est quelconque. Soient $C_1(x)$ et $C_2(x)$ les préconditions respectivement dérivées lors de ces preuves.

8 . Elaboration de la précondition dérivée pour la stratégie **Instanciation de la condition.**

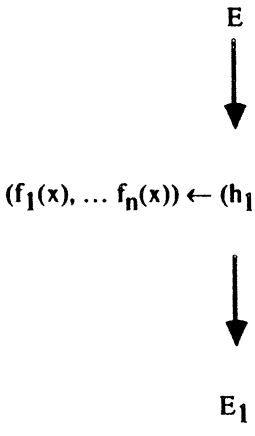
La précondition dérivée est $((Q_0(X) \wedge C_1(x)) \vee (\neg Q_0(X) \wedge C_2(x)) \wedge H(x))$, comme nous l'avons établi pour la règle d'inférence relative à la conditionnelle. Cette formule est sous sa forme la plus simple possible puisque C_1 et C_2 ont déjà été élaborées sous les hypothèses respectives $H(x) \wedge Q_0(X)$ et $H(x) \wedge \neg Q_0(X)$. Remarquons que si l'une des préconditions C_1 ou C_2 se réduit à **faux** alors la précondition dérivée ne se réduit pas nécessairement à **faux**. Ceci n'était pas le cas pour les stratégies que nous avons vues jusqu'à présent. Dès que l'un des sous-problèmes ne pouvait pas être prouvé l'énoncé traité ne pouvait pas être prouvé, lui non plus. Il est néanmoins possible, dans certains cas triviaux, de simplifier la preuve et le programme de la manière suivante :

a) - Si $C_1(x)$ se réduit à **faux** l'arbre de preuve est modifié en :



La précondition retournée est $\neg Q_0(X) \wedge C_2(x) \wedge H(x)$. En effet, dans ce cas les fonctions $h_{1,i}$ n'ayant pas pu être synthétisées il n'est pas nécessaire d'introduire une conditionnelle.

c) - Réciproquement si $C_2(x)$ se réduit à faux l'arbre de preuve est modifié en :



Et la précondition retournée est $Q_0(X) \wedge C_1(x) \wedge H(x)$.

fin-stratégie.

Les optimisations proposées sont succintes. Nous nous sommes intéressés uniquement aux simplifications qui peuvent être détectées à partir des préconditions dérivées. Il serait aussi possible par exemple de s'assurer que la condition introduite n'est pas toujours vraie ou toujours fausse par la demande d'une hypothèse R telle que :

$$R(x) \wedge P(x) \equiv Q_0(X)$$

Si R est **faux** alors il n'est pas nécessaire de chercher à synthétiser les fonctions $h_{1,i}$ et si R est **vrai** il n'est pas nécessaire de synthétiser les fonctions $h_{2,i}$.

3.3.2. Stratégie Instanciation d'une postcondition.

Cette stratégie repose sur le choix des fonctions calculées par la première branche de la condition. Celles-ci étant fixées, ceci permet d'un part d'élaborer la condition, ce sera la précondition dérivée lors de la synthèse de ces fonctions et d'autre part de trouver un énoncé caractérisant les fonctions $h_{2,i}$.

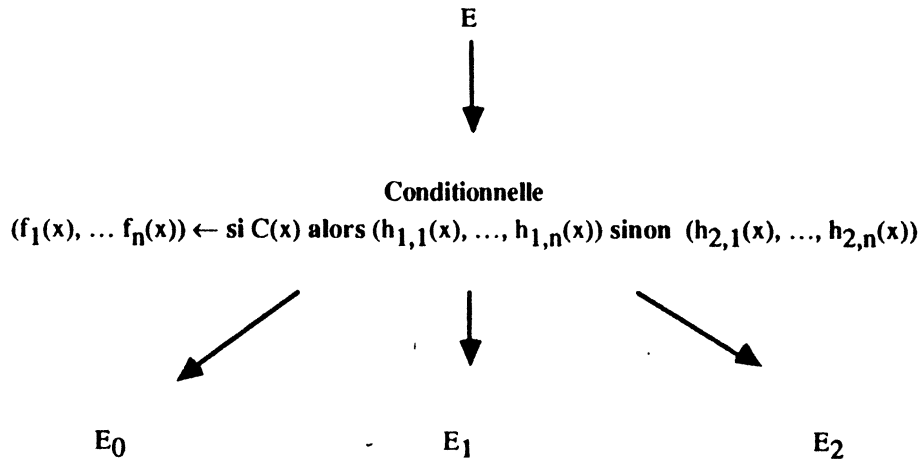
déroulement :

1 - choix de l'énoncé caractérisant les fonctions de la première branche de la conditionnelle. Supposons que cet énoncé soit l'énoncé E_1 suivant :

profils	:	$var_1 = h_{1,1}(x_1 : t_x) \rightarrow t_1, \ldots, var_n = h_{1,n}(x_n : t_x) \rightarrow t_n$
prec	:	$P_1(x)$
post	:	$Q_1(x, var)$

$x_1, \ldots, x_n$ sont les arguments des fonctions $h_{1,i}$, instanciés à partir de l'argument initial x.

2 . Application de la stratégie **Reconnaissance de fonctions de base**. Cette stratégie s'applique sur toutes les fonctions f_i de l'énoncé E et les fonctions reconnues sont les fonctions $h_{1,i}$ décrites par l'énoncé ci-dessus. Soit $H_1(x)$ la précondition dérivée, son existence découle de la stratégie en cours d'application. Si $H_1(x)$ se réduit à **faux** ,il y a échec. Sinon l'arbre construit est :



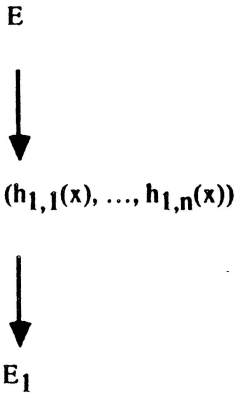
L'énoncé E_1 est :

profils	:	$var_1{=}h_{1,1}(x_1 : t_x) \rightarrow t_1, \ldots, var_n{=}h_{1,n}(x_n : t_x) \rightarrow t_n$
prec	:	$P_1(x) \wedge H_1(x)$
post	:	$Q_1(x, var)$
trace	:	T

Les énoncés E_0 et E_2 ne sont pas encore élaborés.

3 . Preuve de l'énoncé E_1. Soit $H_1'(x)$ la précondition dérivée lors de cette preuve. Si elle se réduit à **faux** il y a échec. Si E_1 est un énoncé déjà prouvé la précondition dérivée $H_1'(x)$ est alors initialisé à **vrai**. Soit $C_1(x)$ la formule $H_1(x) \wedge H_1'(x)$. Elle décrit la classe des données pour laquelle les fonctions choisies par l'utilisateur conviennent pour le problème traité.

4.1. Si cette conjonction se réduit à **vrai** alors l'arbre est modifié en :



La précondition retournée est vrai et il y a arrêt de l'application de la stratégie.

4 . 2. Déterminer la condition. La condition retenue est élaborée à partir de la précondition dérivée $C_1(x)$. L'utilisateur a la possibilité de choisir d'exprimer cette condition sous une autre forme dépendante d'un sous-ensemble du n-uplet initial x. En effet modifier les données de cette condition peut permettre une amélioration du programme. Ce'est le cas par exemple lors de la synthèse de la fonction **max**. La solution car(x) ne convient que lorsque SUP(car(x), cdr(x)) est vrai. Nous avons vu qu'il était préférable de ramener cette condition sous la forme car(z) ≥ z où z représente le résultat max(cdr(x)). Si l'utilisateur choisit de modifier cette condition, soit X l'ensemble des données qu'il a sélectionné. Il y a alors demande d'une hypothèse C_1' telle que $C_1'(X) \wedge T \wedge P(x) \Rightarrow C_1(x)$.

Cette demande peut être décomposée en différentes étapes, en s'aidant du graphe de dépendance des variables sous-jacent à la trace, comme nous l'avons évoqué dans la première partie de ce chapitre.

Si C_1' n'est pas trouvé l'utilisateur décide si l'application de cette stratégie doit continuer ou non. Si la condition $C_1(x)$ trouvé initialement lui convient alors $C_1'(X)$ est identifiée à $C_1(x)$.

5 . Demande d'une hypothèse Q_2 telle que $P(x) \wedge \neg C_1'(X) \wedge Q_2(x, var) \Rightarrow Q(x, var)$

La propriété $\neg C_1'(X)$ doit intervenir lors de la recherche de $Q_2(x, var)$. Ceci garantit qu'il est intéressant d'introduire la conditionnelle choisie.

6 . Si Q_2 n'a pas été trouvée il y a échec sinon les énoncés E_0 et E_2 élaborés sont :

E_0 :	**profil**	:	cond=C(X : t_X) → booléen
	préc	:	P(x)
	post	:	(cond=vrai) ≡ $C_1'(X)$
	trace	:	T

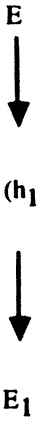
E_2 : **profil** : $var_1=h_{2,1}(x : t_x) \rightarrow t_1, \ldots, var_n=h_{2,n}(x : t_x) \rightarrow t_n$

préc : $P(x) \wedge \neg C_1'(X)$

post : $Q_2(x, var)$

trace : T

7 . Preuve de E_2. Soit $C_2(x)$ la précondition dérivée. Si elle se réduit à **faux** l'arbre est modifié en :



8 . Preuve de E_0. Soit H(x) la précondition dérivée.

9 . Elaboration de la précondition dérivée.

Elle s'obtient en simplifiant la formule $H(x) \wedge (C_1'(X) \vee (\neg C_1'(X) \wedge C_2(x)))$, c'est à dire $H(x) \wedge (C_1'(X) \vee C_2(x))$. Les préconditions C_1' et C_2, obtenues lors de la preuve des énoncés E_1 et E_2, peuvent être simplifiables par la précondition H.

- Demande d'une hypothèse simplificatrice D_1 telle que $H(x) \wedge D_1(x) \wedge T \Rightarrow C_1'(X)$

- Demande d'une hypothèse simplificatrice D_2 telle que $H(x) \wedge D_2(x) \wedge T \Rightarrow C_2(x)$

La précondition retournée est alors $H(x) \wedge (D_1(x) \vee D_2(x))$.

fin-stratégie.

C'est cette stratégie qui est appliquée lors du traitement d'un échec partiel. Les pas 1, 2 et 3 sont déjà exécutés, puisqu'ils correspondent à la preuve effectuée et que l'utilisateur a choisi de conserver. La précondition dérivée lors de cette preuve partielle correspond à la formule $C_1(x)$.

3.3.3. Stratégie Instanciation des postconditions.

Cette stratégie est fondée sur le choix des fonctions permettant le calcul du résultat dans les deux cas de la conditionnelle. Ceci permet de choisir la condition parmi les deux préconditions dérivées. Cette stratégie est souvent employée lorsque la postcondition est sous forme disjonctive, c'est à dire lorsque le lien entre les données et les résultats est explicitement défini par cas. Ceci se produit lorsque les énoncés sont dérivés par le système et que les connaissances utilisées sont elles mêmes décrites par cas. Il est alors possible de déterminer syntaxiquement les énoncés choisis.

déroulement :

1 - choix de l'énoncé caractérisant les fonctions de la première branche de la condition. Soit :

profils : $var_1=h_{1,1}(x : t_x) \rightarrow t_1, \ldots, var_n=h_{1,n}(x : t_x) \rightarrow t_n$
prec : $P_1(x)$
post : $Q_1(x, var)$

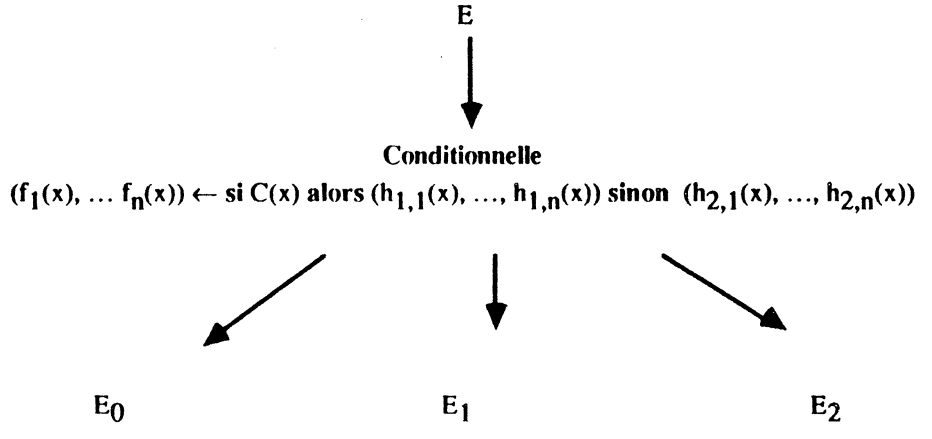
2 - choix de l'énoncé caractérisant les fonctions de la seconde branche de la condition. Soit :

profils : $var_1=h_{2,1}(x : t_x) \rightarrow t_1, \ldots, var_n= h_{2,n}(x : t_x) \rightarrow t_n$
prec : $P_2(x)$
post : $Q_2(x, var)$

3 . Application de la stratégie **Reconnaissance de fonctions de base** pour les fonctions f_i de l'énoncé à partir de l'énoncé E_1 fourni par l'utilisateur. Soit $H_1(x)$ la précondition dérivée obtenue, son existence découle de la stratégie en cours d'application. Si $H_1(x)$ est réduite à faux il y a remise en cause de la stratégie utilisée.

4 . Application de la stratégie **Reconnaissance de fonctions de base** pour les fonctions f_i de l'énoncé à partir de l'énoncé E_2 fourni par l'utilisateur. Soit $H_2(x)$ la précondition dérivée obtenue, son existence découle de la stratégie en cours d'application. Si $H_2(x)$ est réduite à faux il y a remise en cause de la stratégie utilisée.

5 . L'arbre construit est :



Avec :

E_1 : **profils** : $var_1 = h_{1,1}(x : t_x) \rightarrow t_1, \ldots, var_n = h_{1,n}(x : t_x) \rightarrow t_n$
prec : $P_1(x) \wedge H_1(x)$
post : $Q_1(x, var)$
trace : T

E_2 : **profils** : $var_1 = h_{2,1}(x : t_x) \rightarrow t_1, \ldots, var_n = h_{2,n}(x : t_x) \rightarrow t_n$
prec : $P_2(x) \wedge H_2(x)$
post : $Q_2(x, var)$
trace : T

6 . Si nécessaire, preuve des énoncés E_1 et E_2. Soient $H_1'(x)$ et $H_2'(x)$ les préconditions dérivées obtenues lors de ces preuves et soient $C_1(x)$ et $C_2(x)$ les formules $H_1(x) \wedge H_1'(x)$ et $H_2(x) \wedge H_2'(x)$.

a) Si $C_1(x)$ est **vrai** ou si $C_2(x)$ est **faux** l'arbre est modifié en :

$(f_1(x), \ldots f_n(x)) \leftarrow (h_{1,1}(x), \ldots, h_{1,n}(x))$

E_1

La précondition dérivée est alors $C_1(x)$ et il y a arrêt de l'application de cette stratégie.

b) Si $C_1(x)$ est **faux** ou si $C_2(x)$ est **vrai** l'arbre est modifié en :

$(f_1(x), \ldots f_n(x)) \leftarrow (h_{2,1}(x), \ldots, h_{2,n}(x))$

E_2

La précondition dérivée est $C_2(x)$ et il y a arrêt de l'application de cette stratégie.

7 . Déterminer la condition Q_0. Ce choix se fait à partir des deux préconditions obtenues $C_1(x)$ et $C_2(x)$. C'est l'utilisateur qui décide. Comme précédement il est possible de modifier cette condition en fonction des données sur lesquelles elle doit porter. Soit C'(X) la condition obtenue à partir de $C_1(x)$ ou de $C_2(x)$, suivant la condition retenue, et soit X les données choisies par l'utilisateur.

8 . Les fonctions f_i ont alors été synthétisées pour la précondition $C'(X) \vee C_2(x)$ ou $C_1(x) \vee C'(X)$ suivant que C' est issu de C_1 ou de C_2. Nous cherchons à simplifier cette disjonction.

8 . 1 . Si C' est issu de C_1 alors demande d'une hypothèse simplificatrice D telle que :

$$D(x) \wedge P(x) \wedge \neg C'(X) \wedge T \Rightarrow C_2(x)$$

Si D se réduit à **faux** ceci signifie que le domaine de définition des fonctions $h_{2,i}$ est inclus dans le domaine de définition des fonctions $h_{1,i}$ et donc que les fonctions $h_{2,i}$ ne seront jamais évaluées. Dans ce cas l'arbre est modifié en :

E

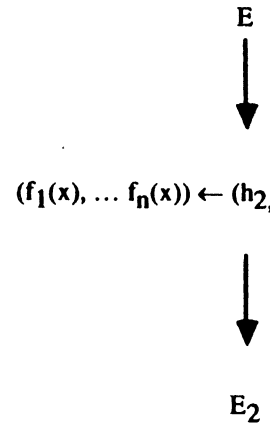
$$(f_1(x), \ldots f_n(x)) \leftarrow (h_{1,1}(x), \ldots, h_{1,n}(x))$$

E_1

8 . 2 . Si C' est issu de C_2 alors demande d'une hypothèse simplificatrice D telle que :

$$D(x) \wedge P(x) \wedge \neg C'(X) \wedge T \Rightarrow C_1(x)$$

De la même manière si D se réduit à **faux** l'arbre est modifié en :



9. Si D ne se réduit pas à **faux** alors l'énoncé E_0 élaboré est :

profil : cond=C(X : t_X) → booléen
préc : P(x)
post : (cond=vrai) ≡ C'(X)
trace : T

Preuve de cet énoncé. Soit H(x) la précondition dérivée.

10 . Elaboration de la précondition dérivée finale. Elle est obtenue à partir de la formule $H(x) \wedge (C'(X) \vee D(x))$. $C'(X) \vee D(x)$ doit alors être simplifié par la précondition H.

- Demande d'une hypothèse simplificatrice D_1 telle que $P(x) \wedge T \wedge H(x) \wedge \mathbf{D_1}(x) \Rightarrow C'(X)$.

- Demande d'une hypothèse simplificatrice D_2 telle que $P(x) \wedge T \wedge H(x) \wedge \mathbf{D_2}(x) \Rightarrow D(x)$.

La précondition retournée est alors $H(x) \wedge (D_1(x) \vee D_2(x))$.

. fin-stratégie.

Cette stratégie peut être utilisée lorsque la postcondition est de la forme $Q_1(x, var) \vee Q_2(x, var)$. Les énoncés choisis au pas 1 et 2 sont alors établis syntaxiquement, leur postcondition respective étant la première et la seconde partie de la disjonction. Dans ce cas les pas 3 et 4 sont triviaux, la preuve des deux énoncés obtenus permettant évidemment la preuve de l'énoncé initial.

Le pendant en logique de la construction conditionnelle est classiquement la disjonction. Dans le système LOPS l'introduction d'une conditionnelle est effectuée par la stratégie GET-DNF qui transforme une formule en une formule équivalente sous forme normale disjonctive. Smith [Smi 85], quant à lui, cherche à caractériser les cas dans lesquels il est intéressant de procéder à une analyse par cas. Il propose des tactiques pour introduire des disjonctions dans les énoncés, et donc pour construire des programmes conditionnels. Par exemple :

- si une donnée ou un résultat appartient à un domaine dont les éléments preuvent être caractérisés par un

ensemble fini $\{t_1, ..., t_n\}$ alors on peut introduire la disjonction $x=t_1 \vee ... \vee x=t_n$;

- si la postcondition ne peut être transformée que si la donnée x vérifie une certaine propriété R et que R(x) ne peut pas être démontrée ou réfutée alors introduire la disjonction $R(x) \vee \neg R(x)$ dans la postcondition.

3.4. Stratégies d'introduction de récursions.

Rappelons que la formule $P(x) \rightarrow Q(x, f(x))$ peut être démontrée par la construction :

$$f(x) \leftarrow \text{si } C(x) \text{ alors } f_0(x) \text{ sinon } g(f(h_{1,1}(x), ..., h_{1,n}(x)), ..., f(h_{k,1}(x), ..., h_{k,n}(x)), x)$$

si et seulement si :

$$P(x) \rightarrow (C(x)=\text{vrai}) \equiv Q_0(x)$$
$$P(x) \wedge Q_0(x) \rightarrow Q_1(x, f_0(x))$$
$$P(x) \wedge \neg Q_0(x) \rightarrow Q_{i,j}(x, h_{i,j}(x))$$
$$P(x) \wedge \neg Q_0(x) \wedge \exists y\, (\Pi_{(i=1, n; j=1, k)} (Q_{i,j}(x, w_{i,j}) \wedge Q(w_i, y_i))) \rightarrow Q_g(y, x, g(y, x))$$
$$P(x) \wedge \neg Q_0(x) \wedge \Pi_{(j=1, k)} Q_{i,j}(x, w_{i,j}) \Rightarrow P(w_i)$$
$$P(x) \wedge Q_0(x) \wedge Q_1(x, r) \Rightarrow Q(x, r)$$
$$P(x) \wedge \neg Q_0(x) \wedge \Pi_{(i=1, n)} (\Pi_{(j=1, k)} Q_{i,j}(x, w_{i,j}) \wedge Q(w_i, y_i))) \wedge Q_g(y, x, r) \Rightarrow Q(x, r)$$

Pour garantir en plus l'arrêt, il faudra trouver un ordre bien fondé **inf** tel que, pour tout i :

$$P(x) \wedge \neg Q_0(x) \wedge \Pi_{(j=1, k)} Q_{i,j}(x, w_{i,j}) \Rightarrow \mathbf{inf}(w_i, x)$$

La récursion est un schéma élaboré à partir de conditionnelle et de compositions fonctionnelles. En effet, la définition :

$$f(x) \leftarrow \text{si } C(x) \text{ alors } f_0(x) \text{ sinon } g(f(h_{1,1}(x), ..., h_{1,n}(x)), ..., f(h_{k,1}(x), ..., h_{k,n}(x)), x)$$

peut se décomposer en :

- une conditionnelle de la forme $f(x) \leftarrow \text{si } C(x) \text{ alors } f_0(x) \text{ sinon } g_1(x)$
- une composition fonctionnelle de la forme $g_1(x) \leftarrow g(k_1(x), ..., k_k(x), x)$
- k compositions fonctionnelles de la forme $k_i(x) \leftarrow f(h_{i,1}(x), ..., h_{i,n}(x))$

Les stratégies pour la récursion peuvent donc s'élaborer à partir des stratégies précédentes en sachant que la fonction de recomposition des k compositions fonctionnelles sont connues, c'est la fonction f. Ce schéma étant déjà plus complexe, les informations demandées à l'utilisateur doivent présenter un aspect cohérent, c'est à dire qu'il doit être plausible d'attendre de sa part qu'il puisse les fournir. Une stratégie associée à un tel schéma peut déjà être vue comme une tactique de programmation. Les deux stratégies que nous proposons reposent respectivement sur le choix :

- d'une construction du résultat basée sur une définition inductive du domaine du résultat.

- d'un sous-ensemble des données et d'une manière de décomposer ces données.

Choisir une construction du domaine du résultat revient à fixer les fonctions f_0 et g du schéma récursif et donc à cela revient à utiliser la stratégie introduction d'une composition fonctionnelle par **Instanciation des fonctions de recomposition**. Donner une décomposition des données revient à fixer la condition C et les fonctions $h_{i,j}$, pour le sous-ensemble des données sur lequel cette décomposition s'applique. Ceci revient à utiliser la stratégie introduction d'une composition fonctionnelle par **Instanciation des fonctions intermédiaires.**

Ces méthodes de programmation sont basées sur les notions de définition et de décomposition de domaines. Classiquement un domaine peut être décrit inductivement à l'aide d'un ensemble de fonctions appelées *Constructeurs* qui permettent d'engendrer, librement ou non, tous les éléments de ce domaine. Ces fonctions décrivent par exemple la structure des objets. Par exemple { nil, cons } et { nil, unit, append } sont deux ensembles de constructeurs permettant par exemple d'engendrer toutes les listes d'éléments de même type. Lors de la description de la stratégie **Construction du résultat**, basée sur le choix d'une définition de domaine, nous nous bornerons aux définitions inductives ne contenant que deux constructeurs dont l'un permet de construire des objets primitifs du domaine et le second permet de construire inductivement les objets de ce domaine. Une définition inductive d'un domaine D sera donc la donnée :

- d'une valeur primitive appartenant à ce domaine ;

- d'une fonction permettant d'obtenir des objets de ce domaine à partir d'autres objets appartenant, entre autres, à ce même domaine.

Il est plus difficile de caractériser la notion de décomposition d'un domaine. Smith décrit intuitivement une décomposition comme une fonction qui associe à un élément d'un type un élément de ce même type mais plus petit dans un ordre bien fondé. Cette définition est très générale. Il est possible de définir plus formellement une décomposition d'un domaine à partir d'une définition inductive de ce domaine, en utilisant des fonctions appelées *Sélecteurs* et *Indicateurs*. Les sélecteurs permettent d'accéder aux parties d'un objet structuré en terme de constructeurs et les indicateurs identifient, pour un objet donné, le dernier constructeur ayant contribué à la création de cet objet. Par exemple **car** et **cdr** sont des sélecteurs associés au constructeur **cons** et permettent d'accéder aux deux arguments de cette fonction. L'indicateur **nul** permet de déterminer si un objet se réduit au constructeur **nil** ou non. Klaeren [Kla 84] définit une décomposition comme une fonction d définie de la manière suivante :

$$d(a) = (F, a_1, \ldots, a_n) \Rightarrow a = F(a_1, \ldots, a_n)$$

Fest une fonction et a_i est un objet construit à partir de a.

Si $\{ C_1, \ldots, C_n \}$ est un ensemble de constructeurs, si $S_{i,1}, \ldots, S_{i,ki}$ est l'ensemble des sélecteurs associés au constructeur C_i et si I_i est l'indicateur associé à ce même constructeur alors une décomposition peut être décrite sous la forme :

$$I_1(x) \Rightarrow x = C_1(S_{1,1}(x), \ldots, S_{1,k1}(x))$$

$$\ldots$$

$$I_n(x) \Rightarrow x = C_n(S_{n,1}(x), \ldots, S_{n,kn}(x))$$

Pour que cette décomposition soit complète nous devons avoir $\Pi_{(i=1\ ,\ n)}\ I_i(x) \equiv$ **vrai**.

Définir une décomposition à partir d'une définition inductive d'un domaine, c'est à dire imposer qu'il soit possible de reconstruire l'objet décomposé à l'aide de constructeurs, est restrictif. Par exemple dans l'algorithme de tri par sélection, la décomposition utilisée décompose une liste en son minimum et en la liste obtenue après élimination de ce minimum. Cette décomposition ne correspond pas à une définition inductive naturelle du domaine des listes. Pour notre part nous n'imposerons donc pas nécessairement qu'il soit possible de reconstruire le résultat à partir des décompositions. Une décomposition sera alors de la forme :

$$I_i(x) : \{ (S_{1,1}(x), \ldots, S_{1,n}(x)), \ldots, (S_{k,1}(x), \ldots, S_{k,n}(x)) \}$$

Ceci signifie que, lorsque $I_i(x)$ est vrai, le n-uplet x peut se décomposer en les n-uplets $(S_{1,1}(x), \ldots, S_{1,n}(x)), \ldots, (S_{k,1}(x), \ldots, S_{k,n}(x))$. Le lien existant entre x et les sélecteurs $S_{i,j}$ sera décrit par un énoncé, le lien ne sera donc pas nécessairement de type fonctionnel. Lors de la description de la stratégie **Décomposition des données**, basée sur le choix d'une décomposition, nous nous limiterons aux décompositions en deux cas. Une décomposition sera alors la donnée :

- d'une valeur du domaine à décomposer et de l'indicateur I associé à cette valeur ;
- d'une décomposition de la forme $\{ (S_{1,1}(x), \ldots, S_{1,n}(x)), \ldots, (S_{k,1}(x), \ldots, S_{k,n}(x)) \}$ qui s'appliquera à tous les éléments ne vérifiant pas l'indicateur I.

Nous présentons maintenant les stratégies **Construction du résultat** et **Décomposition des données** lors de leur application à une seule fonction. Ces deux stratégies combinent respectivement les stratégies relatives à l'introduction de compositions fonctionnelles et de conditionnelles. Puisque l'introduction d'une conditionnelle n'est applicable que sur l'ensemble des fonctions de l'énoncé, l'application de ces stratégies produira donc une définition conditionnelle pour toutes les fonctions décrites dans l'énoncé traité. D'autre part, lorsque l'utilisateur choisit une définition du domaine du résultat ou une décomposition d'un sous-ensemble des données, la construction ou la décomposition du cas de base est optionnelle. En effet, les données et les résultats peuvent n'appartenir qu'à un sous-ensemble du domaine considéré et les valeurs primitives classiques de ce domaine peuvent en être exclues. Dans ce cas le cas de base, propre à ce sous-domaine, sera élaboré par la stratégie. Le schéma général de programme engendré sera alors de la forme :

$f_i(x) \leftarrow$ **si** $C_0(x)$ **alors** $f_0(x)$
sinon si $C_1(x)$ **alors** $f_1(x)$
sinon $g(f_i(h_{1,1}(x), \ldots, h_{1,n}(x)), \ldots, f_i(h_{k,1}(x), \ldots, h_{k,n}(x)), x)$

$f_j(x) \leftarrow$ **si** $C_0(x)$ **alors** $f_{0,j}(x)$
sinon si $C_1(x)$ **alors** $f_{1,j}(x)$
sinon $f_{2,j}(x)$

f_i désigne la fonction sur laquelle est appliquée la stratégie et f_j les fonctions de l'énoncé non affectées directement par la stratégie. C_1 et f_1 décrivent les données et le résultat qui ne correspond pas au cas de base donné par l'utilisateur et qui ne sont pas argument ou résultat d'un appel récursif.

3.4.1. Stratégie Construction du résultat.

Cette stratégie est basée sur le choix d'une définition inductive du domaine du résultat, c'est à dire de deux constructeurs. Ceci revient à instancier les fonctions de recomposition g et f_0 du schéma ci-dessus. Tous les arguments de la fonction g ayant même type que la fonction à synthétiser seront considérés comme élaborés par des appels récursifs. L'application de la stratégie relative à l'introduction d'une composition fonctionnelle **Instanciation de la fonction de recomposition** permettra alors de dériver un énoncé caractérisant les fonctions $h_{i,j}$. La stratégie Construction du résultat peut être vue comme l'enchainement des stratégies suivantes :

- Introduction d'une conditionnelle par instanciation d'une postcondition de la conditionnelle par f_0 ;
- Instanciation de la fonction de recomposition à l'aide de la construction g choisie ;
- Introduction d'une conditionnelle par échec.

Déroulement :

1. Choix d'une construction du domaine du résultat. L'utilisateur fournit :

   - une fonction b permettant de construire les objets primitifs du domaine du résultat. Cette donnée est optionnelle.

   - une fonction g permettant de construire inductivement les objets du domaine du résultat.

Soient les énoncés suivants relatifs aux fonctions b et g :

profil : $var_i = b(z_1 : s_1, \ldots, z_p : s_p) \rightarrow t_0$
préc : $P_b(z_1, \ldots, z_p)$
post : $Q_b(z_1, \ldots, z_p, var_i)$

profil : $var_i = g(y_1 : r_1, \ldots, y_m : r_m) \rightarrow t_g$
préc : $P_g(y_1, \ldots, y_m)$
post : $Q_g(y_1, \ldots, y_m, var_i)$

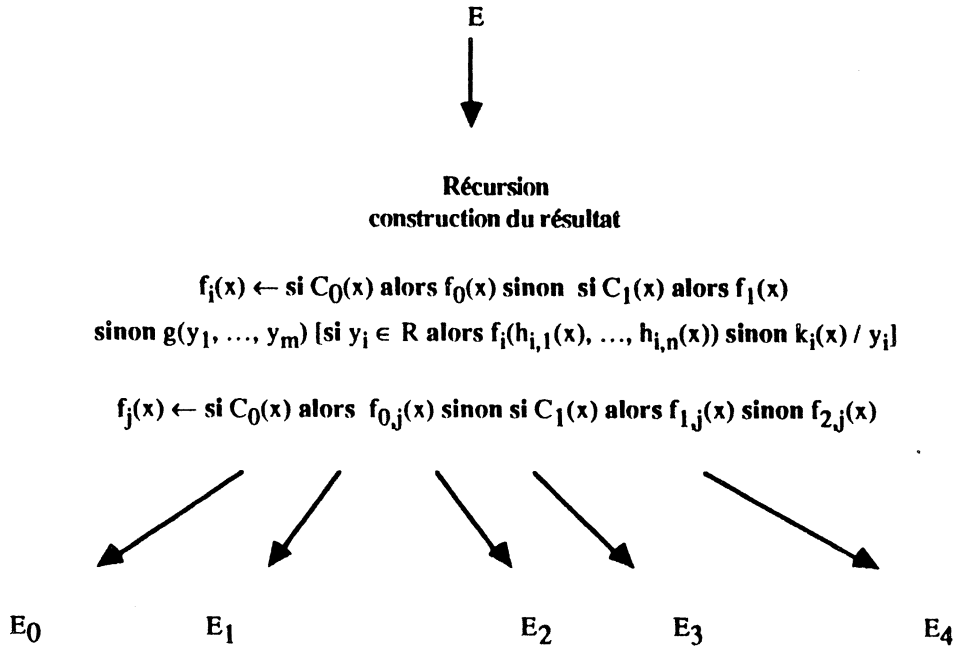
Le système vérifie que le type du résultat de la fonction en cours de construction, t_i, est une instance des types t_g et t_0. Si le filtrage n'est pas possible alors les choix effectués par l'utilisateur sont rejetés sinon soient σ_0 et σ_g les deux substitutions obtenues.

2 . Sélection des appels récursifs. Le critère retenu est le type des données de la fonction g. Toutes les données y_i de type r_i tel que $r_i\sigma_g$ est une instance de t_i sont considérées comme élaborées par un appel récursif. Soient R l'ensemble de ces données. Si cet ensemble est vide il y a échec de la stratégie, sinon :

- Pour chaque élément y_i appartenant à R le système engendre n fonctions $h_{i,j}(x)$ calculant les n arguments de l'appel récursif introduit. y_i désigne donc le résultat $f_i(h_{i,1}(x), \ldots, h_{i,n}(x))$.

- Pour chaque élément y_i n'appartenant pas à R le système engendre une fonction $k_i(x)$ calculant le ième argument de la fonction g, qui ne provient pas d'un appel récursif.

2.1. Si l'utilisateur a donné une construction de base l'arbre engendré est :



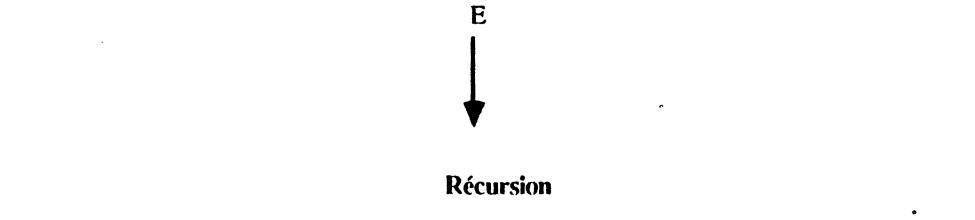
- E_0 est l'énoncé caractérisant la fonction boléenne C_0 ;
- E_1 est l'énoncé caractérisant les fonctions f_0 et $f_{0,j}$;
- E_2 et l'énoncé relatif à C_1 ;
- E_3 est relatif aux fonctions f_1 et $f_{1,j}$;
- E_4 caractérise les fonctions $h_{i,j}$, k_i et $f_{2,j}$.

L'énoncé E_1 est l'énoncé initial dans lequel les fonctions f_j sont renommées en $f_{0,j}$ et la fonction f_i en f_0. C'est à dire :

profil	:	$var_1=f_{0,1}(x : t_x) \rightarrow t_1, \ldots, var_i=f_0(x : t_x) \rightarrow t_i, var_n=f_{0,n}(x : t_x) \rightarrow t_n$
prec	:	P(x)
post	:	Q(x, var)
trace	:	T

Les autres énoncés seront construits ultérieurement.

2.2. Si l'utilisateur n'a pas donné de construction de base l'arbre construit est :



construction du résultat

$f_i(x) \leftarrow$ **si** $C_1(x)$ **alors** $f_1(x)$ **sinon** $g(y_1, \ldots, y_m)$
[**si** $y_i \in R$ **alors** $f_i(h_{i,1}(x), \ldots, h_{i,n}(x))$ **sinon** $k_i(x) / y_i$]

$f_j(x) \leftarrow$ **si** $C_1(x)$ **alors** $f_{1,j}(x)$ **sinon** $f_{2,j}(x)$

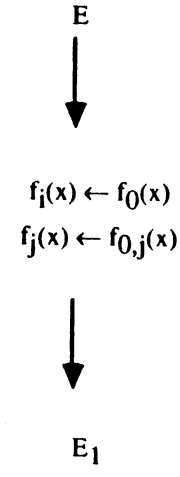
E_2 E_3 E_4

Ces 3 énoncés seront déterminés ultérieurement.

3 . Cette étape n'est effectuée que si l'utilisateur a donné une construction des objets primitifs du domaine du résultat.

3.1. L'énoncé E_1 est alors prouvé par appel de la stratégie introduction d'une composition fonctionnelle par **Instanciation de la fonction de recomposition.** Cette stratégie s'applique sur la fonction f_0, à partir de l'énoncé E_1. La fonction de recomposition retenue est la fonction **b** donnée par l'utilisateur. Soit $H_1(x)$ la précondition dérivée lors de cette preuve. Son existence découle de la stratégie en cours d'application.

a). Si cette précondition est réduite à **vrai** il y a arrêt de l'application de la stratégie **Construction du résultat** et l'arbre est modifié en :



La précondition dérivée est alors **vrai.**

b). Si $H_1(x)$=**faux** c'est comme si l'utilisateur n'avait pas donné de construction de base. L'exécution du pas **3** est stoppée et l'arbre retourné est le même que celui du pas **2.2.**

3.2. $H_1(x)$ sera la condition introduite. L'utilisateur choisit, si il le veut, de transformer cette condition en une autre condition $R_1(X)$ suivant les variables qu'il souhaite voir apparaître dans cette condition, comme dans les stratégies relatives à l'introduction de conditionnelles. Si $H_1(x)$ convient alors $R_1(X)$ est la formule $H_1(x)$ elle-même. L'énoncé E_0 engendré est alors :

profil : cond = $C(x : t_x) \rightarrow$ booléen
prec : $P(x)$
post : cond=vrai $\equiv R_1(X)$
trace : T

Si l'étape 3 n'a pas été exécuté ou si il y a eu échec la précondition $R_1(X)$ est assimilée à la constante **faux**.

4 . Recherche de l'énoncé E_4.

Demande d'une hypothèse Q_2 telle que :

$$P(x) \wedge \neg R_1(X) \wedge Q_2(x, y_1, \ldots, y_m, \text{var\_autre}) \text{ [si } y_i \in R \text{ alors } w_i / y_i]$$
$$\wedge \prod_{(yi \in R)} Q(w_i, y_i) \wedge Q_g(y_1, \ldots, y_m, var_i) \Rightarrow Q(x, var)$$

Les variables w_i désigne le n-uplet $(h_{i,1}(x), \ldots, h_{i,n}(x))$. Dans cette demande il faut donc d'une part transformer le but Q(x, var) en faisant apparaître les variables intermédiaires $y_1, \ldots, y_m$ puis faire disparaître les y_i appartenant à R en utilisant la relation $Q(w_i, y_i)$ pour tous les appels récursifs introduits.

5 . Si Q_2 n'est pas trouvé il y a échec, sinon il y a demande d'une hypothèse P_2 telle que :

$$P_2(y_1, \ldots, y_m) \text{ [si } y_i \in R \text{ alors } w_i / y_i] \wedge \prod_{(yi \in R)} Q(w_i, y_i) \Rightarrow P_g(y_1, \ldots, y_m)$$

Cette demande permet de ramener la précondition de la fonction de recomposition choisie en terme d'une propriété des variables résultats des fonctions $h_{i,j}$ et k_j.

6 . Si P_2 n'est pas trouvé il y a échec sinon il y a demande d'une hypothèse simplificatrice P_2' telle que :

$$(P(x) \wedge \neg R_1(X) \wedge Q_2(x, y_1, \ldots, y_m, \text{var\_autre}) \wedge P_2'(x, y_1, \ldots, y_m, \text{var\_autre})$$
$$\Rightarrow P_2(y_1, \ldots, y_m)) \text{ [si } y_i \in R \text{ alors } w_i / y_i]$$

Puisque cette demande vise une simplification il n'y a pas de traitement d'échec. L'énoncé E_4 engendré est :

profils : $\ldots, \text{var\_autre}_j = f_{2,j}(x : t_x) \rightarrow t_n, \ldots,$
$y_i = k_i(x : t_x) \rightarrow r_i, \ldots,$
$w_{i,j} = h_{i,j}(x : t_x) \rightarrow t_j, \ldots$

prec : $P(x) \wedge \neg R_1(X)$

post : $(Q_2(x, y_1, \ldots, y_m, \text{var\_autre}) \wedge P_2'(x, y_1, \ldots, y_m, \text{var\_autre}))$ [si $y_i \in R$ **alors** w_i / y_i] $\wedge \prod_{(i=1, k)} P(w_{i,1}, \ldots, w_{i,n})$

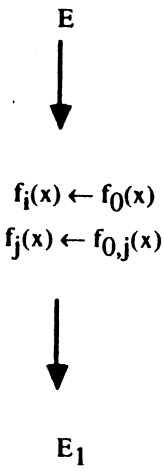
trace : T

Les appels récursifs introduits doivent vérifier la précondition initiale, c'est pourquoi $P(w_i)$ est une propriété attendues des résultats w_i.

7. Preuve de E_4. Soit $H_4(x)$ la précondition dérivée lors de cette preuve. Son existence découle de la stratégie en cours d'application.

8 . Cas $H_4(x)$=**faux**.

Si $R_1(X)$=**faux** il y a échec total de la stratégie sinon l'arbre est modifié en :



Il y a arrêt de l'application de cette stratégie et la précondition dérivée est $R_1(X)$. En effet, dans ce cas, il n'est pas possible de calculer les arguments de la fonction de recomposition ($H_4(x)$=**faux**). Si l'utilisateur a donné un cas de base et que celui-ci est effectivement calculable ($R_1(x)\neq$**faux**) alors les fonctions ne sont calculables que pour les données vérifiant R_1.

9 . Si $H_4(x)\neq$**faux** on cherche à s'assurer de l'arrêt du programme. Le système demande à l'utilisateur de fournir un ordre bien fondé sur le domaine des x vérifiant la précondition initiale P(x). Soit **inf**(a, b) cet ordre.

- Pour chaque appel récursif engendré il y a demande d'une hypothèse H_i telle que :

$$H_i(x) \wedge P(x) \wedge C_4(x) \wedge \neg R_1(X) \wedge (Q_2(x, y_1, \ldots, y_m, \text{var\_autre}) \wedge P_2'(x, y_1, \ldots, y_m, \text{var\_autre}))\ [\text{si } y_i \in R \text{ alors } w_i / y_i] \wedge \prod_{(i=1, k)} P(w_i) \Rightarrow \inf(w_i, x)$$

Cette demande permet uniquement de vérifier que la relation d'ordre donnée convient pour le problème traité. L'arrêt de la récursion n'est garanti que si il a été prouvé par ailleurs que cet ordre est bien fondé sur le domaine considéré. Si l'un des H_i n'a pas été trouvé il y a échec sinon il y a demande d'une hypothèse simplificatrice H telle que :

$$H(x) \wedge P(x) \wedge H_4(x) \wedge \neg R_1(X) \wedge T \Rightarrow \Pi_{(i=1,\,k)}\, H_i(x)$$

Si H(x) est réduite à **vrai** alors l'ordre choisi convient pour toute donnée sinon il ne convient que pour les données vérifiant cette propriété. Dans ce cas c'est l'utilisateur qui fournit, si il le désire, une nouvelle relation d'ordre. Il peut aussi décider de ne pas garantir l'arrêt du programme construit. Dans ce cas **vrai** est affectée à H(x) et le fait que l'arrêt n'est pas garanti est mémorisé dans l'arbre.

La synthèse des fonctions $h_{i,j}$, k_i et $f_{2,i}$ n'aboutit donc que pour la précondition dérivée $H_4(x) \wedge H(x)$. Soit $H_4'(x)$ cette précondition. Il faut maintenant regarder si les fonctions de l'énoncé ont pu être construites pour toute donnée vérifiant la précondition initiale. Si ce n'est pas le cas alors il y aura introduction d'une conditionnelle permettant de traiter les données restantes.

10 . Cas $H_4'(x)$=**vrai** et $R_1(X)$=**faux**. L'arbre est modifié en :

Récursion
construction du résultat
arrêt non garanti

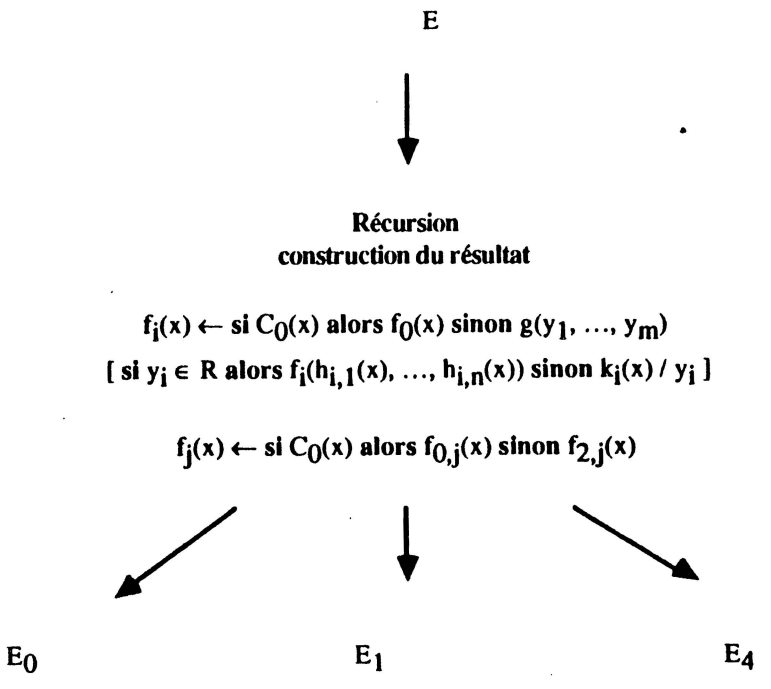
$f_i(x) \leftarrow g(y_1, \ldots, y_m)$ [y_i / **si** $y_i \in R$ **alors** $f_i(h_{i,1}(x), \ldots, h_{i,n}(x))$ **sinon** $k_i(x)$]

$f_j(x) \leftarrow f_{2,j}(x)$

E_4

Dans ce cas l'arrêt n'est pas prouvé puisqu'il n'y a pas de cas de base. La précondition renvoyée est $C_4'(x)$.

11 . Cas $H_4'(x)$=**vrai** et $R_1(X) \neq$**faux**. L'arbre est modifié en :



Les fonctions **b** et **g** données par l'utilisateur permettent de calculer le résultat pour toutes les données possibles, il n'est donc pas nécessaire de conserver la conditionnelle supplémentaire introduite initialement. L'arbre construit au pas 2.1 est donc simplifié. Rappelons que nous avons choisi de construire l'arbre de preuve chaque fois que des choix sont effectués par l'utilisateur même si sa construction est modifié par la suite, ceci afin de disposer, à tout moment, de l'historique de la preuve.

Soit $R_4(x)$ la précondition $H_4'(x)$.

12 . Si $H_4'(x) \neq$ **vrai** il y a appel de la stratégie introduction d'une conditionnelle par **Instanciation d'une postcondition**. Cette stratégie s'applique sur l'énoncé initial pour la précondition $P(x) \wedge (R_1(X) \vee H_4'(x))$ et son application produit un arbre de preuve issu de E_2 et E_3. Soit $R_4(x)$ la précondition dérivée lors de cette preuve.

13 . Preuve de E_0 si cet énoncé a été constitué. Soit $H(x)$ la précondition dérivée.

14 . La précondition finale est élaborée à partir de la formule $H(x) \wedge (R_1(X) \vee R_4(x))$. Rappelons que :

- $R_1(X)$ est la précondition dérivée lors de la synthèse de la fonction f_0 si l'utilisateur a donné un cas de base sinon elle a été initialisée à **faux** (pas 3.2).

- $R_4(x)$ est la précondition élaborée lors de la synthèse des fonctions $h_{i,j}$ et k_i (pas 11) ou lors de la construction de la définition si $C_1(x)$ alors $f_1(x)$ sinon $g(y_1, \ldots, y_m)$ [si $y_i \in R$ alors

$f_i(h_{i,1}(x), \ldots, h_{i,n}(x))$ **sinon** $k_i(x) / y_i$], si ceci a été nécessaire (pas 12).

$H(x) \wedge (C_1(X) \vee R_4(x))$ correspond à la peécondition dérivée lors de l'application de la stratégie introduction d'une conditionnelle par **Instanciation d'une postcondition**. R_1 et R_4 doivent donc être simplifiées par l'hypothèse H(x).

- Demande d'une hypothèse D_1 telle que $D_1(x) \wedge P(x) \wedge H(x) \wedge T \Rightarrow R_1(X)$.
- Demande d'une hypothèse D_2 telle que $D_2(x) \wedge P(x) \wedge H(x) \wedge T \Rightarrow R_4(x)$.

La précondition finale renvoyée est $H(x) \wedge (D_1(x) \vee D_2(x))$.

15 . Si cette précondition dérivée est différente de **vrai** et que la preuve est conservée il faut s'assurer que les arguments des appels récursifs appartiennent aussi au sous-ensemble des données pour lequel cette preuve a été possible.

Si $R_4(x) \neq$ **faux**, c'est à dire si il y a eu effectivement introduction d'appels récursifs, alors, pour chacun des appels introduits, il y a demande d'une hypothèse R_i telle que :

$R_i(x) \wedge P(x) \wedge H(x) \wedge (D_1(x) \vee D_2(x)) \wedge \wedge \prod_{(i=1, k)} P(w_i)$
$(Q_2(x, y_1, \ldots, y_m, \text{var\_autre}) \wedge P_2'(x, y_1, \ldots, y_m, \text{var\_autre}))$ [**si** $y_i \in R$ **alors** w_i / y_i]
$\Rightarrow H(w_i) \wedge (D_1(w_i) \vee D_2(w_i))$

Si l'un des R_i ne peut pas être trouvé il y a échec sinon il y a demande d'une hypothèse simplificatrice R telle que :

$$R(x) \wedge P(x) \wedge H(x) \wedge (D_1(x) \vee D_2(x)) \wedge T \Rightarrow \prod_{(i=1, k)} R_i(x)$$

Si R(x)=**vrai** la précondition dérivée n'est pas modifiée, c'est $H(x) \wedge (D_1(x) \vee D_2(x))$ sinon elle devient $R(x) \wedge H(x) \wedge (D_1(x) \vee D_2(x))$. Si l'utilisateur choisit de conserver encore cette preuve il faut réitérer cette dernière étape jusqu'à obtenir R(x)=**vrai** ou jusqu'à ce que la preuve soit remise en cause.

fin-stratégie.

Pour élaborer cette stratégie nous n'avons fait que reprendre le déroulement classique des stratégies déja exposées en remplaçant les pas relatifs à la preuve des sous-problèmes par l'appel de stratégies adéquates. La stratégie utilisée au premier niveau est l'introduction de conditionnelle par **Instanciation d'une postcondition** dont les principales étapes sont :

1 - Preuve de la première branche de la conditionnelle ;

2 - choix de la condition ;

3 - preuve de la seconde branche de la conditionnelle ;

4- preuve de la condition ;

5 - élaboration de la précondition dérivée.

Les étapes 1 et 2 n'ont lieu que si l'utilisateur a donné une construction pour les objets de base du domaine du résultat. L'étape 3 est un appel de la stratégie introduction d'une composition fonctionnelle par **Instanciation de la fonction de recomposition**. Cette étape se décompose alors en :

3.1 - recherche de l'énoncé des fonctions intermédiaires ;

3.2 - preuve de cet énoncé.

Cet énoncé est aussi prouvé par application de la stratégie introduction d'une composition fonctionnelle par **Instanciation de la fonction de recomposition**. La fonction de recomposition choisie est la fonction f_i, pour laquelle une définition récursive est établie, et cette stratégie s'applique sur toutes les fonctions intermédiaires ayant le même codomaine que f_i. D'autre part, si cet énoncé ne peut être prouvé totalement, nous avons choisi d'introduire une conditionnelle permettant de traiter les autres cas. L'étape 3.2 est donc :

3.2.1 - recherche de l'énoncé des fonctions intermédiaires $h_{i,j}$ et k_i ;

3.2.2 - preuve de cet énoncé ;

3.3.3 - introduction d'une conditionnelle par **Instanciation d'une postcondition**.

Le schéma récursif n'est pas seulement une construction établie à partir de constructions plus primaires mais il est aussi basé sur un raisonnement par récurrence. Nous avons donc intégré, dans la stratégie **Construction du résultat**, les contraintes propres à ce raisonnement. Cela revient essentiellement à vérifier que les hypothèses d'induction $P(w_i) \wedge D(w_i) \Rightarrow Q(w_i, y_i)$ sont valides et que $P(w_i) \wedge D(w_i)$ est vrai pour tout i. Ces deux propriétés permettent alors de déduire les formules $Q(w_i, y_i)$ nécessaires au déroulement de cette stratégie. La validité des hypothèses d'induction repose sur l'existence d'un ordre bien fondé tel que w_i est strictement inférieur à x dans cet ordre, pour tout i. Ceci est traité à l'étape 9. La formule $P(w_i) \wedge D(w_i)$ est établie d'une part lors de la preuve de l'énoncé relatif au calcul des arguments w_i (pas 6), puisque $P(w_i)$ est une des propriétés attendues de ces résultats, et d'autre part lorsque la précondition dérivée finale est établie (pas 15).

Nous avons décrit la stratégie **Construction du résultat** lorsque la construction choisie est réduite à 2 possibilités. Si la construction est la donnée de m fonctions $Cons_i$ alors le schéma de programme engendré sera de la forme :

$f_i(x) \leftarrow$ **si** $C_1(x)$ **alors** $Cons_1(k_{1,1}(x), \ldots, k_{1,i1}(x))$
sinon
si $C_2(x)$ **alors** $Cons_2(k_{2,1}(x), \ldots, k_{2,i2}(x))$
sinon
...
sinon $Cons_m(k_{m,1}(x), \ldots, k_{m,im}(x))$

Les fonctions $k_{i,j}$, ayant même codomaine que f_i, sont de la forme $f_i((h_{i,j,1}(x), \ldots, h_{i,j,n}(x))$. Le déroulement de la stratégie est le même, les fonctions $h_{i,j,k}$ et $k_{i,j,k}$ étant toutes décrites par l'énoncé E_4. Lorsque cette stratégie

s'applique sur plusieurs fonctions, son déroulement est aussi identique. La construction choisie est utilisée pour toutes les fonctions sélectionnées et le système engendre des fonctions $h_{i,j,k}$ et $k_{i,k}$ associées à chacune de ces fonctions.

3.4.2. Stratégie Décomposition des données.

Dans cette stratégie l'utilisateur fixe une décomposition et le sous-ensemble des données sur lequel elle porte. Par exemple l'algorithme classique de la fonction élimination d'un élément n dans un liste repose l uniquement sur une décomposition de l, l'élément à éliminer étant toujours le même. Par contre, l'algorithme de calcul du pgcd par soustractions successives s'applique sur un élément de N^2, et la décomposition sous-jacente fait intervenir ces 2 éléments. Cette décomposition est :

x=0 :	(0, y)
x≥y :	(x-y, y)
sinon :	(y, x)

Nous n'imposons donc pas que l'utilisateur donne une décomposition portant sur la donnée x en entier mais seulement sur la partie qui lui semble nécessaire. Les arguments des appels récursifs, autres que ceux découlant de la décomposition retenue, peuvent néanmoins dépendre de la décomposition choisie, nous introduirons donc de manière systématique une fonction calculant ces arguments. Par exemple la fonction **perm** qui calcule si 2 listes x et y sont des permutations l'une de l'autre peut être décrite par :

perm(x, y) ← **si** x=nil **alors** y=nil
sinon si dans(car(x), y) **alors** perm(cdr(x), elim(car(x), y)) **sinon** faux

Cette définition est basée sur une induction sur le premier élément utilisant la décomposition des listes en terme de nil et cons(car(x), cdr(x)). Imposer que la décomposition porte obligatoirement sur le n-uplet des données en entier nécessiterait, comme le propose Gresse [Gre 84], d'utiliser la décomposition du domaine liste(t) × liste(t) suivante :

x=nil et y=nil :	(nil, nil)
x=nil :	(nil, y)
y=nil :	(x, nil)
dans(car(x), y) :	(cdr(x), elim(car(x), y))
sinon :	(x, y)

elim est une fonction partielle qui élimine un élément dans une liste donnée, elle est indéfinie si cet élément n'est pas présent dans cette liste.

Il semble difficile de supposer que l'utilisateur puisse fournir une telle décomposition sauf si il connait explicitement le programme auquel il veut arriver ... D'autre part, lorsqu'une fonction porte sur plusieurs arguments il est classique de chercher d'introduire une récursion en se basant sur une décomposition d'une des données et de déterminer alors les autres arguments nécessaires. C'est pour cela que nous avons trouvé nécessaire de ne pas faire porter obligatoirement la décomposition sur la donnée en entier mais aussi de ne pas nous limiter aux cas où les arguments, non pris en compte dans la décomposition, sont nécessairement les mêmes que ceux de l'appel initial. Rappelons d'autre part que nous limitons la donnée d'une décomposition à :

- un indicateur I et la valeur de base correspondante ;
- des k-uplets décrivant les éléments obtenus par décomposition.

Le schéma général de la stratégie introduction de récursion par **Décomposition de la donnée** est le suivant :

- Introduction d'une conditionnelle par instanciation de la condition à l'aide de l'indicateur I.

- Introduction d'une composition fonctionnelle par instanciation des fonctions intermédiaires à l'aide des décompositions données.

- Introduction d'une conditionnelle par échec, si nécessaire. Considérer ce cas d'échec permet de ne pas imposer à l'utilisateur de donner un cas de base. Dans certains cas, celui-ci sera déduit automatiquement.

En fait les 2 conditionnelles introduites seront fusionnées en une seule. L'indicateur I fourni par l'utilisateur peut ne permettre que d'instancier partiellement la condition qui sera introduite.

Déroulement :

1. - Choix des données à décomposer. Soit X ce sous-ensemble.

- Choix de la décomposition sous la forme :

- cas de base : une propriété I(X) sur les données et les valeurs primitives de X associées à cette condition. Cette donnée est optionnelle. Si l'utilisateur ne la donne pas la condition I(x) est initialisée à **faux**.

- cas récursion : un ensemble de k-uplets $(S_{i,1}(X), \ldots, S_{i,k}(X))$ qui sont des décompositions de X et un énoncé caractérisant les fonctions $S_{i,j}$. Soient $T_1, \ldots, T_m$ ces k-uplets.

Supposons que l'énoncé donné soit :

profils : $\ldots, w_{i,j} = S_{i,j}(X : t_X) \rightarrow r_{i,j}, \ldots,$
prec : $P_1(X)$
post : $Q_1(X, w)$

w désigne l'ensemble des résultats $w_{i,j}$. Nous supposons que les fonctions $S_{i,j}$ sont définies par un seul énoncé, la constitution de cet énoncé à partir des énoncés propres à chacune de ces fonctions ne posant pas de problème.

2. Le système détermine les données sur lesquelles il est possible d'appliquer les décompositions choisies par l'utilisateur. Si P_1 n'est pas réduite à **vrai** il y a demande d'une hypothèse H telle que :

$$P(x) \wedge \neg I(X) \wedge H(x) \Rightarrow P_1(X)$$

Si H est identiquement **faux** la décomposition choisie est rejetée.

3. Sélection des appels récursifs. Ce choix se fait parmi l'ensemble des k-uplets T_i tel que le type des k-uplets est une instance du type de l'ensemble des données à décomposer, c'est à dire T_X. Soient $R_1, \ldots, R_k$ ces k-uplets. Pour chaque R_i le système engendre l'appel :

$$f_i(x_1, \ldots, x_n)\ [\text{si } x_j \in X \text{ alors } S_{i,j}(X) \text{ sinon } k_{i,j}(x) / x_j]$$

Les fonctions $k_{i,j}$ calculent, pour chacun des appels introduits, les arguments des appels qui ne découlent pas de la décomposition choisie par l'utilisateur. Nous désignons par la suite par w_i l'ensemble des données du i-ème appel récursif qui sont le résultat des fonctions $S_{i,j}$ et par y_i les résultats calculés par les fonctions $k_{i,j}$. $w_i \cup y_i$ désigne les arguments du i-ème appel récursif et w et y désignent respectivement $(w_1, \ldots, w_n)$ et $(y_1, \ldots, y_n)$.

Une fois les appels récursifs engendrés, il faut alors vérifier que la précondition initiale est aussi vraie pour les arguments des appels et qu'il existe un ordre bien fondé associé à la décomposition retenue.

4. Pour chaque appel introduit il y a demande d'une hypothèse A_i telle que :

$$P(x) \wedge H(x) \wedge \neg I(X) \wedge A_i(x, y_i) \wedge Q_1(X, w) \Rightarrow P(w_i \cup y_i)$$

Si les A_i ne sont pas trouvés il y a échec, sinon il y demande d'une hypothèse simplificatrice A telle que :

$$P(x) \wedge H(x) \wedge \neg I(X) \wedge A(x, y) \wedge T \Rightarrow \Pi_{(i=1, k)} A_i(x, y_i)$$

5. L'utilisateur fournit un ordre bien fondé **inf** et le système cherche les hypothèses sous lesquelles les arguments des appels récursifs sont bien inférieurs à la donnée initiale dans cet ordre. Pour chaque appel introduit, il y a demande d'une hypothèse B_i telle que :

$$B_i(x, y) \wedge H(x) \wedge P(x) \wedge \neg I(X) \wedge A(x, y) \wedge Q_1(X, w) \Rightarrow \mathbf{inf}((w_i \cup y_i), x)$$

Il y a ensuite demande d'une hypothèse simplificatrice B telle que :

$$P(x) \wedge H(x) \wedge \neg I(X) \wedge B(x, y) \wedge A(x, y) \wedge T \Rightarrow \Pi_{(i=1, k)} B_i(x, y_i)$$

Si l'utilisateur n'a pas donné une telle relation d'ordre ou si l'hypothèse B établie se réduit à **faux** alors l'arrêt n'est pas garanti.

L'introduction des appels récursifs est donc possible pour la précondition $H(x) \wedge \neg I(X)$ et les fonctions intermédiaires $k_{i,j}$ introduites doivent être telles que $A(x, y) \wedge B(x, y)$. Rappelons que y désigne l'ensemble des résultats calculés par ces fonctions.

6. Recherche des énoncés caractérisant la fonction de recomposition g, les fonctions $k_{i,j}$ introduites et les fonctions initiales f_j qui ne sont pas directement concernées par l'application de la stratégie **Décomposition des données**. Cette étape est analogue à l'application de la stratégie introduction d'une compositionnelle fonctionnelle par **Instanciation des fonctions intermédiaires**.

- Demande d'une hypothèse Q_2 telle que :

$$P(x) \wedge H(x) \wedge \neg I(X) \wedge Q_2(x, z, y, \mathbf{var}) \wedge Q_1(X, w) \wedge \Pi_{(i=1, k)} Q(w_i \cup y_i, z_i) \Rightarrow Q(x, \mathbf{var})$$

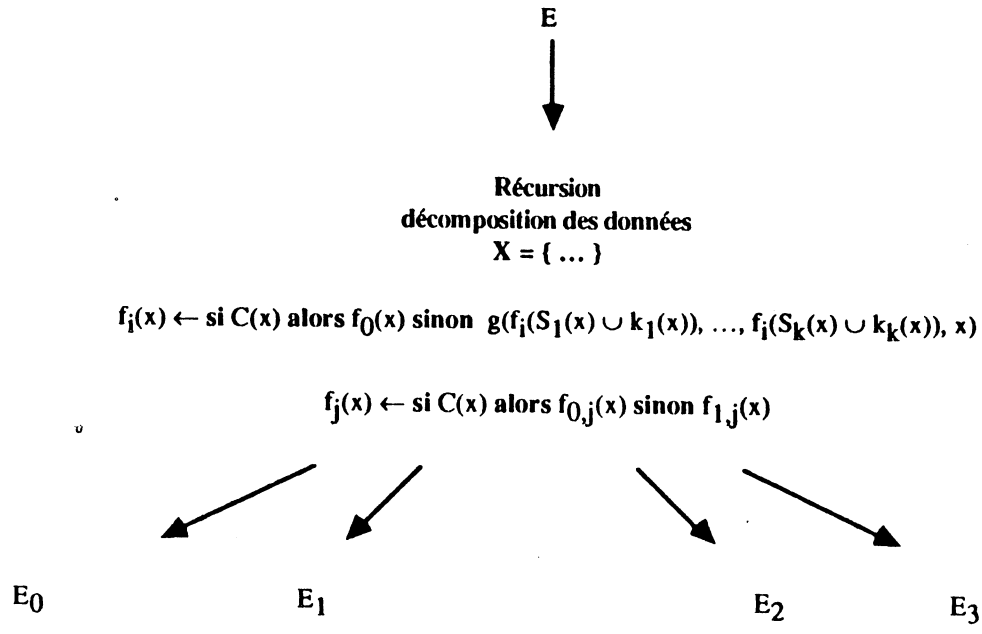
z_i désigne le résultat du ième appel récursif introduit et z les résultats des k appels récursifs.

Si Q_2 n'est pas trouvée il y a échec sinon les fonctions f_j, $k_{i,j}$ et la fonction g sont caractérisées par la formule $Q_2(x, y, z, var) \wedge A(x, y) \wedge B(x, y)$. Les arguments des fonctions f_j et $k_{i,j}$ d'une part et de la fonction g d'autre part ne sont pas les mêmes. Comme dans la stratégie introduction d'une composition fonctionnelle par **Instanciation des fonctions intermédiaires** nous devons appliquer la stratégie **Découper et Ordonner** pour obtenir deux énoncés caractérisant chacun l'une de ces classes de fonctions. Nous appliquons donc cette stratégie pour les ensembles { $k_{i,j}$, f_j } et { g }.

7 . Demande de deux hypothèses Q_3 et Q_4 telles que :

$$H(x) \wedge P(x) \wedge \neg I(X) \wedge \mathbf{Q_3(x, y, var\_autre)} \wedge \mathbf{Q_4(x, y, z, var\_autre, var_j)} \Rightarrow Q_2(x, y, z, var)$$

Si Q_3 n'est pas trouvée il y a échec sinon l'arbre élaboré est :



E_0 est relatif à la fonction C et E_1 aux fonctions f_0 et $f_{0,j}$. E_2 caractérise les fonctions $k_{i,j}$ et $f_{1,j}$ et E_3 caractérise la fonction de recomposition g. $(S_i(x) \cup k_i(x)))$ désigne le n-uplet obtenu à partir de x par application de la substitution [si $x_j \in X$ alors $S_{i,j}(X)$ sinon $k_{i,j}(x) / x_j$]. Les énoncés établis sont les suivants :

E_2 :

profils	:	..., $y_{i,j} = \mathbf{k_{i,j}}(x : t_x) \rightarrow t_i$, ...,
		$var\_autre_j = \mathbf{f_{1,j}}(x : t_x) \rightarrow t_j$, ...
prec	:	$P(x) \wedge \neg I(X) \wedge H(x)$
post	:	$Q_3(x, y, var\_autre) \wedge A(x, y) \wedge B(x, y)$
trace	:	T

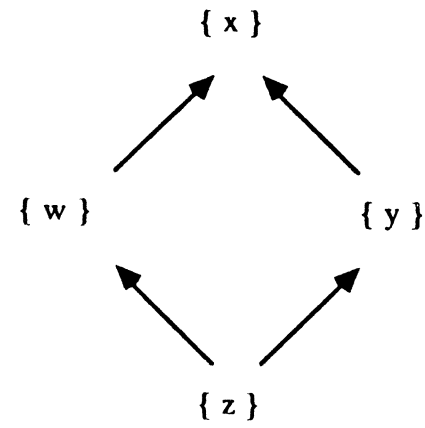
E_3 : **profil** : $var_i = g(z : t_z , x : t_x , y : t_y , var\_autre: t_{autre}) \rightarrow t_i$

prec : $P(x) \wedge \neg I(X) \wedge H(x) \wedge A(x, y) \wedge B(x, y)$

post : $Q_4(x, y, z, var)$

trace : $T, Q_1(X, w), Q_3(x, y, var\_autre), \Pi_{(i=1, k)} Q(w_i \cup y_i, z_i)$

Et le graphe de dépendance des variables est complété par :



Les deux énoncés E_2 et E_3 proviennent de l'application de la stratégie **Découper et Ordonner**. Comme dans cette stratégie, nous commençons par prouver l'énoncé E_3..

8 . Preuve de E_3. Soit $C_3(z, x, y, var\_autre)$ la précondition dérivée.

9 . Si la précondition $C_3(z, x, y, var\_autre)$ est différent de **faux** ou de **vrai** alors elle doit être ramenée en terme d'une formule C_3' portant uniquement sur les variables x, y et var_autre, afin d'ajouter cette relation à la postcondition de l'énoncé E_2.

- Demande d'une hypothèse C_3' telle que :

$$C_3'(x, y, \mathbf{var\_autre}) \wedge P(x) \wedge \neg I(X) \wedge H(x) \wedge Q_1(X, w) \wedge \Pi_{(i=1, k)} Q(w_i \cup y_i, z_i)$$
$$\Rightarrow C_3(z, x, y, var\_autre)$$

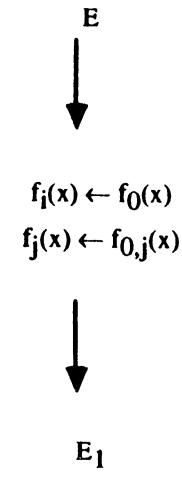
Si C_3' n'est pas trouvé il y a échec sinon on cherche à simplifier cette formule à l'aide de l'énoncé déja élaboré pour E_2. Il y a demande d'une hypothèse simplificatrice R telle que :

$$\mathbf{R(x, y, var\_autre)} \wedge Q_3(x, y, var\_autre) \wedge A(x, y) \wedge B(x, y) \wedge P(x) \wedge \neg I(X) \wedge H(x)$$
$$\Rightarrow C_3'(x, y, var\_autre)$$

R(x, y, var_autre) est alors ajouté à la postcondition de E_2.

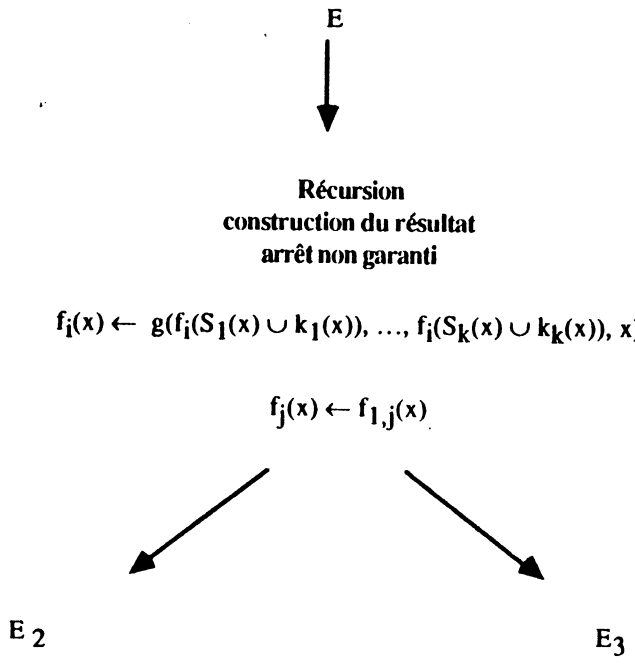
10 . Si C_3 est différent de **faux** il y a preuve de l'énoncé E_2. Soit $C_2(x)$ la précondition obtenue lors de cette preuve. Par défaut $C_2(x)$ est initialisé à **faux**.

11 . Cas $C_2(x)$=**faux**. Si I(X)=**faux** il y a échec total de la stratégie, sinon l'arbre est modifié en :



Rappelons que I(x) représente la classe des données pour lesquelles il existe un cas de base.

12 . Cas $C_2(x)$=**vrai** et I(x)=**faux**. L'arbre est modifié en :



Dans ce cas l'arrêt n'est pas prouvé puisqu'il n'y a pas de cas de base. La précondition renvoyée est **vrai**.

13 . Cas $C_2(x) \neq$ vrai.

L'arbre n'est pas modifié et les énoncés E_0 et E_1 peuvent être élaborés. Les fonctions g, $h_{i,j}$ et $f_{1,j}$ ont alors pu être synthétisées pour la précondition $H(x) \wedge \neg I(x) \wedge C_2(x)$. La conditionnelle introduite portera sur la condition $\neg(H(x) \wedge \neg I(x) \wedge C_2(x))$. L'utilisateur peut choisir de modifier cette condition suivant le sous-ensemble des données X_0 sur lequel il souhaite qu'elle porte. Il y a demande d'une hypothèse R_0 telle que :

$$R_0(X_0) \wedge P(x) \wedge T \Rightarrow \neg(H(x) \wedge \neg I(x) \wedge C_2(x))$$

Si $\neg(H(x) \wedge \neg I(x) \wedge C_2(x))$ convient initialement alors R_0 est initialisé à cette formule. L'énoncé E_0 construit est :

profil	:	cond=C(x : t_x) → booléen
prec	:	P(x)
post	:	(cond=vrai) ≡ $R_0(X_0)$
trace	:	T

14 . Elaboration de l'énoncé E_1. Demande d'une hypothèse Q_0 telle que $Q_0(x, var) \wedge P(x) \wedge R_0(X_0) \Rightarrow Q(x, var)$. $R_0(X_0)$ doit intervenir lors de la recherche de Q_0 afin que l'introduction d'une conditionnelle soit effectivement pertinente.

Si Q_0 n'est pas trouvé il y a échec sinon l'énoncé E_1 élaboré est :

profils	:	..., $var\_autre_j = f_{0,j}(x : t_x) \rightarrow t_j$, ...
		$var_i = f_0(x : t_x) \rightarrow t_i$
prec	:	$P(x) \wedge R_0(X_0)$
post	:	$Q_0(x, var)$
trace	:	T

15. Preuve de E_1. Soit $C_1(x)$ la précondition dérivée. Si elle est identiquement faux l'arbre est modifié comme au pas 12.

16 . Preuve de E_0, si cet énoncé a été constitué. Soit $C_0(x)$ la précondition dérivée.

17. La précondition finale est élaborée à partir de la formule $C_0(x) \wedge (C_1(x) \vee \neg R_0(X_0))$. C_1 et $\neg R_0$ doivent être simplifiés par l'hypothèse $C_0(x)$.

- Demande d'une hypothèse D_1 telle que $D_1(x) \wedge P(x) \wedge C_0(x) \wedge T \Rightarrow C_1(x)$

- Demande d'une hypothèse D_2 telle que $D_2(x) \wedge P(x) \wedge C_0(x) \wedge T \Rightarrow \neg R_0(X_0)$

La précondition finale renvoyée est donc $C_0(x) \wedge (D_1(x) \vee D_2(x))$.

18 . Comme dans la stratégie précédente, si la précondition dérivée est différente de **vrai** et que la preuve est conservée il faut s'assurer que les arguments des appels récursifs appartiennent aussi au sous-ensemble des données pour lequel cette preuve a été possible.

Si $C_2(x) \neq$ **faux**, c'est à dire si il y a eu effectivement introduction d'appels récursifs, alors, pour chacun des appels introduits, il y a demande d'une hypothèse R_i telle que :

$$R_i(x) \wedge P(x) \wedge C_0(x) \wedge (D_1(x) \vee D_2(x)) \wedge Q_1(X, w) \wedge Q_3(x, y, var\_autre)) \wedge A(x, y) \wedge B(x, y)$$
$$\Rightarrow C_0(w_i \cup y_i) \wedge (D_1(w_i \cup y_i) \vee D_2(w_i \cup y_i))$$

Si l'un des R_i ne peut pas être trouvé il y a échec sinon il y a demandé d'une hypothèse simplificatrice R telle que :

$$R(x) \wedge P(x) \wedge C_0(x) \wedge (D_1(x) \vee D_2(x)) \wedge T \Rightarrow \Pi_{(i=1, k)} R_i(x)$$

Si R(x)=**vrai** la précondition dérivée n'est pas modifiée sinon elle devient $R(x) \wedge C_0(x) \wedge (D_1(x) \vee D_2(x))$. Si cette preuve n'est toujours pas remise en cause il faut réitérer cette dernière étape jusqu'à obtenir R(x)=**vrai** ou jusqu'à abandonner cette preuve.

fin-stratégie.

Pour élaborer cette stratégie nous n'avons fait que reprendre le déroulement classique des stratégies déja exposées en remplaçant les pas relatifs à la preuve des sous-problèmes par l'appel de stratégies adéquates. La stratégie utilisée au premier niveau est l'introduction de conditionnelle par **Instanciation de la condition** dont les principales étapes sont :

1 - Recherche de l'énoncé E_1 de la première branche de la conditionnelle ;

2 - Recherche de l'énoncé E_2 de la seconde branche de la conditionnelle ;

3 - preuve de E_2 ;

4 - preuve de E_1 ;

5 - preuve de l'énoncé de la condition ;

6 - élaboration de la précondition dérivée.

L'étape 3 est un appel de la stratégie introduction d'une composition fonctionnelle par **Instanciation des fonctions intermédiaires**, l'instanciation découlant de la décomposition fournie par l'utilisateur. Les contraintes propres à la récursion, existence d'un ordre bien fondé et preuve que les arguments des appels vérifient la précondition nécessaire au bon déroulement de la preuve, sont traitées respectivement aux pas **4**, **5** et **18**.

Lorsque cette stratégie s'applique sur plusieurs fonctions ou lorsque la décomposition choisie est plus complexe le déroulement de cette stratégie n'est pas modifiée si ce n'est qu'il y aura alors plus de fonctions intermédiaires à synthétiser.

CONCLUSION

CONCLUSION

" Formalization is an experimental science "
D. Scott

Les connaissances collaborant à l'élaboration d'un programme sont de trois types :

- Celles relatives à l'activité de programmation. Ces connaissances peuvent être d'ordre très différent. Elles vont, par exemple, de la maitrise du langage de programmation utilisé, de la connaissance de schémas d'algorithmes classiques à l'acquisition de techniques d'optimisation (élimination de récursivités terminales, programmation dynamique, codage ...).

- Celles relatives au domaine du problème. Elles permettent de garantir la correction du programme engendré. Le programmeur n'a généralement pas en tête ces connaissances de manière explicite mais il les utilise lors de la conception et de la mise au point du programme.

- Celles relatives à l'analyse du problème. Ces connaissances doivent plutôt être qualifiées de méta-connaissances puisqu'elles permettent de trouver l'idée algorithmique à partir du problème à traiter, des connaissances de programmation et des connaissances sur le domaine du problème.

Dans cette thèse nous nous sommes intéressés à une représentation des connaissances, dites de programmation, sous forme d'un système déductif et de stratégies. La sémantique du langage cible est définie sous forme axiomatique, en terme d'énoncés de la forme $\mathcal{P}$: $\forall$ x (P(x) $\rightarrow$ Q(x, f(x))) où P et Q sont des formules de la logique du premier ordre. D'autre part nous avons adopté une démarche générale de construction sous la forme :

1) choix d'un schéma de programme ;

2) recherche des propriétés le validant ;

3) élaboration des sous-problèmes à résoudre ;

4) construction de programmes répondant à ces sous-problèmes ;

5) évaluation de la classe des données pour laquelle la construction retenue aboutit ;

6) itération si ce processus n'a abouti que partiellement ou si il a échoué.

Enfin, dans les stratégies proposées, nous avons introduit des optimisations très simples permettant de transformer les programmes construits (par exemple $f(x) \leftarrow$ **si faux alors** $f_1(x)$ **sinon** $f_2(x)$ est simplifié en $f(x) \leftarrow f_2(x)$). Les optimisations proposées sont syntaxiques, dans le sens où elles ne prennent pas en compte des propriétés propres au domaine du problème.

Les connaissances sur les domaines de problème, quant à elles, ont implicitement été assimilées à des théorèmes de théories du premier ordre. Ces théories peuvent être représentées par une ou plusieurs axiomatisations, ceci sous la forme d'axiomes propres à ces théories et de mécanismes déductifs associés à la logique du premier ordre. On peut aussi supposer disposer de théorèmes *in extenso* qui sont jugés intéressants (propriétés de commutativité ou d'associativité de certains opérateurs ...). Parmi ces axiomes nous devons disposer, si le domaine s'y prête, de principes d'induction. D'autre part, l'ensemble des théories attachées aux domaines de problèmes possibles peuvent être structurées [BuG 77].

Nous ne nous sommes pas préoccupés des connaissances relatives à l'analyse du problème. Nous avons supposé disposer d'un interlocuteur " intelligent " capable de décider des choix à faire, c'est à dire de la preuve à développer. Dans le cadre de la synthèse assistée de programmes, cet interlocuteur a été assimilé à l'utilisateur. Il n'existe bien sûr pas de modélisation complète de ces connaissances puisqu'elles reposent sur des concepts tels que l'expérience (apprentissage mais aussi réutilisation, à bon escient, des acquis), l'intuition, l'analogie ... Néanmoins plusieurs approches sont possibles pour guider les choix à faire, les critères utilisés dépendant soit de la forme de l'énoncé à traiter (critères syntaxiques) soit du domaine du problème (critères sémantiques).

Dans le cas de notre approche, un critère syntaxique pourrait être, par exemple, que l'énoncé à prouver "a la forme" d'une des spécifications définissant la sémantique des différentes constructions possibles du langage cible. En effet, ces spécifications ont permis de caractériser les propriétés du domaine du problème sous-jacentes à chaque construction. Si l'énoncé correspond à une de ces spécifications nous sommes alors assurés de trouver, au moins au premier niveau du développement de la preuve, les théorèmes adéquats. Par exemple si la postcondition de l'énoncé est de la forme $Q_1(x, f(x)) \vee Q_2(x, f(x))$ alors f peut être définie par $f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$ **sinon** $f_2(x)$ et les postconditions caractérisant

les fonctions $f_1$et f_2 sont respectivement $Q_1(x, f_1(x))$ et $Q_2(x, f_2(x))$. En effet, les postconditions $L(x, f_1(x))$ et $L_2(x, f_2(x))$ caractérisant f_1 et f_2 doivent être telles que :

(a) $\forall x, y\ (P(x) \wedge Q_0(x) \wedge L_1(x, y) \Rightarrow (Q_1(x, y) \vee Q_2(x, y)))$

(b) $\forall x, y\ (P(x) \wedge Q_0(x) \wedge L_2(x, y) \Rightarrow (Q_1(x, y) \vee Q_2(x, y)))$

$Q_1(x, y)$ et $Q_2(x, y)$ sont bien des solutions possibles pour $L_1(x, y)$ et $L_2(x, y)$ puisque $A \Rightarrow A \vee B$ est une formule valide quelques soient A et B. Un second critère syntaxique peut être de reconnaître dans la postcondition un "morceau de programme". Rappelons qu'un programme peut être exprimé par un énoncé dont la postcondition est la relation $f(x) = T$ où T est le terme du langage cible définissant f pour l'argument x. Reconnaître un lien de cette forme peut donc être une indication pour choisir de construire un programme ayant la structure du terme T. Par exemple si une postcondition contient l'égalité $f(x) =$ **si** C **alors** t_1 **sinon** t_2 où C, t_1 et t_2 sont des termes dont les seules variables libres appartiennent au n-uplet x, alors on peut choisir d'élaborer la définition $f(x) \leftarrow$ **si** $C(x)$ **alors** $f_1(x)$ **sinon** $f_2(x)$. Les postconditions relatives à f_1 et f_2 peuvent être obtenues à partir de la postcondition Q en remplaçant le terme $f(x) =$ **si** C **alors** t_1 **sinon** t_2 respectivement par $f_1(x) = t_1$ et $f_2(x) = t_2$. Ces postconditions sont donc :

$$Q(x, f(x))\ [f(x) = \textbf{si } C \textbf{ alors } t_1 \textbf{ sinon } t_2 / f_1(x) = t_1]\ [f(x) / f_1(x)]$$

$$Q(x, f(x))\ [f(x) = \textbf{si } C \textbf{ alors } t_1 \textbf{ sinon } t_2 / f_2(x) = t_2]\ [f(x) / f_2(x)]$$

Ces deux solutions vérifient bien les implications (a) et (b) relatives à l'introduction de condition.

Une autre approche consiste à utiliser des critères sémantiques, c'est à dire dépendants des propriétés du domaine du problème ou des opérateurs apparaissant dans l'énoncé. La recherche de tels critères en démonstration automatique est un sujet d'étude important, notamment pour les preuves par induction. Ces preuves sont basées sur le principe d'induction structurelle [Bur 69], la difficulté consiste alors à choisir le sous-ensemble des données sur lequel faire porter l'induction ainsi que le schéma d'induction approprié. La stratégie générale consiste à se baser sur les définitions récursives des opérateurs apparaissant dans l'énoncé pour fixer ces paramètres [BoM 79], [BiC 85], [Fra 85], [Biu 86]. Dans [JaP 86] nous montrons sur des exemples comment synthétiser une définition de fonction, donnée en terme d'un ensemble E_1 de constructeurs, en une définition équivalente basée sur un autre ensemble E_2 de constructeurs, en utilisant les équations définissant E_1 en terme de E_2. Ceci permet, une fois le schéma d'induction retenu, de déduire les définitions nécessaires à la mise en place de la preuve. Les critères que nous venons de proposer prennent en compte uniquement le fait que la preuve retenue peut aboutir au premier niveau. Il peut donc exister une preuve plus pertinente, vis à vis du programme qui en découle. D'autre part, ces critères ne sont pas complets puisque soit ils découlent de la syntaxe de l'énoncé qui bien sûr ne reflète pas nécessairement la forme du problème à résoudre soit ils

sont basés sur la connaissance explicite du domaine du problème, qui n'est forcément que partielle. Néanmoins il est nécessaire de développer de tels critères mêmes, et surtout, si ceux-ci ne permettent de faire les choix que dans des cas simples.

Nous ne nous sommes pas non plus préoccupés du problème de la recherche d'hypothèses qui consiste, rappelons le, à trouver les hypothèses permettant de prouver un but B à partir d'un ensemble A de prémisses connues. Ce processus est, dans toute sa généralité, extrèmement complexe. D'une part la réponse espérée doit vérifier certaines contraintes (être la plus simple possible, lier un ensemble de variables entre elles) et d'autre part la forme de raisonnement nécessaire à la recherche de ces hypothèses peut être très variée. On peut par exemple chercher si il n'existe pas directement une instance appropriée de la forme $C \Rightarrow (A \Rightarrow B)$, ou on peut transformer B en utilisant les définitions des opérateurs le constituant ... Il est donc nécessaire de trouver des stratégies, et des critères pour les appliquer, de la même manière que pour le développement de la partie constructive de la preuve. Smith, par exemple, se base sur la forme syntaxique de B pour décomposer la recherche d'une hypothèse en un ensemble de problèmes de recherche d'hypothèses pour des buts plus simples. Pour établir des critères sémantiques, c'est à dire dépendant du domaine du problème, il faudrait disposer d'informations sur les théorèmes contenus explicitement dans la base de théorèmes. Bien que nous ayons dissocié explicitement les deux niveaux de preuve (preuve pour la construction et preuve sur le domaine du problème) la problématique du développement d'une preuve est la même. Le but est du même ordre : trouver une fonction ou trouver une relation et la preuve s'exprime dans les deux cas en terme de recherche d'hypothèses. La complexité de la recherche d'hypothèses est accrue par le fait que les déductions et les axiomes sont plus nombreux que pour la partie constructive de la preuve, néanmoins les stratégies développées pour la synthèse pourraient être applicables pour la recherche d'hypothèses.

Il reste donc beaucoup à faire ... Notamment, nous sommes loin de pouvoir proposer un outil sérieux d'aide à la construction de programmes. En effet, en plus du problème de la recherche des hypothèses nécessaires et des choix à effectuer, il faudrait développer un véritable outil d'aide à la mise au point de programmes. L'utilisateur devrait pouvoir stopper le déroulement d'une preuve, la modifier ... Dans le cadre déductif que nous avons adopté, il faudrait, pour faire cela, représenter les développements comme des objets manipulables et transformables [Sin 87]. Ces objets devraient contenir plus d'informations que l'arbre de preuve que nous avons décrit, notamment il faudrait conserver les propriétés sur le domaine du problème ayant permis le déroulement d'une preuve particulière. D'autre part, il serait intéressant de chercher à utiliser les échecs découlant des choix stratégiques faits, ceci de manière générale. Néanmoins l'axiomatisation que nous avons proposée est une étape nécessaire. A partir de cette axiomatique, il est possible de décrire les développements et de proposer des stratégies de développement plus évoluées ou plus spécialisées. L'axiomatique proposée permet, pour de nouveaux schémas de programmes, de déduire la preuve à effectuer, et peut être même des stratégies de preuve, à partir de celles proposées. Par exemple on pourrait proposer un schéma, qualifié *d'itératif*, de la forme $f(x) \leftarrow$ **si** $C(x)$ **alors** $f_0(x)$ **sinon** $f(h(x))$. Ce schéma est intéressant à considérer puisque, la

récursion étant terminale, il peut être implanté efficacement sous forme itérative. Un tel schéma est une instance du schéma récursif ; la fonction de recomposition est l'identité et le nombre d'appels récursifs est réduit à 1. La preuve à effectuer découle donc directement de celle que nous avons donnée dans le cas général et une stratégie applicable est la stratégie Introduction d'une récursion par instanciation de la fonction de recomposition, puisque la fonction de recomposition est l'identité. Une autre classe intéressante de schémas peut être obtenue en intégrant des transformations de programmes dépendantes de propriétés sur le domaine du problème. Par exemple considérons la définition : f(x) ← **si** C(x) **alors** f_0(x) **sinon** g(f(h(x)), x). On peut montrer, si g est une fonction associative, que cette définition est équivalente aux 2 définitions :

f(x) ← **si** C(x) **alors** f_0(x) **sinon** F(h(x), x)
F(x, n) ← **si** C(x) **alors** g(f_0(x), n) **sinon** F(h(x), g(n, x))

Cette transformation est valide, si g est associative, quelques soient les fonctions C, f_0 et h ([KiS 81], [Bac 85]). Elle permet de réécrire une définition intrinsèquement récursive en une forme itérative, en ajoutant un paramètre. On pourrait proposer un schéma *récursif-itératif*, engendrant directement une telle définition, ceci étant transparent pour l'utilisateur. La preuve associée à l'élaboration de la définition ci-dessus est donc la même que celle utilisée pour la définition f(x) ← **si** C(x) **alors** f_0(x) **sinon** g(f(h(x)), x), il suffit d'ajouter une prémisse décrivant l'associativité à gauche de la fonction g. Cette propriété se traduit par la formule :

$$\forall\, a, b, c, w_1, w_2, r\ (Q_g(b, c, w_1) \wedge Q_g(a, w_1, r) \Rightarrow Q_g(a, b, w_2) \wedge Q_g(c, w_2, r))$$

On peut alors appliquer les mêmes stratégies que pour l'introduction de récursion. Il suffit de vérifier cette contrainte, une fois l'énoncé de g établi.

Il est également possible de proposer des schémas correspondant à des classes de problèmes. Par exemple Bidoit, Guiho et Gresse [BGG 79] ont proposé un système adapté à la synthèse de programmes opérant sur des tableaux, en traitant différentes formes de problèmes sur cette structure de données. Par exemple l'une des formes classiques de problème est la recherche d'un indice i compris entre a et b tel que P(T[i], x) est vrai, P, T, x, a, b étant les paramètres du problème. Un schéma de programme correspondant à cette forme générique de problème est :

f(x, T, a, b) ← g(x, T, a, b, a)
g(x, T, a, b, i) ← **si** i > b **alors** ⊥
sinon si P(T[i], x) **alors** i **sinon** g(x, T, a, b, i+1)

Ce schéma peut, en fait, être généralisé à tous les problèmes dans lequel le résultat est trouvé par énumération de

l'ensemble des solutions possibles. Par exemple la recherche d'un diviseur différent de 1 d'un entier n peut être décrite par le programme suivant :

diviseur(n) ← g(n, 2)
g(n, m) ← **si** m / n **alors** m
sinon si m+1 ≥ n div 2 **alors** n **sinon** g(n, m+1)

Pour un problème de la forme *Trouver r tel que P(r, x)* un schéma général peut donc être :

f(x) ← g(x, elem-init, 1)
g(x, e, indice) ← **si** indice > borne-sup **alors** ⊥
sinon si P(e, x) **alors** e **sinon** g(x, elem-succ(e, indice), indice+1)

Ce schéma est utilisable à condition de trouver une énumération de l'ensemble des solutions possibles. Une telle énumération est décrite ici par une fonction *elem-succ* qui renvoie la solution possible suivante, à partir de l'élément *e* qui vient d'être énuméré et d'un compteur *indice*. L'énumération est initialisée par la fonction *elem-init* qui fournit le premier élément de l'énumération. *borne-sup* teste la fin de l'énumération, si celle-ci est finie. La preuve à mettre en place découle des règles de la composition fonctionnelle (pour la définition de f) et de la règle de la récursion (pour la définition de g). On pourrait proposer un schéma *énumération + test* basé sur le choix d'une fonction d'énumération. Il faudrait pouvoir caractériser plus précisément ce qu'est une fonction d'énumération, en fonction du problème à résoudre.

L'intégration de nouveaux schémas ne pose pas en théorie de difficultés puisque la preuve à effectuer découle de la forme syntaxique de ces schémas. Certains schémas peuvent être basés sur des propriétés des fonctions construites (associativité, fonction d'énumération). Dans notre cadre, ces propriétés doivent nécessairement être décrites par des formules du premier ordre. Or ceci n'est pas toujours possible et, même lorsque c'est le cas, en rechercher des instances ne reviendra pas toujours à rechercher les hypothèses nécessaires à la preuve d'une formule de la forme A ⇒ B. Par exemple la transformation d'un programme récursif en sa version itérative est plus générale que celle que nous avons donnée. Elle permet en fait de réécrire toute définition de la forme f(x) ← **si** C(x) **alors** $f_0(x)$ **sinon** g(f(h(x)), x) en :

f(x) ← F(x, $f_0(x)$)
F(x, n) ← **si** C(x) **alors** n **sinon** F(f(h(x)), g'(n, x))

Cette transformation est valide à condition qu'il existe une fonction g' telle que g(x, g'(y, z)) = g'(g(x, y), z) et g(x, $f_0(x)$) = g'($f_0(x)$, x). Ces propriétés décrivent la fonction g de manière récursive. Nous ne pourrons nous

intéresser à de telles transformations que si nous sommes capables, au moins en partie, de traiter des énoncés définissant récursivement les objets cherchés. Rappelons que ceci fait de la synthèse un problème du second ordre puisqu'il peut être nécessaire de raisonner sur la structure du programme en cours de construction (référence exemples de synthèses du pgcd, chapitre 2 section 2.4). La possibilité d'effectuer des preuves du second ordre rendra donc le processus de synthèse plus " créatif ".

BIBLIOGRAPHIE

[Abr 84] J. ABRIAL
Spécifier ou comment matérialiser l'abstrait.
TSI, vol 3, no 3, 1984.

[AHU 74] A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN
The design and Analysis of Computer Algorithms.
Addison Wesley, 1974.

[AHU 82] A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN
Data structures and Algorithms.
Addison Wesley, 1982.

[BaF 82] A. BAAR, E. FEIGENBAUM
The handbook of Artificial Intelligence.
vol 2, A. Baar et E. Feigenbaum eds, PITMAN PRESS, 1982.

[Bac 78] J. BACKUS
Can programming be liberated from the von Neumann style?
A functional style and its algebra of programs.
Comm. of the ACM, vol 21, no 8, aout 1978.

[Bac 85] J. BACKUS
From function level semantics to program transformation and optimization.
Mathematical Foundations of software development, TABSOFT Berlin,
Vol 1, CAAP'85, LNCS 185, 1985.

[Bar 79] D. BARSTOW
An experiment in Knowledge-based Automatic Programming.
Artificial Intelligence, no 12, pp 73-119, 1979

[BCF 85] R. BRENA, R.CAFERRA, B. FRONHÖFER, C. GRESSE, P. JACQUET, M-L. POTET
Program synthesis through problem splitting: a method for subproblem characterization.
Computers and Artificial Intelligence, vol 4, no 5, pp 421-429, 1985.

[BeE 86] B. BERT, R. ECHAHED
Design and Implementation of a generic, logic and functional programming language.
Proc. of the European Symposium on Programming, Saarbrücken,
LNCS 213, pp 119-132, 1986.

[Ben 85] S. BENSALEM
Algèbre de programmes dans un univers typé.
Thèse de Docteur de troisième cycle,
Institut Polytechnique de Grenoble, décembre 85.

[Bib 78] W. BIBEL
On strategies for the synthesis of algorithms.
Proceeding on Artificial Intelligence and Simulation of Behavior,
Hambourg (G), 18-20 juillet 1978.

[Bib 80] W. BIBEL
Syntax-directed, semantics-supported program synthesis.
Artificial Intelligence, no 14, 1980.

[BiC 85] M. BIDOIT, M-A CHOQUER
Preuves de formules conditionnelles par récurrence.
5ième Congrés AFCET " Reconnaissance des formes et Intelligence artificielle ",
Grenoble, 27-29 novembre 1985.

[BiH 84] W. BIBEL, K.M. HORNING
LOPS - a system based on a strategical approach to program synthesis.
in Automatic program construction techniques, A. Biermann,
G. Guiho, Y. Kadratoff eds, MacMillan, New-York 1984.

[Bie 85] A.W. BIERMANN
Automatic programming: a tutorial on formal methodologies.
Journal of Symbolic Computation, vol 1, no 2, 1985.

[BiK 76] A.W. BIERMANN, R. KRISHNASWANY
Constructing Programs from Example computations.
IEEE Trans. Soft. Eng., SE2, octobre 1976.

[Biu 86] S. BIUNDO
A synthesis system mechanizing proofs by induction.
ECAI 7, Brigthon, 1986.

[BoM 79] R.S. BOYER, J.S. MOORE
A Computational Logic.
Academic Press, New York, 1979.

[Bru 68] N.G. de BRUIJN
The mathematical language AUTOMATH, its usage and some of its extensions.
Symposium on Automatic Demonstration, IRIA, Versailles, 1968.
Présenté aussi dans Springer-Verlag Lecture Notes in Mathematics 125, pp 29-61, 1970.

[BuD 77] R.M. BURSTALL, J. DARLINGTON
A transformation system for developping recursive programs.
Journal of the ACM, vol 24, no 1, janvier 1977.

[BuG 77] R.M. BURSTALL, J.A. GOGUEN
Putting theories to make specifications.
IJCAI 77, MIT, Cambridge, Massachusetts, 22-25 aout 1977.

[BuG 80] R. BURSTALL, J. GOGUEN
The semantics of CLEAR, a specification language.
Proc. of Advanced Course on Abstract Software Specifications.
LNCS, no 86, pp 292-332, Copenhague, 1980.

[Bur 69] R.M. BURSTALL
Proving properties of programs by structural induction.
Computer journal 12, pp 41-48, 1969.

[Car 63] J. McCarthy
A basis for a mathematical theory of computation.
Computer programming anf Formal systems, P. Braffort, D. Hirschberg eds,
Studies in Logic on the Foundations of Mathematics.
North-Holland pub. C. Amsterdam, pp 33-70, 1963.

[ClD 78] K. L. CLARK, J. DARLINGTON
Algorithm classification through synthesis.
The Computer Journal, vol 23, no 1, juin 1978.

[CoH 84] Th. COQUAND, G. HUET
A theory of Constructions.
Version préliminaire, International Symposium on Semantics of Data Types, Sophia-Antipolis, juin 1984.

[Com 86] H. COMMON
Sufficient completeness, term rewriting systems and anti-unification.
CAD 8, LNCS, 1986.

[Coo 78] S. COOK
Soundness and completeness of an axiom system for program verification.
SIAM J. COMPUT., vol 17, no 1, février 1978.

[Dar 75] J. DARLINGTON
Application of program transformation to program synthesis.
Arc et Sénans, 1-3 juillet 1975

[DeB 80] J. DE BAKKER
Mathematical theory of program correctness
Prentice Hall, 1980.

[DeD 80] A. DEMERS, J. DONAHUE
Datatypes, parameters and type checking.
7th ACM Symposium on Principles of Programming Languages, Las Vegas, 1980.

[DeM 77] N. DERSHOWITZ, Z. MANNA
The evolution of programs: automatic program modification.
IEEE Trans. Software Engineering, SE 3(6), pp 377-385, novembre 1977.

[Der 82] N. DERSHOWITZ
Applications of the Knuth-Bendix Completion Procedure.
Proc. of the Séminaire d'Informatique Théorique, Paris, France, decembre 1982.

[Der 83] N. DERSHOWITZ
Computing with rewrite systems.
Aerospace Report, no ATR-83 (8478), 1983.
Paru aussi dans Information and controle, 1985.

[Der 85] N. DERSHOWITZ
Synthesis by Completion.
IJCAI 85, 1985.

[Dij 76] E.W. DIJKSTRA
A discipline of Programming.
Englewood Cliff, Prentice Hall, 1976.

[Don 82] M. O'DONNEL
A critique of the Foundations of Hoare style programming logics.
Comm. of the ACM, vol 25, no 12, decembre 1982.

[DoV 87] D. DOLLE, J.R. VIGOUROUX
Projet SCALP.
Rapport de Projet troisième année ENSIMAG, juin 1987.

[Epp 85] D. EPPSTEIN
A Heuristic Approach to Program Inversion.
IJCAI 85, 1985.

[FGJ 85] K. FUTATSUGI, J. GOGUEN, J-P. JOUANNAUD, J. MESEGUER
Principles of OBJ2.
in Proceedings on Principles of Programming Languages,
Association for Computing Machinery, pp 52-66, 1985.

[Flo 67] R.W. FLOYD
Assigning meanings to programs.
in J.T Schwartz ed, Mathematical aspects of Computer Science,
Proc. Symposia in Applied Mathematics 19, Providence (R.I), Amer. Math. Soc., pp 19-32, 1967.

[Fra 84] M. FRANOVA
Program Synthesis and Constructive Proofs.
Cybernetics and Systems Research 2, R. Trappl (eds),
Elsevier Science Publishers (North Holland), 1984.

[Fra 85] M. FRANOVA
Preuves constructives dans des domaines constructibles.
5e congrès AFCET " Reconnaissances des formes et Intelligence
Artificielle " , GRENOBLE, 27-29 novembre 1985.

[FrF 86] B. FRONHÖFER, U. FURBACH
Knuth-Bendix completion versus fold/unfold: a comparative study in program synthesis.
Bericht nr 8604, Universität der Bundeswehr München,
Fakultät für Informatik, Avril 1986.

[Fro 85] B. FRONHÖFER
The LOPS-Approach: towards new syntheses of algorithms.
ÖGAI-85, Vienna, Austria, H. Trost, J. Relli eds,
Informatik-Fachberichte, no 106, Springer,
Berlin, pp 164-172, septembre 1985.

[Ger 75] S.L. GERHART
Knowledge about programs: a model and case study.
International Conference in reliable software, pp 88-94,
Los Angeles, avril 1975.

[GMW 79] M. GORDON, R. MILNER, C. WADSWORTH
Edinburgh LCF.
Lecture Notes in Computer Science, vol 78, Springer-Verlag, New York, 1979.

[Got 79] S. GOTO
Program synthesis from natural deduction proofs.
IJCAI 79, 1979.

[Gra 86] Anna GRAM
Raisonner pour programmer.
Dunod 1986.

[Gre 69] C. GREEN
Application of theorem proving to problem solving.
IJCAI, Washington (D.C), mai 7-9 1969.

[Gre 76] C. GREEN
The design of the PSI program synthesis system.
Proc. 2nd International Conference on Software Engineering,
San Francisco, octobre 1976.

[Gre 84] C. GRESSE
Contribution à la programmation automatique. "CATY: un système de construction assistée de programmes".
Thèse de Docteur d'Etat, Université de PARIS-SUD, centre d'Orsay, mars 1984.

[Gri 81] D. GRIES
The science of programming.
Springer-Verlag, 1981.

[HiS 86] J.R. HINDLEY, J. SELDIN
Introduction to Combinators and λ-Calculus.
London Mathematical Society, Student texts 1,
Cambridge University Press, 1986.

[Hoa 69] C.A.R. HOARE
An axiomatic basis for computer programming.
Comm. ACM, no 12, pp 576-580, 1969.

[Hog 78] C.J. HOGGER
Program synthesis in predicate logic.
Proceeding on Artificial Intelligence and Simulation of Behavior,
Hambourg (G), 18-20 juillet 1978.

[Hog 81] C.J. HOGGER
Derivation of Logic Programming.
Journal of the ACM, vol 28, pp 272-392, avril 1981.

[HoW 73] C.A.R. HOARE, N. WIRTH
An axiomatic definition of the programming language PASCAL.
Acta Informatica 2, Springer-Verlag 73, pp 335-355, 1973.

[How 80] W.A. HOWARD
The formulaes-as-types notion of construction.
To H.B. Curry : Essays on Combinatory Logic, lambda Calculus and formalism.
J.P Seldin, J.R. Hindley eds, 1980.

[Hsi 82] J. HSIANG
Topics in Automated Theorem Proving and Program Generation.
Thèse, Department of Computer Science,
University of Illinois at Urbana-Champaign, Urbana, Illinois, décembre 1982.

[Jac 88] P. JACQUET
Program synthesis by completion with dependent subtypes.
9th Conference on Automated Deduction, LNCS 310, 1988.

[JaP 86] P. JACQUET, M-L POTET
Program synthesis = proof method + knowledge.
7th European Conference on Artificial Intelligence, Brighton, 1986.

[Joh 75] S.C. JOHNSON
Yacc : Yet Another Compiler Compiler.
Computing Science Technical Report 32,
Bell Laboratories, Murray Hill N. J., 1975.

[Jon 80] C.B. JONES
Software Development: a rigourous approach.
Prentice Hall, 1980.

[Kan 83] E. KANT
On the Efficient Synthesis of Efficient Programs.
Artificial Intelligence, no 20, pp 253-305, 1983.

[KaN 83] E. KANT, A. NEWELL
An automatic algorithm designer: an initial implementation.
AAAI 83, 1983.

[Kan 85] E. KANT
Understanding and automating algorithm design.
IJCAI 85, 1985.

[KaS 83] D. KAPUR, G. SIVAHUMAR
Experiments with and architecture of RRL, a Rewrite Rule Laboratory.
Proc. NSF workshop on the Rewrite Rule Laboratory,
Schenectady, NY, septembre 1983.

[KiS 81] R.B. KIEBURTZ, J. SHULTIS
Transformations of FP programs schemes.
ACM proceedings of the 1981 conference on functionals
programming languages and computer architecture, 18-22 octobre 1981.

[Kla 84] H.A. KLAEREN
A constructive method for abstract algebraic software specification.
Theoritical Computer Science, pp 139-204, aout 1984.

[KnB 70] D.E. KNUTH, P.B. BENDIX
Simple word problems in universal algebras.
In Computational Problems in Abstract Algebra, J. Leeche, ed. Pergamon Press,
pp 263-297, 1970.

[KoP 83] Y. KODRATOFF, M. PICARD
Complétion de système de réécriture et synthèse de programmes à partir de leur spécification.
Journées BIGRE 83, 1983.

[KoZ 85] E. KOUNALIS, H. ZHANG
A general completeness test for equational specifications.
Centre de recherche en Informatique de Nancy, Nancy, France, 1985.

[Kow 74] R. KOWALSKI
Predicate logic as programming language.
Information processing 74, North-Holland publishing company, 1974.

[Kow 79] R. KOWALSKI
ALGORITHM = LOGIC + CONTROL.
Comm. of the ACM, vol 22, no 7, juillet 1979.

[Les 83] P. LESCANNE
Computer experiments with the REVE term rewriting system generator.
Proc. Tenth symposium on Principles Languages, Austin (texas), janvier 1983.

[Liv 78] C. LIVERCY.
Théorie des programmes : schémas, preuves, sémantique.
Dunod Informatique, 1978.

[Man 74] Z. MANNA
Mathematical theory of computation.
Mc Graw Hill, 1974.

[MaP 69] Z. MANNA, A. PNUELI
Formalization of properties of recursively defined functions.
ACM symposium on Theory of Computing, 1969.

[Mar 75] P. MARTIN-LÖF
An intuitionistic Theory of Types: predicative part.
Logic Colloquium 73, H.E. Rose, J.C. Shepherdson eds, North-Holland, Amsterdam, pp 73-118, 1975.

[MaW 71] Z. MANNA, R. WALDINGER
Toward automatic program synthesis.
Comm. ACM, vol 14, no 3, mars 1971.

[MaW 75] Z. MANNA, R. WALDINGER
Knowledge and Reasoning in Program Synthesis.
Artificial Intelligence no 6, pp 175-208, 1975.

[MaW 78] Z. MANNA, R. WALDINGER
The Logic of Computer Programming.
IEEE Transactions on software engineering, vol SE-24, no 3, mai 1978.

[MaW 79] Z. MANNA, R. WALDINGER
Synthesis: Dreams $\Rightarrow$ Programs.
IEEE Transactions on software engineering, vol SE-5, no 4, juillet 1979.

[MaW 80] Z. MANNA, R. WALDINGER
A deductive approach to program synthesis.
ACM Transactions on Programming Languages, vol 2, no 1, janvier 1980.

[MaW 81] Z. MANNA, R. WALDINGER
Deductive Synthesis of the unification algorithm.
Science of Computer Programming 1,
North-Holland Publishing Company, pp 5-48, 1981.

[Mil 78] R. MILNER
A theory of type polymorphism in programming.
Journal of Computer and Systems Sciences, no 16, pp 158-167, 1978.

[MMW 84] Z. MANNA, Y. MALACHI, R. WALDINGER
Tablog: the deductive-tableau programming language.
Technical Note, no 328, Computer Science Department,
Stanford University, Stanford,
CA and Artificial Intelligence Center, SRI International,
Menlo PARK (CA), septembre 84.

[Mcc 77] B.P. MacCUNE
THE PSI PROGRAM MODEL BUILDER : Synthesis of Very High-Level Programs.
Proc. of the Symposium on Artificial Intelligence Programming Languages.
SIGPLAN notices, vol 12, no 8, aout 1977.

[Mur 82] N. MURRAY
Completely nonclausal theorem proving.
Artificial Intelligence, vol 18, no 1, pp 67-85, 1982.

[Nie 84] H. NIELSON
Hoare's Logic for Run-time Analysis of Programs.
Thèse, Department of Computer Science,
University of Edinburgh, octobre 1984.

[NoP 83] B. NÖRSTROM, K. PETERSONN
Types and Specifications.
Information processing 83, R.E.A. Mason ed,
Elsevier Science Publishers B.V., north-Holland, IFIP 83, 1983.

[Par 69] D. PARK
Fixpoint induction and proofs of program properties.
Machine Intelligente 5, pp 59-78, 1969.

[Per 84] H. PERDRIX
Synthèse de programmes à partir de leurs spécifications.
Rapport de recherche no 187, Université de PARIS-SUD,
septembre 1984.

[Pot 84] M.L POTET
Etude de la synthèse automatique de programmes à partir d'une spécification complète.
Rapport Recherche, no 12-LIFIA, no 470-IMAG, novembre 1984.

[Rob 65] J.A. ROBINSON
A machine oriented logic based on the resolution principle.
JACM, vol 12, no 1, pp 23-41, janvier 1965.

[Sat 79] M. SATO
Towards a mathematical theory of program synthesis.
IJCAI 79, 1979.

[Sch 86] M. SCHMIDT-SCHAUSS
Unification in Many-sorted Equational theories.
8ième International Conference on Automated Deduction,
LNCS 230, pp 538-552, 1986.

[ScS 71] D. SCOTT, C. STRACHEY
Towards a mathematical semantics of computer language.
Technical Monograph PRG 6, Computing Laboratory,
OXFORD University, 1971.

[Sho 84] Y. SHOHAM
Knowledge inversion.
AAAI 84, 1984.

[Sim 63] H.A. SIMON
Experiments with an Heuristic Compiler.
JACM, pp 482-506, octobre 1963.

[Sin 87] M. SINTZOFF
Expressing program developments in a design calculus.
Logic of Programming and Calculi of discrete Design, Broy ed,
Springer-Verlag Berlin, 1987.

[Smi 82a] D.R. SMITH
Top-Down Synthesis of simple divide and conquer algorithms.
Technical Report, NPS 52-82-11, Dept. of Computer Science,
Naval Postgraduate School, Monterey (CA), 1982.

[Smi 82b] D. SMITH
Derived preconditions and their use in program synthesis.
6th Conf. on Aut. Deduc., LNCS 138,
Springer-Verlag N. Y., 1982.

[Smi 85a] D. R. SMITH
The design of divide and conquer algoritms.
Science of Computer Programming, no 5, pp 37-58, 1985.

[Smi 85b] D. R. SMITH
Top-down Synthesis of Divide-and-Conquer Algorithms.
Artificial Intelligence Journal, vol 27, p 43-96, septembre 1985.

[SyS 75] L. SYKLOSSY, D.A. SYKES
Automatic program synthesis from example problems.
IJCAI 4, TBILISSI (URSS), pp 268-278, 1975.

[Thi 84] J.J. THIEL
Stop losing sleep over incomplete data type specifications.
Proc. Eleventh Symposium on Principles of Programming Languages,
Salt Lake City, UT, janvier 84.

[Tyu 77] E.H. TYUGU
A programming system with automatic program synthesis.
Lecture Notes in Computer Science, no 47,
Methods of Algorithmic Language Inplementation,
pp 251-267, Springer-Verlag, 1977.

[UlM 77] J.W. ULRICH, R. MOLL
Programs synthesis by analogy.
SIGPLAN Notices, vol 12, no 8, Aout 1977.

[WaL 69] R. WALDINGER, R. LEE
PROW: a step toward automatic program writing.
IJCAI, Washington (DC), 7-9 mai, 1969.

[Wal 83] C. WALTHER
A many-sorted calculus based on Resolution and Paramodulation.
IJCAI 8, Karlsruhe, 1983.

REFERENCES

BIBLIOGRAPHIQUES

BAKKER (J.W)
Correctness of programs with function procedures.
LNCS 131, Logics of Programming, Springer-Verlag.

BERGSTRA (J.A) ET TUCKER (J.V)
The axiomatic semantics of programs based on Hoare's logic.
Acta Informatica, no 21, 1984.

BERGSTRA (J.A), TIURYN (J) ET TUCKER (J.V)
Floyd's principle, correctness theories and program equivalence.
TCS no 17, Noth-Holland Publishing Company, 1982.

BROY (M)
Program construction by transformations : a family tree of sorting programs.
A.W. Biermann et G. Guiho (eds), Computer Program synthesis Methodologies, 1983.

BUNDY (A)
The use of explicit plans to guide inductive proofs.
9th International Conference on Automated Deduction, LNCS 310, mai 1988.

BURSTALL (R.M.) et GOGUEN (J.A.)
Putting theories together to make specifications.
IJCAI 77, MIT, Cambridge, Massachusetts, 1977.

CIP LANGUAGE GROUP
The munich Project CIP.
LNCS, no 183, Springer-Verlag.

DERSHOWITZ (N) ET MANNA (Z)
The evolution of programs : automatic program modification.
IEEE transactions on software engineering, vol SE-3, no 6, novembre 1977.

ERIKSSON (L-H)
Synthesis of a unification algorithm in a logic programming calculus.
Journal of Logic Programming, 1984.

KRIEG-BRÜCKNER (B)
Algebraic formalisation of program development by transformation.
ESOP, Nancy (FR), mars 1988.

PARTSCH (R)
Transformational program development in a particular domain.

Science of Computer Programming, North-Holland, vol 7, 1986.

PARTSCH (H) ET STEINBRÜGGEN (R)
Program Transformation Systems
Computing Surveys, vol 12, no 3, septembre 1983.

REMY (J-L)
Construction, évaluation et amélioration systématiques de structure de données.
RAIRO Informatique théorique / theoretical Informatics, vol 14, no 1, 1980.

TRAUGOTT (J)
Deductive synthesis of sorting programs.
8th international Conference on Automated Deduction, Oxford, 1986.

SRIVAS (M.K.)
Automatic synthesis of implementation for abstract data types from algebraic specifications.
MIT / LCS / tech. Rep-276, Laboratory for Computer Science, MIT, juin 1982.

VON HENKE (F.W)
On generating Program from data types : an approach to automatic programming.
Colloques INRIA sur la Construction, l'amélioration et la vérification de programmes.
Arc et Sénans, 1975.

ANNEXES

SYNTHESE DU TRI PAR INSERTION

ET

SYNTHESE DU TRI PAR SELECTION

SYNTHESE D'UN ALGORITHME DE TRI PAR INSERTION

Enoncé:

profil	:	**tri = tri** (x: liste(entier)) → liste(entier)
préc	:	vrai
post	:	perm(x, tri) ∧ ord(tri)

Nous choisissons d'appliquer la stratégie " **décomposition d'une donnée** ".

• **Synthèse de la fonction tri.**

1. - Choix des données à décomposer : { **x** }.

   - Choix de la décomposition.

      - Cas de base.

         Indicateur : **x=nil**
         Valeur : **nil**.

      - Cas récursion.

         - Décomposition : { **car(x)** }, { **cdr(x)** }.
         - Enoncés :

profil :	**car**(x: liste(t)) → t	**profil :**	**cdr**(x: liste(t)) → liste(t)
préc :	x≠nil	**préc :**	x≠nil
post :	car(x) = **car**(x)	**post :**	cdr(x) = **cdr**(x)

2. - hypothèse en terme de **x** telle que **x≠nil** ⇒ **x≠nil** ?
   → vrai (on peut appliquer les sélecteurs car et cdr).

3. Choix des appels récursifs : { **cdr(x)** }. L'appel récursif introduit est **tri(cdr(x))**.

4. La précondition initiale étant **vrai** il n'est pas nécessaire de prouver que l'argument de l'appel récursif vérifie aussi cette précondition, ceci étant trivialement vrai.

5. Demande d'un ordre bien fondé **inf** tel que **x≠nil** ∧ **w=cdr(x)** ⇒ **inf(w, x)**. L'ordre classique, basé sur la longueur des listes, convient.

6. Recherche de l'énoncé caractérisant la fonction de recomposition.

- hypothèse en terme de **x, z** et **tri** telle que **x≠nil** ∧ **ord(z)** ∧ **perm(cdr(x), z)** ⇒ **perm(x, tri)** ?

profil : r = perm(x, tri : liste(t)) → booléen
préc : vrai
post : r = **si** x=nil **alors** tri=nil
sinon si dans(car(x), tri) **alors** perm(cdr(x), elim(car(x),tri))
sinon faux

→ z=elim(car(x), tri) ∧ dans(car(x), tri)

7. L'arbre engendré est alors :

Récursion
Décomposition des données

X = {x}
tri(x) ← **si** C_1(x) **alors** f_1(x) **sinon** g_1(tri(cdr(x)), x)

E_0 (énoncé de C_1) | E_1 (énoncé de f_1) | E_2 (énoncé de g_1)

Avec :

E_0 : **profil** : c = C_1(x: liste(entier)) → booléen
préc : vrai
post : (c=vrai) ≡ (x=nil ∨ ...)

E_1 : **profil** : tri = f_1(x: liste(entier)) → liste(entier)
préc : x=nil ∨ ...
post : ... ∧ ord(tri)

E_2 : **profil** : tri = g_1(z, x : liste(entier)) → liste(entier)
préc : x≠nil ∧ ord(z)
post : z=elim(car(x), tri) ∧ dans(car(x), tri) ∧ ord(tri)
trace : perm(cdr(x), z)

Il reste à prouver dans l'ordre l'énoncé E_2, E_1 et E_0 et à élaborer la précondition dérivée finale. Nous présentons maintenant une preuve de l'énoncé E_2.

Synthèse de la fonction g_1

Enoncé :

profil	:	tri = g_1(z, x : liste(entier)) → liste(entier)
préc	:	x≠nil ∧ ord(z)
post	:	z=elim(car(x), tri) ∧ dans(car(x), tri) ∧ ord(tri)
trace	:	perm(cdr(x), z)

La donnée x intervient dans la postcondition uniquement sous la forme car(x). Nous choisissons donc d'appliquer la stratégie introduction d'une composition fonctionnelle par **Instanciation des fonctions intermédiaires.**

• **Déroulement.**

1. Choix de la fonction intermédiaire **car** et de son argument **x** .

2. - hypothèse en terme de **x** telle que **x≠nil** ⇒**x≠nil** ? (on peut appliquer la fonction **car**)
   → vrai

3. - hypothèse en terme de **n et tri** telle que **x≠nil** ∧ **n=car(x)** ⇒ **z=elim(car(x), tri)** ?
   → z=elim(n, tri)

   - hypothèse en terme de **n et tri** telle que **x≠nil** ∧ **n=car(x)** ⇒**dans(car(x), tri)** ?
   → dans(n, tri)

L'arbre de preuve est :

Composition fonctionnelle
instanciation des fonctions intermédiaires
g_1(z, x) ← insert(z, car(x))

profil	:	tri = **insert**(z : liste(entier), n : entier) → liste(entier)
préc	:	ord(z)
post	:	z=elim(n, tri) ∧ dans(n, tri) ∧ ord(tri)
trace	:	perm(cdr(x), z), n=car(x), x≠nil

La fonction introduite est la fonction d'insertion d'un élément dans une liste triée, c'est pour cela que nous l'avons nommée **insert**. x n'intervient plus dans la postcondition, les hypothèses sur x sont donc conservées non plus dans la précondition mais dans la trace de l'énoncé.

Il reste à prouver cet énoncé (pas 5 .2 de la stratégie **Instanciation des fonctions intermédiaires**) puis à élaborer la précondition dérivée-finale. Nous présentons maintenant la synthèse de la fonction **insert**. La stratégie choisie est la stratégie **Décomposition des données.**

Synthèse de la fonction insert

• Déroulement.

1. - Choix des données à décomposer : { z }.

   - Choix de la décomposition.

      - Cas de base.

         Indicateur : **z=nil.**
         Valeur : **nil.**

      - cas récursion.

         Décomposition: { **car(z)** }, { **cdr(z)** }.
         Enoncés :

profil :	**car**(z : liste(t)) → t	**profil :**	**cdr**(z : liste(t)) → liste(t)
préc :	z≠nil	**préc :**	z≠nil
post :	car(z) = **car**(z)	**post :**	cdr(z) = **cdr**(z)

2. - hypothèse en terme de z telle que **ord(z) ⇒ z≠nil** ?
   → z≠nil

3. Choix des appels récursifs : { **cdr(z)** }. Il y a un seul appel récursif introduit, c'est insert(cdr(z), k(z, n)), où k est une nouvelle fonction à synthétiser. Elle permet de calculer le second argument de l'appel récursif introduit.

4. - hypothèse en terme de z telle que **z≠nil ∧ ord(z) ⇒ ord(cdr(z))** ? (l'appel récursif vérifie la précondition initiale)
   → vrai

5. Demande d'un ordre bien fondé **inf** tel que **z≠nil ∧ w=cdr(z) ⇒ inf((w, k), (x, n))**. L'ordre classique, basé sur la longueur des listes, convient.

6. Recherche de l'énoncé relatif à la fonction k et à la fonction de recomposition g_2.

   - hypothèse en terme de **z, n, w, k et insert** telle que

   ord(z) ∧ z≠nil ∧ cdr(z)=elim(k, w) ∧ dans(k, w) ∧ ord(w) ⇒ z=elim(n, insert) ?

   profil : elim(n : t, insert : liste(t)) → liste(t)
   préc : dans(n, insert)
   post : elim(n, insert)= **si** n=car(insert) **alors** cdr(insert)
   sinon car(insert) . elim(n, cdr(insert))

$\rightarrow$ (insert=n . z) $\vee$ (insert=car(z).w $\wedge$ car(z)$\neq$n $\wedge$ k=n)

La postcondition est donc :

((insert=n . z) $\vee$ (insert=car(z).w $\wedge$ car(z)$\neq$n $\wedge$ k=n)) $\wedge$ ord(insert)

7. La fonction k doit nécessairement être calculée avant la fonction g_2 puisque cette dernière utilise le résultat k(z, n). Nous appliquons donc la stratégie **Découper et Ordonner** pour les instances { k } et { insert }.

- hypothèses en terme de **z, n, w et k** et en terme de **z, n, w et insert** telles que :

z$\neq$nil $\wedge$ ord(z) $\Rightarrow$ ((insert=n . z) $\vee$ (insert=car(z).w $\wedge$ car(z)$\neq$n $\wedge$ k=n)) $\wedge$ ord(insert) ?

$\rightarrow$ **k=n**

$\rightarrow$ (insert=n . z) $\vee$ (insert=car(z) . w $\wedge$ car(z)$\neq$n)

L'arbre engendré est alors :

Récursion
Décomposition des données
X= { z }
insert(z, n) $\leftarrow$ **si** C_2(z, n) **alors** f_2(z, n) **sinon** g_2(insert(cdr(z), k(z, n)), z, n)

E_0	E_1	E_2	E_3
(énoncé de C_2)	(énoncé de f_2)	(énoncé de k)	(énoncé de g_2)

Avec :

E_2 :

profil	:	k=k(n : entier) $\rightarrow$ entier
préc	:	vrai
post	:	k = n
trace	:	perm(cdr(x), z), n=car(x), x$\neq$nil, ord(z), z$\neq$nil

E_3 :

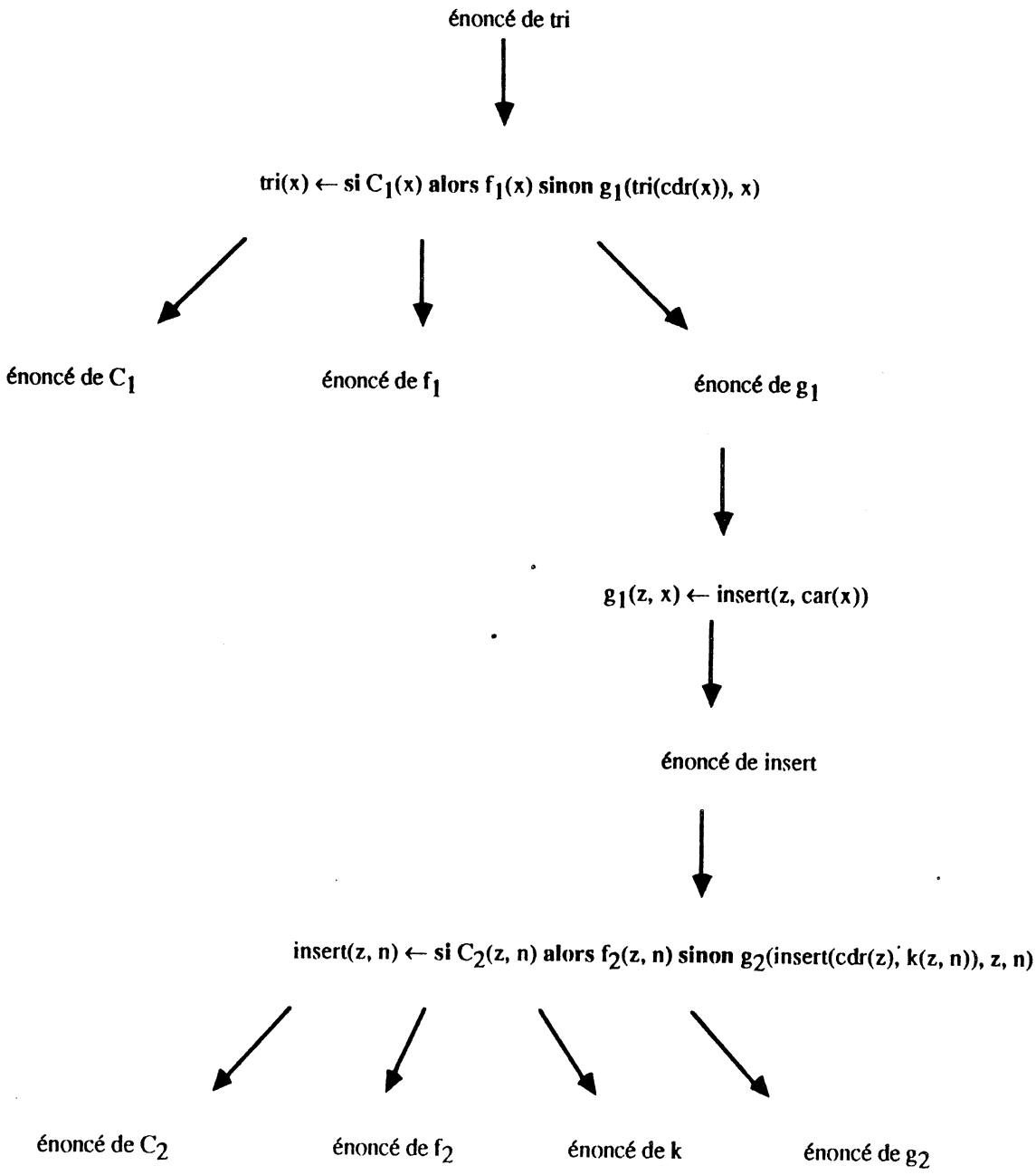
profil	:	insert = g_2(w, z : liste(entier), n : entier) $\rightarrow$ liste(entier)
préc	:	ord(z) $\wedge$ z$\neq$nil $\wedge$ ord(w)
post	:	((insert=n . z) $\vee$ (insert=car(z) . w $\wedge$ car(z)$\neq$n)) $\wedge$ ord(insert)
trace	:	perm(cdr(x), z), n=car(x), x$\neq$nil, cdr(z)=elim(k, w), dans(k, w)

L'argument z n'apparaissant plus dans la postcondition de E_2, il a été supprimé du profil des fonctions et les propriétés relatives à z transférées dans la trace.

Il reste alors à prouver les énoncés E_3 et E_2, puis à déterminer les énoncés E_1 et E_0 et ensuite à élaborer la précondition dérivée finale.

RECAPITULATIF :

L'arbre construit à ce niveau de la preuve est :



Synthèse de la fonction g_2

Enoncé :

profil	:	insert = g_2(w, z : liste(entier), n : entier) → liste(entier)
préc	:	ord(z) ∧ z≠nil ∧ ord(w)
post	:	((insert=n . z) ∨ (insert=car(z) . w ∧ car(z)≠n)) ∧ ord(insert)
trace	:	perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w),dans(k, w), k=n

La postcondition étant sous forme disjonctive nous choisissons d'introduire une conditionnelle à l'aide de la stratégie **Instanciation des postconditions.**

• **Déroulement.**

1. et 2. Par la forme disjonctive de la postcondition nous pouvons choisir de constituer les deux énoncés E_1 et E_2 suivants :

profil	:	insert=h_1(z : liste(entier), n : entier) → liste(entier)
préc	:	ord(z) ∧ z≠nil
post	:	insert=n . z ∧ ord(insert)
trace	:	perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w), dans(k, w), ord(w), k=n

w n'apparait pas dans la postcondition, cette donnée a donc été otée du profil de h_1.

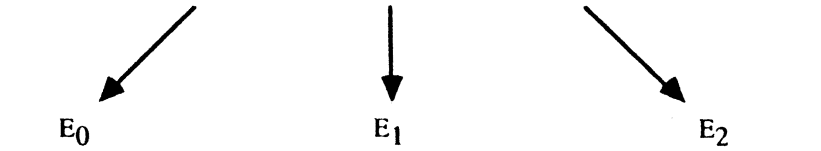
profil	:	insert=h_2(w, z : liste(entier), n : entier) → liste(entier)
préc	:	ord(z) ∧ z≠nil ∧ ord(w)
post	:	insert=car(z) . w ∧ car(z)≠n ∧ ord(insert)
trace	:	perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w),dans(k, w), k=n

3. et 4. Ces deux énoncés ayant été déterminés syntaxiquement à partir de l'énoncé initial il n'est pas nécessaire de vérifier qu'ils permettent la preuve de l'énoncé initial. Ceci est garanti par leur construction.

5. L'arbre produit est :

conditionnelle

g_2(w, z, n) ← **si** C_3(w, z, n) **alors** h_1(w, z, n) **sinon** h_2(w, z, n)



E_0	E_1	E_2
(énoncé de C_3)	(énoncé de h_1)	(énoncé de h_2)

E_1 et E_2 sont les deux énoncés établis aux pas 1 et 2 et E_0 sera déterminé une fois E_1 et E_2 prouvés.

6. Preuve de E_1. La stratégie utilisée est la reconnaissance d'une fonction de base.

1'. Choix de la fonction de base **cons** dont l'énoncé est :

profil : **cons**(y_1 : t , y_2 : liste(t)) → liste(t)
prec : vrai
post : **cons**(y_1, y_2) = y_1 . y_2

Les arguments choisis sont **{n, z}**.

2'. - hypothèse en terme de **z, n et insert** telle que **ord(z) ∧ z≠nil ∧ insert=n . z ⇒ insert=n . z** ?
→ vrai

- hypothèse en terme de **z, n et insert** telle que **ord(z) ∧ z≠nil ∧ insert=n . z ⇒ ord(insert)** ?
→ n ≤ car(z)

La définition engendrée est donc $h_1(z, n) \leftarrow n . z$ et la précondition dérivée est **n ≤ car(z)**.

fin-pas 6.

Il reste alors à prouver l'énoncé E_2, à déterminer la condition de la conditionnelle, à prouver E_0 et à élaborer la précondition dérivée finale. Nous présentons maintenant une preuve de E_2.

Synthèse de la fonction h_2

Enoncé :

profil	:	insert=h_2(w, z : liste(entier), n : entier) → liste(entier)
préc	:	ord(z) ∧ z≠nil ∧ ord(w)
post	:	insert=car(z) . w ∧ car(z)≠n ∧ ord(insert)
trace	:	perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w),dans(k, w), k=n

z n'intervient que en terme de car(z) nous appliquons alors la stratégie introduction d'une composition fonctionnelle par **Instanciation des fonctions intermédiaires.**

• **Déroulement.**

1. Choix de la fonction intermédiaire **car** et de son argument { **z** }.

2. - hypothèse en terme de **w, z et n** telle que **ord(z) ∧ z≠nil ⇒ z≠nil** ? (on peut appliquer la fonction **car** sur l'argument **z**)
→ vrai

3. - hypothèse en terme de **w, m, z, insert et n** telle que
ord(z) ∧ z≠nil ∧ m = car(z) ∧ ord(w) ⇒ insert=car(z) . w ?
→ insert = m . w

- hypothèse en terme de **w, m, z, insert et n** telle que
ord(z) ∧ z≠nil ∧ m = car(z) ∧ ord(w) ⇒ car(z)≠n ?
→ m≠n

L'arbre construit est donc :

composition fonctionnelle
$h_2(w, z, n) \leftarrow h_2'(w, car(z), n)$

E_1

Avec E_1 :

profil	:	insert = h_2'(w : liste(entier), m, n : entier) → liste(entier)
préc	:	ord(w)
post	:	insert=m . w ∧ m≠n ∧ ord(insert)
trace	:	perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w),dans(k, w), k=n, m=car(z), ord(z), z≠nil

4. Synthèse de h_2'.

En raison du lien **insert = m . w** nous appliquons la stratégie **Reconnaissance d'une fonction de base.**

1'. Choix de la fonction **cons** pour les arguments **{m, w}**.

2'. - hypothèse en terme de **m, n, w et insert** telle que **ord(w) ∧ insert = m . w ⇒ insert = m . w** ?
→ vrai

- hypothèse en terme de **m, n, w et insert** telle que **ord(w) ∧ insert = m . w ⇒ ord(insert)** ?
→ (w=nil) ∨ (w≠nil ∧ m ≤ car(w))

3'. La reconnaissance de la fonction **cons** n'aboutit que pour la précondition (w=nil ∨ (w≠nil ∧ m ≤ car(w))) ∧ m≠n. Nous cherchons à simplifier cette précondition à partir des hypothèses disponibles dans la section **trace**

- hypothèse en terme de **m et w** telle que :

cdr(z)=elim(k, w) ∧ dans(k, w) ⇒ (w=nil) ∨ (w≠nil ∧ m ≤ car(w)) ?

→ m ≤ car(w)

En effet de **dans(k, w)** il est possible de déduire **w≠nil** puisque **dans(k, nil)** est toujours faux. Nous ne cherchons pas à simplifier m≠n puidqueces objets n'apparaissant pas dans la trace de l'énoncé. La précondition dérivée est donc **m ≤ car(w) ∧ m≠n.**

fin-pas 4.

5. Recherche de la précondition dérivée pour la synthèse de h_2 définie par $h_2(w, z, n) \leftarrow h_2'(w, car(z), n)$. Il faut ramener la précondition obtenue ci-dessus en terme des données de h_2 c'est à dire en terme des variables w, z et n, ceci en utilisant le lien m=car(z).

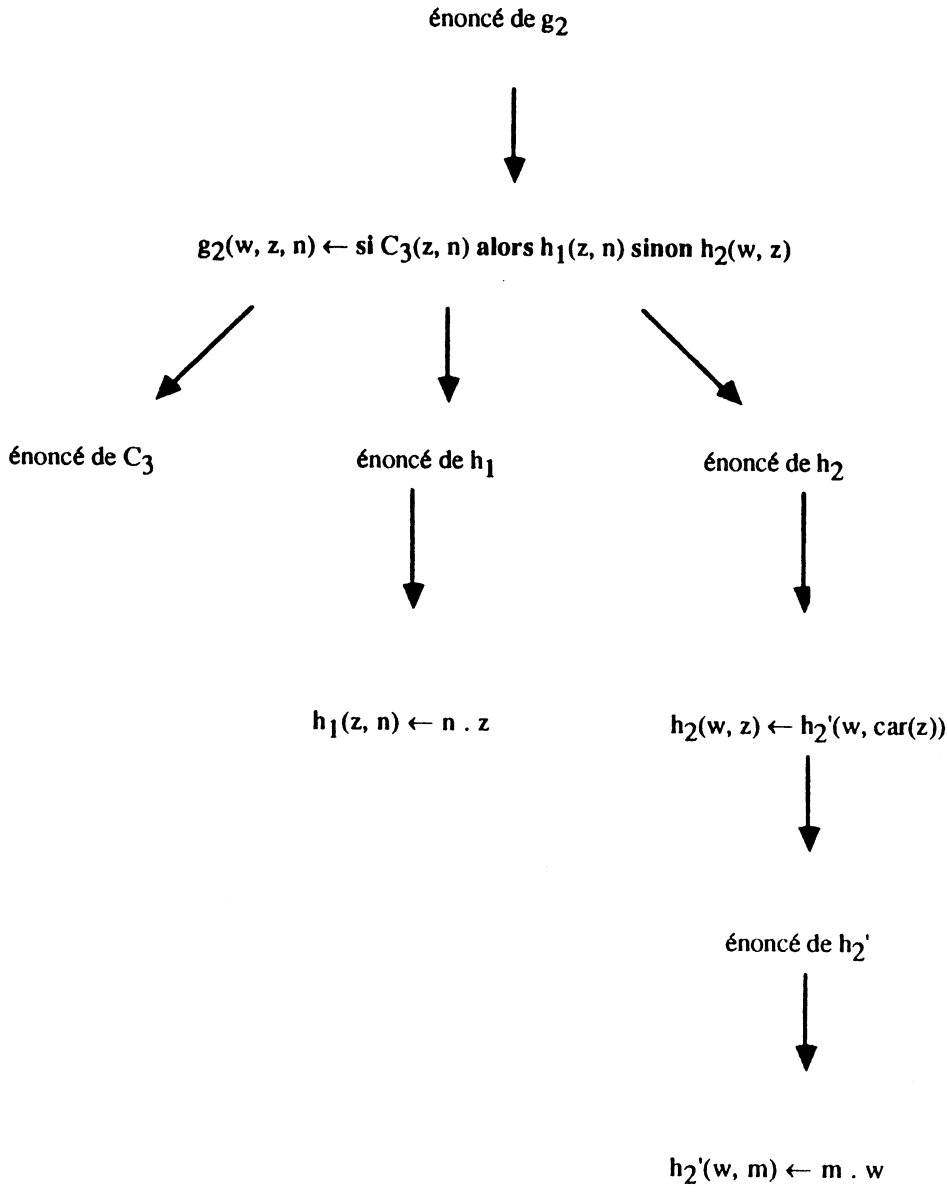
- hypothèse en terme de **m et w** telle que **m=car(z) ⇒ m ≤ car(w) ∧ m≠n** ?
→ car(z) ≤ car(w) ∧ car(z)≠n

La précondition dérivée pour la synthèse de la fonction h_2 est donc **car(z) ≤ car(w) ∧ car(z)≠n.**

fin-synthèse de h_2.

RECAPITULATIF :

L'arbre de preuve issu de l'énoncé relatif à la fonction g_2 est :



Pour terminer la synthèse de la fonction g_2 il ne reste plus qu'à terminer l'application de la stratégie introduction d'une conditionnelle par **Instanciation des postconditions** en choisissant la condition C_3 de la conditionnelle introduite.

Fin de la synthèse de la fonction g_2.

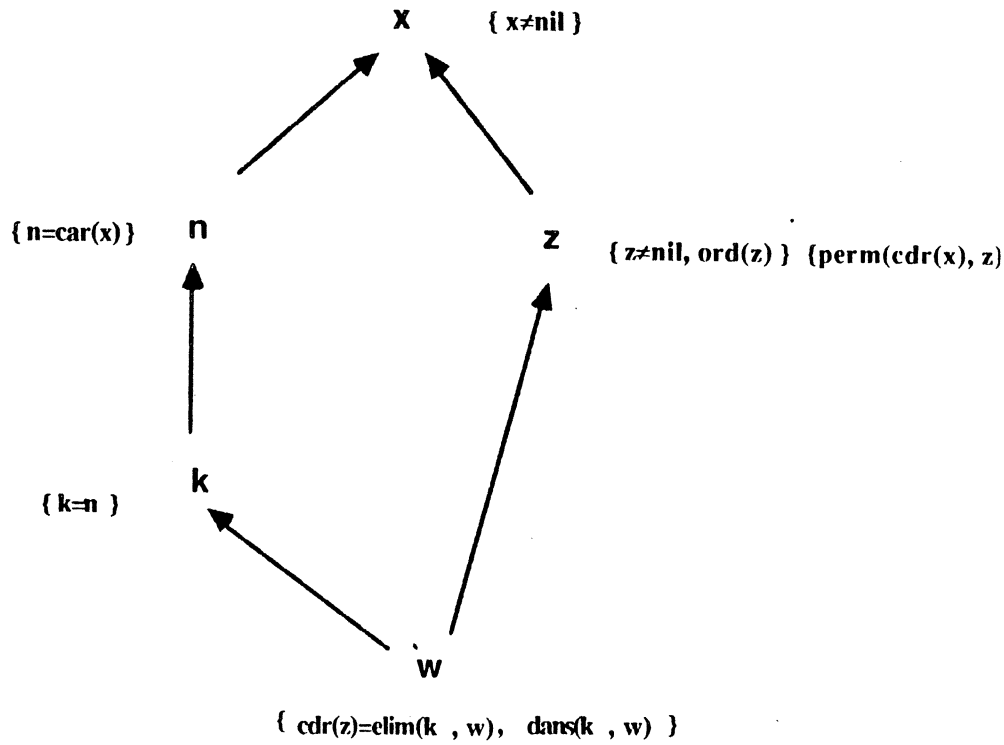
7. Les deux préconditions dérivées obtenues sont respectivement $n \leq car(z)$ et $car(z) \leq car(w) \wedge car(z) \neq n$. Nous choisissons de retenir la première pour la condition C_3.

8.1. Nous cherchons à simplifier la précondition dérivée $car(z) \leq car(w) \wedge car(z) \neq n$ en utilisant l'information $\neg(n \leq car(z))$ et les hypothèses de la trace.

- Demande d'une hypothèse en terme de n, z et w telle que $\neg(n \leq car(z)) \Rightarrow car(z) \neq n$?
→ vrai

- Demande d'une hypothèse en terme de n, z et w telle que $\neg(n \leq car(z)) \wedge trace \Rightarrow car(z) \leq car(w)$?

La trace disponible est : { perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w),dans(k, w), k=n, ord(z), z≠nil }. Le graphe de dépendance des variables est :



Pour utiliser l'hypothèse $n > car(z)$ il suffit de procéder de la manière suivante :

1 - ramener le but à prouver en terme de z et k à partir du lien existant entre w, z et k ;

2 - simplifier la formule obtenue en 1 à partir des propriétés de k et z ;

3 - ramener la formule obtenue en 2 en terme n et z en utilisant le lien entre k et n ;

4 - simplifier la formule obtenue en 3 en utilisant l'hypothèse $n > car(z)$.

Ceci donne lieu aux demandes suivantes :

- hypothèse en terme de **k et z** telle que **dans(k, w) ∧ cdr(z) = elim(k, w) ⇒ car(z) ≤ car(w)** ?
→ inf(car(z), cdr(z)) ∧ car(z) ≤ k

- hypothèse en terme de **k et z** telle que **z≠nil ∧ ord(z) ⇒ inf(car(z), cdr(z)) ∧ car(z) ≤ k** ?
→ car(z) ≤ k

- hypothèse en terme de **n et z** telle que **n = k ⇒ car(z) ≤ k** ?
→ car(z) ≤ n

- hypothèse en terme de **n et z** telle que **n > car(z) ⇒ car(z) ≤ n** ?
→ vrai

La fonction h_2 permet donc de calculer le résultat pour toutes les données telles que la condition retenue est fausse.

9. Synthèse de C_3 à partir de l'énoncé :

profil	:	cond = **C_3**(z : liste(entier), n: entier) → booléen
préc	:	ord(z) ∧ z≠nil
post	:	(cond=vrai) ≡ n ≤ car(z)
trace	:	perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w), k=n, dans(k, w), ord(w)

En raison du lien n ≤ car(z) nous utilisons la stratégie introduction d'une composition fonctionnelle par **Instanciation de la fonction de recomposition**.

1'. Choix de la fonction de recomposition ≤.

2'. Demande d'une hypothèse en terme de **y_1, y_2, z et n** telle que :

(cond=vrai) ≡ $y_1 \leq y_2$ ⇒ (cond=vrai) ≡ n ≤ car(z) ?

→ y_1=n ∧ y_2=car(z)

3'. Nous obtenons la définition $C_3(z, n) \leftarrow h_3(z, n) \leq h_4(z, n)$ et l'énoncé caractérisant les fonctions h_3 et h_4 est :

profils	:	y_1 = **h_3**(z : liste(entier), n: entier) → entier ;
		y_2 = **h_4**(z : liste(entier), n: entier) → entier ;
préc	:	ord(z) ∧ z≠nil
post	:	y_1 = n ∧ y_2 = car(z)
trace	:	perm(cdr(x), z), n=car(x), x≠nil, cdr(z)=elim(k, w), k=n, dans(k, w), ord(w)

4'. Synthèse de ces deux fonctions. La reconnaissance de la fonction identité pour l'argument n et de la fonction **car** pour l'argument x permet d'engendrer les définitions $h_3(z, n) \leftarrow n$ et $h_4(z, n) \leftarrow car(z)$ et les préconditions dérivées se réduisent toutes deux à **vrai**.

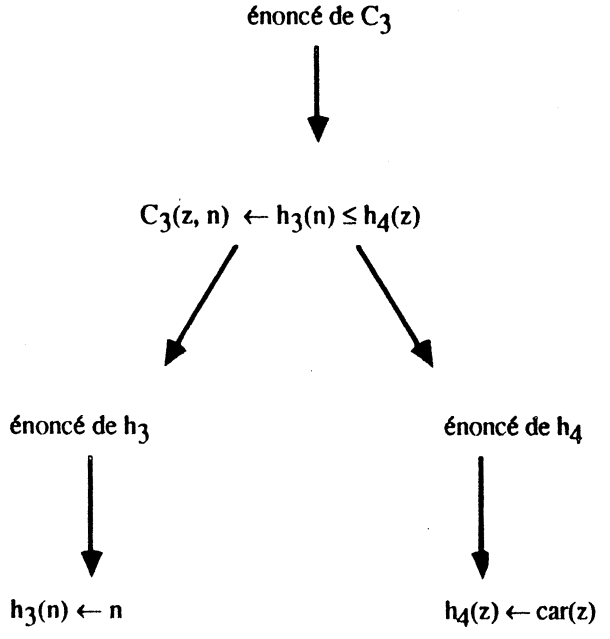
5'. La précondition dérivée lors de la synthèse de C_3 est donc **vrai**.

fin-pas 9.

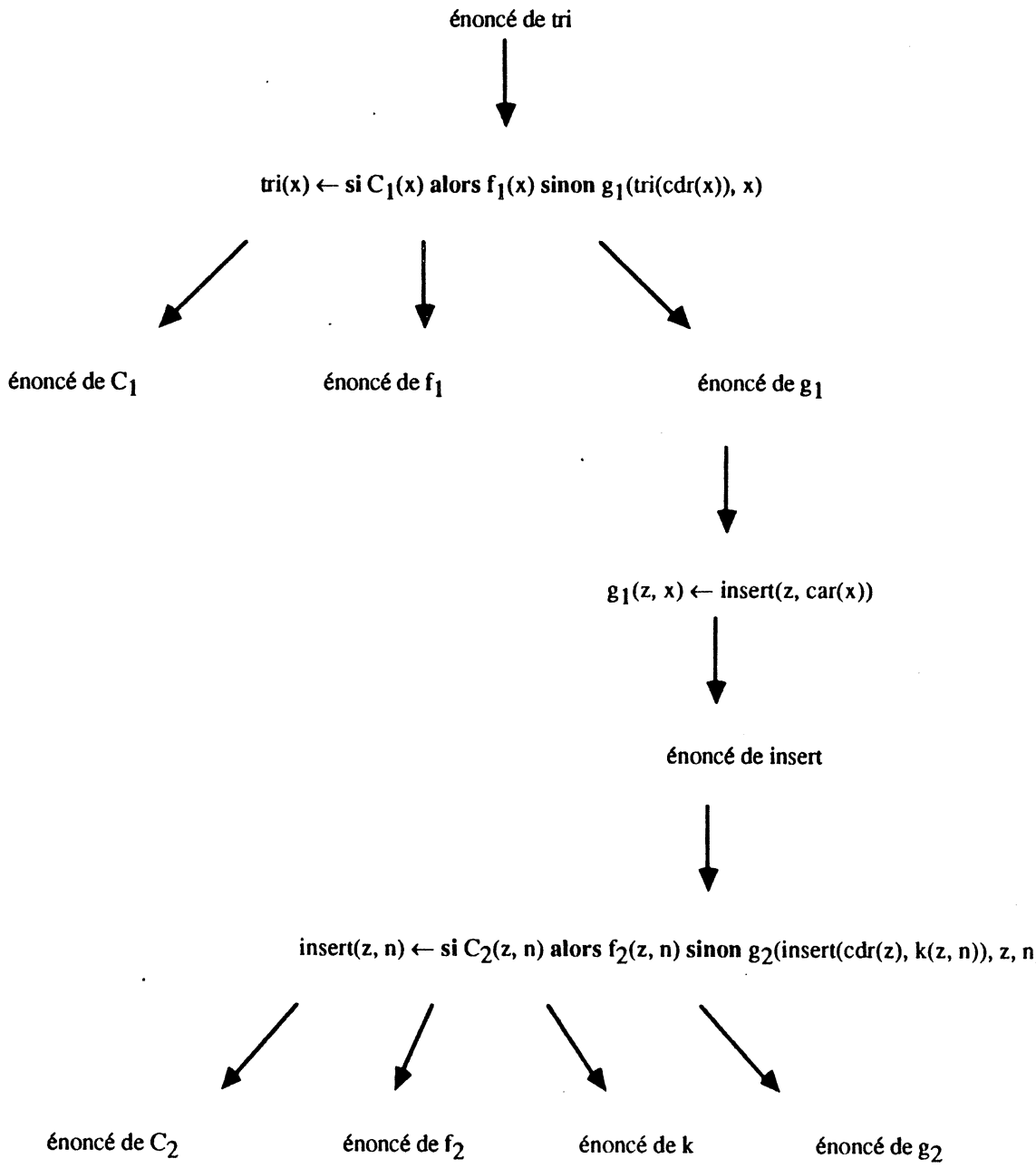
10 . Elaboration de la précondition dérivée pour la synthèse de la fonction g_2. Elle est égale à $n \leq car(z) \vee$ **vrai**, ce qui se simplifie en **vrai**.

fin-synthèse de g_2.

L'arbre issu de g_2, donné page 11, est donc complété par :



L'arbre de preuve qu'il reste à compléter, donné page 6, est :



L'arbre issu de l'énoncé de g_2 est complet. Il reste donc à synthétiser dans l'ordre les fonctions k, f_2, C_2, f_1 et C_1.

• **Synthèse de k.**

La construction de g_2 a aboutie pour la précondition **vrai**. L'énoncé de la fonction k n'est donc pas modifié, c'est, rappelons le :

profil	:	k = k(n : entier) → entier
préc	:	vrai
post	:	k = n
trace	:	perm(cdr(x), z), n=car(x), x≠nil, ord(z), z≠nil

Ici la reconnaissance de la fonction de base identité produit la définition k(n) ← n et la précondition dérivée est la constante **vrai**.

• **Synthèse de f_2.**

La précondition de la fonction f_2 est **ord(z) ∧ z=nil**. Ceci découle des choix faits lors de l'application de la stratégie **Décomposition des données** pour la fonction insert. Puisque la précondition dérivée lors de la synthèse de la fonction k est réduite à **vrai**, cette précondition n'est pas modifiée. Il ne reste plus qu'à établir la postcondition caractérisant la fonction f_2.

- Demande d'une hypothèse en terme de **n, z et insert** telle que **ord(z) ∧ z=nil ⇒ z=elim(n, insert)** ?
→ **insert = n . nil**

L'énoncé constitué est :

profil	:	insert = f_2(n : entier) →liste(entier)
préc	:	vrai
post	:	insert = n . nil ∧ dans(n, insert) ∧ ord(insert)
trace	:	perm(cdr(x), z), n=car(x), x≠nil, ord(z), z=nil

La variable z qui est initialement une donnée de la fonction f_2 n'apparait pas dans la postcondition, les hypothèses sur z sont donc conservées dans la trace. Pour prouver cet énoncé nous utilisons la stratégie **Reconnaissance d'une fonction de base** en raison du lien insert = n . nil.

Déroulement :

1. Choix de la fonction **cons** et des arguments **{n, nil}**.

2. - hypothèse en terme de **n** telle que **insert = n . nil ⇒ insert = n . nil** ?
→ vrai

- hypothèse en terme de **n** telle que **insert = n . nil ⇒ dans(n, insert)** ?

→ vrai

- hypothèse en terme de **n** telle que **insert = n . nil** ⇒ **ord(insert)** ?
→ vrai

La définition obtenue est donc $f_2(n) \leftarrow$ n . nil et la précondition dérivée se réduit à **vrai**.

fin-synthèse de f_2.

• **Synthèse de C_2.**

La précondition de f_2 étant réduite à **vrai**, l'énoncé de C_2 est :

profil	:	cond = C_2(z: liste(entier)) → booléen
préc	:	ord(z)
post	:	(cond=vrai) ≡ z=nil
trace	:	perm(cdr(x), z), n=car(x), x≠nil

Stratégie **Reconnaissance de fonction de base**. La fonction reconnue est la fonction **nul**.

1. Choix de la fonction **nul** pour l'argument **z** à partir de l'énoncé :

profil	:	r = **nul**(z : liste(t)) → booléen
préc	:	vrai
post	:	(r=vrai) ≡ (z=nil)

2. - hypothèse en terme de **x** telle que (cond=vrai) ≡ z=nil ⇒ (cond=vrai) ≡ z=nil ?
→ vrai

On obtient $C_2(z) \leftarrow$ nul(z) et la précondition dérivée est **vrai**.

fin-synthèse de C_2.

Ceci termine la synthèse de la fonction **insert** avec la précondition **vrai**. La précondition établie initialement pour la fonction f_1, qui est x=nil, n'est donc pas modifiée. Il en découle que la précondition dérivée pour la synthèse de g_1 (voir arbre précédent) est aussi réduite à **vrai**.

• **Synthèse de f_1.**

Il ne reste qu'à déterminer la postcondition de l'énoncé caractérisant f_1.

- Demande d'une hypothèse en terme de **x et tri** telle que **x=nil ⇒ perm(x, tri)** ?
→ **tri=nil**

L'énoncé de f_1 est donc :

profil	:	tri=f_1 → liste(entier)
préc	:	vrai
post	:	tri = nil ∧ ord(tri)
trace	:	x=nil

La donnée initiale x n'apparait pas dans la postcondition, elle est donc otée du profil et f_1 devient une fonction constante.

Stratégie reconnaissance de la fonction primitive nil.

- **hypothèse en terme de tri telle que tri=nil ⇒ ∧ tri=nil ?**
→ **vrai**

- **hypothèse en terme de tri telle que tri=nil ⇒ ord(tri) ?**
→ vrai

On obtient f_1 ← nil pour la précondition dérivée **vrai.**

fin-synthèse de f_1.

• **Synthèse de C_1.**

L'énoncé est :

profil :	C=C_1(x: liste(entier)) → booléen
préc :	vrai
post :	(C=vrai) ≡ (x=nil)

On applique la stratégie reconnaissance de la fonction de base **nul** pour l'argument **x**. On obtient $C_1(x)$ ← nul(x) et la précondition dérivée est **vrai.**

fin-synthèse de C_1.

Tous les énoncés ont maintenant été prouvés, la synthèse de la fonction **tri** est terminée. Le programme en résultant est :

$tri(x) \leftarrow$ **si** $C_1(x)$ **alors** f_1 **sinon** $g_1(tri(cdr(x)), x)$

$C_1(x) \leftarrow nul(x)$

$f_1 \leftarrow nil$

$g_1(z, x) \leftarrow insert(z, car(x))$

$insert(z, n) \leftarrow$ **si** $C_2(z)$ **alors** $f_2(n)$ **sinon** $g_2(insert(cdr(z), k(n)), z, n)$

$C_2(z) \leftarrow nul(z)$

$f_2(n) \leftarrow n \,.\, nil$

$k(n) \leftarrow n$

$g_2(w, z, n) \leftarrow$ **si** $C_3(z, n)$ **alors** $h_1(z, n)$ **sinon** $h_2(w, z)$

$C_3(z, n) \leftarrow h_3(n) \leq h_4(z)$

$h_3(n) \leftarrow n$

$h_4(z) \leftarrow car(z)$

$h_1(z, n) \leftarrow n \,.\, z$

$h_2(w, z) \leftarrow h_2'(w, car(z))$

$h_2'(w, m) \leftarrow m \,.\, w$

En remplaçant les appels de fonctions non récursives par leur définition le programme se réécrit en :

$tri(x) \leftarrow$ **si** $nul(x)$ **alors** nil **sinon** $insert(tri(cdr(x)), car(x))$

$insert(z, n) \leftarrow$ **si** $nul(z)$ **alors** $n \,.\, nil$ **sinon si** $n \leq car(z)$ **alors** $n \,.\, z$ **sinon** $car(z) \,.\, insert(cdr(z), n)$

SYNTHESE D'UN ALGORITHME DE TRI PAR SELECTION

Enoncé:

profil : tri = **tri** (x : liste(entier)) → liste(entier)
préc : vrai
post : perm(x, tri) ∧ ord(tri)

Nous choisissons d'utiliser la stratégie introduction de récursion par **Construction du résultat .**

•Synthèse de la fonction tri.

1. Choix:

- construction de base :

profil : **nil** → liste(t)
préc : vrai
post : **nil** =nil

- construction récursive :

profil : cons = **cons**(y_1 : t, y_2: liste(t)) → liste(t)
préc : vrai
post : cons = y_1 . y_2

2. Sélection des appels récursifs : { y_2 }.

- L'ensemble R est réduit à { y_2 }. Il y a introduction d'une fonction permettant de calculer l'argument de l'appel récursif introduit. Nous nommons **reste** cette fonction dont le résultat est y_2.

L'ensemble A est { y_1 }. Il y a introduction d'une fonction permettant de calculer cet argument à partir de la donnée. Soit **min** cette fonction.

La définition de tri engendrée est donc : tri(x) ← **si** C_0(x) **alors** f_0(x) **sinon** min(x) . tri(reste(x)).

3. Synthèse du cas de base. On applique la stratégie introduction d'une composition fonctionnelle par **Instanciation de la fonction de recomposition** à l'aide de la fonction constante **nil**.

1'. - hypothèse en terme de x telle que **tri=nil ⇒ ord(tri)** ?
→ vrai

- hypothèse en terme de x telle que **tri=nil ⇒ perm(x, tri)** ?
→ x=nil

La définition engendrée est $f_0(x) \leftarrow$nil et cette définition n'est correcte que pour la précondition dérivée **x=nil**.

4. Recherche de l'énoncé caractérisant **min** et **reste**.

- hypothèse en terme de **min** et w telle que **ord(w) ∧ tri= min . w ⇒ ord(tri)** ?

→ w=nil ∨ (w≠nil ∧ min ≤ car(w))

- hypothèse en terme de **x, min** et **reste** telle que :

perm(reste, w) ∧ ord(w) ⇒ w=nil ∨ (w≠nil ∧ min ≤ car(w)) ?

→ inf(min, reste)

- hypothèse en terme de **x, min et reste** telle que :

perm(reste, w) ∧ ord(w) ∧ tri= min . w ⇒ perm(x, tri) ?

profil : r = perm(x, tri : liste(t)) → booléen
préc : vrai
post : r = **si** tri=nil **alors** x=nil
sinon si dans(car(tri), x) **alors** perm(elim(car(tri), x), cdr(tri))
sinon faux

→ reste=elim(min, x) ∧ dans(min, x)

5 et 6. Ces deux étapes ne sont pas nécessaires puisque la précondition de la fonction cons est réduite à vrai.

7. L'énoncé caractérisant les fonctions **min** et **reste** est alors :

profils : min=**min**(x : liste(entier)) → entier ;
reste=**reste**(x : liste(entier)) → liste(entier)
préc : x≠nil
post : reste=elim(min, x) ∧ dans(min, x) ∧ inf(min, reste)

Nous présentons maintenant une synthèse des fonctions **min** et **reste**.

Synthèse des fonctions min et reste

Nous choisissons de synthétiser les fonctions **min** et **reste** par une même stratégie. La structure de leur définition sera donc commune. La stratégie choisie est l'introduction de récursion par **Décomposition des données.**

• Déroulement.

1. Choix des variables à décomposer : { **x** }

   Choix de la décomposition :

   - cas récursion: { **car(x), cdr(x)** }.

   - Enoncés :

   profil : **car**(x : liste(t)) → t ;
   préc : x≠nil
   post : **car**(x)=car(x)

   profil : **cdr**(x : liste(t)) → liste(t) ;
   préc : x≠nil
   post : **cdr**(x)=car(x)

2. - hypothèse en terme de **x** telle que **x≠nil** ⇒ **x≠nil** ? (on peut appliquer les sélecteurs **car** et **cdr**)
   → vrai

3. Sélection des appels récursifs possibles : **{cdr(x) }**. Les deux appels récursifs engendrés sont donc min(cdr(x)) et reste(cdr(x)).

4. - hypothèse en terme de **x** telle que **x≠nil** ∧ **w=cdr(x)** ⇒ **w≠nil** ? (cdr(x) vérifie bien la précondition initiale de min et reste)
   → cdr(x)≠nil

5. Demande d'un ordre bien fondé **inf** tel que **x≠nil** ∧ **w=cdr(x)** ⇒ **inf(w, x)**. L'ordre classique sur les listes convient.

6. Recherche de la postcondition des deux fonctions de recomposition g_1 et g_2.

   - hypothèse en terme de **x, $\mathbf{min_1}$, $\mathbf{reste_1}$, min** et **reste** telle que

   x≠nil ∧ **cdr(x)≠nil** ∧ **$\mathbf{reste_1}$=elim($\mathbf{min_1}$, cdr(x))** ∧ **dans($\mathbf{min_1}$, cdr(x))** ∧ **inf($\mathbf{min_1}$, $\mathbf{reste_1}$)**
   ⇒ **dans(min, x)** ?

   → min=car(x) ∨ (min≠car(x) ∧ min=$\mathbf{min_1}$)

$reste_1$ et min_1 désignent respectivement les résultats min(cdr(x)) et reste(cdr(x)).

- hypothèse en terme de **x, min_1, $reste_1$, min** et **reste** telle que :

$$x \neq nil \wedge cdr(x) \neq nil \wedge reste_1 = elim(min_1, cdr(x)) \wedge dans(min_1, cdr(x)) \wedge inf(min_1, reste_1) \Rightarrow reste = elim(min, x) \ ?$$

$\rightarrow (min = car(x) \wedge reste = cdr(x)) \vee (min \neq car(x) \wedge reste = car(x) . reste_1)$

La postcondition élaborée est alors la conjonction de ces deux réponses.

L'arbre construit est :

Récursion
Décomposition des données
X = {x}

(min(x), reste(x)) ← **si** $C_1(x)$ **alors** ($min_0(x)$, $reste_0(x)$)
sinon (g_1(min(cdr(x)), reste(cdr(x)), x), g_2(min(cdr(x)), reste(cdr(x)), x))

E_0	E_1	E_2
(énoncé de C_1)	(énoncé de min_0 et $reste_0$)	(énoncé de g_1 et g_2)

Avec E_2 :

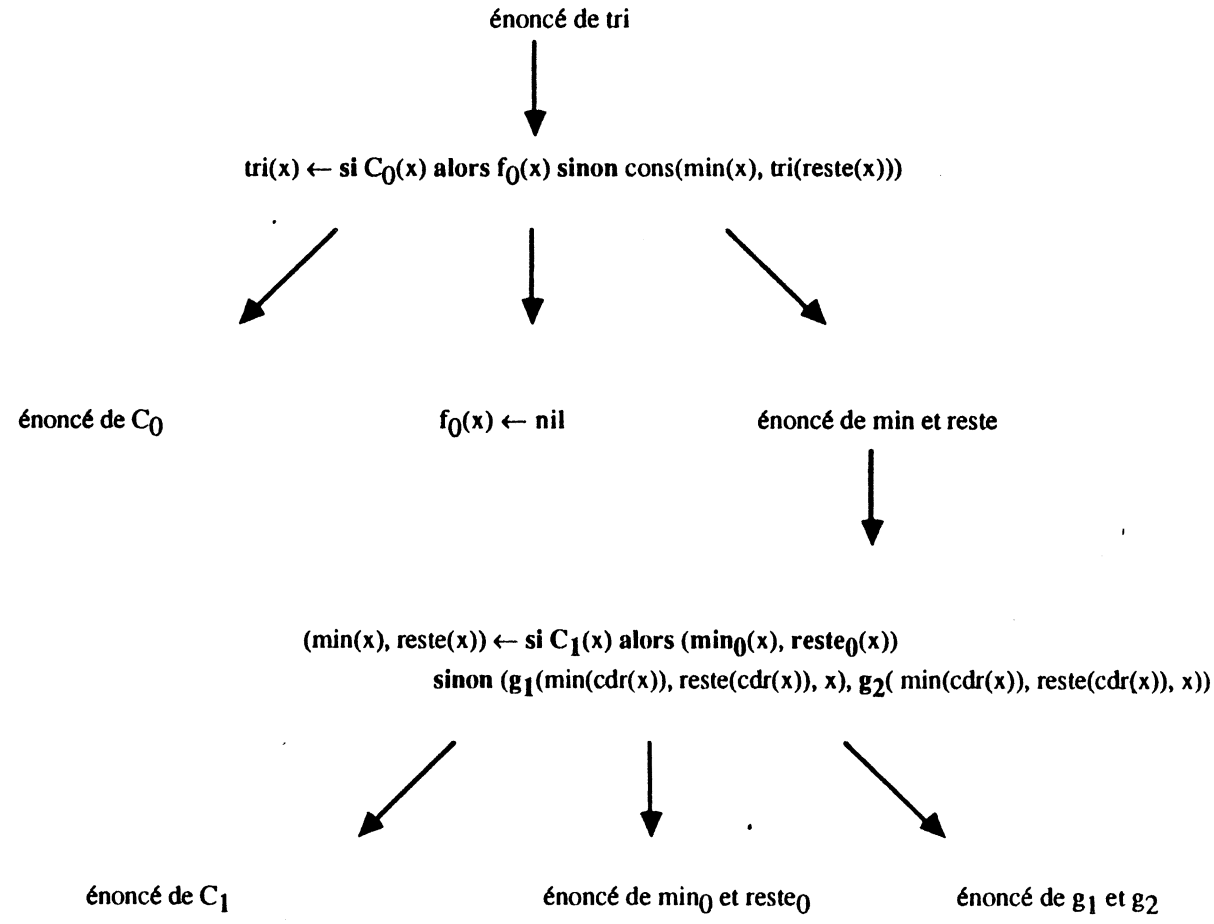
profils : min = g_1(min_1 : entier; $reste_1$, x : liste(entier)) → entier ;
reste = g_2(min_1 : entier , $reste_1$, x : liste(entier)) → liste(entier)

préc : $x \neq nil \wedge cdr(x) \neq nil \wedge inf(min_1, reste_1)$

post : $((min = car(x) \wedge reste = cdr(x)) \vee (min \neq car(x) \wedge min = min_1 \wedge reste = car(x) . reste_1)) \wedge inf(min, reste)$

trace : $reste_1 = elim(min_1, cdr(x))$, $dans(min_1, cdr(x))$

L'arbre élaboré à ce niveau de la preuve est :



La précondition dérivée lors de la synthèse des fonctions g_1 et g_2, si elle n'est pas réduite à vrai, pourra modifier les énoncés des fonctions min_0, $reste_0$ et C_1 qui devront être synthétisées à leur tour. Nous présentons maintenant une synthèse des fonctions g_1 et g_2.

Synthèse de g_1 et g_2

Enoncé :

profils : $min = g_1(min_1 : entier ; reste_1, x : liste(entier)) \rightarrow entier$;
$reste = g_2(min_1 : entier ; reste_1, x : liste(entier)) \rightarrow liste(entier)$

préc : $x \neq nil \wedge cdr(x) \neq nil \wedge inf(min_1, reste_1)$

post : $((min=car(x) \wedge reste=cdr(x)) \vee (min \neq car(x) \wedge min=min_1 \wedge reste=car(x) . reste_1)) \wedge inf(min, reste)$

trace : $reste_1=elim(min_1, cdr(x))$, $dans(min_1, cdr(x))$

Nous utilisons la stratégie introduction d'une conditionnelle par **Instanciation des postconditions**. Ce choix peut être justifié par la forme disjonctive de la postcondition.

• **Déroulement.**

1 et 2. Par des critères syntaxiques nous constituons les énoncés relatifs aux deux branches de la conditionnelle, les postconditions étant respectivement la première et la seconde partie de la disjonction. Soient E_1 et E_2 les 2 énoncés suivants :

E_1 :

profils : $min = g_{11}(x : liste(entier)) \rightarrow entier$;
$reste = g_{12}(x : liste(entier)) \rightarrow liste(entier)$

préc : $x \neq nil \wedge cdr(x) \neq nil$

post : $min=car(x) \wedge reste=cdr(x) \wedge inf(min, reste)$

trace : $reste_1=elim(min_1, cdr(x))$, $dans(min_1, cdr(x))$, $inf(min_1, reste_1)$

Les données $reste_1$ et min_1 n'apparaissent plus dans la postcondition de E_1, elles ont donc été enlevées du profil de g_{11} et g_{21}.

E_2 :

profils : $min = g_{21}(min_1 : entier , reste_1, x : liste(entier)) \rightarrow entier$;
$reste = g_{22}(min_1 : entier , reste_1, x : liste(entier)) \rightarrow liste(entier)$

préc : $x \neq nil \wedge cdr(x) \neq nil \wedge inf(min_1, reste_1)$

post : $min \neq car(x) \wedge min=min_1 \wedge reste=car(x) . reste_1 \wedge inf(min, reste)$

trace : $reste_1=elim(min_1, cdr(x))$, $dans(min_1, cdr(x))$

3 et 4. Ces deux énoncés permettent bien la preuve de l'énoncé initial, ceci étant vrai par construction.

5. L'arbre construit est :

Conditionnelle
instanciation des deux cas

$(g_1(min_1, reste_1, x), g_2(min_1, reste_1, x)) \leftarrow$
si $C_2(min_1, reste_1, x)$ **alors** $(g_{11}(min_1, reste_1, x), g_{12}(min_1, reste_1, x))$
sinon $(g_{21}(min_1, reste_1, x), g_{22}(min_1, reste_1, x))$

E_0 E_1 E_2

E_1 et E_2 sont les énoncés établis aux pas 1 et 2. L'énoncé E_0 sera constitué une fois E_1 et E_2 prouvés. Nous présentons maintenant une preuve de E_1. Cette preuve est faite en deux étapes : la première est relative à la construction de g_{11} et la seconde est relative à la construction de g_{12}.

• **Synthèse de la fonction g_{11}.**

En raison du lien min=car(x) nous choisissons la stratégie **Reconnaissance de fonction de base** que nous appliquons à la fonction g_{11}.

• **Déroulement.**

1. Choix de la fonction **car** et de son argument **{ x }**.

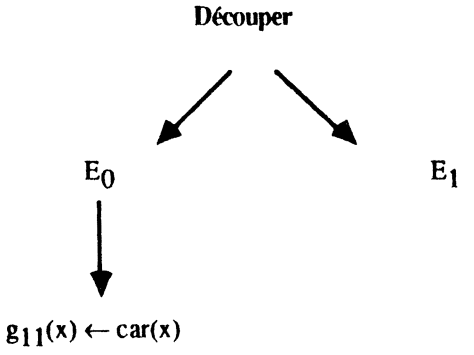
profil : car(x : liste(t)) → t
préc : x≠nil
post : car(x)=car(x)

2. - hypothèse en terme de **x** telle que **x≠nil ∧ cdr(x)≠nil ⇒ x≠nil** ? (on peut appliquer la fonction **car** à l'argument x)
→ vrai

3. - hypothèse en terme de **x, reste et min** telle que **x≠nil ∧ cdr(x)≠nil ∧ min=car(x) ⇒min=car(x)** ?
→ vrai

- hypothèse en terme de **x, min et reste** telle que **x≠nil ∧ cdr(x)≠nil ∧ min=car(x) ⇒inf(min, reste)** ?
→ inf(car(x), reste)

La reconnaissance de la fonction car pour g_{11} n'aboutit que pour la précondition dérivée **reste=cdr(x) ∧ inf(car(x), reste)**. L'arbre construit est donc :



E_0 est l'énoncé de la fonction car dans lequel car est renommé en g_{11} et l'énoncé E_1 est :

profil	:	reste = g_{12}(x : liste(entier)) → liste(entier)
préc	:	x≠nil ∧ cdr(x)≠nil
post	:	reste=cdr(x) ∧ inf(car(x), reste)
trace	:	$reste_1$=elim(min_1, cdr(x)), dans(min_1, cdr(x)), inf(min_1, $reste_1$)

4. Preuve de E_1. En raison de la relation reste=cdr(x) nous utilisons la stratégie **Reconnaissance d'une fonction de base.**

1'. Choix de la fonction de base **cdr** et de son argument {x}.

2'. - hypothèse en terme de x telle que **x≠nil ∧ cdr(x)≠nil ⇒ x≠nil** ?
→ vrai

3'. - hypothèse en terme de x et reste telle que **x≠nil ∧ cdr(x)≠nil ∧ reste=cdr(x) ⇒ reste=cdr(x)** ?
→ vrai

- hypothèse en terme de x et reste telle que **x≠nil ∧ cdr(x)≠nil ∧ reste=cdr(x) ⇒ inf(car(x), reste)** ?
→ inf(car(x), cdr(x))

La précondition dérivée est donc **inf(car(x), cdr(x))**. Elle ne peut pas être simplifiée à l'aide de la trace puisqu'elle porte sur les données initiales du problème. La définition engendrée est donc g_{12}(x) ← cdr(x).

5. La précondition dérivée lors de la synthèse des fonctions g_{11} et g_{12} est donc **inf(car(x), cdr(x))**.

Synthèse de g_{21} et g_{22}

Enoncé :

profils	:	min = g_{21}(min$_1$: entier , reste$_1$, x : liste(entier)) → entier ;
		reste = g_{22}(min$_1$: entier , reste$_1$, x : liste(entier)) → liste(entier)
préc	:	x≠nil ∧ cdr(x)≠nil ∧ inf(min$_1$, reste$_1$)
post	:	min≠car(x) ∧ min=min$_1$ ∧ reste=car(x) . reste$_1$ ∧ inf(min, reste)
trace	:	reste$_1$=elim(min$_1$, cdr(x)), dans(min$_1$, cdr(x))

Nous appliquons la stratégie **Reconnaissance d'une fonction de base** pour g_{21} en raison du lien min = min$_1$.

• **Synthèse de g_{21}.**

1. Choix de la fonction de base **id** et de son argument { **min$_1$** }.

profil : r = id(n : t) → t
préc : vrai
post: r = n

2. - hypothèse en terme de **x, reste$_1$, min$_1$, min** et **reste** telle que

x≠nil ∧ cdr(x)≠nil ∧ min=min$_1$ ∧ inf(min$_1$, reste$_1$) ⇒min=min$_1$?

→ vrai

- hypothèse en terme de **x, reste$_1$, min$_1$, min** et **reste** telle que

x≠nil ∧ cdr(x)≠nil ∧ min=min$_1$ ∧ inf(min$_1$, reste$_1$) ⇒ min≠car(x) ?

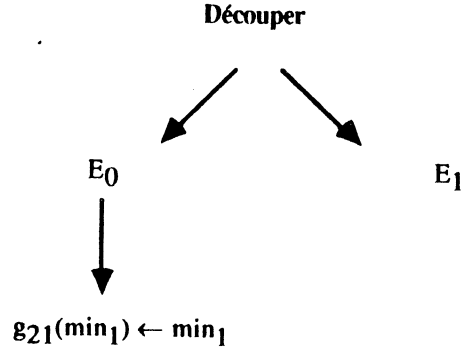
→ min$_1$≠car(x)

- hypothèse en terme de **x, reste$_1$, min$_1$, min** et **reste** telle que

x≠nil ∧ cdr(x)≠nil ∧ min=min$_1$ ∧ inf(min$_1$, reste$_1$) ⇒ inf(min, reste) ?

→ inf(min$_1$, reste)

3. La définition engendrée pour g_{21} est g_{21}(min$_1$) ← min$_1$ et la précondition dérivée est **min$_1$≠car(x) ∧ inf(min$_1$, reste) ∧ reste=car(x) . reste$_1$**. L'arbre de preuve est donc :



avec E_1 :

profil	:	reste = $g_{22}(min_1$: entier , $reste_1$, x : liste(entier)) $\rightarrow$ liste(entier)
préc	:	$x \neq nil \wedge cdr(x) \neq nil \wedge inf(min_1, reste_1)$
post	:	$min_1 \neq car(x) \wedge reste = car(x) \,.\, reste_1 \wedge inf(min_1, reste)$
trace	:	$reste_1 = elim(min_1, cdr(x))$, $dans(min_1, cdr(x))$

4. Preuve de E_1.

x n'intervient que sous la forme car(x). Nous choisissons donc d'introduire une composition fonctionnelle par instanciation des fonctions intermédiaires.

1'. Choix de la fonction intermédiaire **car** et de son argument { **x** }.

profil :	car(x : liste(t)) $\rightarrow$ liste(t)
préc :	$x \neq nil$
post:	car(x) = car(x)

2'. - hypothèse en terme de **x, $reste_1$ et min_1** telle que :

$$x \neq nil \wedge cdr(x) \neq nil \wedge inf(min_1, reste_1) \Rightarrow x \neq nil \ ?$$

$\rightarrow$ vrai

- hypothèse en terme de **x, $reste_1$, m, min_1 et reste** telle que

$$x \neq nil \wedge cdr(x) \neq nil \wedge m = car(x) \wedge inf(min_1, reste_1) \Rightarrow reste = car(x) \,.\, reste_1 \ ?$$

$\rightarrow$ reste = m . $reste_1$

- hypothèse en terme de **x, $reste_1$, m, min_1 et reste** telle que

$$x \neq nil \wedge cdr(x) \neq nil \wedge m = car(x) \wedge inf(min_1, reste_1) \Rightarrow min_1 \neq car(x) \ ?$$

$\rightarrow$ $min_1 \neq m$

3'. L'arbre issu de E_1 est alors :

Composition fonctionnelle

$g_{22}(min_1, reste_1, x) \leftarrow g_{22}'(min_1, reste_1, car(x))$

E_1'

Avec E_1' :

profil	:	reste = g_{22}' (min_1 : entier , $reste_1$: liste(entier), m : entier) → liste(entier)
prec	:	inf(min_1, $reste_1$)
post	:	reste = m . $reste_1$ ∧ $min_1 \neq m$ ∧ inf(min_1, reste)
trace	:	$reste_1$=elim(min_1, cdr(x)), dans(min_1, cdr(x)), inf(min_1, $reste_1$), m=car(x), x≠nil, cdr(x)≠nil

4'. Synthèse de la fonction g_{22}'. Nous choisissons d'appliquer la stratégie **Reconnaissance d'une fonction de base** en raison du lien reste = m . $reste_1$.

1". Choix de la fonction de base **cons** et de ses arguments { **m, $reste_1$** }.

profil	:	cons(y_1 : t, y_2 : liste(t)) → liste(t)
prec	:	vrai
post	:	cons(y_1, y_2) = y_1 . y_2

2". - hypothèse en terme de **m, min_1, $reste_1$** et **reste** telle que:

reste = m . $reste_1$ ∧ inf(min_1, $reste_1$) ⇒ reste = m . $reste_1$?

→ vrai

- hypothèse en terme de **m, min_1, $reste_1$** et **reste** telle que :

reste = m . $reste_1$ ∧ inf(min_1, $reste_1$) ⇒ $min_1 \neq m$?

→ $min_1 \neq m$

- hypothèse en terme de **m, min_1, $reste_1$** et **reste** telle que :

reste = min . $reste_1$ ∧ inf(min_1, $reste_1$) ⇒inf(min_1, reste) ?

→ $min_1 \leq m$

La synthèse de la fonction g_{22}' est terminée et la reconnaissance de la fonction **cons** n'aboutit donc que pour la précondition dérivée $\mathbf{min_1 < m}$.

4'. Recherche de la précondition dérivée pour la synthèse de g_{22}. Rappelons que la définition élaborée est $g_{22}(reste_1, x, min_1) \leftarrow g_{22}'(reste_1, car(x), min_1)$. Il faut donc ramener la précondition dérivée $\mathbf{min_1 < m}$ en terme des données $reste_1$, x et min_1.

- hypothèse en terme de **x**, $\mathbf{min_1}$ et $\mathbf{reste_1}$ telle que $\mathbf{m=car(x) \Rightarrow min_1 < m}$?

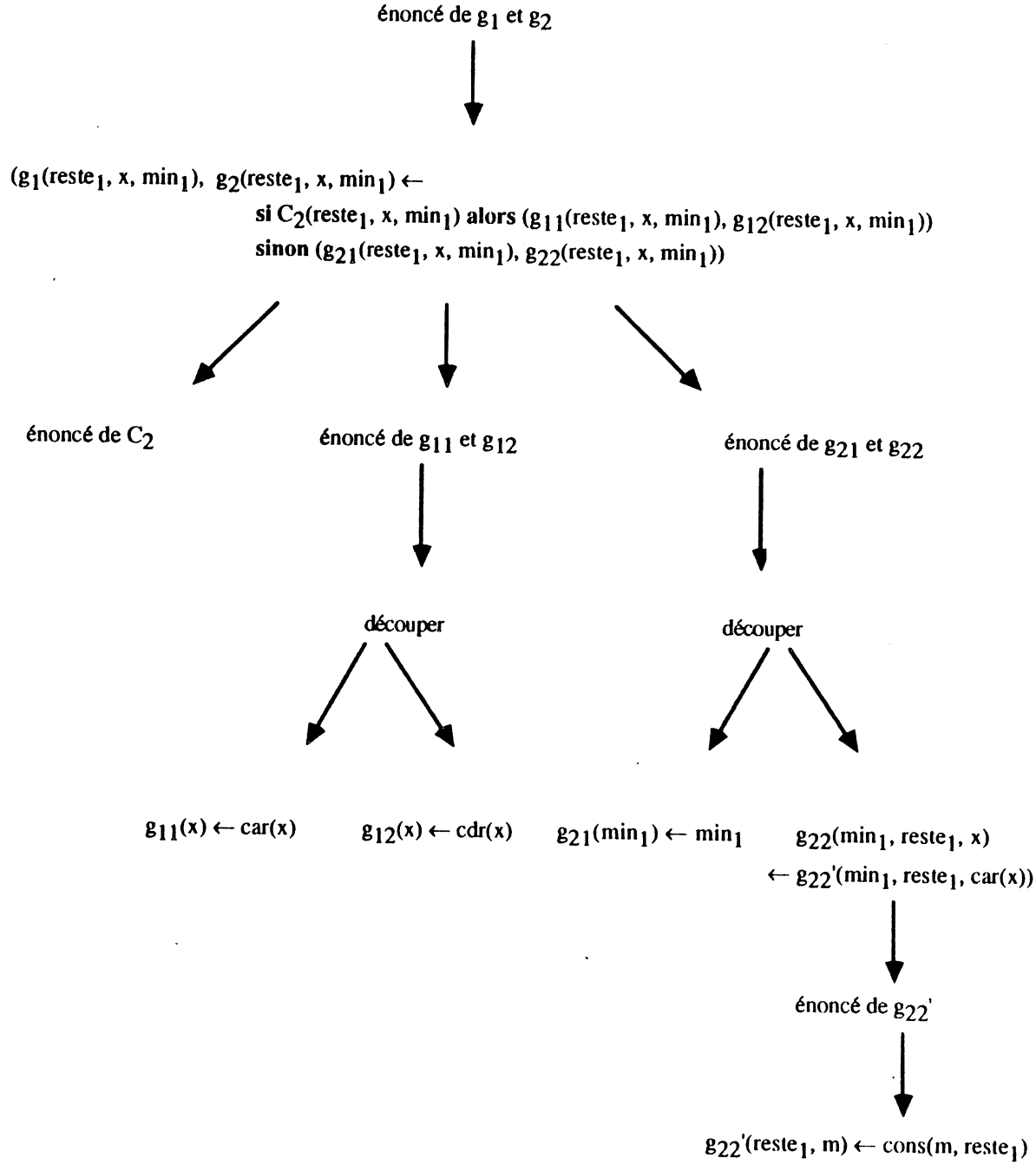
$\rightarrow min_1 < car(x)$

La précondition dérivée est donc $\mathbf{min_1 < car(x)}$.

Ceci termine la synthèse des fonctions g_{21} et g_{22} pour la précondition dérivée $\mathbf{min_1 < car(x)}$.

Synthèse du tri par sélection

L'arbre issu de g_1 et g_2 est :



Nous avons donc synthétisé les fonctions g_{11}, g_{21}, g_{12} et g_{22}, il reste à établir la fonction C_2 à partir des deux préconditions dérivées **inf(car(x), cdr(x))** et $\mathbf{min_1 < car(x)}$.

Fin de la synthèse de g_1 et de g_2

7. Nous retenons la condition $\mathbf{min_1 < car(x)}$.

8.2. Simplification de la formule **inf(car(x), cdr(x))** en sachant $\neg(min_1 < car(x))$.

- hypothèse en terme de $\mathbf{min_1}$ et x telle que $\neg(\mathbf{min_1 < car(x)}) \wedge \mathbf{trace} \Rightarrow \mathbf{inf(car(x), cdr(x))}$?

La trace est { $reste_1$=elim(min_1, cdr(x)), dans(min_1, cdr(x)), inf(min_1, $reste_1$) }. Les seules variables liées par le graphe de dépendance sont les deux ensembles { x } et { min_1, $reste_1$ }. La démarche adoptée est alors la suivante :

1 - faire apparaître min_1 en utilisant les liens $\mathbf{reste_1=elim(min_1, cdr(x))}$ et $\mathbf{dans(min_1, cdr(x))}$.

2 - utiliser l'hypothèse $min_1 \geq car(x)$ et les hypothèses sur min_1 pour simplifier la formule obtenue en 1.

Ceci donne lieu aux demandes suivantes:

- hypothèse en terme de $\mathbf{min_1}$ et x telle que :

$$\mathbf{reste_1=elim(min_1, cdr(x))} \wedge \mathbf{dans(min_1, cdr(x))} \Rightarrow \mathbf{inf(car(x), cdr(x))} \text{ ?}$$

$\rightarrow$ $inf(car(x), reste_1) \wedge car(x) \leq min_1$

- hypothèse en terme de $\mathbf{min_1}$ et x telle que

$$\mathbf{inf(min_1, reste_1)} \wedge \mathbf{min_1 \geq car(x)} \Rightarrow \mathbf{inf(car(x), reste_1)} \text{ ?}$$

$\rightarrow$ vrai

- hypothèse en terme de $\mathbf{min_1}$ et x telle que :

$$\mathbf{inf(min_1, reste_1)} \wedge \mathbf{min_1 \geq car(x)} \Rightarrow \mathbf{car(x) \leq min_1} \text{ ?}$$

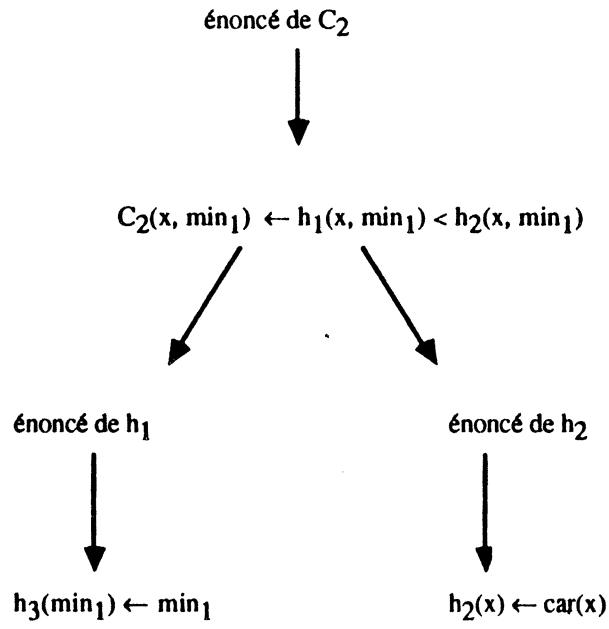
$\rightarrow$ vrai

La condition inf(car(x), cdr(x)) est donc la négation de $\neg(min_1 < car(x))$ dans le contexte de la preuve effectuée. Les fonctions g_{11} et g_{12} permettent donc de calculer le résultat pour toutes les données vérifiant la précondition initiale et telles que $min_1 < car(x)$ s'évalue à **faux**.

9. Synthèse de C_2 à partir de l'énoncé :

profil	:	cond = C_2(x : liste(entier), min_1 : entier) → booléen
prec	:	x≠nil ∧ cdr(x)≠nil
post	:	(cond=vrai) ≡ (min_1 < car(x))

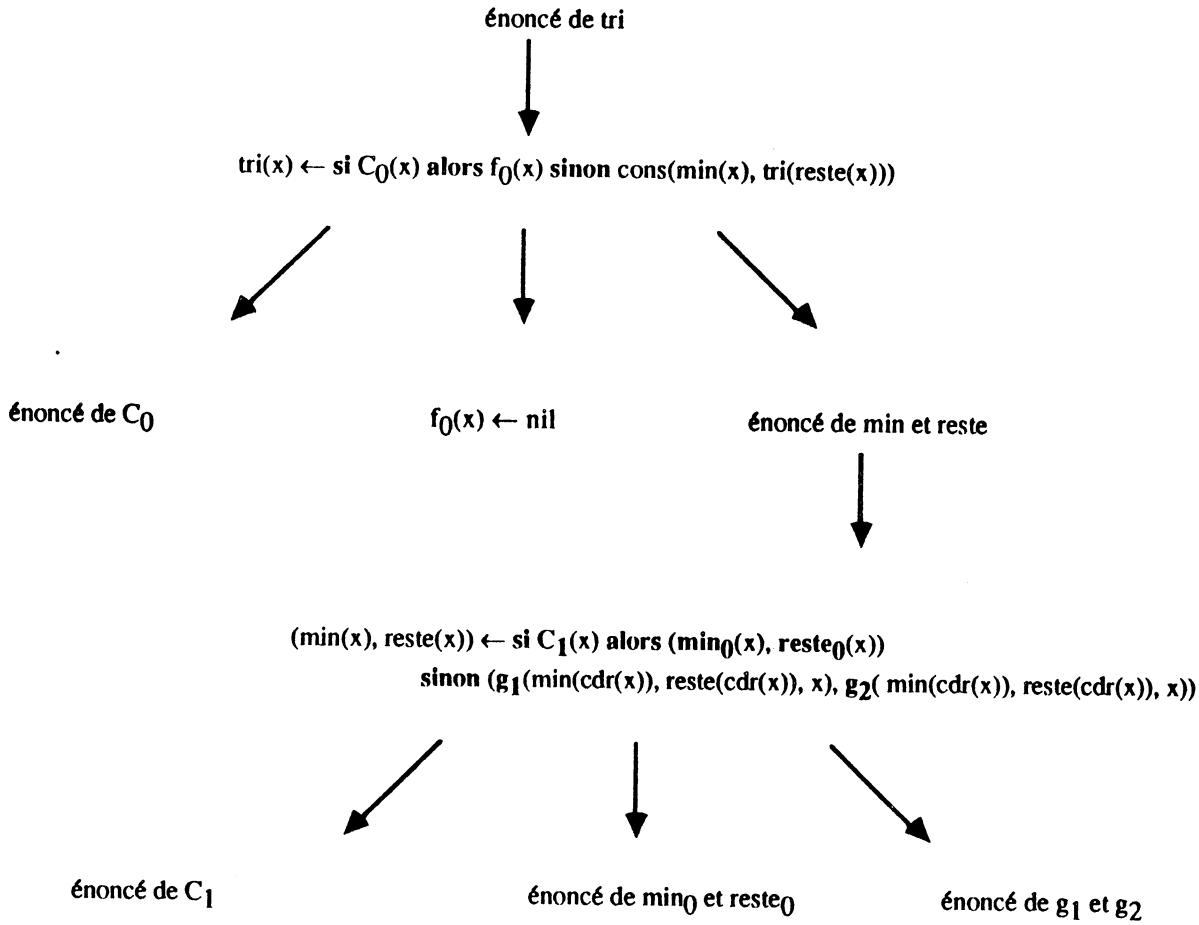
Nous ne détaillons pas la synthèse de cette fonction il suffit d'appliquer par exemple la stratégie d'introduction d'une composition fonctionnelle en instanciant la fonction de recomposition par <. La précondition dérivée se réduit alors à **vrai**. L'arbre construit, issu de l'énoncé de C_2, est :



10. La synthèse des fonctions g_1 et g_2 est donc terminée. La précondition dérivée est **min_1<car(x)** ∨ **vrai** c'est à dire **vrai**.

RECAPITULATIF :

L'arbre de preuve qu'il reste à compléter, déjà donné page 5, est :



Il reste dans un premier temps à synthètiser les fonctions min_0, $reste_0$ et C_1. La précondition dérivée lors de la synthèse de g_1 et g_2 étant réduite à **vrai** les préconditions associées aux fonctions min_0 et $reste_0$ ne sont pas modifiées. Une fois ces deux fonctions synthétisées, il ne restera plus qu'à construire la fonction C_0 à partir de la précondition dérivée obtenue pour la synthèse complète des fonctions min et reste.

• **Synthèse de min_0 et $reste_0$.**

La précondition de l'énoncé associé à ces fonctions est x≠nil ∧ cdr(x)=nil. x≠nil est la précondition initiale des fonctions **min** et **reste** et cdr(x)=nil a été établie lors de la synthèse des fonctions g_1 et g_2. La postcondition caractérisant les fonctions $\mathbf{min_0}$ et $\mathbf{reste_0}$ est obtenue en simplifiant la postcondition initiale dans(min, x) ∧ reste=elim(min, x) ∧inf(min, reste) à l'aide de la précondition x≠nil ∧ cdr(x)=nil.

- hypothèse en terme de **min, reste et x** telle que **x≠nil ∧ cdr(x)=nil ⇒ dans(min, x)** ?
→ min=car(x)

- hypothèse en terme de **min, reste et x** telle que **x≠nil ∧ cdr(x)=nil ⇒ reste=elim(min, x)** ?
→ min=car(x) ∧ reste=nil

L'énoncé des fonctions min_0 et $reste_0$ est donc :

profils	:	min = $\mathbf{min_0}$(x : liste(entier)) → entier ;
		reste = $\mathbf{reste_0}$(x : liste(entier)) → liste(entier)
préc	:	x≠nil ∧ cdr(x)=nil
post	:	min=car(x) ∧ reste=nil ∧ inf(min, reste)

La synthèse de ces 2 fonctions ne pose pas de problème. Les définitions obtenues sont $min_0(x) \leftarrow car(x)$ et $reste_0(x) \leftarrow$ nil et la précondition dérivée se réduit à **vrai** puisque inf(car(x), nil) est toujours vrai.

• **Synthèse de C_1.**

Son énoncé est :

profil	:	cond = $\mathbf{C_1}$(x : liste(entier)) → booléen
préc	:	x≠nil
post	:	(cond=vrai) ≡ (cdr(x)=nil)

C_1 peut être synthétisée par la définition $C_1(x) \leftarrow$ nul(cdr(x)). La précondition dérivée se réduit à **vrai**. La précondition dérivée pour la synthèse des fonction min et reste est donc aussi réduite à **vrai**. Il n'est pas nécessaire d'introduire une conditionnelle supplémentaire dans la définition de la fonction tri puisque tous les cas ont été traités. Il ne reste plus qu'à construire la fonction C_0 à partir de l'énoncé :

profil	:	cond = $\mathbf{C_0}$(x : liste(entier)) → booléen
préc	:	vrai
post	:	(cond=vrai) ≡ (x=nil)

La synthèse de C_0 ne pose pas de problème et aboutit pour la précondition **vrai**. Tous les sous-problèmes ont été prouvés. L'énoncé initial est donc vrai et le programme résultant de la preuve effectuée est :

tri(x) ← **si** C_0(x) **alors** f_0(x) **sinon** cons(min(x), tri(reste(x)))

C_0(x) ← nul(x)

f_0(x) ← nil

(min(x), reste(x)) ← **si** C_1(x) **alors** (min_0(x), $reste_0$(x))
sinon (g_1(min(cdr(x)), reste(cdr(x)), x), g_2(min(cdr(x)), reste(cdr(x)), x))

C_1(x) ← nul(cdr(x))

(min_0(x), $reste_0$(x)) ← (car(x), nil)

(g_1(min_1, $reste_1$, x), g_2(min_1, $reste_1$, x)) ←
si C_2(min_1, x) **alors** (g_{11}(x), g_{12}(x))
sinon (g_{21}(min_1), g_{22}($reste_1$, x))

C_2(min_1, x) ← min_1 < car(x)

g_{11}(x) ← car(x)

g_{12}(x) ← cdr(x)

g_{21}(min_1) ← min_1

g_{22}($reste_1$, x) ← g_{22}'($reste_1$, car(x))

g_{22}'($reste_1$, m) ← cons(m, $reste_1$)

Ce programme peut se réécrire en :

tri(x) ← **si** nul(x) **alors** nil **sinon** cons(min(x), tri(reste(x)))

(min(x), reste(x)) ← **si** nul(cdr(x)) **alors** (car(x), nil)
sinon
si min(cdr(x)) < car(x) **alors** (car(x), cdr(x))
sinon (min(cdr(x)), cons(car(x), reste(cdr(x))))

département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs
. M. SINTZOFF,
. J.L REMY

Mademoiselle POTET Marie-Laure

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 2 juin 1988

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble

Auteur : Marie-Laure Potet

Etablissement : Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle

Titre : Preuves et stratégies pour la synthèse déductive de programmes.

Résumé : Pour maitriser la combinatoire liée à la synthèse déductive de programmes nous avons choisi de décomposer le processus nécessaire à l'élaboration d'un programme en deux étapes principales : la première est relative au choix de la preuve à mettre en place et à son déroulement, la seconde est relative à la recherche des propriétés qui doivent exister sur le domaine du problème, afin de valider les choix faits. Cette décomposition nous a permis de développer des stratégies interactives, dans lesquelles l'utilisateur est responsable du choix de la forme du programme attendu. Le rôle des stratégies est alors d'exécuter la preuve sous-jacente à ces choix et de caractériser, le mieux possible, les propriétés nécessaires sur le domaine du problème. Cette caractérisation permet d'envisager la déduction de ces propriétés mêmes lorsqu'elles ne sont pas explicitement disponibles et donc de " découvrir " de nouveaux algorithmes. Les stratégies que nous proposons sont basées sur un système formel de preuve que nous avons développé et elles sont attachées à l'élaboration de schémas de programmes très simples (condition, récursion et composition fonctionnelle).

Mots clés : Synthèse de programmes, démonstration automatique