Université Paris vii

École doctorale Informatique, Télécommunications et Electronique
de Paris vii(EDITE)

Comparaison entre la transformation et l'extraction de programmes logiques

Par Anne BOUVEROT

Thèse de doctorat en INFORMATIQUE

Dirigée par Laurent FRIBOURG

Présentée et soutenue publiquement le 8 Mars 1991 à l'Université Paris vii

Devant un jury composé de :

Guy COUSINEAU, Prof	Université de paris vii, France	Président et Rapporteur
Maurizio MARTELLI , Prof	Université de Genova , Italie	Rapporteur
Pierre-Louis CURIEN, DR-CNRS	Université Paris vii, France	Rapporteur
Jean-Pierre FINANCE, Pro	Université de Nancy 1	Rapporteur
Laurent FRIBOURG, CR-CNRS	Université Paris vii	Directeur de thèse

Résumé

Titre : Comparaison entre la transformation et l'extraction de programmes logiques.

Mots clés : Programmation logique ; Programmes primitifs-récursifs ; Développement de programmes ; Preuve de programmes ; Transformation de programmes ; Extraction de programmes ; Pliage/dépliage.

Résumé : L'exécution étendue généralise l'exécution de Prolog à des buts implicatifs. Un but implicatif est une implication entre deux conjonctions d'atomes contenant des variables universelles et existentielles. L'extraction d'un programme à partir d'une preuve par exécution étendue consiste à prouver un but implicatif initial et à extraire de cette preuve un programme logique. Ce programme extrait calcule les variables existentielles en fonction des variables universelles et vérifie des propriétés de correction partielle et de terminaison par rapport au but initial. La transformation de programmes consiste à remplacer un programme par un autre programme équivalent ou spécialisé. La méthode de transformation par pliage/dépliage permet de transformer des programmes récursifs logiques tout en préservant l'équivalence entre les programmes. La méthode d'extraction à partir d'une preuve par exécution étendue et celle de transformation par pliage/dépliage permettent chacune d'obtenir un programme logique, respectivement à partir d'un but implicatif et d'une clause définie. Suivant la forme de la définition initiale du but initial, on distingue les cas de composition par référence à la composition de fonctions et les cas de "tupling". Pour une conjonction de prédicats primitifs-récursifs, quand il y a compatibilité, on observe une propriété d'équivalence entre la transformation et l'extraction : pour des exemples comparables, les programmes obtenus sont identiques. Lorsqu'il n'y a pas de compatibilité, en pliage/dépliage, il est nécessaire de deviner puis d'utiliser un lemme pour forcer le pliage. En exécution étendue, on obtient au cours de la preuve un but intermédiaire et la propriété de présentation par rapport à ce but permet de forcer le pliage dans le processus de transformation.

Abstract

Title : Extended execution and transformation of logic programs.

Keywords : Extended execution ; Logic programming ; Implicative goals ; Program extraction ; Program transformation ; Folding/unfolding.

Abstract : Extended execution is a generalization of the execution of Prolog to implicative goals. An implicative goal is an implication between two conjunctions of atoms that contain universal and existential variables. Extracting a program from an extended execution proof consists in proving an initial implicative goal and in extracting a logic program from that proof. The extracted program calculates the existential variables from the universal variables and verifies some properties of partial correctness and termination with regard to the initial goal. Transforming programs consists in replacing a program by another one which is equivalent to it or specializes it. The transforming method of folding/unfolding allows to transform recursive logic programs and preserves the equivalence between programs. The extraction method based on extended execution proofs and the transformation method of folding/unfolding each give a logic program, respectively from an implicative goal and a defined clause. According to the form of the initial definition or the initial goal, we distinguish between cases of composition, referring to the composition of functions, and cases of tupling. For a conjunction of primitive-recursive predicates, when compatibility holds, there is a property of equivalence between transformation and extraction : the programs obtained from comparable examples are identical. When compatibility doesn't hold, we need folding/unfolding, to guess and use a lemma in order to force a folding. In extended execution, the proof gives an intermediate goal and the property of presentation with regard to this goal allows to force the folding in the transformation process.

En Allemagne, en Espagne, en Etats-Unis

Elle doit être défendue ;

Il faut la soutenir en France ;

En Italie, bien-sûr, il s'agit de la discuter…

…qu'est-ce que c'est ?

Remerciements

Je remercie Guy Cousineau, Professeur d'Informatique à l'Université de Paris 7, d'avoir accepté d'être président du jury et surtout d'avoir bien voulu rapporter cette thèse en un temps très court

Ringrazio tantissimo Maurizio Martelli, Professore di Matematica all'Universita di Genova, 'rapporteur' anche lui, molto vloce e anche molto cortese. Mi fa veramente piacere che egli faccia parte della commissione.

Je suis reconnaissante à Pierre-Louis Curien, Directeur de Recherche au CNRS et à Jean-Pirre Finance, Professeur d'Informatique à l'Université de Nancy 1 ,quiont accepté de faire partie de mon jury.

Je remercie particulièrement Laurent Fribourg Chargé de recherche au CNRS qui a été mon directeur de thèse ; à tous les stades et surtout à la fin de mon travail et a suggéré, critiqué, amélioré, lu etrelu, discuté, précisé, revu, corrigé, conseillé, relu encore, modifié, approfondi, vérifié...

Je remercie Pierre Cregut pour avoir fait du 'type-cheking', protesté rigoureusement contre toute explication peu claire et, même, vérifié la place desvirgules d'après E. et Mme Bled.

Merci aussi à Caroline Lavatelli qui a relu les premiers chapitres.

J'ai travaillé à cette thèse au Laboratoire d'Informatique de l'ENS, plus précisement dans l'équipe Formel, au Pavillon, et j'ai beacoup apprécié la compagnie de ses différents membres.

Je télécommunique ma reconnaissance à tous ceux qui, à France-Télécom, ont fait en sorte que je puisse terminer cette thèse.

Ringrazio anche numerosi Italiani : tuttiquelli che mi hanno accolta al Dipartimento di Informatica di Pisa nella primavera 1990, e Roberto Di Cosmo e Giuseppe Castagna. I confezionatori di tiramisu per il mio 'pot' di tesi.

Je tiens enfin à remercier tous ceux et toutes celles qui ont bien voulu se prêter au test suivant, dit "de. l'introduction" : "Hein qu'elle est compréhensible, la première moitié de la première page ?"- indépendamment de leur réponse , bien-sûr.

Table des matières

Introduction

Présentation générale

Motivation. Lors de l'écriture d'un programme, la partie la plus intéressante et aussi la plus courte est de trouver un algorithme puis de le coder dans le langage de programmation choisi. Il faut ensuite essayer de faire fonctionner le programme et corriger les erreurs que l'on trouve inévitablement, c'est-à-dire vérifier que le programme fait bien point par point ce que l'on voulait qu'il fasse.

L'objectif de la transformation et de l'extraction de programmes est de permettre le développement de programmes dont la correction soit automatiquement assurée. On construit un programme à partir d'une spécification précise et formelle de ce que l'on cherche à calculer. Le processus de transformation ou d'extraction garantit que le programme obtenu calcule exactement ce qu'exprime sa spécification.

Le domaine d'application est ici celui de la programmation logique. La transformation de programmes y a déjà été étudiée, souvent par transposition de méthodes provenant du domaine fonctionnel. Les méthodes d'extraction à partir de preuves y sont en revanche assez nouvelles.

Idée générale. L'exécution d'un programme logique peut être vue comme la preuve grâce à la règle de résolution d'un but défini initial ne contenant que des variables existentielles. On extrait de cette preuve une substitution des variables du but. En exécution étendue un but implicatif initial, contenant des variables existentielles et universelles, est prouvé par des règles généralisant la résolution. De la preuve, on extrait un programme. Certaines clauses de ce programme ne font qu'appliquer une substitution aux variables existentielles : ce sont les clauses associées aux règles qui étendent directement la résolution. Les autres clauses concernent les variables universelles, sont définies de façon plus compliquée et sont associées à une règle d'induction.

La transformation par pliage/dépliage s'applique à des programmes récursifs et permet d'améliorer l'interaction des boucles récursives. En programmation logique, certaines spécifications peuvent être écrites sous forme de programmes peu efficaces ; en augmentant leur efficacité par transformation, on fait de la synthèse de programmes.

Les deux méthodes différentes très nettement. En extraction à partir d'une preuve on impose un schéma récursif pour le programme extrait en commençant la preuve par une induction, alors qu'en transformation on fait de l'évaluation partielle et on cherche à retrouver une structure récursive pour terminer la transformation par un pliage. L'objet de

ce travail est de comparer précisément ces deux manières de procéder.

Présentation détaillée

On se place dans un langage du premier ordre qui permet de définir les programmes logiques, composés de clauses définies et de buts définis. L'exécution des programmes logiques se fait grâce à la résolutions SLD, et la propriété importante associée est que les conséquences logiques d'un programme sont exactement les buts que l'on peut prouver par résolution SLD [**LlO87**].

L'exécution étendue [**KS86**] ne s'applique pas seulement à des buts définis mais à des buts implicatifs. **Un but implicatif** est une implication entre deux conjonctions d'atomes. En tant que formule du premier ordre, il contient des variables universelles et existentielles, mais les variables existentielles ne doivent apparaître que dans sa conclusion. Un but implicatif (G) a la forme suivante :

(G) $\forall$ X $\exists$ Y $[H(X) => C(X, Y)]$

Où l'hypothèse H et la conclusion C de l'implication sont des conjonctions d'atomes.

X représente les variables universelles et Y les variables existentielles.

Les règles d'exécution étendue qui seront utilisées ici sont l'inférence de clause définie et la simplification, qui étendent la résolution SLD au cas des buts implicatifs., et l'induction structurelle. On utilisera aussi une règle de réarrangement pour modifier la quantification de certaines variables ou transférer une partie de l'hypothèse dans la conclusion.

La propriété intéressante de l'exécution par résolution SLD se généralise : les conséquences logiques de la complétion d'un programme sont exactement les buts implicatifs prouvables par exécution étendue. De plus, sous certaines conditions, une preuve par exécution étendue préserve l'équivalence entre les formules [**Fri88**].

L'extraction d'un programme à partir d'une preuve consiste à effectuer une preuve d'une formule de spécification et à extraire de cette preuve un programme. On passe ainsi d'une spécification à un programme exécutable qui calcule ce qui est spécifié.

Lorsque la méthode de preuve choisie est l'exécution étendue, on effectue une preuve d'un but implicatif initial et, lors de l'application de chaque règle, on génère une ou plusieurs clause(s) définie(s) pour garder trace des opérations effectuées [Fri90]. A la fin de la preuve, on a un ensemble de clauses qui constitue le programme extrait.

Plus précisément, si l'on part d'un but implicatif (G) quelconque de variables universelles X et existentielles Y, on lui associe un **prédicat d'entrée-sortie** $es_G(X, Y)$ qui servira à garder une trace de la preuve. On peut extraire d'une preuve par exécution étendue de (G) un programme pour $es_G(X, Y)$ qui calcule les variables existentielles Y en fonction des variables universelles X. Ce programme vérifie des propriétés de correction partielle, de terminaison et de présentation par rapport à sa spécification.

Le prédicat d'entré-sortie contient comme arguments toutes les variables du but initial. Suivant l'utilisation que l'on souhaite faire du programme extrait, il se peut que certains

de ces arguments soient superflus. La troncature du programme extrait est l'élimination syntaxique de ces arguments. Sous des conditions de fonctionnalité, c'est une opération qui préserve la correction partielle et la terminaison du programme extrait par rapport au but implicatif initial.

La transformation de programmes consiste à remplacer un programme par un autre programme qui lui est équivalent ou qui le spécialise, c'est-à-dire qui calcule tout ou partie de ce qui était calculable. On emploie généralement des règles de transformation qui modifient le programme par étapes successives. L'objectif est d'améliorer le programme initial, mais il est rarement formalisé : il faut vérifier au cas par cas que le programme final est plus efficace.

la méthode de transformation par **pliage/dépliage** permet de transformer des programmes récursifs fonctionnels ou logique et consiste en l'application d'une des trois règles de définition, de dépliage ou de pliage. Pour des programmes logiques, elle préserve l'équivalence entre les programmes. Afin de pouvoir au cours d'une transformation utiliser un lemme, par exemple l'associativité d'un prédicat, on définit aussi une règle de **remplacement** du corps d'une clause.

Suivant la forme de la définition initiale d'une transformation par pliage/dépliage , on distingue [**PP88**] les cas de composition, par référence à la composition de fonction et le cas de "tupling" [1].

Lorsque l'on se restreint à des cas particuliers comme l'introduction de listes de différence[ZG88], l'utilisation systématique de l'associativité [**BH87**] ou la recherche de prédicats récursifs-terminaux [**Deb8** ; **Azi87**], on peut rendre le processus de transformation déterministe. On peut inversement chercher à étendre le domaine d'application des transformations et transformer ainsi certaines formules du premier ordre en programmes logiques [**Day87** ; **KH86** ; **ST84** ; **LP88**].

La méthode d'éxtraction à partir d'une preuve par exécution étendre et celle de transformation par pliage/dépliage permettent chacune d'obtenir un programme logique, respectivement à partir d'un but implicatif et d'un ensemble de clauses définies. Elle s'occupent toutes deux de développement de programmes logiques [**Dev90**]. Ces deux méthodes mettent en oeuvre des techniques différentes : preuve par induction et règles d'inférence de clause définie et d simplification pour l'exécution étendue, et pour la transormation évaluation partielle ou dépliage et introduction de récursion ou pliage. Elles poursuivent également des objectifs différents : rendre un programme déjà exécutable plus efficace tout en préservant son sens pour la transformation par pliage/dépliage, et obtenir un programme exécutable qui soit correct par rapport à une spécification simple pour l'extraction. Cependant, lorsque le but initial en exécution étendue et la clause de définition initiale en pliage/dépliage représentent la même conjonction d'atomes, il arrive que les programmes obtenus soient identiques. Cette observation incite à comparer les deux approches de façon précise sur des exemples détaillés.

On se restreint à des prédicats primitifs récursifs en un argument, et on considère des conjonctions de tels prédicats. Dans les cas de composition ou de "tupling" compatible, on a

1. certaines termes anglais comme "tuplin" et "backtracking , pour lesquels il n'y a pas de traduction française évidente ou qui sont effectivement utilisées dans la littérature resteront en anglais

une propriété d'équivalence entre la trnsformation et l'extraction. À partir respectivement d'un but implicatif et d'une clause de définition comparables, une preuve par exécution étendue et un transformation par pliage/dépliage réussissent, et les programmes obtenus sont identiques.

L'utilisation du caractère fonctionnel d'un prédicat en pliage/dépliage correspond en exécution étendue à une règle de simplification qui garde son hypothèse.

Lorsqu'il n'y a pas compatibilité, on a besoin en pliage/dépliage de deviner puis d'utiliser **un lemme** pour réussir à plier : on parle alors de pliage forcé. Dans les exemples correspondants d'exécution étendue on obtient au cours de la preuve un but implicatif qu'il faudra prouver par une nouvelle induction : la propriété de préservation du prédicat d'entré-sortie par rapport à ce but intermédiaire permet de forcer le pliage dans le processus de transformation, sans avoir à deviner aucun lemme.

Lorsque l'on veut obtenir un programme **récursif terminal** en un argument, il faut que la variable correspondante soit universelle dans le but implicatif initial, et il faut pouvoir trnquer le prédicat auxiliaire par rapport à cette variable.

Au cours de l'étude puis de la comparaison de méthodes d'extraction de programmes à partir de preuves et de transformation de programmes, on est naturellement amené à considérer de nombreux exemples. L'automatisation des règles de preuve, d'extraction et de transformation permet de rendre plus aisée la manipulation de tels exemples, et d'assurer l'exactitude des opérations effectuées.

Les exemples présentés dans cette thèse ont tous été effectivement obtenus comme résultat d'une session avec un système interactif de transformation par pliage/dépliage ou d'extarction à partir d'une preuve par d'exécution étendue. Ces deux systèmes sont implantés [2] en Prolog.

2. on trouve suivant les ouvrages un des deux termes "implémenter" et "implanter" . Le premier vient de l'anglais "to implement", et le second signifie [**CRR90**] : introduire et faire se développer quelque chose d'une manière durable dans (un nouveau milieu). Les deux systèmes dont il est question ici est donc été introduits et se sont développés dans ... le langage Prolog.

Plan de la thèse

On suivra le plan suivant, dans lequel les propriétés principales sont indiquées :

Programmes logiques, exécution. Dans la première partie du chapitre 1 on définit le langage du premier ordre utilisé et on rappelle brièvement l'exécution des programmes logiques par résolution SLD. La deuxième partie présente l'exécution étendue. La propriété 1.2.1 (Kanamori) exprime la correction de l'exécution étendue, et la propriété 1.2.2 (Fribourg) donne les conditions sous lesquelles une preuve préserve l'équivalence entre les formules.

Extraction de programmes logiques à partir d'une preuve par exécution étendue. Le chapitre 2 contient la définition des règles d'extraction associées aux règles d'exécution étendue. Les propriétés 2.2.1 (Fribourg) et 2.2.2 précisent sous quelles conditions la correction du programme extrait est garantie. Ce chapitre définit aussi la troncature et la stratégie de propagation aux prédicats auxiliaires. La propriété 2.2.3 (Fribourg) donne les conditions sous lesquelles un programme extrait, après troncature, continuera à être correct par rapport au but initial. Toutes les preuves apparaissent.

Transformation de programmes logiques. La méthode de pliage/dépliage est exposée au début du chapitre 3. Les propriétés 3.1.1 (Tamaki-sato) 3.1.2 (Kanamori-Kawamura) et 3.1.3 (Seki) expriment respectivement la préservation du plus petit modèle de Herbrand,du multi-ensemble de succès et de l'ensemble d'échec fini. Des stratégies, dont la composition et le 'tupling', sont présentées.

La deuxième partie du chapitre consiste en des cas particuliers de transformations déterministes, et la troisième partie présente des systèmes permettant de transformer certaines formules du premier ordre en programmes logiques.

Comparaison entre la transformation et l'extraction. Le chapitre 4 contient une comparaison entre la transformation par pliage/dépliage et l'extraction à partir d'une preuve par exécution étendue.

Les propriétés 4.1.1 et 4.1.2, concernant respectivement les cas de 'tupling', expriment l'équivalence entre les deux méthodes lorsqu'il y a compatibilité. L'exemple de la suite de Fibonacci illustre la correspondance entre l'utilisation du caractère fonctionnel d'un prédicat en pliage/dépliage et la simplification gardant l'hypothèse en exécution étendue.

Les propriétés 4.2.1 et 4.2.2 traitent des cas où en pliage/dépliage on a besoin de deviner et d'utiliser un lemme. La propriété de préservation de prédicat d'entrée-sortie d'un but intermédiaire de la preuve par exécution étendue permet de forcer le dépliage.

Enfin, la propriété 4.2.3 concerne les cas non compatibles où l'on veut obtenir un programme récursif terminal.

Systèmes d'extraction et de transformation. Le système interactif ExExE d'extraction à partir d'une preuve par exécution étendue et celui de transformation par pliage/dépliage sont présentés au chapitre 5. Le code Prolog du système ExExE est présenté en annexe.

Chapitre 1

Programmes logiques, exécution

Après avoir défini le langage du premier ordre dans lequel on se placera, on introduit les programmes logiques, formés de clauses de Horn. L'exécution des programmes logiques est basée sur la résolution SLD, et les conséquences logiques d'un programme sont exactement les buts que l'on peut prouver par résolution SLD. L'exécution étendue généralise la résolution SLD, et s'applique à des buts implicatifs. Les conséquences logiques de la complétion d'un programme sont exactement les buts implicatifs prouvables par exécution étendue.

1.1 Clauses de Horn

1.1.1 Généralités

Langage du premier ordre

On se donne comme vocabulaire un ensemble de variables, que l'on dénotera par X, Y, Z, T ... avec éventuellement des primes et des indices, et un ensemble de symboles de fonction, munis chacun d'une arité. Une constante est une fonction d'arité nulle. On dispose alors d'un ensemble de termes, défini de la façon suivante : toute variable est un terme, et si f est une fonction d'arité n, si t_1...,t_n sont des termes, f(t_1...,t_n)est également un terme. Par la suite t, u, v, w représenteront des termes.

On se donne ensuite un ensemble de symboles de prédicats, munis chacun d'une arité ; on les représentera par une chaine de caractères en minuscules, avec éventuellement des indices : p, q, app, es_F.

On définit la notion d'atome : si p est un symbole de prédicat d'arite n, et si t_1,...,t_n sont des termes, alors p(t_1,...,t_n) est un atome. Par la suite, les atomes seront représentés par A, B, éventuellement indicés.

On considère enfin les quantificateurs $\forall$ et $\exists$, et les connecteurs logiques $\wedge$, $\vee$, $\neg$, $\Rightarrow$, et $\Leftrightarrow$. L'ensemble des **formules du premier ordre** est défini comme suite :

- Tout atome est une formule,
- Si F est une formule, $\neg F$, (F) et $\exists\ X\ F$ aussi,
- Si F et G sont des formules, $F \vee G$ aussi.
- $\forall\ XF$ est une abréviation pour $\neg(\exists\ X \neg F)$,
- $F \Rightarrow G$ est une abréviation pour $\neg F \vee G$.
- $F \wedge G$ est une abréviation pour $\neg(F \Rightarrow \neg G)$.
- $F \leftrightarrow G$ est une abréviation pour $(F \Rightarrow G) \wedge (G \Rightarrow F)$

Les formules sont notées F, G, et les conjonctions d'atomes H, C.

On définit une fonction unaire *vars* qui donne l'ensemble des variables de son argument, que ce soit un terme ou une formule.

On manipulera aussi des ensembles. On note un ensemble en extension en énumérant ses éléments entre accolades $\{a, b, c\}$, et l'ensemble vide est $\emptyset$. On utilise l'union $\cup$ et l'intersection $\cap$ de deux ensembles ; on note l'appartenance d'un élément à un ensemble par $\in$. La suppression d'une partie d'un ensemble par $\setminus$; un ensemble peut être un sous-ensemble $\subset$ ou un sur-ensemble $\supset$ d'un autre.

Substitutions,unification

Une *substitution* θ est un ensemble fini $\{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$, où les X_i sont des variables distinctes et les t_i des termes. Une substitution peut s'appliquer à un terme ou une formule, et sera notée α, β, γ, σ, ou θ. L'application de θ à F consiste à remplacer dans F chaque occurrence d'une variable X_i par t_i, et on la note de façon postfixée $F\theta$.

Si F'= $F\theta$,on dit que θ est un *filtre* de F vers F',et que F' est une *instance* de F.

Un *renommag* est une substitution pour laquelle chaque t_i est une variable différente des X_j,.ces variables étant deux à deux distinctes.

La composition de deux substitutions θ et σ se note par juxtaposition $\theta\sigma$.

On dit que F et G sont des *variantes* l'une de l'autre s'il existe deux substitutions θ et σ telles que $F = G\theta$ et $G = F\sigma$.

Une substitution θ est un *unificateur* de F et G si $F\theta = G\theta$. On a un **unificateur le plus général** ou **'mgu'** ('most general unifier') si pour tout autre unificateur σ de F et G, il existe une substitution γ telle que $\sigma = \theta\gamma$.

1.1.2 Programmes logiques

Clauses de Horn

Une clause de Horn est une clause définie ou un but défini :

- Une ***clause définie*** est une formule de la forme $(C)\ \neg A_0 \vee A_1 \vee \ldots \vee A_n$, avec $n \geq 0$, où les A_i sont des atomes ; on utilise la notation suivante :
  $(C)\ \ A_0$:- $A_1, \ldots, A_n$.
  On appelle A_0 la *tête* de la clause et $A_1, \ldots, A_n$ son *corps*.
- Un *but défini* est une conjonction d'atomes $(G)\ A_1 \wedge \ldots \wedge A_n$, donc une formule *positive* ; on note la *négation* d'un but défini comme une clause sans tête :
  $(\neg G)$:- $A_1, \ldots, A_n$.

Une clause sera notée (C), et un but (G). Un *programme logique* est un ensemble de clauses définies et sera noté P, Q.

La *définition* d'un prédicat p dans un programme logique P est l'ensemble des clauses de P dont la tête a p pour prédicat. Intuitivement, la **complétion** d'un programme P consiste à considérer que la définition de chaque prédicat est de la forme "si et seulement si". On donne ici seulement un exemple (cf. [**Llo87** ; **chapitre 3**] pour une définition plus formelle). Le programme suivant constitue la définition du prédicat app :

- (C1) $app([\,], Y, Y)$.
- (C2) $app([A|X], Y, [A|Z])$:- $app(X, Y, Z))$.

Sa complétion peut s'écrire :

$$\begin{aligned} \text{app}(X, Y, Z) \Longleftrightarrow\ & (X = [\,]\ \wedge\ Z = Y) \\ & \vee\ (\exists\ X_1,\ Z_1\ X = [A|X_1]\ \wedge\ Z = [A|Z_1]\ \wedge\ \text{app}(X_1, Y, Z_1)) \end{aligned}$$

Ainsi, si l'on a app(X,Y,Z), avec $X = [A|X']$), alors forcément $Z = [A|Z']$ pour un certain Z', et app(X',Y,Z'). On dira lorsque l'on fait ce raisonnement que l'on utilise la partie "seulement si" implicite de la clause $(C2)$.

Une clause définie (C) A :- A_1, ..., A_n est dite récursive si l'un au moins des A_i a le même symbole de prédicat que A, et récursive linéaire si un seul A_i vérifie cela, par opposition à la récursion multiple.

Plus petit modèle de Herbrand :

Les programmes logiques sont munis d'une sémantique déclarative, donnée par la théorie des modèles. On rappelle simplement qu'elle associe à un programme logique P son **plus petit modèle de Herbrand** noté M_p, composé des atomes clos qui sont séquences logiques de P. Pour toutes ces notions, voir [**Llo87**].

Propriétés d'un prédicat

On dit qu'un prédicat p défini par un programme P :

- est ***fonctionnel*** par rapport à (un de) ses argument(s) X si $p(X) \wedge p(Y) \Rightarrow X = Y$ est vrai dans M_p,
- ***termine*** par rapport à (un de) ses argument(s) X si $\exists X\ p(X)$ est vrai dans M_p.
- est ***déconnecté*** en X d'une formule F dans laquelle il apparaît s'il termine par rapport à X et si X n'apparaît pas ailleurs dans F.

Un prédicat est utilisé pour calculer à partir de certains de ses arguments, considérés comme des données, le reste de ses arguments, considérés comme des résultats. On appellera *mode* la distinction entre les arguments d'entrée ou données et les arguments de *sortie* ou résultats.

1.1.3 Résolution SLD

Les programmes logiques sont également munis d'une sémantique opérationnelle (procédurale), donnée par la résolution SLD. Elle coïncide avec la sémantique déclarative, c'est-à-dire que l'ensemble de succès d'un programme P, défini par la résolution SLD, est égal au plus petit modèle de Herbrand de P [**Llo7**].

Le principe de résolution de Robinson est le suivant : on ajoute à un programme P la négation d'un but défini $(G) A_1 \wedge \ldots \wedge A_n$ sous la forme $(\neg G)$:- $A_1, ..., A_n$, et on montre que $P \cup \{\neg G\}$ est contradictoire en dérivant la clause vide par résolution. Le but (G) est alors conséquence logique de P. À cause de ce principe, il arrive souvent que l'on confonde un but défini et sa négation, et que l'on appelle but défini une formule négative :- $A_1, \ldots A_n$. L'exécution des programmes logiques se fait par résolution.

Résolution, dérivation, réfutation SLD

La résolution SLD s'applique à un but défini (G) et une clause définie (C) qui n'ont aucune variable en commun :

(G) $A_1 \wedge \ldots \wedge A_m \wedge \ldots \wedge A_n$.

(C) B :- $B_1, \ldots, B_p$.

On sélectionne un atome du but, par exemple A_m, qui soit unifiable avec la tête de (C) par un 'mgu' σ : $A_m\sigma = B\sigma$. La résolution SLD produit un nouveau but défini (G') appelé résolvant :

(G') $(A_1 \wedge \ldots A_{m-1} \wedge B_1 \wedge \ldots \wedge B_p \wedge A_{m+1} \wedge \ldots \wedge \ldots A_n)$ σ

Une dérivation SLD à partir d'un programme logique P et d'un but défini (G), ou encore à partir de $P \cup \{\neg G\}$, est une suite pour $i \geq 0$ de buts définis (G_i), de variantes [1] (C_i) de clauses de P et de substitutions σ_i tels que :

- $G_0 = G$.
- G_i est le résultat d'une résolution SLD à partir de (G_{i-1}) et (C_i) avec le 'mgu' σ_i.

Une *dérivation* SLD peut être finie ou infinie. Si elle est finie, elle peut se terminer par un succès, c'est-à-dire aboutir à la clause vide, ou par un échec. Une ***réfutation SLD*** est une dérivation SLD qui termine avec succès. Si $\sigma_1, \ldots, \sigma_n$ sont les substitutions associées à une réfutation SLD de (G), une ***substitution réponse*** σ est la restriction aux variables de (G) de la composition $\sigma_1 \ldots \sigma_n$.

Un ***arbre de recherche ou arbre SLD*** est une représentation arborescente de l'ensemble des dérivations SLD pour $P \cup \{\neg G\}$. Un nœud est un but défini avec la mention de l'atome sélectionné par la règle de résolution. Il contient trois sortes de branches :

- des branches de succès qui terminent sur la clause vide.
- des branches d'échec qui terminent sur un but défini dont l'atome sélectionné ne peut s'unifier avec aucune des têtes de clause de P.
- des branches infinies.

Caractérisation d'un programme

On peut caractériser l'information qu'un programme contient de plusieurs manières.

L'ensemble de succès d'un programme P est composé des atomes A tels qu'il existe une réfutation SLD pour $P \cup \{\neg A\}$.

Le multi-ensemble de succès (*success multiset*) [**KK88**] d'un programme P est le multi-ensemble de toutes les paires atome-substitution (A, σ) telles qu'il existe une réfutation SLD pour $P \cup \{\neg A\}$ de substitution réponse σ ; il est noté SM_P.

L'ensemble d'échec fini ('finite failure set') [**Llo87** ; **chapitre 3**] d'un programme P, noté F_P, caractérise l'information négative ; c'est l'ensemble des atomes A tels qu'il existe une dérivation SLD pour $P \cup \{\neg A\}$ qui termine sur un échec de façon finie.

Correction de la résolution SLD

La résolution SLD vérifie les propriétés suivantes [**Llo87**] :

Propriété 1.1.1 (Validité de la résolution SLD) Si P est un programme logique et $(G)A_1, \wedge \ldots \wedge \ A_n$, un but défini, si θ est la substitution réponse associée à une réfutation SLD de $P \cup \{\neg G\}$, alors $(A_1 \wedge \ldots \wedge \ A_n)\theta$ est une conséquence logique de P.

1. c'est-à-dire après recommage des variables pour éviter toute confusion

Propriété 1.1.2 (Complétude de la résolution SLD) Si P est un programme logique et $(G)(A_1 \wedge \ldots \wedge\ A_n)$ un but défini, et si θ une substitution des variables de (G) telle que $(A_1 \wedge \ldots \wedge\ A_n)\ \theta$ soit une conséquence logique de P, alors il existe une réfutation SLD de $P \cup \neg G$ de substitution réponse σ, et une autre substitution γ telles que $\theta = \sigma\gamma$.

On obtient comme corollaire l'égalité entre l'ensemble de succès et le plus petit modèle de Herbrand M_p d'un programme logique P donné.

1.2 Exécution étendue

En exécution étendue, on part d'un but implicatif contenant des variables existentielles et universelles. On le prouve grâce à des règles généralisant la résolution SLD. On se sert d'un principe d'induction, structurelle ou par point fixe.

1.2.1 Preuve par exécution étendue

Kanamori et al. ont introduit l'exécution étendue, qui est une extension de l'exécution des clauses de Horn par résolution SLD au cas où les buts de départ sont des X-formules [**KS86**]. On se restreindra ici, comme dans[**Fri88**], à des buts implicatifs :

$$\forall\ X\ (A_1 \wedge \ldots \wedge A_n \Rightarrow \exists\ Y\ B_1 \wedge \ldots \wedge B_m)$$

où les A_i, B_j sont des atomes ; Y est le vecteur des variables existentielles de la formule, qui ne doivent pas intervenir dans l'hypothèse de l'implication, et X la vecteur des variables universelles. Par exemple :

$$(G)\ \forall\ X_1\ , X_2\ a(X_1, X_2) \wedge b(X_2) \Rightarrow \exists\ Y\ c(X_1, Y)$$

Dans [**KS88**], les variables universelles sont dites libres, et les variables existentielles sont déclarées 'undecided'. Une clause définie est un but implicatif avec une conclusion atomique et sans variable existentielle. Un but défini est un but implicatif sans hypothèse. Par la suite, on omet les quantificateurs dans les formules : les variables existentielles sont notées $?(X)$ et les autres variables sont universelles. Le but (G) se notera :

$$(G)\ a(X_1, X_2) \wedge b(X_2) \Rightarrow c(X_1, ?(Y))$$

Exemple : longueur de deux listes

À partir du prédicat app de concaténation de deux listes et du prédicat leng qui donne la longueur d'une liste :

$$\begin{array}{l} app([\,], Y, Y). \\ app([A|X], Y, [A|Z]) \text{ :- } app(X, Y, Z). \\ leng([\,], 0). \\ leng([A|X], s(N)) \text{ :- } leng(X, N). \end{array}$$

on cherche à prouver le but implicatif :

$$\text{(G)} \quad \text{app}(X, Y, ?(Z)) \wedge\ leng(?(Z), ?(T))$$

On commence par une induction structurelle sur X de type list [2] :

list([ ]).
list($A[X]$) :-list(X).

qui génère deux nouveaux buts :

(G1) app([ ], Y, Z) $\wedge\ leng(?(Z), ?(T))$
(G2) app(X, Y, Z_1) $\wedge\ leng(Z_1, T_1) \Rightarrow$ app($[A|X], Y, ?(Z)) \wedge\ leng(?(Z), ?(T))$

Les variables existentielles Z et T sont renommées en de nouvelles variables universelles Z_1 et T_1, et la variable universelle Y est inchangée. Dans le cas de base, on applique la règle de dci, ou inférence de clause définie, avec la première clause définissant app, et on obtient en unifiant la variable existentielle Z à Y :

(H) $leng$(Y, ?T)

Dans le cas récursif, on applique une dci au premier atome de la conclusion, en utilisant la substitution (Z) $\leftarrow [A|Z']$ qui ne modifie que la variable existentielle Z. La nouvelle variable Z' est existentielle :

$$\text{app}(X, Y, Z_1) \wedge\ leng(Z_1, T_1) \Rightarrow \text{app}(X, Y, ?(Z')) \wedge\ leng([A|?(Z')], ?(T))$$

On utilise ensuite la règle de simplification, qui permet d'éliminer deux atomes unifiables placés chacun d'un côté de l'implication, à condition que leur unification ne modifie effectivement que des variables existentielles. Ici, par exemple, app(X, Y, Z_1) et app($X, Y, ?(Z')$) sont unifiables par la substitution $(Z' \leftarrow Z_1)$. On obtient :

(J) $leng(Z_1, T_1) \Rightarrow\ leng([A|Z_1], ?(T))$

On applique alors une autre dci, avec la substitution $(T \leftarrow s(T'))$:

(K) $leng(Z_1, T_1) \Rightarrow\ leng(Z_1, ?(T'))$

On simplifie ensuite ce but en unifiant T' et T_1, ce qui donne le but vide, noté $\square$. Pour (H), on effectue alors une nouvelle induction structurelle, sur Y de type liste :

(H1) $leng([\,], ?(T))$
(H2) $leng[Y, T_1] \Rightarrow\ leng([A|Y]?, (T))$

Une dci appliquée à H_1 donne $\square$. Pour H_2, on applique également une dci et on obtient une forme (L), que l'on simplifie en $\square$. On a alors terminé la preuve, que l'on peut représenter par un **arbre de preuve** comme sur la figure 1.1.

2. on suppose connaître les types des variables apparaissant dans les prédicats "connus"

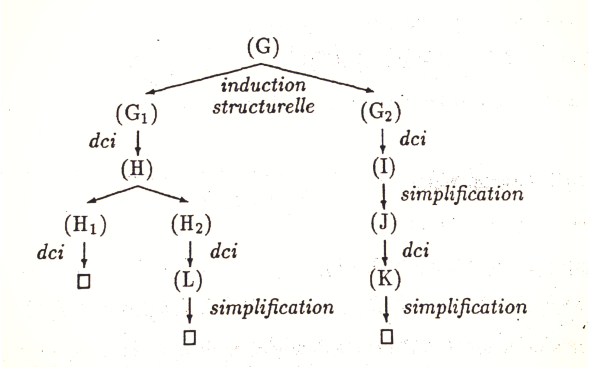

FIGURE 1.1 – Arbre de preuve

Formalisation

On considère des buts implicatifs de la forme $(G)H(X) \Rightarrow C(X, ?(Y))$, où X et Y sont les vecteurs des variables universelles et existentielles de la formule, et H et C sont des conjonctions d'atomes dont les prédicats sont définis dans un programme P.

Dans l'exemple de la longueur de deux listes, on a présenté les règles d'inférence de clause définie, de simplification, et d'induction structurelle. Le système d'exécution étendue originel de [**KS86**] n'utilise pas l'induction structurelle, mais l'induction par point fixe (*computational induction*) [**Cla79**]. Il utilise aussi une règle supplémentaire de nfi, ou *'negation as failure inference'*, dont on peut se passer si l'on utilise l'induction structurelle et une règle de réarrangement d'une formule. On présentera ici, outre la dci, la simplification et l'induction structurelle, les règles de réarrangement et d'induction par point fixe.

On définit la notion d'***unificateur existentiel***[3] de deux termes A et B : c'est un unificateur qui n'instancie que des variables existentielles de A et B.

Inférence de clause définie ou dci. Il s'agit simplement d'une extension de la règle de résolution SLD. Étant donnés un but implicatif (G) et une clause (C) de P :

$$\begin{array}{ll} (G) & H \Rightarrow (A_1 \wedge A_2 \wedge \ldots \wedge A_n)(?(Y)) \\ (C) & B \text{ :-} B_1, \ldots, B_m. \end{array}$$

tels que B soit unifiable, par exemple, avec A_1 : $A_1\sigma = B\sigma$, où σ est un 'mgu' *existentiel*, on fait une *dci* de (G) à A_1 en remplaçant A_1 par le corps de (C) :

$$(G') \quad H \Rightarrow (B_1 \wedge \ldots \wedge B_m \wedge A_2 \wedge \ldots \wedge A_n)(?(Y)\sigma)$$

Toute variable introduite par cette règle sera considérée comme existentielle.

La règle de [**KK88**] est déterministe : si plusieurs clauses de P permettent une dérivation, on en choisit une seule. On applique ici cette règle de façon non-déterministe avec

3. un tel unificateur est dit décidant ('deciding') dans [**KS86**] car il concerne les variables indécidés

toutes les clauses (C) de P qui le permettent, obtenant ainsi un ensemble de nouvelles formules. [4]

Simplification Étant donnée la formule :

$$(G) \quad A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow (B_1 \wedge B_2 \wedge \ldots \wedge B_m)(?(Y))$$

on unifie un atome de son hypothèse, par exemple A_1, avec un atome de sa conclusion, par exemple B_1, par un "mgu" existentiel σ : $A_1\sigma = B_1\sigma$, et on les élimine ; cela donne :

$$(G') \quad A_2 \wedge \ldots \wedge A_n \Rightarrow (B_2 \wedge \ldots \wedge B_m)(?(Y)\sigma)$$

Les nouvelles variables introduites par σ sont existentielles.
Une version légèrement différente, la simplification gardant l'hypothèse [**Fri88**], élimine la conclusion mais pas l'hypothèse.
On peut interpréter cette règle comme une forme de résolution SLD pour laquelle on utilise l'hypothèse A_1 de (G) comme clause définie.

Induction structurelle. C'est une forme d'induction simple [**Bur69**] qui requiert la connaissance des types des variables. Pour plus de clarté, on considère seulement le cas du type *list*. On part d'un but implicatif sans hypothèse [5], contenant une variable universelle X de type list :

$$\begin{array}{ll} (G) & \forall\, X :\ \text{list}\, C((?Y)) \\ & \text{list}([\,]). \\ & \text{list}([A/X]) \text{ :-list}(X). \end{array}$$

On la remplace par des formules correspondant aux deux clauses définissant *list* :

$$\begin{array}{ll} (G1) & C(?(Y))\{X \leftarrow [\,]\} \\ (G2) & C\{X \leftarrow X_1, Y \leftarrow Y_1\} \Rightarrow C(?(Y))\{X \leftarrow [A|X_1]\}; \end{array}$$

où A et X_1 sont de nouvelles variables universelles (X_1 de type list), et Y_1 est un nouveau vecteur de variables universelles renommant Y. Les variables universelles de la formule autres que X restent constantes lors de l'induction. On parlera d'induction structurelle avec "constantes".

4. la dci non-deterministe est une extension de la règle de [**KK88**]. Cependant, dans la plupart des exemples traités, une seule clause permet une dci, et les deux règles sont équivalentes.

5. on peut autoriser une hypothèse, à condition que la variable d'induction n'y figure pas [Fri90]

Induction par point fixe On reprend l'exemple de la longueur de deux listes à partir de :

$$(G) \quad \text{app}(X, Y, Z) \Rightarrow \text{leng}(Z, ?(T))$$

Le prédicat app constitue la plus petite relation qui contienne le triplet $([\,], Y, Y)$, et qui contienne $([A|X], Y, [A|Z])$ dès qu'elle contient (X, Y, Z). Pour prouver un but de la forme $\text{app}(X, Y, Z) \Rightarrow C(X, Y, Z)$ en utilisant une induction par point fixe [**Cla79**] par rapport à app, il faut que l'ensemble des solutions de $\text{app}(X, Y, Z)$ soit inclus dans celui de $C(X, Y, Z)$, donc que C vérifie le schéma récursif définissant app. Pour (G'), l'induction par point fixe produit les deux lemmes :

$$(G'_1) \quad \text{leng}(Y, ?(T))$$
$$(G'_2) \quad \text{leng}(Z, T_1) \Rightarrow \text{leng}([A|Z], ?(T))$$

Les buts (G'_1) et (G'_2) sont identiques respectivement aux buts (H) et (J) obtenus après induction structurelle. On a économisé une dci pour chaque lemme et une simplification dans le cas récursif : l'induction par point fixe est donc un peu plus rapide.

Plus formellement [**Fri88**], l'induction par point fixe s'applique à un but implicatif avec une hypothèse atomique.

$$(G) \quad A(X) \Rightarrow C(X, ?(Y))$$

Soit $P_A = \{C^1, \ldots, C^n\}$ le programme définissant A.

$$(C^i) \quad A(X\theta^1) : -A(X\theta^1_1), \ldots, A(X\theta^1_m), Rmd^i.$$

où Rmd^i représente les atomes non-récursifs. On suppose que $A(X)$ est en forme générale, c'est-à-dire avec des variables pour tous ses arguments (c'est une simplification par rapport à [**Fri88**]). L'induction par point fixe produit alors les buts (G_i), pour : i = 1…n :

$$(G_i) \quad C(X{\theta_1}^i, Y_1) \wedge \ldots \wedge C(X\theta^i_{m_i}, y_{m_i}) \wedge Rmd^i \Rightarrow C(X\theta^i, ?(Y)).$$

Les variables existentielles Y de C sont renommées en de nouvelles variables universelles pour chaque apparition de C dans l'hypothèse. Les variables universelles restent constantes lors de l'induction.

Règle de réarrangement D'après la définition que l'on a donnée de l'induction structurelle, elle s'applique à re formule sans hypothèse ; pour l'induction par point fixe au contraire il faut une hypothèse atomique. Pour obtenir une formule de la forme souhaitée, on est amené à **transférer des atomes** entre l'hypothèse et la conclusion ; pour préserver le sens de la formule, il faut alors parfois **changer la quantification** de certaines variables. C'est un des cas d'utilisation de la règle de réarrangement.

Étant donné un but implicatif :

$$(G) \quad H_1 \wedge H_2 \Rightarrow C(?(Y))$$

où H_1, H_2, et C sont des conjonctions d'atomes, chacune éventuellement vide, la règle de réarrangement [**Fri88**] permet de transférer une partie de l'hypothèse, par exemple H_1, vers la conclusion, et de changer la quantification de certaines variables de H_1 ; Si Z est le vecteur des variables existentielles de (G'), on obtient :

$$(G') \quad H_2 \Rightarrow (H_1 \wedge C)(?(Z))$$

Par exemple, pour prouver :

$$(G) \quad \text{app}(X, Y, Z) \Rightarrow \text{leng}(Z, ?(T))$$

par induction structurelle, on fait un transfert ; pour obtenir une formule qui ait inituitivement le même sens, on existentialise Z :

$$(G') \quad \text{app}(X, Y, ?(Z)) \wedge \text{leng}(?(Z), ?(T))$$

1.2.2 Correction de l'exécution étendue

On part d'un but implicatif $(G) \quad H(X) \Rightarrow C(X, ?(Y))$ qui spécifie une relation entre X et Y, et on en fait une preuve par exécution étendue. Les atomes sont définis dans un programme P de plus petit modèle de Herbrand M_P. On a des propriétés de correction qui généralisent le cas de l'exécution par résolution SLD.

Correction

On peut montrer la complétude et la validité de l'exécution étendue [**Kan86**], ce qui généralise les propriétés 1.1.1 et 1.1.2 :

Propriété 1.2.1 (Kanamori) Un but implicatif (G) dont les atomes sont définis dans un programme P est prouvable par exécution étendue si et seulement si (G) est conséquence logique de la complétion de P.

Dans l'exemple de la longueur de deux listes, le but initial :

$$(G) \quad \text{app}(X, Y, ?(Z)) \wedge \text{leng}(?(Z), ?(T))$$

est une conséquence logique de la complétion du programme définissant app et leng.

Equivalence entre les formules

Deux formules (F) et (G) sont dites ***équivalentes*** [**Fri88**] si $(F) \Leftrightarrow (G)$ est vrai dans M_P. La règle d'induction structurelle préserve l'équivalence entre les formules (on suppose

que l'on part d'une formule correctement typée). La règle de dci deterministe de [**KS86**] préserve l'équivalence quelle que soit la clause choisie ; la dci non-deterministe la préserve aussi, a fortiori.

Les règles de simplification, d'induction par point fixe et de réarrangement ont en revanche besoin d'être restreintes pour préserver l'équivalence. Donnons des exemples.

• Dans la propriété d'associativité suivante :

$$\text{app}(X, U, [B|T] \Rightarrow \text{app}(X, [A], ?(Y)) \wedge \text{app}(?(Y), ?(Z), [B|T])$$

app(X, U, T) et app(?(Y), ?(Z), T) sont unifiables par une substitution existentielle, mais la simplification correspondante donne : app(X,[A], X).

• Une induction par point fixe à partir de :

$$fib_1(X, Y, Z) \Rightarrow fib(X, Z)$$

avec les clauses récursives suivantes pour fib et fib_l :

$$fib(s(s(X)), Z)) \text{ :-} fib(X, T), fib(s(X), Y), add(T, Y, Z).$$
$$fib_1(s(X), Y, Z)) \text{ :-} fib_1(X, T, Y), add(T, Y, Z).$$

donne un lemme récursif faux, car n'importe quel Z plus grand que Y satisfait l'hypothèse et non la conclusion :

$$fib(X, Y) \wedge add(T, Y, Z) \Rightarrow fib(s(X), Z)$$

• Un transfert à partir de :

$$app(X, Y, Z) \Rightarrow leng(Z, ?(T))$$

qui n'existentialise pas la variable Z donne une formule fausse.

$$app(X, Y, Z) \wedge leng(Z, ?(T))$$

Propiété 1.2.2 (Fribourg) : Une simplication par rapport à un atome $p(X)$ préserve l'équivalente si $p(X)$ est fonctionnel par rapport à X et s'il est déconnecté en X dans l'hypothèse. L'induction par point fixe préserve l'équivalence entre les formules au cours de la preuve si l'on garde les hypothèse d'induction
Le but $R \wedge p(X) \Rightarrow q(X, ?(Y))$ est équivalent à celui l'on obtient après transfert et exixtentialisation de X :$R \Rightarrow p(?(X)) \wedge q(?(X), ?(Y))$, si $p(X)$ est fonctionnel et termine par rapport à X, et si X n'apparaît pas dans R.

Cependant cette définition de l'équivalence ne prend pas en compte la préservation de l'ensemble des solutions d'un but.Par exemple, une dci déterministe peut perdre des solutions pour les variables existentielles. Si on a pour fib les deux clauses de base :

$fib(0, s(0)).$
$fib(s(0), s(0)).$

le but $fib(?(X), s(0))$ admet pour X deux solutions 0 et $s(0)$, et une *dci* déterministe en choisissant une de ces deux clauses, élimine une des valeurs possibles pour X. Une *dci* non-déterministe, par contre, fournit les deux solutions.

Chapitre 2

Extraction de programmes logiques à partir d'une preuve par exécution étendue

L'extraction de programmes de preuves consiste à prouver une spécification et à en extraire un programme exécutable calculant tout ou partie de ce qui est spécifié.

En fonctionnel, il est possible de représenter une preuve par un terme fonctionnel : l'extraction d'un programme consiste ensuite à simplifier la preuve (par élimination de coupures). On peut citer plusieurs systèmes permettant l'extraction de programmes fonctionnels à partir de preuves : le système de développement de programmes NuPrl [**CAB86**], le système PRIZ [**MT90**], le système PX [**HN88**] qui permet l'extraction de programmes LISP, et la méthode de [**Pau89**] dans le calcul des constructions [**CH88**]. Le système NuPrl a été récemment étendu au cas des programmes logiques dans [**BSW90**].

En exécution étendue, on effectue une preuve d'un but implicatif initial, et on génère une ou plusieurs clause(s) définie(s) pour garder trace de l'application de chaque règle. À la fin de la preuve, on a un ensemble de clauses qui constitue le programme extrait. Ce programme vérifie des propriétés de correction par rapport à sa spécification.

2.1 Méthode d'extraction

Étant donné un but implicatif $(G) \quad H(X) \Rightarrow C(X, Y)$, on peut extraire de sa preuve par exécution étendue un programme calculant les variables existentielles Y en fonction des variables universelles X. Ceci se fait par l'intermédiaire d'un ***prédicat d'entrée-sortie*** $es_G(X, Y)$ qui concerne toutes les variables de (G) [Fri90].

2.1.1 Exemples : longueur de deux listes

A partir du prédicat app de concaténation de deux listes et du prédicat $leng$ qui donne la longueur d'une liste (cf. chapitre1), on cherche à synthétiser un prédicat qui donne la longueur cumulée de deux listes, en considérant qu'il s'agit de la longueur de la concaténation de ces deux listes. Pour cela, on s'intéresse au but implicatif :

$$(G)app(X, Y, ?(Z)) \wedge leng(?(Z), ?(T))$$

où Z est une variable existentielle intermédiaire. Une telle spécification constitue en fait un programme permettant de calculer la longueur T totale des listes X et Y ; mais la construction de la liste Z au cours du calcul est inutile si l'on n'est intéressé que par le résultat final T : on va chercher à éliminer cette variable.

On associe au but initial un prédicat $es_G(X, Y, Z, T)$ grâce auquel on va pouvoir garder une trace de la preuve, et dont on espère qu'il permettra de calculer directement T en fonction de X et Y. On commence par une induction structurelle sur X de type *liste*, qui génère les deux nouveaux buts :

$$(G1)\ app([\,], Y, Z) \wedge\ leng(?(Z), ?(T))$$
$$(G2)\ app(X, Y, Z_1) \wedge\ leng(Z_1, T_1) \Rightarrow app([A|X], Y, ?(Z)) \wedge\ leng(?(Z), ?(T))$$

On garde la trace de ce que l'on vient de faire dans les clauses d'entrée-sortie :

$$es_G([\,], Y, Z, T) \text{ :- } es_{G1}(Y, Z, T).$$
$$es_G([A|X], Y, Z, T) \text{ :- } es_G(X, Y, Z_1, T_1),\ es_{G2}(X, Y, Z_1, T_1, A, Z, T).$$

ou es_{G1} servira à garder trace de la preuve du cas de base et es_{G2} du cas récursif. Dans le cas de base, on applique la règle de dci avec la première clause définissant app :

$$(H)\ leng(Y, ?(T))$$

avec comme clause associée, pour garder trace de la substitution $(Z \leftarrow Y)$:

$$es_{G1}(Y, Y, T) \text{ :- } es_H(Y, T).$$

On pourrait continuer la preuve des (H) par une induction structurelle sur Y de type list ; On obtiendrait pour es_H des clauses identiques à celles qui définissent le prédicat $leng$. On décide plutôt de s'arrêter là dans la preuve de (G_1) en définissant es_H par :

$$es_H(Y, T) \text{ :- } leng(Y, T).$$

On dit que l'on a appliqué la règle de ***postulat***. Dans le cas récursif, on applique la règle de *dci* à $app([A|X], Y, ?(Z))$:

$$(I)\ app(X, Y, Z_1) \wedge\ leng(Z_1, T_1) \Rightarrow app(X, Y, ?(Z')) \wedge\ leng([A|?(Z')], ?(T)).$$

On a une clause pour garder trace de l'application de cette règle :

$$es_{G2}(X, Y, Z_1, T_1, A, [A|Z'], T) \text{ :- } es_I(X, Y, Z_1, T_1, Z', A, T).$$

On effectue ensuite une *simplification* entre $app(X, Y, Z_1)$ et $\exists\ Z'\ app(X, Y, Z')$:

$$(J)\ leng(Z_1, T_1) \Rightarrow\ leng([A|Z_1], ?(T))$$
$$es_I(X, Y, Z_1, T_1, Z_1, A, T) \text{ :- } es_J(Z_1, T_1, A, T)$$

On effectue ensuite une *dci* à $leng\ ([A|Z_1], ?(T))$:

$$(K)\ leng(Z_1, T_1) \Rightarrow leng(Z_1, ?(T'))$$
$$es_J(Z_1, T_1, A, s(T')) \text{ :- } es_K(Z_1, T_1, T').$$

On fait ensuite une simplification qui donne la formule $\square$, et on a comme clause associée :

$$es_k(Z_1, T_1, T_1).$$

On a alors terminé la preuve. On peut remarquer que l'arbre de preuve de la figure 2.1 est un sur-arbre de celui de la figure 1.1 puisque l'on a postulé (H)

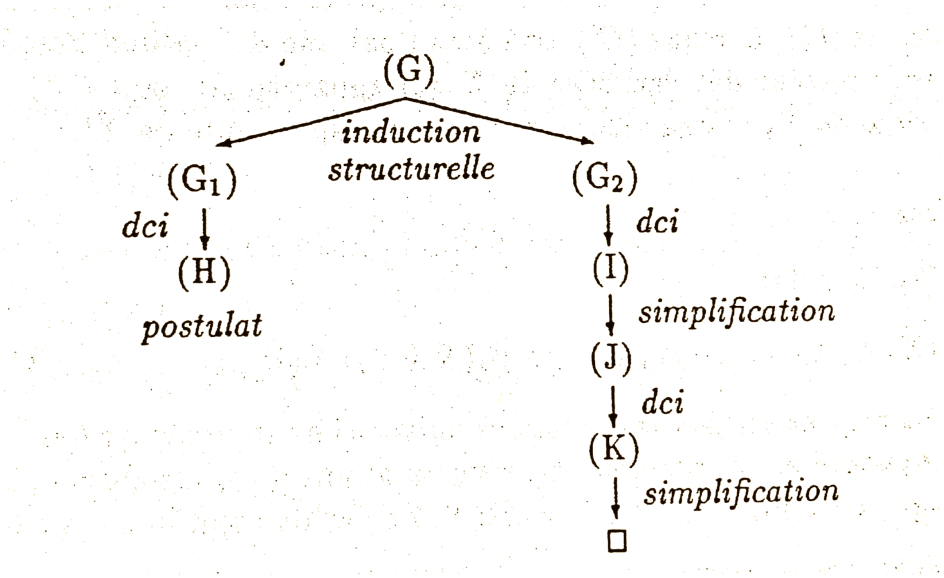

FIGURE 2.1 – Arbre de preuve

Le programme extrait est le suivant :

$$
\begin{aligned}
&es_G([\,], Y, Z, T) \text{ :- } es_{G1}(Y, Z, T).\\
&es_G([A|X], Y, Z, T) \text{ :- } es_G(X, Y, Z_1, T_1), es_{G2}(X, Y, Z_1, T_1, A, Z, T).\\
&es_{G1}(Y, Y, T) \text{ :- } es_H(Y, T).\\
&es_H(Y, T) \text{ :- } leng(Y, T).\\
&es_{G2}(X, Y, Z_1, T_1, A, [A|Z'], T) \text{ :- } es_I(X, Y, Z_1, T_1, Z', A, T).\\
&es_I(X, Y, Z_1, T_1, A, T) \text{ :- } es_J(Z_1, T_1, A, T).\\
&es_J(Z_1, T_1, A, s(T')) \text{ :- } es_K(Z_1, T_1, T').\\
&es_K(Z_1, T_1, T_1).
\end{aligned}
$$

Cela se simplifie facilement, par des dépliages (cf. chapitre 3), en :

$$
\begin{aligned}
&es_G([\,], Y, Y, T) \text{ :- } leng(Y, T).\\
&es_G([A|X], Y, [A|Z], s(T)) \text{ :- } es_G(X, Y, Z, T).
\end{aligned}
$$

Dans ce programme, la liste Z est toujours construite au cours du calcul, même si maintenant on voit clairement que cela n'est pas nécessaire au calcul de T.

2.1.2 Formalisation

On part d'un but implicatif $(G)\ H(X) \Rightarrow C(X, ?(Y))$, où H et C sont des conjonctions d'atomes dont les prédicats sont définis dans un programme P. On lui associe un prédicat d'entrée-sortie $es_G(X, Y)$. Les règles que l'on formalise ci-après sont la simplification, la *dci*, l'induction structurelle et par point fixe, le réarrangement, ainsi qu'une règle d'élimination d'hypothèse et une règle de postulat.

On note $\overline{es_G}$ pour $esc_G(X, Y)$ si X et Y sont respectivement les variables universelles et existentielles de (G). Lorsque (G') est obtenu par une *dci* ou une simplification, X' désignera le sous-vecteur des variables de X qui apparaissent dans (G'). Si t désigne une instance close de X, t' sera l'instance close correspondante de X'.

Simplification

Étant donnée la formule :

$$(G)\ A_1(X) \wedge (A_2 \wedge \ldots \wedge A_n)(X') \Rightarrow B_1\ (X, ?(Y)) \wedge\ (B_2 \wedge \ldots \wedge B_m)(X', ?(Y))$$

où X et Y sont respectivement les variables universelles et existentielles, on unifie un atome de son hypothèse, par exemple A_1, avec un atome de sa conclusion, par exemple B_1, par un 'mgu' existentiel σ : $A_1(X) = B_1(X, Y\sigma)$. On supprime ces deux atomes :

$$(G')\ (A_2 \wedge \ldots \wedge A_n)(X') \Rightarrow (B_2 \wedge \ldots \wedge B_m)(X', ?(Y\sigma))$$

Par convention, $?(t)$ désigne un vecteur t de termes dont certaines variables sont quantifiées existentiellement. Les variables existentielles de $?(Y\sigma)$ sont les variables de Y qui n'ont pas été liées par σ. Si (G') a pour variables existentielles Z, i.e. les variables existentielles de $?(Y\sigma)$ apparaissant dans (G'), la clause d'entrée-sortie est :

$es_G(X, Y\sigma)$:- $es_{G'}(X', Z)$.

Une version légèrement différente, **la simplification gardant l'hypothèse** [**Fri88**], consiste à éliminer la conclusion B_1, mais pas l'hypothèse A_1.

Inférence de clause définie ou dci

Etant donnés un but (G), et une clause (C) de P :

$(G)\ H(X') \Rightarrow A_1\ (X, ?(Y) \wedge\ (A_2 \wedge \cdots \wedge A_m)(X', ?(Y))$
$(C)\ B(L, M)$:- $(B_1, \ldots, B_m)(L, N)$.

si A_1 et B sont unifiables par $A_1\ (X, Y\alpha) = B\ \ (L\beta, M\beta)$, une *dci* donne :

$(G')\ H(X') \Rightarrow\ (B_1\ \wedge \ldots \wedge\ B_m)(?(L\beta), ?(N)) \wedge (A_2 \wedge \ldots \wedge A_m)(X', ?(Y\alpha))$

Les variables existentielles de (G'), notées Z, sont les variables de $?(N)$ ainsi que les variables de (G') apparaissant dans le vecteur de termes $L\beta \cup Y\alpha$ qui n'apparaissaient pas dans (G). Si es_G et es'_G sont les prédicats d'entrée-sortie associés à (G) et (G') respectivement, on génère pour garder trace de la preuve la clause d'entrée-sortie :

$es_G(X, Y\alpha)$:- $es_{G'}(X', Z)$.

La règle de [**KS86**] est *déterministe* : si plusieurs clauses de P permettent une dci, on en choisit une seule. On applique ici cette règle de façon *non-déterministe*, avec toutes les clauses (C) de P qui le permettent, obtenant ainsi un ensemble de clauses associées.

Induction structurelle

On considère seulement le cas du type *list*. Une induction structurelle, à partir de :

$(G)\ \forall\ X : list\ C(?(Y))$

donne les deux sous-buts et les deux clauses d'entrée-sortie :

$(G1)\ C\ (?(Y))\{X \leftarrow [\,]\}$
$(G2)\ C\{X \leftarrow X_1, Y \leftarrow Y_1\} \Rightarrow C(?(Y))\{X \leftarrow [A|X_1]\}$.
$\overline{es_G}\ \{X \leftarrow [\,]\}$:- $\overline{es_{G1}}$. $\overline{es_G}\ \{X \leftarrow$

La règle d'induction structurelle est associée à une construction de clauses d'entrée-sortie plus complexe que lors d'une *dci* ou d'une simplification, où, comme pour une réfutation SLD, on garde trace des substitutions utilisées.

Induction par point fixe

La définition est plus restreinte qu'au chapitre 1, afin d'obtenir la propriété de correction 2.2.2. Soit $A(X)$ en forme générale, défini par les clauses (C^1), ..., (C^p) :

$$(C^i)\ A\ (f_i(L_i, M_i)) : -A(g_i(L_i, N_i)), Rmd_i(L'_i, M'_i, N_i, O_i).$$

Où Rmd_i représente les atomes non-récursifs, les vecteurs de variables L_i, M_i, N_i et O_i sont deux à deux disjoints et L'_i (resp. M'_i) est le sous-vecteur des variables de L_i (resp. M_i) apparaissant dans Rmd_i. On suppose que la définition de A satisfait les conditions suivantes de **bien-fondé sémantique et de réciprocité [Fribourg]** :

$$(BF_i)\ Rmd_i, (L'_i, M'_i, N_i, O_i) \Rightarrow g_i\ (L_i, N_i) \ll\ f_i\ (L_i, M_i).$$
$$(R_i)\ A(fi(L_i, M_i)) \wedge Rmd_i(L'_i, M'_i, N_i, O_i) \Rightarrow A(g_i(L_i, N_i)).$$

où $\ll$ est un ordre bien fondé et stable par substitution. Par exemple, le prédicat $pgcd$ qui calcule le plus grand diviseur commun à deux entiers :

$pgcd(X, Y, Z)$:- $pgcd(X - Y, Y, Z), X \geq Y > 0.$
$pgcd(X, Y, Z)$:- $pgcd(X, Y - X, Z), Y \geq X > 0.$
$pgcd(0, Y, Y).$
$pgcd(X, 0, X).$

satisfait ces conditions, en prenant pour $\ll$ l'ordre < sur le premier élément du triplet :

$$(BF)\ X \geq Y > 0 \Rightarrow (X - Y, Y, Z) \ll\ (X, Y, Z).$$
$$(R)\ pgcd(X, Y, Z) \wedge X \geq Y > 0 \Rightarrow pgcd(X - Y, Y, Z).$$

A partir de :

$$(G)\ A(X) \Rightarrow C(X, ?(Y)),$$

une induction par point fixe produit :

$$(G_i)\ C(g_i(L_i, N_i), Y_1) \wedge Rmd_i(L'_i, M'_i, N_i, O_i) \Rightarrow C(f_i(L_i, M_i), ?(Y))$$

pour i = 1 . . . p, et la clause associée est :

$$es_G\ (f_i(L_i, M_i), Y) : -es_G\ (g_i(L_i, N_i), Y_1),\ Rmd_i\ (L'_i, M'_i, N_i, O_i), es_{Gi}\ (L_i, M_i, N_i, O_1, Y_1, Y).$$

où $Y_1 \ldots, Y_m$, sont de nouveaux vecteurs de variables universelles renommant les variables existentielles Y, toutes deux à deux distinctes.

Règle de réarrangement

Etant donné un but implicatif :

$$(G)\ H_1 \wedge H_2 \ \Rightarrow\ C(?(Y))$$

où H_1, H_2 et C sont des conjonctions d'atomes, un réarrangement transfère H_1 dans la conclusion et change la quantification. Si Z est le nouveau vecteur des variables existentielles, on a :

$$(G')\ H_2 \ \Rightarrow\ (H_1 \wedge C)(?(Z)),$$

On peut considérer que le nouveau but a le même prédicat d'entrée-sortie associé que l'ancien car il n'y a aucune nouvelle variable et aucune instantiation d'une variable existante par une substitution. On prend comme clause d'entrée-sortie : $\overline{es_G} : - \overline{es_{G'}}$.

Un cas particulier de réarrangement est le **transfert fonctionnel**. On suppose que $H_1(X,T)$ est fonctionnel par rapport à T, et on part de :

$$(G)\ H_1(X,T) \wedge H_2(X) \Rightarrow\ C(X,T,?(Y)),$$

On existentialise T lors du transfert de H_1 :

$$(G')\ H_2(X) \Rightarrow\ H_1\ (X,?(T)) \wedge C\ (X,?(T),?(Y)),$$

La clause d'entrée-sortie est alors $es_G\ (X,T,Y) : -es_{G'}(X,T,Y)$.

Règle d'élimination d'hypothèse

Cette règle [**Fri88**] consiste simplement à éliminer un ou plusieurs atome(s) de l'hypothèse d'un but implicatif. Pour préserver l'équivalence entre les formules, il faut que la partie de l'hypothèse à supprimer soit déconnectée (cf. chapitre 1) en un de ses arguments. À partir du but :

$$(G)\ A(X,Y) \wedge H(Y,Z) \Rightarrow C\ (Y,Z,?(T))$$

on obtient en supprimant $A(X,Y)$:

$$(G')\ H(Y.Z) \Rightarrow C\ (Y,Z,?(T))$$

et la clause d'entrée-sortie est $es_G(X, Y, Z, T) :- es_{G'}(Y, Z, T)$.

Règle de postulat

On peut à tout moment postuler un but sans hypothèse :

$(G)\ (A_1, \wedge \ldots \wedge A_n)(X, ?(Y))$

en le prenant pour corps de son prédicat d'entrée-sortie :

$es_G\ (X, Y)$:- $A_1\ (X, Y), \ldots, A_n(X, Y)$.

2.1.3 Troncature

Lorsque l'on veut utiliser un programme pour calculer certains de ses arguments (résultats) en fonction de certains autres (données), il se peut qu'il comporte des arguments "superflus", c'est-à-dire des arguments dont la présence n'est pas essentielle au calcul des résultats et dont la manipulation est en ce sens inutile. La troncature consiste alors tout simplement à éliminer ces arguments. Dans l'exemple de la longueur de deux listes on était arrivé pour $es_G(X, Y, Z, T)$ au programme suivant :

$es_G([\,], Y, Y, T)$:- $leng\ (Y, T)$.
$es_G([A|X], Y, [A|Z], s(T))$:- $es_G(X, Y, Z, T)$.

En tronquant es_G par rapport à son troisième argument Z, on obtient un programme définissant $es'_G(X, Y, T)$. Ses deux premiers arguments X et Y étant donnés, ce nouveau programme permet de calculer la longueur de leur concaténation sans effectuer le travail intermédiaire inutile concernant Z :

$es'_G([\,], Y, T)$:- $leng(Y, T)$.
$es'_G([A|X], Y, s(T))$:- $es'_G(X, Y, T)$.

Définition : forme tronquée

$p(X, Y)$ est un atome, X et Y étant des vecteurs de variables, une forme tronquée de p par rapport aux arguments Y est $p'(X)$, où p' est un nouveau nom de prédicat. Si P est un programme définissant $p(X, Y)$, on obtient une forme tronquée de P par rapport aux arguments Y de p en remplaçant dans P toute instance de $p(X, Y)$ par l'instance correspondante de p'(X) ; on peut aussi vouloir "propager" la troncature et tronquer un atome auxiliaire $q(Z, T)$ par rapport à certains de ses arguments, par exemple T, et on remplace alors toute instance de $q(Z, T)$ par l'instance correspondante de $q'(Z)$. Par exemple, si l'on veut tronquer le programme suivant :

$p(X, [A|Y], Z)$:- $p(X_1, Y, Z_1)$. $p_1(X, A, Z, X_1, Z_1)$.
$p_1(X, A, Z, X_1, [A1|Z])$.
$p_1([B|X], A, Z, [B|X_1], Z_1)$:- $p_1(X, A, Z, X_1, Z_1)$.

par rapport au premier argument de p, on tronque aussi par rapport aux premier et quatrième arguments de p_1 :

$p'([A|Y], Z)$:- $p'(Y, Z_1), p'_1\ (A, Z, Z_1)$.
$p'_1(A, Z, [A|Z])$.
$p'_1(A, Z, Z_1)$:- $p'_1(A, Z, Z_1)$.

Stratégie de propagation de la troncature

La définition de la troncature permet la propagation mais ne la spécife pas. Lorsque l'on a obtenu un programme par extraction, certaines variables du prédicat es_G sont "inintéressantes" et l'on tronque es_G par rapport aux arguments correspondants ; on ne sait pas par contre par rapport à quels arguments on a intérét à tronquer les prédicats auxiliaires. Pour cela, on peut appliquer la stratégie de propagation suivante.

On suppose pour simplifier que l'on désire tronquer le prédicat p par rapport à son premier argument. Une clause défnissant p est de la forme :

$(C)\ p(t_0, \ldots)$:- $p(t_1, \ldots), \ldots, p(t_n, \ldots), q(\ldots, u, \ldots), \ldots.$

où $t_o \ldots, t_n$,u ($n \geq 0$) sont des termes. On tronque d'abord p, ce qui donne :

$(C')\ p'(\ldots)$:- $p'(\ldots), \ldots, p'(\ldots), q(\ldots, u, \ldots), \ldots.$

On appelle X le vecteur des variables apparaissant dans les t_i. On propage la troncature au prédicat axiliaire q, à l'argument correspondant au terme u, si :

• on a effectivemeat une propagation,i.e. les variables de u intervenaient dans la première troncature $vars(u) \subset X$.

• cette propagation est justifiée par la troncature précédente, i.e. les variables de u n'apparaissent plus que dans q [1] : $vars(u) \cap\ (vars(C')|vars(q(\ldots, u, \ldots))) = \varnothing$

Dans l'exemple précédent, on tronque d'abord p par rapport à son premier argument :

$p'([A|Y], Z)$:-$p'(Y, Z_1), p_1(X, A, Z, X_1, Z_1)$.

Les termes $u_0 = X$ et $u_1 = X_1$ vérifient les conditions qu'on vient de donner, donc on tronque p_1 par rapport à ses premier et quatrième arguments. Par contre, dans l'exemple de la longueur de deux listes, quand on tronque es_G par rapport à son troisième argument, on *ne propage pas* la troncature à $leng(Y, T)$ dans la première clause car Y garde une occurrence dans es'_G.

Par la suite, on omettra de donner un nouveau nom au prédicat tronqué : il se distinguera du prédicat non tronqué uniquement par le nombre de ses arguments.

2.2 Correction

On part d'un but implicatif $(G)\ H(X) \Rightarrow C(X, ?(Y))$ spécifiant une relation entre X et Y. On le prouve par exécution étendue et on extrait de la preuve un programme pour $es_G(X, Y)$ que l'on simplifie par des dépliages et qre l'on tronque par rapport à certaines variables. La simplification par dépliages préserve toujours l'équivalence des programmes (cf. chapitre 3). On appelle P le programme qui définit les prédicats sur lesquels on travaille, de plus petit modèle de Herbrand M_P.

2.2.1 Extraction

Lorsque 1'on a terminé la preuve d'un but implicatif $(G)\ (H) \Rightarrow C\ (?(Y))$ dont les atomes sont définis par un programme P et de prédicat associé $\overline{es_G}$, c'est-à-dire lorsque l'on aboutit à un arbre de preuve dont toutes les feuilles sont $\Box$ ou bien ont donné lieu à un postulat, on a un progrmme extrait Q . On peut simplifer Q par des dépliages, obtenant ainsi un programme équivalent mais plus simple Q' . On dit que Q, ou Q' :

• est **partiellement correct** par rapport à (G) si dans $M_{P\cup Q}$:$\overline{es_G} \wedge H \Rightarrow C$.

• termine par rapport à (G) si dans $M_{P\cup Q}$: $H \Rightarrow \exists\, Y \overline{es_G}$

.

On a alors les propriétés de correction suivantes ([Fri90] pour la première) :

Propriété 2.2.1 (Fribourg) Lors d'une preuve utilisant les règles de dci, simplification et induction structurelle, le programme extrait termine et est partiellement correct par rapport au but de départ.

La règle de réarrangement ne garantit que la correction partielle, la terminaison devant étre vérifiée directement sur le programme extrait.

1. q est déconnecté en u du reste de (C').

Propriétés 2.2.2 Une preuve utilisant la règle d'induction par point fixe par rapport à une définition d'atome satisfaisant les conditions de bien-fondé sémantique et de réciprocité donne un programme extrait partiellement correct et qui termine .

Une preuve utilisant la règle de transfert fonctionnel donne un programme extrait partiellement correct et qui termine.

La règle de postulat garantit toujours la correction partielle, et garantit la terminaison si la conjonction d'atomes postulée est vrai.

L'élimination d'une hypothèse garantit la correction partielle et la terminaison.

Dans l'exemple de la longueur de deux listes, le programme obtenu est partiellement correct et termine par rapport à la formule de spécification. Afin de pouvoir prouver ces propriétés, on introduit les définitions suivantes pour une règle R permettant de passer d'un but (G) à des buts $(G_1), \ldots, G(_p)$. L'ensemble des clauses associées Q_R préserve la correction partielle (resp. la terminaison) si pour tout programme Q_{G_i} partiellement correct (resp. terminant) par rapport à (G_i), i étant compris entre 1 et p, le programme $Q_R \cup Q_{G_1} \cup \ldots \cup Q_{G_p}$, est partiellement correct (resp. termine) par rapport à (G).
On commence par un lemme qui servira pour les règles de dci et simplification.

Lemme 2.2.1 Soit un but (G) de variables universelles X et existentielles Y. On applique une règle (dci, simplification) faisant intervenir une ou plusieurs substitutions existentielles $\sigma_1 \ldots \sigma_p$, et conduisant aux buts $(G_1) \ldots (G_p)$. On a :

1. pour $k = 1, \ldots, p$, on note $Y_k \cup Z_k$ le vecteur de variables existentielles de $?(Y_{\sigma_k})$ où Z_k apparaît dans G_k mais Y_k non, et N_k le vecteur de variables internes de (Cl_k) (vide pour une simplification). On peut alors écrire $Y_{\sigma_k} = f_k(X, Y_k, Z_k)$ pour une fonction f_k. Si $X' \subset X$ est le vecteur de variables universelles restant dans (G'), la kième clause d'entrée-sortie est :
   $(ES_k)\ es_G(X, f_k(X, Y_k, Z_k))$:- $es_{G_k}(X', Z_k, N_k)$.
2. pour k compris entre 1 et p, si s est une instance close de X' et si v_k et n_k sont des termes clos tels que $es_{G_K}(s, v_k, n_k)$, alors il existe un terme clos u tel que l'on ait $es_G(r, u)$ pour toute extension close r de s.
3. si t est une instance close de X, t' l'instance close correspondante de X' et u un terme clos tels que $es_G(t, u)$, il existe k compris entre 1 et p et u_k, v_k et n_k termes clos tels que u soit de la forme $f_k(t, u_k, v_k)$ et qu'on ait $es_G(t', v_k, n_k)$.

Preuve

1. La clause d'entrée-sortie associée à l'obtention de (G_k) est :
   $(ES_k)\ es_G(X, Y_{\sigma_k})$:- $es_{G_k}(X', Z_k, N_k)$.
   Comme $Y_k \cup Z_k$ représente les variables existentielles de $?(Y_{\sigma_k})$ et que les variables universelles sont prises dans X, on peut noter $Y_{\sigma_k} = f_k(X, Y_k, Z_k)$.

2. Si s, v_k et n_k sont donnés tels que $es_{G_k}(s, v_k, n_k)$, en utilisant la clause d'entrée-sortie (ES_k) on a $es_G(X, f_k(X, u_k, v_k)$ pour un certain u_k. On en déduit pour toute extension close r de s : $es_G(r, f_k(r, u_k, v_k)$ soit en posant $u = f_k\ (r, u_k, v_k)$ et $es_k\ (r, u)$.

3. On a $es_G(t, u)$, c'est donc qu'une clause (ES_k) a servi à l'inférer, pour un certain k compris entre 1 et p. En utilisant la partie "seulement si" implicite de (ES_k), on déduit alors que u est de la forme $f_k(t, u_k, v_k)$ pour des termes clos u_k et v_k, et qu'il existe un n_k tel que $es_{G_k}(t', v_k, n_k)$. □

Preuve : règle de simplification

La formule initiale est $(G)\ B(X) \wedge H(X') \Rightarrow A(X, ?(Y)) \wedge C(X', ?(Y))$, où A et B sont des atomes, H et C des conjonctions d'atomes, et X et Y les vecteurs des variables universelles et existentielles de (G). Son prédicat d'entrée-sortie est $es_G(X, Y)$. On suppose que A et B sont unifiables de façon existentielle par $A(X, Y\sigma) = B(X)$, où σ ne s'applique pas à B et élimine les variables existentielles Y qui apparaissent effectivement dans A, car B ne contient pas de variable existentielle.

Pour traiter à la fois le cas de la simplification "normale" et celui de la simplification gardant l'hypothèse, on note entre crochets ce qui concerne la deuxième version de la règle. Une simplification donne alors :

$(G')\ [B(X)\wedge]\ H(X') \Rightarrow C(X', ?(Y\sigma))$

On peut remarquer que les variables existentielles de $?(Y\sigma)$, que l'on note ici Z, apparaissent toutes dans (G'). D'après le premier point du lemme 2.2.1, on a alors $Y\sigma = f(X, Z)$, N est vide et la clause d'entrée-sortie est :

Soit $Q_{simpl} = \{ES\}$ le programme associé.

• Q_{simpl} préserve la terminaison. Si $Q_{G'}$ est un programme pour $\overline{es_{G'}}$ tel que :

$[B(X)\wedge]\ H(X') \Rightarrow\ \exists Z\ es_{G'}(X', Z)$, dans $M_{P\cup Q_{G'}}$ (Term)

on veut montrer que $Q_G = Q_{simpl} \cup Q_{G'}$ vérifie :

$B(X) \wedge H(X') \Rightarrow \exists\, Y\ es_G\ (X, Y)$, dans $M_{P\cup Q_G}$

On suppose donnés $Q_{G'}$ vérifiant $(Term)$ et un terme clos t tel que $B(t) \wedge H(t')$, et on cherche un terme clos u tel que $es_G\ (t, u)$. On a $[B(t)\wedge]H(t')$; en se servant de $(Term)$, il existe un v clos tel que $es_G\ (t', v)$. D'après le deuxième point du lemme 2.2.1, il existe alors un u clos tel que l'on ait $es_G\ (t, u)$ pour toute extension r de t', donc en particulier $es_G\ (r, u)$.

• Q_{simpl} préserve la correction partielle. Pour tout $Q_{G'}$ définissant $\overline{es_{G'}}$ tel que :

$$es_{G'}\ (X', Z) \wedge [B(X)\wedge]\ H(X') \Rightarrow C\ (X', f(X, Z)),\ \text{dans}\ \ M_{P\cup Q_{G'}} \quad \text{(Corr)}$$

on veut montrer que $Q_G = \ Q_{simpl} \cup Q_{G'}$ vérifie :

$$es_G\ (X.Y) \wedge B(X) \wedge H(X') \Rightarrow\ A\ (X, Y) \wedge C\ (X', Y), \text{dans}\ M_{P\cup Q_G}$$

On suppose donné $Q_{G'}$ vérifiant $(Corr)$, ainsi que des termes clos t et u tels que $es_g\ (t, u)$ et $B(t) \wedge H(t')$, et on cherche à montrer $A(t, u) \wedge C(t', u)$. On a $es_G(t, u)$, donc d'après le troisième point du lemme 2.2.1 le terme u est de la forme $f(t, v)$ pour un terme clos v, et on a $es_{G'}(t', v)$.

Les hypothèses $[B(t)\wedge]\ H(t')$ et $es_{G'}(t, v)$ de (Corr) étant vérifiées, on en déduit $C(t', f(t, v))$,soit $C(t', u)$.

Puisque l'on a fait une simplification, on a l'égalité $B(X) = A(X, f(X, Z))$. On a ici $B(t)$, donc aussi $A(t, f(t, Z)$; mais en fait, comme une simplification ne fait qu'éliminer les variables existentielles de A, Z n'apparaît pas et il s'agit d'un terme clos. On a bien finalement $A(t, u) \wedge C(t', u)$. □

La preuve concernant la dci déterministe apparaît dans [Fri90].

Preuve : règle de dci non-déterministe

Le but initial est de la forme $(G)\ \ H(X') \Rightarrow\ \ A\ (X, ?(Y)) \wedge C\ (X', ?(Y))$, où A est un atome, H et C des conjonctions d'atomes, et X et Y les vecteurs des variables universelles et existentielles, avec $X' \subset X$. Le prédicat associé est $es_G\ (X, Y)$. Pour $i = 1, ..., p$, on suppose que l'on a une clause de P :

$$(Cl_i) \text{ :- } B^i(L_i, M_i)(B^i_1, \ldots, B^i_{p_i})(L_i, N_i)$$

telle que A et B' soient unifiables par un 'mgu' $\alpha_i \cup \beta_i$, avec α_i existentiel : $A\ (X, Y\alpha_i) = B'\ (L_i\beta_i, M_i\beta_i,)$. La *dci* donne alors le nouveau but :

$$(G_i)\ H(X') \Rightarrow (B^i_1 \wedge \ldots (B^i_{p_i})(?(L_i\beta_i), ?(N_i))\ \wedge C(X', ?(Y\alpha_i,))$$

Les notations du premier point du lemme 2.2.1 se retrouvent comme suit : les variables internes N_i de (Cl_i) sont existentielles ; les variables existentielles de $?(Y\alpha_i)$ sont $Y_i \cup Z_i$. où Z_i, représente les variables de $L_i\beta_i$ qui seront existentielles dans (G_i) et Y_i celles de $M_i\beta_i$: qui n'apparaissent pas dans (Gi). On écrit alors $L_i\beta_i = g_i(X, Z_i)$ et $M_i\beta_i = h_i, (X, Y_i)$. On a $Y\alpha_i = f_i(X, Y_i, Z_i)$ et la clause d'entrée-sortie est :

$(ES_i)\ es_G(X, f_i(X, Y_i, Z_i))$:- $es_{G_i}(X', Z_i, N_i)$.

Soit $Q_{dci} = \{ES_1, \ldots, ES_p\}$ le programme associé à l'application de la règle de dci.

• Q_{dci} préserve la terminaison. Si le programme Q_{G_i}, définissant $\overline{es_{G_i}}$, est tel que :

$H(X') \Rightarrow \exists\, Z_i, N_i\ \ es_{G_i}\ (X', Z_i, N_i)$, dans $M_{P \cup Q_{G_i}} \quad (Term_i)$

on veut montrer que $Q_G = Q_{dci} \cup Q_{G_i} \cup ... \cup Q_{G_p}$ vérifie :

$H(X') \Rightarrow \exists\, Y,\ es_G(X, Y)$, dans $M_{P \cup Q_G}$

On suppose donné $Q_{G_1} \ldots Q_{G_p}$, vérifiant $(Term_i) \ldots (Term_p)$, et un terme clost t'tel que $H(t')$. On cherche u clos et t extension de t' telsque $es_G\ (t, u)$. On a $H(t')$: en se servant par exemple de $(Term_1)$, il existe v_1 et n_1 tels que $es_{G_1}\ (t', v_1, n_1)$. D'après le deuxième point du lemme 2.2.1, appliqué pour (G_1), il existe u clos tel que l'on ait $es_G(r, u)$ pour toute extension r de t'. On peut donc prendre pour t toute extension de t'.

• Q_{dci} préserve la correction partielle. Pour tout Q_G définissant $\overline{es_{G_i}}$tels que :

$es_{G_i}\ (X', Z_i, N_i) \wedge H(X') \Rightarrow\ (B_1^i \wedge \cdots \wedge B_{p_i}^i)(g_i(X, Z_i), N_i)$
$\quad \wedge C\ (X', f_i(X, Y_i, Z_i))$, dans $M_{P \cup Q_{G_i}}\ (Corr_i)$

on veut montrer que $Q_G = Q_{dci} \cup Q_{G_1} \cup \ldots \cup Q_{G_p}$ vérifie :

$es_G\ (X, Y) \wedge H(X') \Rightarrow A\ (X, Y) \wedge C\ (X', Y)$ dans $M_{P \cup Q_G}$

On suppose donné $Q_{G_1} \ldots Q_{G_p}$, vérifiant $(Corr_i) \ldots (Corr_p)$, ainsi que t et u clos tels que $es_G(t, u)$ et $H(t')$, et on cherche à montrer $A\ (t, u) \wedge C(t', u)$.

On a $es_G\ (t, u)$, donc d'après le troisième point du lemme 2.2.1, il existe k compris entre 1 et p tel que u soit de la forme $f_k(t, u_k, v_k)$ pour des termes clos u_k et v_k, et il existe n_k clos tel que $es_{G_k}(t', v_k, n_k)$.

Les hypothèses $es_{G_k}\ (t', v_k, n_k)$ et $H(t')$ de $(Corr_k)$ étant vérifiées, on en déduit : $(B_1^k \wedge \cdots \wedge B_{p_k}^k)(g_k(t, v_k), n_k)$ et $C(t', f_k\ (t, Y_k, v_k))$ pour tout Y_k, donc en particulier $C(t', f_k\ (t, u_k, v_k))$.

On a $(B_1^k \wedge \ldots \wedge B_{p_k}^k)(g_k\,(t, v_k), n_k)$, qui est le corps de la clause (Cl_k) de (P) ; on en déduit la tête $B^k\,(g_k(t, v_k), h_k(t, Y_k))$ pour tout Y_k, donc en particulier $B^k\,(g_k(t, v_k), h_k(t, u_k))$.

Puisqu'on a fait une dci, on a l'égalité $B^k\,(g_k(t, v_k), h_k\,(t, u_k))$ = $A\,(t, f_k\,(t, u_k, v_k))$.

On avait déjà $C\,(t', f_k\,(t, u_k, v_k))$, et comme $f_k\,(t, u_k, v_k) = u$ on a en fait $A\,(t, u) \wedge C\,(t', u)$ □.

Preuve : règle d'induction structurelle

On traite le cas du type *list*. Le but initial est $(G)\ X$: list $C(X, Y, ?(Z))$, où C est une conjonction d'atomes et Y représente les variables universelles autres que X. Le prédicat associé est $es_G(X, Y, Z)$. Une induction structurelle sur X : list donne :

$(G_1)\ C([\,], Y, ?(Z))$.
$(G_2)\ C(X_1, Y, Z_1) \Rightarrow\ C([A|X_1], Y, ?(Z))$.

de prédicat d'entrée-sortie $es_{G_1}(Y, Z)$ et $es_{G_1}(X_1, Y, Z_1, A, Z)$ (on rapelle que les prédicats d'entrée-sortie sont notés en prenant les variables dans leur ordre d'apparition de gauche à droite). Les clauses associées sont :

$(ES_1)\ es_G([\,], Y, Z)$:- $es_G(Y, Z))$.
$(ES_2)\ es_G([A|X_1], Y, Z)$:- $es_G(X_1, Y, Z)), es_{G_2}(X_1, Y, Z_1, A, Z)$.

Soit $Q_{struct} = Es_1, Es_2$ le programme associé.

• Q_{struct} préserve la correction partielle. Pour tous programmes Q_{G_1}, Q_{G_2} définissant $\overline{es_{G_1}}$, es_{G_2} tels que :

$es_{G_1}(Y, Z) \Rightarrow C([\,], Y, Z)$, dans $M_{P \cup Q_{G_1}}$ $\quad (Corr_1)$
$es_{G_2}(X_1, Y, Z_1, A, Z) \wedge C(X_1, Y, Z_1) \Rightarrow C([A|X_1], Y, Z)$, dans $M_{P \cup Q_{G_2}}$ $\quad (Corr_2)$

on veut montrer que $Q_G = Q_{struct} \cup Q_{G_1} \cup\ \cup Q_{G_2}$, vérifie :

$es_G(X, Y, Z) \Rightarrow C(X, Y, Z)$, dans $M_{P \cup Q_G}$ $\quad (Corr)$

On suppose donnés Q_{G_1} et Q_{G_2} vérifant $(Corr_1)$ et $(Corr_2)$. Pour des termes clos t, u et v, on suppose $es_G(t, u, v)$ et on montre $(Corr)$ par récurrence sur la taille de la liste t .

- Si t= [ ], on a $es_G([\,], u, v)$ et, en utilisant la partie "seulement-si implicite de (ES_1), on déduit $es_{G_1}(u, v)$. D'après $(Corr_1)$ on a alors $C([\,], u, v)$.

- Si $t = [a|t_1]$, on utilise la partie "seulement-si" implicite de (ES_2) pour déduire de $es_G\,([a/t_1], u, v)$ les deux atomes $es_{G_2}\,(t_1, u, v_1, a, v)$ et $es_G\,(t_1, u, v_1)$ pour un certain terme clos v_1. On déduit $C\,(t_1, u, v_1)$ du deuxième de ces atomes grâce à l'hypothèse de récurrence. En utilisant $(Corr2)$, on a alors $C([a|t_1], u, v)$.

• Q_{struct} préserve la terminaison. Pour tous programmes Q_{G_1}, et Q_{G_2}, définissant : $\overline{es_{G_1}}$ et $\overline{es_{G_2}}$ tels que :

$\exists\, Z\ es_{G_1}(Y, Z)$, dans $M_{P\cup Q_{G_1}}$. $(Term_1)$
$C\,(X_1, Y, Z_1) \Rightarrow\ \exists\, Z\ es_{G_2}\,(X_1, Y, Z_1, A, Z)$, dans $M_{P\cup Q_{G_2}}\ (Term_2)$

on veut montrer que $Q_G = Q_{struct} \cup Q_{G_1} \cup\ \cup Q_{G_2}$, vériñe :

$\exists\, Z\ es_G(X, Y, Z)$, dans $M_{P\cup Q_G}$.

On suppose donnés Q_{G_1}, et Q_{G_2}, vérifiant $(Term_1)$ et $(Term_2)$. On se donne des termes clos t et u, et on prouve par récurrence sur la taille de la liste t qu'il existe un terme clos v tel que $es_G(t, u, v)$.

- Si t= [ ], d'après $(Term_1)$ il existe un v tel que , $es_{G_1}(u, v)$, et en utilisant la clause (ES_1) on a $es_G(t, u, v)$.

- Si $t = [a|t_1]$, on utilise l'hypothèse de récurrence pour trouver un v_1 clos tel que $es_G(t_1, u, v_1)$. On se sert alors du fait que l'induction structurelle préserve la correction partielle pour prouver $C(t_1, u, v_1)$, ce qui permet d'utiliser$(Term2)$ et d'en déduire $es_{G_2}(t_1, u, v_1, a, v)$ pour un certain v. En utilisant(ES_2) on peut alors déduire $es_G([a|t_1], u, v)$. □

Preuve : règle d'induction par point fixe.

Le but initial est $(G)\ A(X) \Rightarrow\ C(X, ?(Y))$, où C est une conjonctions d'atomes, $A(X)$ est en forme générale et A est défini par les clauses $(C^1), \ldots (C^p)$, satisfaisant les conditions de bien-fondé sémantique (BF_i) et de réciprocité (R_i) pour tout i :

$(C^i)\ A(f_i(L_i, M_i))$:- $A(g_i(L_i, N_i)), Rmd_i(L'_i, M'_i, N_i, O_i)$.

où L_i, M_i, N_i, O_i sont deux à deux disjoints et $L'_i(resp.M'_i)$ est le sous-vecteur des variables de $L_i\ (resp.M_i)$ apparaissant dans Rmd_i. L'induction par point fxe donne :

$(G_i)\ C(g_i(L_i, N_i), Y_i) \wedge Rmd_i(L_i, M_i, N_i, 0_i) \Rightarrow C(f_i(L_i, M_i), ?(Y))$.

Les p clauses d'entrée-sortie associées sont :

$(ES_i)\ es_G(f_(L_i, M_i), Y)$:- $es_G(g_i(L_i, N_i), Y_1), es_G(L_i, N_i, Y_i, M_i, O_i, Y),$
$Rmd_i(L'_i, M'_i, N_i, O_i)$.

Soit Q_{pfixe} = $\{ES_1, \ldots, ES_p\}$ le programme associé.

• Q_{pfixe} préserve la correction parielle. Pour tous programmes $Q_{G_1}, \ldots, Q_{G_p}$ défnissant $\overline{es_{G_1}}, \ldots, \overline{es_{G_p}}$ tels que :

$$es_{G_i}(L_i, N_i, Y_1, M_i, O_i, Y) \wedge C(g_i(L_i, N_i), Y_1) \wedge Rmd_i(L'_i, M'_i, N_i, O_i) \Rightarrow C(f_i(L_i, M_i), Y), \text{dans } M_{P \cup Q_{G_i}} \quad (Corr_i)$$

on veut montrer que $Q_G = Q_{pfixe} \cup Q_{G_1} \cup \ldots \cup Q_{G_n}$ vérife :

$$es_G(X, Y) \ \wedge \ A(X) \Rightarrow \ C(X, Y) \text{dans } M_{P \cup Q_G} \quad (Corr)$$

On suppose donnés $Q_{G_1}, \ldots, Q_{G_p}$ vérifant $(Corr_1), \ldots, (Corr_p)$. On suppose de plus $es_G(t, u)$ et $A(t)$ pour des termes clos t et u, et on montre $(Corr)$ par récurrence sur t grâce à l'ordre bien fondé $\ll$

.

Comme es_G est défini parQ_{pfixe}, une clause (ES_k) a été utilisée pour inférer $es_G(t, u)$, et on a $t = f_k(l_k, m_k)$ pour des termes clos l_k et m_k. On a alors $es_G(g_k(l_k, m_k), u_1)$, Rmd_k $(l'_k, m'_k, n_K o_k)$ et $es_{G_k}(l_k, n_k, u_1, m_k, o_k, u)$ avec n_k ,o_k et u_1 clos (les termes l_k et m_k sont des instances closes de L_k et M_k et l'_k et m'_k sont les instances correspondantes de L'_k et M'_k)

D'après la condition de bien-fondé sémantique (BF_k) on a $g_k(l_k, n_k) \ll \ t$. En appliquant l'hypothèse de récurence on a $C(g_k(l_k, n_k), u_1)$.

D'aprés la condition de reciprocité (R_k) on a $A(g_k(l_k, n_k))$. Toutes les hypothèses de $(Corr_k)$ sont alors vérifiés, d'où $C(f_k(l_k, m_k), u)$, c'est-à-dire $C(t, u)$.

•Q_{pfixe} préserve la terminaison. Pour tous programmes $Q_{G_1} \ldots, Q_{G_p}$ définissant $\overline{es_{G_1}}, \ldots, \overline{es_{G_p}}$ tels que :

$$C(g_i(L_i, N_i), Y_1) \wedge Rmd_i(L'_i, M'_i, N_i, O_i) \Rightarrow \exists \, Y \ es_{G_i}(L_i, N_i, Y_1, M_i, O_i, Y), \text{dans } M_{P \cup Q_{G_i}} \quad (Term_i)$$

on veut montrer que $Q_G = Q_{pfixe} \cup Q_{G_1} \ \cup \ldots \cup Q_{G_n}$ vérifie :

$$A(X) \ \Rightarrow \exists \, Y es_G(X, Y), \text{dans } M_{P \cup Q_G} \quad (Term)$$

On suppose donnés $Q_{G_1}, \ldots, Q_{G_p}$ vérifant $(Term_1), \ldots, (Term, p)$, et $A(t)$ pour un terme clos t. On prouve $(Term)$ par récurrence sur t grâce à l'ordre $\ll$.

On a $A(t)$. A étant défini par les clauses (C^i), une clause (C^k) a été utilisée pour l'inférer, et on a forcément $t = f_k(l_k, m_k)$ pour des termes clos l_k et m_k. On a alors $A(g_k(l_k, n_k))$ et $Rmd_k(l'_k, m'_k, n_k, o_k)$, pour des termes clos n_k et o_k (l'_k et m'_k sont les instances de L'_k et M'_k correspondant à l_k et m_k).

D'après la condition de bien-fondé sémantique (BF_k) on a $g_k(l_k, n_k) \ll t$ En appliquant l'hypothèse de récurrence on trouve v_1 clos tel que $es_G(g_k(l_k, n_k), v_1)$.

On se sert alors du fait que l'induction par point fixe préserve la correction partielle pour obtenir $C(g_k(l_k, n_k), u_1)$.

Toutes les hypothèses de $(Term_k)$ sont maintenant vérfiées, et on en déduit qu'il existe un u tel que $es_G(l_k, n_k, u_1, m_k, o_k, u)$. En utilisant la clause (ES^k) dont le corps est vérifié, on a alors $es_G(t, u)$. □

Preuve : règle de réarrangement

Le but initial est de la forme $(G)(H_1 \wedge H_2)(X) \Rightarrow C(X, ?(Y))$, où H_1, H_2 et C sont des conjonctions d'atomes. Le prédicat associé est $es_G(X, Y)$. Un réarrangement donne :

$(G')\ H_2 \Rightarrow (H_1 \wedge C)(?(Z))$.

où Z est l'ensemble des variables existentielles de (G'). La clause d'entrée-sortie est :

$(ES)\ es_G(X, Y)$:- $es_{G'}(X, Y)$.

Pour montrer que (ES) préserve la correction partielle, il sufit de montrer que pour tout programme $Q_{G'}$ défnissant $\overline{es_{G'}}$ qui vérifie :

$\overline{es_{G'}} \wedge H_2 \Rightarrow H_1 \wedge C$, dans $M_{P \cup Q_{G'}}$ $\qquad (Corr)$

si $Q_G = Q_{G'} \cup \{\text{ES}\}$, on a dans $M_{P \cup Q_G}$: $\overline{es_G} \wedge H_1 \wedge H_2 \Rightarrow C$, ce qui est vrai en utilisant la partie "seulement si" implicite de (ES). □

Preuve : règle de transfert fonctionnel

Le but initial est de la forme $(G)\ H_1(X, T) \wedge H_2(X) \Rightarrow C(X, T, ?(Y))$, où $H_1(X, T)$ est fonctionnel en T. Un transfert fonctionnel donne le nouveau but :

$(G')\ H_2\ (X) \Rightarrow H_1\ (X, ?(T)) \wedge C\ (X, ?(T), ?(Y))$

et la clause d'entrée-sortie est : $(ES)\ es_G(X, T, Y)$:- $es_{G'}(X, T, Y)$.

Cette règle préserve la correction partielle comme cas particulier de réarrangement.

On veut ensuite montrer que $(ES$ préserve la terminaison. Pour tout programme $Q_{G'}$ définissant $\overline{es_{G'}}$ qui vérifie :

$$H_2(X) \Rightarrow \exists\, T, Y\ es_{G'}(X, T, Y), \text{dans} M_{P \cup Q_{G'}} \qquad (Term).$$

on doit avoir dans $M_{P \cup Q_{G'} \cup \{\text{ES}\}} : H_1(X, T) \wedge H_2(X) \Rightarrow \exists\, Y\ es_G(X, T, Y)$.

On se donne des termes clos t et w tels que $H_1(t, w) \wedge H_2(t)$. Par (Term), il existe alors w' et u clos tels que $es_{G'}(t, w', u)$, ce qui implique en particulier que l'on ait $H_1(t, w')$ à cause de (Corr). Comme $H_1(X, T)$ est fonctionnel en T, on a $w' = w$, donc on a $es_{G'}(t, w, u)$. En utilisant (ES), on a alors $es_G(t, w, u)$. □

Preuve : règle d'élimination d'hypothèse

La formule initiale est $(G)\ A(X, Y) \wedge\ H(Y, Z)\ \Rightarrow\ C(Y, Z, ?(T))$, où A, H et C sont des conjonctions d'atomes et T est le vecteur des variables existentielles de (G). Le prédicat d'entrée-sortie est $es_G(X, Y, Z, T)$. En éliminant l'hypothèse A on obtient le but (G') $H(Y, Z)\ \Rightarrow\ C(Y, Z, ?(T))$, et la clause d'entrée-sortie constituant le programme associe Q_{elim} est $es_G(X, Y, Z, T)$:- $es_{G'}(Y, Z, T)$.

• Q_{elim} préserve la terminaison. Il faut montrer que pour tout programme $Q_{G'}$ vérifiant $H(Y, Z) \Rightarrow \exists\, T\, es_{G'}(Y, Z, T), Q_G = Q_{G'} \cup Q_{elim}$. Vérife $A(X, Y) \wedge H(Y, Z) \Rightarrow \exists\, T\, es_G(X, Y, Z, T)$. On se donne des termes clos t, u et v tels que $A(t, u) \wedge H(u, v)$. On a $H(u, v)$, donc il existe w tel que $es_{G'}(u, v, w)$ En utilisant Q_{elim}, on a alors $es_G(X, u, v, w)$ pour tout X, et en particulier $es_G(t, u, v, w)$

• Q_{elim} préserve la correction partielle, Pour cela, il faut montrer que pour tout programme $Q_{G'}$ définissant $es_{G'}$ et vérifiant $es_{G'}(Y, Z, T) \wedge\ H(Y, Z)\ \Rightarrow\ C(Y, Z, T)$. C'est vrai en utilisant la partie "seudement si" implicite de la clause d'entrée-sorte. □

Preuve : règle de postulat

La formule initiale est $(G)(A_1 \wedge \ldots \wedge\ A_n)(X, ?(Y))$, où les A_i, sont des atomes, et X est le vecteur des variables universelles de (G). Son prédicat d'entrée-sortie est $es_G(X, Y)$.

Un postulat consiste à prendre comme clause d'entrée-sortie :

$$(ES)\ es_G(X, Y) \text{ :-} A_1\ (X, Y), \ldots, A_n(X, Y).$$

Soit alors $Q_{post} = \{ES\}$ le programme associé.

• Q_{post} préserve la terminaison, On veut montrer que $\exists\, Y es_G(X,Y)$ est vrai dans $M_{P\cup Q_{post}}$, D'après (ES), cela revient à montrer $\exists\ Y\ (A_1 \wedge \ldots \wedge A_n)(X,Y)$. Si (G) est vrai, Q_{post} préserve la terminaison .

• Q_{post} préserve la correction partielle. Pour cela, il faut montrer que $es_G(X,Y) \Rightarrow (A_1 \wedge \ldots \wedge A_n)(X,Y)$ est vrai dans $M_{P\cup Q_{post}}$. C'est vrai en utilisant la partie "seulement si" implicite de (ES). □

2.2.2 Troncature

On extrait d'une preuve de (G) un programme pour es_G, dont on suppose qu'il est partiellement correct et qu'il termine, puis on tronque es_G par rapport à des variables existentielles de (G), On peut montrer que les propriétés de correction partielle et de terminaison sont conservées si le nouveau programme est fonctionnel par rapport aux variables existentielles de (G) quil contient encore. On a les résultats suivants [Fri90] :

Lemme 2.2.2 (Fribourg) Si le prédicat d'entrée-sortie $es_G(X,Y,Z)$ associé à le formule $(G)\ H \Rightarrow\ C(?(Y),?(Z))$ est défini dans Q, et si Q' est une forme tronquée de Q définissant $es'_G(X,Y)$, on a :

- $p(X,Y,Z)$ vrai dans $M_{P\cup Q_G} \Rightarrow\ p'(X,Y)$ vrai dans $M_{P\cup Q'}$;
- si de plus :

1) Q termine par rapport à (G),
2) $es'_G(X,Y)$ est fonctionnel par rapport à Y,
alors $H \wedge es'_G(X,Y)$ vrai dans $M_{P\cup Q'} \Rightarrow\ \exists\, Z\ es_G(X,Y,Z)$ vrai dans $M_{P\cup Q}$.

Preuve

• On peut montrer par une induction par point fxe sur p que $p(X,Y,Z) \Rightarrow\ p'(X,Y)$ est vrai dans $M_{P\cup Q\cup Q'}$. En effet, à chaque clause pour p correspond une clause pour p' qui permet le succès de l'induction.

• On suppose $H(X) \wedge\ \ es'_G(X,Y)$. Comme Q termine, il existe Y' et Z tels qu'on ait $es_G(X,Y',Z)$. D'après la première partie du lemme, on a alors $es'_G(X,Y')$ Or es'_G est fonctionnel, donc $Y' = Y$, et on a bien $es_G(X,Y,Z)$. □

Propriété 2.2.3 (Fribourg) Si Q est partiellement correct et termine par rapport a $(G)(H_X) \Rightarrow C(X,?(Y),?(Z))$ pour le prédicat $es_G(X,Y,Z)$, si la forme tronquée $es'_G(X,Y)$ est définie dans Q', et si $es'_G(X,Y)$ est fonctionnel par ropport à Y, alors Q' est partiellement corret et temire par rapport à (G).

Preuve Soit Q partiellement correct et terminant par rapport à (G), et $es'_G(X,Y)$ fonctionnel par rapport à Y.

• Q' termine par rapport à (G). On suppose $H(X)$ vrai, et on cherche Y tel que $es'_G(X,Y)$. Comme Q termine, on a : $\exists\, Y, Z\ es_G(X,Y,Z)$; d'après le premier point du lemme 2.2.2, on a alors $es'_G(X,Y)$.

• Q' est partiellement correct par rapport à (G). On suppose $es'_G(X,Y) \wedge H(X)$, et on veut prouver $C(X,Y,Z)$ dans $M_{P\cup Q'}$. D'après le deuxième point du lemme 2.2.2, il existe Z tel que $es_G(X,Y,Z)$; comme Q est partiellement correct par rapport à (G) et qu'on a H, on a alors $C(X,Y,Z)$. □

Dans l'exemple de la longueur de deux listes, le programme après troncature est partiellement correct et termine car :

1) es_G termine : $\forall\ X,Y\ \exists\ Z,T\ es_G(X,Y,Z,T)$,
2) $es'_G(X,Y,T)$ est fonctionnel par rapport à T.

Chapitre 3

Transformation de programmes logiques

La transformation de programmes consiste à remplacer, généralement par étapes, un programme par un autre programme qui lui est équivalent ou qui le spécialise, c'est-à-dire qui permet de calculer tout ou partie de ce qui était calculable. On espère améliorer ainsi le premier programme, même si cette notion n'est pas très précise : on vérifie au cas par cas que l'on obtient un programme "plus efficace".

Le cadre dans lequel on se place est la logique du premier ordre ; on s'intéresse d'abord à des programmes logiques, on expose la méthode de transformation par pliages et dépliages et des cas particuliers de transformations détermisistes ; puis on présente des systèmes permettant de transformer certaines formules du premier ordre en programmes logiques.

3.1 Transformation par pliage et dépliage

Burstall et Darlington ont étudié dans un cadre fonctionnel [**BD77**], un programme étant un ensemble d'équations récursives, des règles de **pliage** ('folding') et de dépliage ('unfolding'). Leur système ne préserve que la correction partielle. Tamaki et Sato [**TS84**] ont transposé ces règles dans le domaine.de la programmation logique, et ont prouvé qu'eles préservaient la correction totale au sens du plus petit modède de Herbrand : ce résultat a ensuite été étendu [**KK88** ; **Sek89**]. Pettorossi et Proietti [**PP88**] ont présenté des stratégies d'ordonnancement des règles de Tamaki et Sato.

3.1.1 Règles de pliage/dépliage

Les règles présentées sont la défnition d'une nouvelle clause, le dépliage d'une clause à un atome de son corps, le pliage d'une clause par une autre, et le remplacement d'une partie du corps d'une clause [**TS84**].

Exemple : longueur de deux listes

On introduit les règles par un exemple. A partir des prédicats *app* de concaténation de deux listes et *leng* qui donne la longueur d'une liste :

$$(1)\ app([\,], Y, Y).$$
$$(2)\ app([A|X], Y, [A|Z])\ \text{:-}\ app(X, Y, Z).$$
$$(3)\ leng([\,], 0).$$
$$(4)\ leng([A|X], s(N))\ \text{:-}\ leng(X, N).$$

on d**définit** un nouveav prédicat *appleng* qui calcule la longueur de la concaténation de deux listes, et on cherche une définition plus directe et plus efficace de ce prédicat :

$$(5)\ appleng(X, Y, Z) : - app(X, Y, T), leng(T, Z).$$

Dans (5) *app* génère la liste T, soit X concaténé à Y, et *leng* s'en sert pour calculer l'entier Z. La liste T est donc construite de façon intermédiaire au cours du calcul et n'apparait pas dans le résultat final : il parait intéressant de l'éliminer.

La règle de **dépliage** consiste à évaluer symboliquement un prédicat dans le corps d'une clause. On peut ici déplier le premier prédicat du corps de la clause (5) avec les clauses (1) et (2) définissant *app*, et obtenir ainsi deux clauses qui remplaceront la clause (5) :

$$(6)\ appleng([\,], Y, Z)\ \text{:-}\ leng(Y, Z).$$
$$(7)\ appleng([A|X], YZ)\ \text{:-}\ app(X, Y, T), leng([A|T], Z).$$

On peut de nouveau déplier la clause (7), en la remplaçant par (8) :

$$(8)\ appleng([A|X], Y, s(Z))\ \text{:-}\ app(X, Y, T), leng(T, Z).$$

Le corps de la clause (8) est alors le même que celui de la clause (5) de départ. On le remplace par la tête de la clause (5), introduisant ainsi un appel récursif c'est la règle de **pliage** :

$$(9)\ appleng([A|X], Y, s(Z))\ \text{:-}\ appleng(X, Y, Z)$$

Le prédicat *appleng* défini par la clause (5) peut donc être calculé récursivement par les clauses (6) et (9), Les prédicats *app* et *leng* de la spécification initiale étaient chacun défini récursivement ; par cette transformation on a amené la récursion au niveau supérieur, celui de *appleng*.

Formalisation

De façon plus formelle, on considère qu'on définit un nouveau prédicat à partir de prédicats connus (soit parce que ce nouveau prédicat a un intérêt en soi, soit pour représenter une partie du corps d'une clause pour laquelle on cherche une définition plus efficace), et on transforme cette définition. Les règles de transformation sont la définition, le dépliage et le pliage. Au cours de la transformation on note P le programme et D l'**ensemble des définitions** ; au début P est le programme initial et D est vide.

Définition On définit une clause (C) :

$$(C)\ p(X_1, \ldots, X_m) \text{ :- } A_1, \ldots, A_n.$$

où p est un nouveau nom de prédicat, les A_j des atomes dont les prédicats appartiennent au programme initial et $X_1, \ldots, X_m$ (deux à deux distincts) un sous-ensemble des variables des A_j, en ajoutant (C) à D et à P.

Dépliage Etant données deux clauses (C) et (C_1) du programme P :

$$(C)\ A \text{ :- } A_1, \ldots, A_n.$$
$$(C_1)\ B \text{ :- } B_1, \ldots, B_m.$$

on déplie (C) avec (C_1), à un prédicat apparaissant dans le programme initial. par exemple A_1, en unifiant A_1 et la tête de (C_1) par $A_1\theta = B\theta$, et en construisant (C'_1) ce qui correspond à la règle de résolution :

$$(C'_1)\ (A \text{ :- } B_1, \ldots, B_m, A_2, \ldots, An)\theta.$$

On déplie (C) avec toutes les clauses $(C_1), \ldots, (C_p)$ dont la tête s'unifie avec A_1, et on obtient les clauses $(C'_1), \ldots, (C'_p)$. On remplace alors (C) dans P par $(C'_1), \ldots, (C'_p)$ et on laisse D inchangé.

Pliage Etant données une clause (C) du programme P et une clause (C') de D (une défnition précédemment introduite) :

$$(C)\ A \text{ :- } A_1, \ldots, A_n.$$
$$(C')\ B \text{ :- } B_1, \ldots, B_k.$$

on plie (C) à une partie, par exemple $A_1, \ldots, A_k$, de son corps dont le corps de (C') est une instance : $(B_1, \ldots, B_k)\theta = A_1, \ldots, A_k$, Ce qui donne la clause (C'') :

$$(C'')\ A \text{ :- } B\theta, A_{k+1}, \ldots, A_n.$$

On remplace (C') par (C'') dans P et on laisse D inchangé.

Pour que le pliage soit correct, c'est à dire pour que le programme à un instant donné représente bien les mêmes prédicats que ceux qu'on a définis, il faut qu'un dépliage de (C'') par (C') donne un renommage de (C) [**PP88**; **PP90b**]. Formellement, cela signife qu'il faut vérifier les deux conditions suivantes [**TS84**] :

• il faut que $k < n$ ou que (C) soit *pliable*, pour éviter de plier une clause avec elle-même. Une clause de définition est *non pliable*; les clauses résultant d'un dépliage sont *pliables*; après un pliage, (C'') est *pliable* si (C) l'était.

• θ doit remplacer les variables internes [1] de (C') par des variables, distinctes, de (C), lesquelles ne doivent apparaitre ni dans le prédicat de tête A, ni dans les $A_{k+1}, \ldots, A_n$.

3.1.2 Correction, lemmes

Correction

Une transformation par étapes consiste à appliquer à un programme initial P une suite de défnision(s), dépliage(s), pliage(s). Au cours de la transformation on introduit un ensemble de définitions D; le programme final $Tr(P)$ calculera donc plus de choses que P. Pour définir la notion de transformation correcte, on considère alors que toutes les définitions ont été données au départ et qu'on a une transformation :

$$(P \cup D) \rightarrow Tr(p).$$

On cherche à formaliser le fait que $Tr(P)$ "calcule la même chose" que $P \cup D$. Pour cela, on introduit une fonction sémantique Sem, et on dit que la transformation préserve :

- la **correction partielle** si $Sem(Tr(P)) \subset Sem(P \cup D)$.
- la **correction totale** si $Sem(Tr(P)) = Sem(P \cup D)$.

Si l'on s'intéresse à des programmes Prolog plutôt qu'à des clauses définies, on peut prendre pour Sem la sémantique opérationnelle de Prolog. Sinon, on peut choisir pour Sem (P) le plus petit modèle de Herbrand M_p de P, qui coïncide avec l'ensemble de succès [**Llo87**]. Les règles de définition, dépliage et pliage préservent la correction totale par rapport à M_p :

Propriété 3.1.1 (Tamaki-Sato) Une transformation utilisant les règles de définition, dépliage et pliage, et introduisant un ensemble de défnitions D, donne un programme P tel que $M_{Tr_{(P)}} = M_{P \cup D}$.

$Sem(P)$ peut anssi présenter le multi-ensemble de succès SM_P de P :

1. les variables internes d'une clause sont celles qui apparaissent dans son corps mais pas dans son prédicat de tête

Propriété 3.1.2 (Kanamori- Kawamura) Une transformation utilisant les règles de définition, dépliage et pliage, et introduisant un ensemble de définitions D, donne un programme P tel que $SM_{Tr_{(P)}} = SM_{P \cup D}$.

On peut aussi s'intéresser à l'information négative en prenant pour $Sem(P)$ l'ensemble d'échec fini F_P de P. Si l'on restreint les pliages aux atomes qui ont déjà été dépliés au moins une fois [**Sek89**], le pliage/dépliage préserve également F_P :

Propriété 3.1.3 (Seki) Une transformation utilisant les règles de définition, dépliage et pliage restreint aux atomes déjà dépliés, et introduisant un ensemble de définitions D, donne un programme P tel que $F_{Tr(P)} = F_{P \cup D}$.

Dans l'exemple de *appleng*, le programme final est équivalent au programme initial en ce qu'ils calculent les mêmes réponses (propriété 3.1.1), le mēme nombre de fois (propriété 3.1.2), et en ce qu'ils s'arrêtent sur un échec de la même façon (propriété 3.1.3).

Règle de remplacement

Les règles de pliage/dépliage seules ne suffisent pas dans de nombreux cas ; elles ne permettent pas par exemple d'utiliser l'associativité d'un prédicat, d'ajouter des conséquences d'une clause, d'éliminer des prédicats inutiles. Étant donnée une clause (C) et un lemme (L) relatif à une partie du corps de (C) et vrai dans $M_{p \cup D}$:

$$(C)\ A \text{ :- } A_1, \ldots, A_k, \ldots, A_n.$$
$$(L)\ (A_1 \wedge \ldots \wedge A_k) \Leftrightarrow\ (B_1 \wedge \ldots \wedge B_m)$$

la règle de remplacement [**TS84**] consiste à utiliser le lemme pour obtenir (C') :

$$(C')\ A \text{ :- } B_1, \ldots, B_m, A_{k+1}, \ldots, A_n.$$

Dans ce cas, seule la correction partielle est préservée et il faut vérifier à la fin de la transformation qu'aucun cas de non-terminaison n'a été introduit [**TS84**].

Propriété 3.1.4 (Tamaki-Sato) Lors d'une transformation utilisant les règles de définition, dépliage, pliage et remplacement, on a $M_{Tr_{(P)}} = M_{P \cup D}$.

Si l'on restreint l'utilisation de la règle de remplacement, elle préserve la correction totale [**TS84**] ; mais les restrictions données sont très spécifiques et techniques. Par exemple, on peut utiliser la propriété d'associativité du prédicat d'addition :

$$add(0, X, X).$$
$$add(s(X), Y, s(Z)) \text{ :- } add(X, Y, Z).$$

pour remplacer $add(X, Y, U) \wedge add(U, Z, T)$ par $add(Y, Z, V) \wedge add(X, V, T)$, et les conditions sont satisfaites, mais le remplacement inverse est beaucoup plus délicat à justifier.

Problèmes

On considère que l'on a transformé un programme avec succès lorsque l'on a effectué une suite d'étapes aboutissant à un pliage. Cela peut correspondre à la fusion des définitions récursives de deux prédicats différents, lorsque ces prédicats sont connectés et que les récursions sont compatibles *(appleng)*. Cela peut consister à améliorer l'efficacité d'un programme. Ces cas de **réussite d'une transformation** dépendent de la structure du programme initial ; il est intéressant de chercher à les caractériser.

On dispose de règles dont l'utilisation confronte à de multiples choix. Quelle définition faut-il introduire : tout ou partie du corps d'une clause, en généralisant ou non les arguments instanciés ? Quels dépliages doit-on effectuer, et dans quel ordre ? A quel moment applique-t-on une propriété comme l'associativité ? Quand vaut-il mieux plier : dès que possible, après un remplacement ? La réponse à ces questions passe par la recherche de **stratégies de transformation** et de classes de programmes pour lesquelles un algorithme de transformation peut être défini.

3.1.3 Réussite, stratégies

Pettorossi et Proietti [**PP85** ; **PP90b**] ont travaillé sur les règles de pliage/dépliage de Tamaki et Sato. Ils ont apporté des contributions aux problèmes de réussite des transformations et de non-déterminisme.

Succès des transformations

Pettorossi et Proietti ont donné, pour des prédicats particuliers, des conditions sous lesquelles un dépliage sera possible et une indication de l'ordre des dépliages pour y arriver. On se restreint à la classe des programmes récursifs linéaires, c'est-à-dire qu'on exclut la récursion multiple, et on exclut de plus la récursion mutuelle. On considère une clause de la forme :

$$h(\ldots) \text{ :- } p(\ldots, X, \ldots),\ q(\ldots, X, \ldots).$$

où p et q sont définis par deux programmes logiques disjoints. De plus, les définitions de p et q doivent être non croissantes par rapport à leur argument X, c'est-à-dire qu'on doit pouvoir comparer les valeurs de cet argument de part et d'autre d'une clause récursive par un ordre $\prec$ qui peut être par exemple l'ordre sous-terme strict, ou l'inclusion stricte de l'ensemble des variables d'un terme ; ou alors que ces deux valeurs doivent être identiques. Par exemple $app(X, Y, Z)$ est non croissant par rapport à X et Y car dans sa clause récursive :

$$(2)\ app([A|X], Y, [A|Z]) \text{ :- } app(X, Y, Z).$$

on a $X \prec [A|X]$ (pour l'un des deux ordres au choix), et $Y = Y$

.

Le fait de réussir à plier signifie que les définitions récursives sur X de p et q sont "compatibles", et qu'effectuer des dépliages concernant un prédicat puis l'autre permet d'obtenir une clause pliable. En formalisant cette idée on peut définir **une règle de sélection des dépliages** qui guide le cours de la transformation. Si l'on peut comparer par $\prec$ l'argument de p et celui de q qui sont liés, on sélectionne pour le dépliage celui qui a l'argument le plus grand. Lorsque l'on débute la transformation, c'est- à-dire lorsque p et q ont en commun un argument non instancié (ici la variable X), le premier dépliage n'est pas spécifé car on n'a pas $X \prec X$. Ainsi, pour $appleng$, le choix de app pour le premier dépliage est arbitraire vis-à-vis de la règle de sélection. On obtient après ce dépliage la dause (7) :

$$(7)\ appleng([A|X], Y, Z) :- app(X, Y, U), leng([A|U], Z).$$

On pourrait vouloir déplier l'un ou l'autre des prédicats. Or on a $U \prec [A/U]$, donc on sélectionne *leng* pour le dépliage. On se trouve alors dans une situation où un pliage est possible et on s'empresse d'en profiter.

Cette règle de sélection est **complète** pour la classe de programmes considéres : s'il existe un moyen de terminer la transformation par un pliage, on y parviendra.

Stratégies de transformation

D'une façon générale, Pettorossi et Proietti voient le processus de transformation de la manière suivante :

1. la définition initiale est un cas de composition ou de 'tupling',
2. on effectue des dépliages en cherchant à plier dès que possible,
3. si l'on ne parvient pas à plier, deux heuristiques peuvent permettre de terminer quand même la transformation :

- lorsque la transformation boucle, on pose une nouvelle définition,
- lorsqu'il y a 'mismatching', on effectue une généralisation.

La stratégie de **composition** consiste à éliminer une variable interne inutile, et peut en cela éviter la construction de structures de données intermédiaires. La stratégie de 'tupling' consiste à regrouper plusieurs prédicats traversant la même structure de données.

Dans le cas où une transformation ne termine pas parce que l'on génère une clause contenant un groupe de prédicats déjà apparu, la stratégie de **bouclage** s'applique. Lorsque l'on n'arrive pas à effectuer un pliage parce que, bien que les bons prédicats aient été retrouvés, il y a incompatibilité au niveau d'un argument, on peut effectuer une **généralisation**.

Composition *La définition de applen en est un exemple :*

$$(5)\ appleng(X, Y, Z) \text{ :- } app(X, Y, T), leng(T, Z)$$

Les deux prédicats de son corps sont liés par une seule variable T, interne [2]. La liste T est construite de façon intermédiaire au cours du résultat final Z. On l'élimine par pliage/dépliage :

$$(6)\ appleng([\,], Y, Z) \text{ :- } leng(Y, Z).$$
$$(9)\ appleng([A|X], Y, s(Z)) \text{ :- } appleng(X, Y, Z).$$

La spécification $appleng$ parcourt X, puis T soit X concaténé à Y. Le programme obtenu par pliage/dépliage parcourt X, puis seulement Y : les deux parcours redondants de X ont été fusionnés. On a bien amélioré l'efficacité en évitant la construction de la liste intermédiaire T.

Tupling Considérons l'exemple de la suite de Fibonacci :

$$(f1)\ fib(0, s(0)).$$
$$(f2)\ fib(s(0), s(0)).$$
$$(f3)\ fig(s(s(X)), Y) \text{ :- } fib(s(X), Z), fib(X, T), add(Z, T, Y).$$

La clause récursive parcourt l'entier X, une fois pour construire Z et une fois pour construire T. On a deux prédicats liés par une variable X qui n'est pas interne (c'est ici une variable d'entrée du programme). On se trouve dans un cas de tupling [3] et on effectue la définition suivante :

$$(f4)\ p(X, Z, T) \text{ :- } fib(s(X), Z), fib(X, T).$$

qui permet de plier (f3) :

$$(f5)\ fib(s(s(X)), Y) \text{ :- } p(X, Z, T), add(Z, T, Y).$$

En dépliant cette (f4) à fib(s(X), Z), on obtient :

$$p(0, s(0), X) \text{ :- } fib(0, X).$$
$$p(s(X), Z, T) \text{ :- } fib(s(X), U), fib(X, V), add(U, V, Z), fib(s(X), T).$$

2. cela correspond en fonctionnel à un cas de composition de fonctions : leng(app(x, y)), et la transformation consiste à obtenir une définition récursive directe de appleng = leng o app

3. en fonctionnel , la stratégie de 'tupling' consiste à créer une paire (f(x).g(x)) où f et g sont deux fonctions d'argument x ; par exemple, pour fib :(fib(s(x)),fib(x)).

La première clause se déplie en :

$$(f6) \quad p(0, s(0), s(0)).$$

En utilisant le fait que $fib(Y, Y)$ **est fonctionnel par rapport** à Y, c'est-à-dire que $fib(s(X), U)$ et $fib(s(X), T)$ impliquent $T = U$, on remplace la deuxième clause par :

$$p(s(X), Z, T) \text{ :- } fib(X, V), add(T, V, Z), fib(s(X), T).$$

qui peut se plier :

$$(f7)\ p(s(X), Z, T) \text{ :- } p(X, T, V), add(T, V, Z).$$

Le nouveau programme pour *fib*, composé des clauses (f1), (f2), (f5), (f6) et (f7), parcourt X une fois de moins : les deux parcours initiaux ont été fusionnés [4] . On a bien amélioré l'efficacité.

Boucles Le prédicat *souss* définit la notion de sous-suite :

$$souss([\,], X).$$
$$souss([A|X], [A|Y]) \text{ :- } souss(X, Y).$$
$$souss(X, [A|Y]) \text{ :- } souss(X, Y).$$

Le prédicat *csouss* l'utilise pour calculer les sous-suites conmunes à deux suites :

$$csouss(X, Y, Z) \text{ :- } souss(X, Y), souss(X, Z).$$

Le premier dépliage donne trois clauses :

$$(s1)\ csouss([\,], X, Y) \text{ :- } souss([\,], Y).$$
$$(s2)\ csouss([A|X], [A|Y], Z) \text{ :- } souss(X, Y), souss([A|X], Z).$$
$$(s3)\ csouss(X, [A|Y], Z) \text{ :- } souss(X, Y), souss(X, Z).$$

La clause $(s2)$ ne peut pas être pliée ; on effectue donc un nouveau dépliage. Le prédicat $souss(X, Y)$ est non-croissant par rapport à X, sa définition est récursive linéaire et $X \prec [A|X]$ donc on sélectionne $souss(X, [A|Y])$:

$$(s4)\ csouss([A|X], [A|Y], [A|Z]) \text{ :- } souss(X, Y), souss(X, Z).$$
$$(s5)\ csouss([A|X], [A|Y], [B|Z]) \text{ :- } sous(X, Y), souss([A|X], Z).$$

4. l'optimisation est similaire pour la stratégie de composition ; cela explique que les cas de composition et de 'tupling' soient regroupés sous l'appellation *fusion de boucles* dans [**Deb88**].

Le corps de la clause $(s5)$ est le même que celui de la clause $(s2)$. On ne peut toujours pas plier, et le processus de transformation est entré dans une boucle. La solution consiste à introduire une nouvelle définition avant le dépliage de $(s2)$, qui factorisera le corps commun :

$$csouss1(A, X, Y, Z) \text{ :- } souss(X, Y), souss([A|X], Z).$$
$$csouss1([A|X], [A|Y], Z) \text{ :- } csouss1(A, X, Y, Z).$$

La définition de *csouss1* se déplie en :

$$csouss1(A, X, Y, [A|Z]) \text{ :- } souss(X, Y), souss(X, Z).$$
$$csouss1(A, X, Y, [B|Z]) \text{ :- } sous(X, Y), souss([A|X], Z).$$

Ces deux clauses sont alors pliables,respectivement par la première et par la deuxième défnition :

$$csouss1(A, X, Y, [A|Z]) \text{ :- } csouss(X, Y, Z).$$
$$csouss1(A, X, Y, [B|Z]) \text{ :- } csouss1(A, X, Y, Z).$$

et la transformation a pu cette fois terminer.

Dans [**PP90a** ; **PP90L**], Pettorossi et Proietti étudient plus en détail l'introduction de nouvelles définitions au cours d'une transformation.

Généralisation Le prédicat naïf d'inversion d'une liste :

$$(r1)\ rev([\,], [\,]).$$
$$(r2)\ rev([A|X], Y) \text{ :- } rev(X, Z), app(Z, [A], Y).$$

est candidat à la stratégie de composition, car la variable Z semble n'intervenir que de façon intermédiaire. On peut donc poser la définition suivante :

$$rev1(X, A, Y) \text{ :- } rev(X, Z), app(Z, [A], Y).$$

Un premier dépliage donne les deux clauses :

$$rev1([\,], A, Y) \text{ :- } app([\,], [A], Y).$$
$$rev1([B|X], A, Y) \text{ :- } rev(X, T), app(T, [B], Z), app(Z, [A], Y).$$

En utilisant l'associativité de *app*, la deuxième clause se transforme en :

$$rev1([B|X], A, Y) \text{ :- } rev(X, T), app(T, [B, A], Y).$$

Cette nouvelle clause a la même forme que la définition de $rev1$, mais on ne peut pas effectuer un pliage car $[B, A]$ n'a pas la même structure que $[A]$: on dit qu'il y a 'mismatching'. On recommence alors la transformation en posant une définition généralisant l'argument correspondant en une variable :

$$rev(X, C, Y) \text{ :- } rev(X, Z), app(Z, C, Y).$$

cette nouvelle définition permet de plier $(r2)$ en :

$$(r3)\ rev([A|X], Y) \text{ :- } rev(X, [A], Y).$$

et se déplie en :

$$rev([\,], C, Y) \text{ :- } app([\,], C, Y).$$
$$rev([B|X], C, Y) \text{ :- } rev(X, T), app(T, [B], Z), app(Z, C, Y).$$

dont la deuxième clause se transforme, en utilisant l'associativité de app, en :

$$rev([B|X], C, Y) :- rev(X, T), app(T, [B|C], Y).$$

Cette fois-ci on peut effectuer un pliage, et obtenir :

$$rev([B|X], C, Y) :- rev(X, [B|C], Y).$$

On a maintenant un prédicat qui inverse une liste grâce à un accumulateur (le premier argument de rev), et dont la définition complète est formée par (r1), (r3) et :

$$rev([\,], C, C).$$
$$rev([B|X], C, Y) :- rev(X, [B|C], Y).$$

Le rev naïf est quadratique en la taille de son premier argument : si X contient n éléments, $rev(X, Y)$ demande $(n + 1)(n + 2)/2$ étapes de calcul, alors que le rev avec accumulateur est linéaire : il parcourt une fois seulement la liste X. On a nettement amélioré l'efficacité.

On est arrivé à un programme **récursif-terminal** pour rev. On parle de prédicat récursif-terminal uniquement en fonction d'un mode d'utilisation donné, pour formaliser la notion d'"appel récursif en fin de calcul". Un prédicat $p(X, Y)$, d'arguments d'entrée X et de sortie Y, est dit récursif terminal si sa clause récursive est de la forme :

p(X,Y) :- calc(X,Z), p(Z,Y) [5]

où *calc* est une conjonction de prédicats qui calcule Z en fonction de X et représente le calcul fait avant l'appel récursif. Un tel prédicat p peut assez facilement être implémenté comme un calcul itératif, dégageant de l'espace mémoire (pile), d'où un gain d'efficacité ; c'est pourquoi on parle aussi de prédicat **quasi-itératif** [**Azi87**].

5. cela correspond, en notation fonctionnelle, à un appel récursif de la forme f(x) :- f(g(x)) et non f(x) :- h(f(x)). Dans [**SS86** ; **chapitre 8**], la définition est plus restrictive : l'appel récursif doit être seul dans le corps (pas de calc), ou bien calc doit être composé uniquement de prédicats-système Prolog.

Pour traiter cet exemple, les règles de pliage/dépliage et l'associativité de app suffisent. Néanmoins le non-déterminisme apparaît clairement : on ne sait pas quelle définition faire ; on aurait pu plier avant d'utiliser l'associativité. Le problème du choix de la définition initiale est particulièrement intéressant : on pourrait chercher des méthodes qui font des généralisations dès le début. D'autre part, un appel plus direct du rev avec accumulateur serait le suivant (à la place des clauses (r1) et (r3)) :

$$rev(X, Y) \text{ :- } rev(X, [\,], Y).$$

Cette clause s'appelle clause **de mise-à-jour** ('updating'), et ne peut pas s'obtenir avec les règles de pliage/dépliage.

3.2 Restrictions déterministes

Pour des classes particulières de programmes logiques, il est possible d'obtenir de véritables **algorithmes** de transformation qui fonctionnent de façon déterministe. On peut appliquer du pliage/dépliage contrôlé, auquel on ajoute d'autres règles, ou transformer un programme en fonction de sa structure. Les exemples suivants utilisent essentiellement le fait qu'un prédicat soit associatif.

3.2.1 Listes de différence

Zhang et Grant [**ZG88**] ont adapté les règles de Tamaki et Sato au cas particulier où des **listes de différence** ('d-lists') peuvent être introduites.

Une liste de différence est une paire $L\backslash E$ où E est la fin de la liste L, $L\backslash E$ représente la liste L amputée de E. Par exemple, les listes de différence : $[a, b]\backslash[\,]$, $[a, b, c]\backslash[c]$ et $[a, b|E]\backslash E$ correspondent toutes trois à la liste $[a, b]$. Avec cette structure, la concaténation de listes se fait en temps constant par : $X\backslash Y$ concaténé à $Y\backslash Z$ donne $X\backslash Z$. On n'utilise plus le prédicat app qui lui est linéaire en la longueur de la première liste, d'où un gain d'efficacité.

Zhang et Grant présentent un algorithme qui spécifie quelles définitions introduire et dans quel ordre utiliser le pliage, le dépliage et l'associativité de app. Cet algorithme termine et préserve l'équivalence des programmes ; le gain d'efficacité n'est pas évalué. Lorsque dans le corps d'une clause apparaît un couple de la forme :

$$p(X_1, \ldots, X_n, Z), app(Z, X, Y).$$

la définition suivante est introduite :

$$p_1(X1, \ldots, X_n, Y, X) \text{ :- } p(X_1, \ldots, X_n, Z), app(Z, X, Y).$$

Dans l'exemple de rev :

$$rev([\,],[\,]).$$
$$rev([A|X],Y) \text{ :- } rev(X,Z), app(Z,[A],Y).$$

la clause récursive se prête à une telle définition :

$$rev(X,Y,C) \text{ :- } rev(X,Z), app(Z,C,Y).$$

Le terme $[A]$ est automatiquement généralisé à C, parce que toutes les définitions sont posées sous la forme la plus générale, avec des variables pour tous les arguments.

Le premier dépliage se fait à $p(X_1,\ldots,X_n,Z)$, soit ici à $rev(X,Z)$; on obtient :

$$(1)\ rev([\,],Y.C) \text{ :- } app([\,],C,Y).$$
$$(2)\ rev([A|X],Y,C) \text{ :- } rev(X,T), app(T,[A],Z), app(Z,C,Y).$$

L'étape suivante consiste à **évaluer partiellement** le prédicat app quand c'est possible, et éventuellement à utiliser son **associativité.** L'évaluation partielle est un cas particulier de dépliage sans spécialisation, c'est-à-dire employant uniquement du filtrage et non de l'unification **[LS87]**. Ici $app([],C,Y)$ est déjà instancié, donc un dépliage et une évaluation partielle font exactement la même chose. La clause (1) devient :

$$rev([\,],Y,Y).$$

L'associativité de app est formalisée de la façon suivante : lorsque l'on a dans le corps d'une clause deux prédicats $app(X,Y,U_1)$ et $app(U_2,Z,T)$ tels que U_2 soit la ***fin***[6] de U_1, on les remplace par $app(Y,Z,V)$, $app(X,V,T_1)$, où T_1 est la concaténation du ***début***[7] de U_1 et de T. Dans le cas (le plus courant) où $U_1 = U_2$, on obtient la définition classique de l'associativité de *app*, qui consiste à remplacer $app(X,Y,U), app(U,Z,T)$ par $app(Y,Z,V)$, $app(X,V,T)$. On peut ici utiliser l'associativité pour la clause (2) :

rev([A|X],Y,C) :- rev(X,T), app([A],C,V),app(T,V,Y).

Puis on évalue partiellement *app*, pour obtenir d'abord :

$$rev([A|X],Y,C) \text{ :- } rev(X,T), app([\,],C,W), app(T,[A|W],Y).$$

et ensuite :

$$(3)\ rev([A|X],Y,C) \text{ :- } rev(X,T), app(T,[A|C],Y).$$

6. par exemple, *fin*(X) = *fin*([C|X]) = *fin*(a,b|X])=X.
7. *deb*([a,b|X]) = [a,b]), *deb*([C|X]) = [C], *deb*(X) = [ ]

La dernière étape consiste à plier, si c'est possible, avec la définition introduite, et dans ce cas à introduire une clause de **mise-à-jour :**

$$P(X_1, \ldots, X_n, Z) :- p_1(X_1, \ldots, X_n, Y, [\,]).$$

on peut ici plier la clause(3) :

$$rev([A|X], Y, C) \text{ :- } rev(X, Y, [A|C]).$$
$$rev(X, Y) \text{ :- } rev(X, Y, [\,]).$$

Par rapport à la méthode pliage/dépliage, les problèmes dégagés par l'exemple de *rev* sont résolus : on a ici un algorithme qui spécifie exactement l'ordre et la forme des règles ; de plus, la définition de départ est sous une forme générale, évitant l'apparition de 'mismatching', et on introduit une clause de mise-à-jour à la fin. Cependant on se limite aux seuls exemples dans lesquels le prédicat *app*, travaillant sur des listes, Intervient.

3.2.2 Associativité

Brough et Hogger [**BH87**] s'intéressent à des cas voisins de ceux que traitent Zhang et Grant : ils transforment des programmes en utilisant **l'associativité** d'un prédicat. Les transformations se font par des **dérivations schématiques** de programmes.

Ils partent de clauses dont la tête est mise sous forme générale : elle ne contient que des variables et les unifications sont spécifiées explicitement dans le corps ; en fonction de la forme schématique de ce programme initial, ils donnent directement le résultat de la transformation. Par exemple, dans le cas du prédicat rev, le programme est d'abord mis sous la forme :

$$rev(X, Y) \text{ :- } X = [\,], Y = [\,].$$
$$rev(X, Y) \text{ :- } X = [A|X_1], Z = [A], rev(X_1, Y_1), app(Y_1, Z, Y).$$

Ce programme est un cas particulier d'un schéma donné dans [**BH87**] ; la méthode consiste alors à introduire un nouveau prédicat *rev1* comme suit :

$$(1)\ rev(X, Y) \text{ :- } rev_1(Y, [\,], X).$$
$$(2)\ rev_1(T, U, X) \text{ :- } X = [\,], Y = [\,], app(Y, U, T).$$
$$(3)\ rev_1(T, U, X) \text{ :- } X = [A|X_1], Z = [A], app(Z, U, X_2), rev_1(T, X_2, X_1).$$

Pour écrire la clause (1), on utilise l'existence d'un élément neutre à droite pour la concatenation : $app(X, [\,], X)$. Le prédicat associatif, ici **app**, est ensuite évalué partiellement chaque fois que c'est possible, et en instanciant les têtes de clauses on obtient :

$$rev_1(Y, Y, [\,]).$$
$$rev_1(T, U, [A|X]) \text{ :- } rev_1(T, [A|U], X).$$

La généralisation de $[A]$ à une variable se fait automatiquement du fait de l'écriture du programme sous forme de schéma. La clause (1) est la clause de **mise-à-jour**. Cette méthode permet de transformer des programmes récursifs (linéaires) en programmes **récursifs terminaux**, donc d'améliorer leur efficacité.

Chaque fois que l'on utilise l'associativité d'un prédicat, on introduit des **structures de différence** [8] [**SS86** ; **chapitre 15**] :

- pour l'associativité de app, une liste est représentée par la "différence" de deux listes : $[a, b] = [a, b|L]\backslash L$,
- pour l'associativité de l'addition, un entier est considéré comme la différence (soustraction) de deux entiers : $s(s(0)) = s(s(N)) - N$,
- pour l'associativité de la multiplication, un entier est représenté par le quotient de deux entiers : s(s(0)) = ((s(0)) x N)/N.

Si l'on modifie un programme pour qu'il manipule des structures de différence, la nouvelle variable introduite (L ou N pour les structures précédentes) peut être utilisée comme un accumulateur, ce qui peut conduire à un programme plus efficace.

3.2.3 Récursion presque-terminale

Debray [**Deb84** ; **Deb88**] et Azibi [**Azi87**] s'intéressent à des stratégies de transformation de programmes **presque-récursifs-terminaux** en programmes récursifs terminaux, en utilisant les règles de pliage/dépliage. Ils se restreignent à des programmes Prolog récursifs linéaires. Un programme presque-récursif-terminal a comme clause récursive :

$$p \text{ :- } q_1, \ldots, q_n,\ p,\ eval(t_1\ o\ t_2,\ X).\ (n \geq 0)$$

où *eval* représente une conjonction de prédicats évaluables, appliquée aux expressions t_1 et t_2 et dont le résultat est unifié avec X. Ces prédicats sont supposés **associatifs et distributifs** les uns par rapport aux autres s'il y en a au moins deux. La clause récursive du *rev naïf* est presque-récursive-terminale :

$$rev([A|X], Y) \text{ :- } rev(X, Z), app(Z, [A], Y).$$

si l'on considère app comme un prédicat évaluable appliqué à Z et à $[A]$.

8. mais on peut introduire de telles structures sans utiliser d'associativité.

La stratégie de transformation consiste à prendre comme définition initiale la fin de la clause presque-récursive-terminale à partir de l'appel récursif, en effectuant une **généralisation**, mais pas de façon systématique comme pour les systèmes précédents. Debray considère les sous-expressions closes maximales de t_1 et t_2, et les remplace par de nouvelles variables ; néanmoins il ne peut généraliser ainsi un argument ne contenant pas de constante, comme par exemple ici $[A]$. Azibi améliore cette stratégie en associant une variable à chaque argument contenant une variable, sauf s'il s'agit d'une variable intermédiaire entre l'appel récursif et $eval$. Pour rev, on associe une variable à X, Y et à $[A]$, généralisant ainsi le deuxième argument de app :

$$rev(X, Y, C) :- rev(X, Z), app(Z, C, Y).$$

Il ne s'agit donc pas ici d'une généralisation aveugle comme dans [**ZG88** ; **BH87**] ; il ne s'agit pas non plus d'une généralisation motivée par un échec, comme pour [**PP88**], mais d'une généralisation basée sur l'analyse du programme initial.

On déplie la nouvelle définition avec la définition initiale, on applique l'associativité et éventuellement la distributivité, puis on plie ; enfin on plie la clause récursive initiale avec la nouvelle définition. On obtient ainsi le même programme que Pettorossi et Proietti. Il n'y a pas de mise-à-jour.

Des conditions précises d'application de la stratégie sont dégagées, et la correction dérive des résultats de Tamaki et Sato. Debray et Azibi sont les seuls à s'intéresser au cas de plusieurs prédicats en fin de calcul, et à l'utilisation de la distributivité. Les exemples traités concernent surtout le cas des entiers et les prédicats d'addition et de multiplication, bien que ce ne soit pas une limite de la méthode.

3.2.4 Comparaison

Les trois systèmes utilisent des **structures de différence**. Pour Zhang et Grant il s'agit seulement de listes de différence ; les autres ne mentionnent pas explicitement cette utilisation. La méthode de Brough et Hogger ne donne pas comme cas particulier les résultats Zhang et Grant, qui visent d'abord à introduire des listes de différence et n'utilisent pas systématiquement l'associativité.

Ils traitent également tous trois de **prédicats associatifs**. Pour Brough et Hogger c'est la notion centrale, Zhang et Grant transforment avec associativité mais aussi sans, et Azibi et Debray envisagent la distributivité en plus de l'associativité.

Les transformations sont basées sur les règles de pliage et dépliage, sauf pour Brough et Hogger qui considèrent des schémas de programmes.

La **généralisation** de certains arguments à des variables se fait systématiquement dans [**ZG88** ; **BH87**], alors que Debray et Azibi procèdent plus finement. Son application est chaque fois préalable à la transformation, ce qui diffère de la stratégie de Pettorossi et Proietti, où la généralisation est motivée par un échec. Une clause de mise-à-jour est introduite dans la méthode de Zhang et Grant et dans celle de Brough et Hogger, mais ce n'est pas le cas dans les autres systèmes.

3.3 Spécifications en logique du premier ordre

Toutes les méthodes exposées précédemment transforment uniquement des clauses définies ou des programmes Prolog. Les programmes de départ étant déjà exécutables, il s'agit de les rendre plus efficaces. Néanmoins, pour plus de pouvoir expressif, on peut partir de **formules du premier ordre**, que l'on considère comme des spécifications à transformer en programmes. Un des intérêts de la programmation logique est en effet de fournir une représentation uniforme pour les spécifications et les programmes. Les méthodes suivantes transforment des spécifications de la forme :

$$p(X) :- formule(X).$$

où p est un nouveau prédicat défini par cette spécification et formule (X) appartient à une classe de formules du premier ordre. Il s'agit de **clauses généralisées** [9], dont le corps n'est plus une simple conjonction d'atomes.

3.3.1 Transformation directe

Dayantis [**Day87**] s'intéresse au cas particulier de spécifications de la forme :

$$p(Y, Z)(\forall X\ q(X, Y, Z) \Rightarrow\ a(X, Z)).$$

notées $ERR[Y]$, où a est un prédicat quelconque défini par un programme logique, et q une relation $RR[Y, X]$, c'est-à-dire en particulier définie récursivement sur Y indépendamment de X. En fait, cela correspond à un prédicat comme $mem(X, L)$, qui est $RR[L, X]$, car sa définition utilise X uniquement comme un paramètre :

$$mem(X, [X|L]).$$
$$mem(X, [Y|L]) :- mem(X, L).$$

9. de telles clauses sont appelées 'program statements' dans [**Llos7** ; **chapitre 4**], et programmes du premier ordre dans [**Sat90**].

La spécification du prédicat $sousl(L, M)$, qui vérifie qu'une liste est une sous-liste d'une autre, est $ERR[L]$:

$$sousl(L, M) : - (\forall X mem(X, L) \Rightarrow\ mem(X, M)).$$

Dayantis donne un algorithme, utilisant des **schémas de programmes**, qui permet de transformer une telle spécification en un programme logique. La correction totale est préservée et l'algorithme termine.

La définition de mem est d'abord réécrite :

$$mem(X, L) : - L = [Y|L_1], X = Y.$$
$$mem(X, L) : - L = [Y|L_1], mem(X, L_1).$$

On obtient la clause non récursive pour $sousl$ en se servant du fait qu'il n'y a pas de clause définissant $mem(X, [\,])$, c'est-à-dire qu'on utilise, implicitement, le fait que L soit une liste :

$$sousl(L, M) : - L = [\,].$$

La clause récursive est construite grâce aux (deux) clauses définissant $mem(X, [Y|L_1])$:

$$sousl(L, M) : - L[Y|L_1], q(Y, M), sousl(L_1, M).$$

où q est défini par une nouvelle spécification, plus simple que celle pour $sousl$

$$q(Y, M) : - (\forall\ X\ X = Y \Rightarrow mem(X, M)).$$

On dérive ensuite pour q le programme :

$$q(Y, M) : - mem(Y, M).$$

et le programme pour $sousl$ se simplifie en :

$$sousl([\,], M).$$
$$sousl([Y|L], M) : - mem(Y, M), sousl(L, M).$$

On obtient ainsi un ensemble de clauses définies pour $sousl(L, M)$, alors qu'on n'avait au départ qu'une spécification non exécutable.

Cette technique de transformation a l'avantage d'être assez directe, mais, à cause de la définition des prédicats RR, ne s'applique que dans un nombre de cas très restreint.

3.3.2 Pliage/dépliage généralisé

Kanamori et Horiuchi [**KH86**] ont étendu la méthode de pliage/dépliage de Tamaki et Sato à des clauses généralisées, qu'ils appellent "formules définies", de la forme :

$$A :- G_1, \ldots, G_m.$$

où chaque G_i ; est une formule du premier ordre, sans variable existentielle mais éventuellement avec des **variables universelles**, ne contenant que des prédicats définis par un programme logique, et telle que les G_i ; n'aient pas de variable universelle en commun.

Par exemple dans :

$$(1)\ sousl(L, M) :- list(L), (\forall X mem(X, L) \Rightarrow\ mem(X, M)).$$

L et M sont des variables globales, il n'y a pas de variable existentielle, et X est une variable universelle. On mentionne explicitement que L soit de type $list$. Kanamori et Horiuchi définissent les notions de **sous-formule positive et négative** : $mem(X, L)$ est une sous-formule négative de $\forall X\ mem(X, L) \Rightarrow\ mem(X, M)$, et $mem(X, M)$ en est une sous-formule positive. A partir de ces notions, ils définissent des règles de définition, de dépliage positif et négatif et de pliage qui généralisent les règles de Tamaki et Sato.

La clause (1) résulte d'une définition. **Un dépliage positif** est un dépliage classique à un atome positif ; après un dépliage positif à l'atome $list(L)$ la clause (1) donne :

$$(2)\ sousl([\,]) :- (\forall X mem(X, [\,]) \Rightarrow\ mem(X, M)).$$
$$(3)\ sousl([A|L], M) :- list(L), (\forall X mem(X, [A|L]) \Rightarrow\ mem(X, M)).$$

Un **dépliage négatif** s'effectue à un atome négatif, et consiste à regrouper dans le corps de la clause finale plusieurs formules, obtenues à partir de la clause initiale et de clauses du programme par l'intermédiaire de substitutions n'instanciant aucune variable globale, ou **substitutions négatives** [10]. Pour la clause (3), $mem(X, [A|L])$ peut s'unifier avec chacune des têtes de clause définissant mem ; on le remplace d'abord par $faux$, puis successivement par les corps prai et $mem(X, L)$:

$$sousl([A|L], M) :- list(L), (\forall X faux mem(X, M)),$$
$$(vrai \Rightarrow\ mem(A, M)), (\forall X mem(X, L) \Rightarrow\ mem(X.M)).$$

Cette clause se simplifie en :

$$(4)\ sousl([A|L], M) :- list(L); mem(A, M), (\forall X mem(X, L) \Rightarrow\ mem(X, M)).$$

Pour la clause (2), $mem(X, [\,])$ ne peut s'unifier avec aucune tête de clause définissant mem, on le remplace donc seulement par $faux$:

$$sousl([\,], M) :- (\forall X faux \Rightarrow\ mem(X, M)).$$

10. cette définition est à rapprocher de celle de substitation existentielle ('deciding') en exécutios étendue [**KS86**], cf. chapitre 1.

qui se simplifie en :

$$sousl([\,], M).$$

Le pliage s'effectue suivant le même principe qu'auparavant ; on peut ainsi plier la clause (4) par la clause (1) et obtenir une clause récursive pour $sousl$:

$$sousl([A|L], M) :- list(L), mem(A, M), sousl(L, M).$$

Moyennant certaines conditions restrictives pour l'emploi de ces règles, une transformation par étapes préserve la correction totale des programmes (an sens de la propriété 3.1.1). Ces conditions sont celles du pliage [**TS84**], avec en plus le fait qu'un dépliage négatif ne peut se faire qu'à un atome qui **termine** [11], en utilisant une substitution négative.

3.3.3 Technique de double négation

Sato et Tamaki [**ST84**] ont mis au point une méthode, dite de **double négation** et faisant appel à du pliage/dépliage, pour obtenir un programme logique à partir d'une généralisée dont le corps est quantifié universellement :

$$p(Y) :- \forall X\, formule(X, Y).$$

où $formule(X, Y)$ est une formule logique du premier ordre quelconque. La spécification de $sousl$ est de cette forme :

$$sousl(L, M) :- (\forall X mem(X, L) \Rightarrow\ mem(X, M)).$$

On en prend tout d'abord la négation logique :

$$\neg sousl(L, M) :- mem(X, L), \neg mem(X, M).$$

On élimine ensuite la variable interne X par pliage/dépliage :

$$\neg sousl([X|L], M) :- mem(X, M).$$
$$\neg sousl([Z|L], M) :- \neg sousl(L, M).$$

Puis on passe à la **complétion**, c'est-à-dire qu'on écrit l'ensemble des clauses comme une seule formule du premier ordre :

$$\neg sousl(L, M) \Leftrightarrow\ (\exists\, X, L_1\ L = [X|L_1), \neg mem(X, M)) \\ \vee\ (\exists\, Z, L_1\ L = [Z|L_1], \neg sousl(L_1, M)).$$

11. un atome au sens de [**KH86**] si chaque fois qu'on instancie ses variables globales à des termes clos, on peut le prouver où le réfuter en un temps fini.

et on nie les deux membres de l'équivalence :

$$sousl(L, M) \Leftrightarrow\ (\forall\ X, L_1\ L \neq [X|L_1] \vee mem(X, M)) \\ \wedge\ (\forall\ Z, L_1,\ L \neq [Z|L_1] \vee sousl(L_1, M)).$$

Une technique de négation, en plusieurs étapes, permet d'obtenir le résultat final. On explicite d'abord les cas de disunification :

$$sousl(L, M) \Leftrightarrow\ ((\forall\ X, L_1\ L \neq [X|L_1]) \vee (\exists\ X, L_1, L\ =\ [X|L_1], mem(X, M))) \\ \wedge\ ((\forall\ Z, L_1\ L \neq [Z|L_1]) \vee (\exists\ Z, L_1\ L\ =\ [Z|L_1], sousl(L_1, M))).$$

puis, en distribuant $\vee$ par rapport à $\wedge$:

$$sousl(L, M) \Leftrightarrow\ (\forall\ X, L_1\ L \neq\ [X|L_1]) \\ \vee\ (\exists\ X, L_1\ L = [X|L_1], mem(X, M), sousl(L_1, M)).$$

En exprimant cela sous forme de clauses, et en se servant du fait L est une liste, donc que $\forall\ X, L_1\ L \neq [X|L_1]$ est équivalent à $L = [\,]$, on obtient :

$$sousl([\,], M).$$
$$sousl([X|L], M) :- mem(X, M), sousl(L, M).$$

La phase d'élimination des variables internes ne réussit pas toujours, mais le reste est systématique. La transformation ne garantit que la correction partielle ; mais si de

plus le programme P obtenu est **dichotomique** [12] on a la correction totale (au sens de la propriété 3.1.1). L'avantage de cette méthode est de pouvoir s'appliquer de façon assez générale, ce qui implique en contrepartie que l'on ne soit pas certain d'aboutir.

On peut remarquer que par rapport à la technique de double négation, l'algorithme de Dayantis signifie que l'on peut éliminer les variables internes X d'une clause :

$$p(Y, Z) :- q(X, Y, Z), a(X, Z).$$

quand q est une relation $RR[Y, X]$. C'est en fait un résultat évident dans la mesure où il suffit de déplier q sans déplier a ; on peut alors immédiatement effectuer un pliage.

3.3.4 Stratégies de pliage

12. Un programme P est dichotomique si toute instanciation close de tout prédicat de P peut être prouvée ou infirmée par P en un temps fini ; cela correspond à la notion d'atome qui termine de Kanamori et Horiuchi.

Lau et Prestwich [**LP88**] étudient des stratégies d'introduction de la récursion dans des spécifications en logique du premier ordre, de façon à obtenir des programmes logiques. Une spécification est une clause généralisée quelconque.

Ils utilisent des règles de déduction logique et ne préservent que la correction partielle. Ils emploient tout de même le vocabulaire de la transformation de programmes, en se référant au domaine fonctionnel [**BD77**] (et non aux règles de Tamaki et Sato) : ils appellent **problème de pliage** le fait de chercher à obtenir une définition récursive pour un prédicat. Ils partent d'une formule logique, mise sous une certaine forme normale, accompagnée d'une spécification des appels récursifs souhaités. Par exemple, la spécification de $sousl$:

$$sousl(L, M) : - (\forall\ X\ mem(X, L) \Rightarrow\ mem(X, M)).$$

est mise sous la forme :

$$sousl(L, M) : - (\forall\ X\ mem(X, L)\ \vee\ mem(X, M)).$$

On peut écrire un problème de pliage de plusieurs façons. Pour $sousl$, deux tels problèmes sont les suivants (où $\diamond$ est l'opérateur qui note la concaténation de deux listes) :

(P) **fold** $sousl([Y|L], M)$ **to** $sousl(L, N)$
(P') **fold** $sousl(L_1 \diamond\ L_2, M)$ **to** $sousl(L_1, N_1), sousl(L_2, N_2))$

On sépare la liste L en deux morceaux : dans le premier cas on considère sa tête et son corps, et on voudrait un appel récursif du corps ; dans le deuxième cas on la sépare en deux listes et on voudrait un appel récursif pour chacune. On suppose qu'on a, outre un programme pour mem, un programme pour $\neq mem$:

$$\neg mem(X, [Y|L]) : -(X\ =\ Y), \neg mem(X, L)$$

A partir de la spécification (P), on applique une stratégie de définition qui produit deux sous-problèmes de pliage :

(P_1) **fold** $\neg mem(X, [Y|L])$ **to** $\{\neg\text{mem}(X, L)\}$
(P_2) **fold** $mem(X, M)$ **to** $\text{mem}(X, N)$

P_2 est immédiat modulo l'unification de M et N. P_1 nécessite un dépliage (stratégie d'implication), puis une unification comme pour P_2.

La clause générée, après simplifications par dépliages, est :

$$sousl([X|L], M) : - mem(X, M), sousl(L, M).$$

Il s'agit d'une méthode très générale, qui préserve la correction partielle : on peut partir d'une formule du premier ordre quelconque, et obtenir des définitions récursives différentes à partir d'une même spécification.

Par rapport à la technique de double-négation de Sato et Tamaki, le domaine de spécification est plus vaste, les transformations sont directes car il n'y a pas de phase de négation, mais il faut disposer de programmes pour la négation de certains prédicats.

Plusieurs stratégies sont définies, et elles réduisent effectivement l'espace de recherche, mais il reste beaucoup de non-déterminisme.

3.3.5 Comparaison

Dans tous ces exemples, on vérifie que plus on peut transformer des **formules générales**, moins on est assuré de préserver la **correction**, et plus il y a de **non- déterminisme**.

Dans [**Day87**], on a un domaine de spécification très restreint, et une transformation déterministe préserve l'équivalence ; avec le pliage/dépliage généralisé [**KH86**], on élargit un peu le champ de spécification, et on préserve l'équivalence moyennant la vérification de conditions restrictives ; la double-négation de [**ST84**] antorise des spécifications très générales, est algorithmique (seul le passage de pliage/dépliage n'est pas assuré de réussir) et peut préserver la correction totale dans certains cas ; dans [**LP88**], on part de formules du premier ordre quelconques, on préserve seulement la correction partielle et il reste beaucoup de non-déterminisme.

A l'exception de [**Day87**], toutes ces méthodes utilisent, étendent ou s'inspirent du pliage/dépliage.

Pliage/dépliage du premier ordre Récemment, Sato a présenté dans [**Sat90**] un formalisme qui étend le système de pliage/dépliage de Tamaki et Sato (et celui de [**KH86]**] à des **clauses généralisées quelconques**, qu'il appelle programmes du premier ordre. On peut ainsi transformer un ensemble de clauses généralisées en un autre ensemble de clauses généralisées. C'est intéressant surtout dans le cas particulier où l'on arrive à un programme sous forme de clauses définies.

Les règles de dépliage positif et négatif de [**KH86**] sont regroupées dans une même règle de dépliage ; les deux autres règles sont le pliage et le remplacement. On ne peut appliquer ces règles que sous certaines conditions, qui recouvrent celles de [**TS84** ; **KH86**], et en partant de programmes initiaux stricts [13]. L'ensemble des littéraux prouvables à partir d'un programme strict est préservé ; si l'on transforme des clauses définies les ensemble de succès et d'échec fini sont donc préservés.

13. un programme est strict s'il ne contient aucun prédicat dépendant à la fois positivement et négativement d'un autre (la définition est de Kunen).

Chapitre 4

Systèmes d'extraction et de transformation

Pour étudier puis comparer la transformation des programmes logiques et l'extraction à partir de la preuve d'une spécification, on manipule de nombreux exemples. Un système interactif de transformation ou d'extraction est alors appréciable.

Deux tels systèmes ont été implantés : l'un pour l'extraction d'un programme logique à partir d'une preuve par exécution étendue partant d'un but implicatif, appelé plus brièvement ExExE, et l'autre pour la transformation par pliage/dépliage d'un programme logique. Le langage de programmation choisi est Prolog, par homogénéité avec le domaine étudié. Plus précisément, on utilise le langage interprété C-Prolog [1] [**WBBP84**].

On présente d'abord le système d'extraction, puis celui de transformation. Par convention, les fontes seront les suivantes :

- texte tapé par l'utilisateur,
- texte affiché par le programme.

4.1 Système ExExE d'extraction et d'exécution étendue

Les règles d'exécution étendue de [**KS86**] et le formalisme d'extraction de [Fri90] sont implantés, selon la présentation faite dans les deux premiers chapitres.

4.1.1 Présentation du système

Structures manipulées Les structures manipulées sont les **buts implicatifs** à prouver et les clauses d'entrée-sortie qui gardent trace de la preuve déjà effectuée. Les connecteurs

1. les programmes ont aussi été modifiés pour le langage compilé SICStus Prolog [**CW89**], extérieurement très semblable.

d'implication et de conjonction sont notés, en infixe, par => et & [2]. Les variables existentielles sont distinguées par un point d'interrogation ?(X) des autres variables qui sont universelles. Les clauses sont représentées, comme en **C-Prolog**, en séparant la tête et le corps par le connecteur :- et les atomes du corps par une virgule. Les listes sont entourées de crochets : [], [a, b, c], [A|X].

Variables

Les variables sont traitées comme des variables Prolog, mais pour plus de lisibilité sont affichées avec des lettres majuscules. Pour cela le but initial, qui peut être donné avec n'importe quels noms de variables, est réécrit avec des variables apparaissant dans l'ordre suivant : X, Y, Z, T, U, V, W, A, B, C, D... Sur un exemple, cela donne :

»»»»»» REGLE : init.

»»»»»» BUT :

| : app(As, Bs, ?(AsBs)).

but initial :

(F) app(X,Y, ?(Z))

Algorithme d'unification modifié

Du fait de la distinction entre variables universelles et existentielles et de la manipulation de **substitutions existentielles**, l'unification de Prolog n'est pas adaptée. Un algorithme d'unification modifié est programmé explicitement. On utilise l'algorithme classique d'unification de Robinson, sans test d'occurence comme dans la majorité des systèmes Prolog, et adapté de façon à calculer deux substitutions concernant respectivement les variables universelles et existentielles.

Règles

Les règles d'exécution étendue et le processus d'extraction associé sont programmés d'après [**KS86**; Fri90]. On dispose des règles :

- d'induction, structurelle et par point fixe :
  pour l'induction structurelle, l'utilisateur choisit la variable d'induction parmi les variables universelles du but, lequel doit être une conjonction d'atomes; pour l'induction par point fixe, le but doit posséder une hypothèse atomique;

- de dci, déterministe et non-déterministe :

2. Le connecteur de conjonction Λ utilisé jusqu'à présent n'est pas un caractère standard; on se rabat sur le "et" commercial : &

les atomes de la conclusion sont proposés à l'utilisateur ; dans le cas d'une dci déterministe, si plusieurs clauses du programme conviennent, l'utilisateur doit en choisir une ;

- de simplification, gardant ou non l'hypothèse :
  toutes les paires d'un atome de l'hypothèse et d'un atome de la conclu- sion unifiables par une substitution existentielle sont proposées.

Une règle permet de postuler une conjonction d'atomes. On peut changer la quantification d'une variable et transférer un atome de l'hypothèse vers la conclusion d'un but. On peut demander à voir les clauses d'entrée-sortie déjà générées, éventuellement après simplification par dépliages. La troncature d'un prédicat, avec propagation aux prédicats utilisés dans sa définition, est également programmée.

Boucle interactive

Une boucle interactive gère les instructions données par l'utilisateur et applique les différentes règles. Des règles de "confort" sont ajoutées, permettant le choix du programme initial à charger, donnant la définition d'un prédicat particulier dans ce programme, présentant un exemple simple et complet de preuve et d'extraction, rappelant le but initial ou le but courant, interrompant ou recommençant la preuve. Une aide en ligne permet d'obtenir la liste des instructions utilisables à un moment donné dans le cours de la preuve. Quant une règle échoue un message d'erreur est affiché et le système redemande une règle. Lors d'une simplification, on peut abandonner après que les paires possibles aient proposées.

4.1.2 Session commentée

Le programme d'exécution étendue s'appelle exexe. Le processus d'extraction d'un programme pour *appleng* se présente comme suit :

```
| ?- [exexe].

************************************************

      exexe: Extraction de programmes

  a partir de preuves par Execution etendue

************************************************

>>> Pour debuter, taper exexe.
```

```
exexe consulted 84180 bytes 6.73333 sec.

yes

| ?- exexe.

>>> Le fichier "pred" contient des definitions de predicats et des
>>> informations sur leurs arguments (types, definition recursive).
```

Le fichier *pred* correspond au programme logique initial. Outre les définitions de prédicats, il contient la mention des types de leurs arguments et de ceux sur lesquels ils sont définis récursivement ; ces informations sont données par le prédicat $type{-}rec$. La règle *def* permet d'obtenir seulement la définition d'un prédicat, et la règle *info* toutes les informations à son sujet se trouvant dans *pred*, c'est-à-dire sa définition et les clauses pour $type - rec$ le concernant.

```
>>>> Preferez-vous charger un autre fichier (o/n)? n.
pred reconsulted 5860 bytes 483335 sec.

>>>> Taper: g. ou: guide pour obtenir de l'aide.

>>>>>>>>>>> REGLE: def(app).
app([ ],Y,Y).
app([A|X],Y, [A|Z]):- app(X,Y,Z).

>>>>>>>>>>> REGLE: info(leng).
leng([ ],0).
leng([A|X],S(N)) :- leng(X,N).
        type_rec(leng,1,list,true).
        type_rec(leng,2,nat,false).
```

Les règles utilisables avant d'avoir commencé la preuve sont les suivantes :

```
>>>>>>>>>>> REGLE: guide.
>>>> Regles disponibles:
charge(F) : charge fichier F
charge    : donne les noms des fichiers charges
def(P)    : donne les clauses definissant le predicat P
info(P)   : donne les clauses concernant le predicat P
info(P,F) : donne les clauses du fichier F concernant le predicat
exemple   : montre un exemple de session
init      : demande le but initial a prouver
f, fin    : sortie du systeme
```

Induction structurelle

On commence par exposer cas d'une preuve par induction structurelle. Le but initial devra alors être une conjonction d'atomes :

```
>>>>>>>>>> REGLE: init.

>>>>>>>>>> BUT:
|: app(X,Y,?(Z))&leng(?(Z),?(T)).

but initial:
(G) app (X,Y,?(Z))&leng(?(Z),?(T))
```

La liste des règles et instructions que l'on peut utiliser, maintenant que le but à prouver a été donné, s'obtient toujours par guide :

```
>>>>>>>>>> REGLE: guide.
>>>> Regles disponibles:
charge    : donne les noms des fichiers charges
def (P)   : donne les clauses definissant le predicat P
info(P)   : donne les clauses concernant le predicat P
info(P,F) : donne les clauses du fichier F concernant le predicat P
exemple   : montre un exemple de session
exist     : existentialise une variable de la conclusion
univ      : universalise une variable de la conclusion
transfert : transfere un atome dans la conclusion
struct    : regle d'induction structurelle
comput    : regle d'induction par point fixe
dci       : regle de 'definite clause inference' non-deterministe
dcir      : regle de 'definite clause inference' deterministe
simpl     : regle de simplification
simpl_g   : regle de simplification gardant l'hypothese
postul    : regle de postulat
b_cour    : affiche le but courant
b_init    : affiche le but initial
tout_es   : affiche la liste de toutes les clauses generees
depl_es   : affiche la liste des clauses generees, apres depliages
efface    : fin de la preuve, on peut alors en commencer une autre
recomm    : recommence la preuve, avec le meme but initial
f. fin    : sortie du systeme
```

Pour l'induction structurelle, seules les variables universelles sur lesquelles un prédicat du but est défini récursivement sont proposées, accompagnées de leur type :

```
>>>>>>>>>> REGLE: struct.
variable d'induction structurelle choisie:
X:list(X)

nouveaux buts:
(G1) app([],Y,?(Z))&leng(?(Z),?(T))
(G2) (app(X,Y,W)&leng (W,V))=>app([U|X],Y,?(Z))&leng(?(Z),?(T))

clauses associees:
esG([],Y,Z,T):-esG1(Y,Z,T).
esG([U|X],Y,Z,T):-esG (X,Y,W,V),esG2(X,Y,W,V,U,Z,T).
```

Chaque but a un nom et un prédicat associé. Quand il reste plusieurs buts à prouver, l'utilisateur choisit celui sur lequel il veut travailler.

```
>> quel but?

G1: app([],Y,?(Z))&leng(?(Z),?(T))
G2: (app(X,Y,W)&leng (W,V))=>app([U|X],Y,?(Z))&leng(?(Z),?(T))
|: G1.
```

On commence par appliquer la règle de dci. Si un atome de la conclusion a un argument récursif instancié il est choisi pour la dci, sinon l'utilisateur en choisit un dans la conclusion.

```
>>>>>>>>>>> REGLE: dci.

   atome positif choisi:
app([],Y,?(Z))

   nouveau but:
(H) leng(Y,?(T))

   clause associee:
esG1 (Y,Y,T):-esH(Y,T).

>> quel but?
H: leng(Y,?(T))
G2: (app(X,Y,W)&leng(W,V))=>app([U|X],Y,?(Z))&leng(?(Z),?(T))

|: H.
```

On peut décider d'arrêter la preuve d'un but par un postulat :

```
>>>>>>>>>> REGLE: postul.

   clause associee:
esH(Y,T):-leng(Y,T).

>>>> 1 succes partiel
   nouveau but courant:
(G2) (app(X,Y,W)&leng (W,V))=>app([U|X],Y,?(Z))&leng(?(Z),?(T))
```

Pour le but (G2), on effectue une dci ; l'atome instancié à un argument récursif est automatiquement choisi.

```
>>>>>>>>>> REGLE: dci.

   atome positif choisi:
app([U|X],Y,?(Z))

   nouveau but:
(I) (app(X,Y,W)&leng (W,V))=>app(X,Y,?(Z))&leng([U|?(Z)],?(T))

   clause associee:
esG2(X,Y,W,V,U,[U|Z],T):-esI(X,Y,W,V,Z,U,T).

   nouveau but courant:
(I) (app(X,Y,W)&leng(W,V))=>app(X,Y,?(Z))&leng([U|?(Z)],?(T))
```

On fait ensuite une simplification. La paire d'atomes app(X, Y, W), app(X, Y, ?(Z)) est automatiquement choisie.

```
>>>>>>>>>> REGLE: simpl.,

   paire d'atomes choisie:
app(X,Y,W), app(X,Y,?(Z))

>> confirmation (o/n): o.

   nouveau but:
(J) leng(Z,V)=>leng([U|Z],?(T))

   clause associse:
es_I(X,Y,Z,V,Z,U,T):-es_J(Z,V,U,T).

   nouveau but courant:
```

```
(J) leng(Z,V)=>leng([U|Z],?(T))
```

On fait ensuite de nouveau une dci et une simplification, cette fois pour le prédicat leng. Cela termine la preuve.

```
>>>>>>>>>> REGLE: dci.
   atom Positif choisi:
leng([U|Z],?(T))

   nouveau but:
(K) leng(Z,V)=>leng(Z,?(T))

   clause associee:
es_J(Z,V,U,s(T)):-es_K(Z,V,T).

   nouveau but courant:
(K) leng(Z,V)=>leng(Z,?(T))

>>>>>>>>>> REGLE: simpl.

   paire d'atomes choisie:
leng (Z,V), lang (Z,?(T))

>> confirmation (o/n): o.

   nouveau but:
      vrai

   clause associee:
es_K(Z,T,T).

>>>>>>>>>> succes total!
```

A la fin de la preuve, il est possible de voir la liste de toutes les clauses générées ou de voir directement le résultat de leur simplification par dépliages. Dans chaque cas, on rappelle le but dont on est parti ainsi que son prédicat d'entrée-sortie associé.

```
>>>> Voulez-vous voir toutes les clauses d'entree-sortie (tout_es)
>>>> ou seulement leur simplification par depliages (deples)?

>>>>>>>>>> REGLE: tout_es.
   but prouve:
(G) app(X,Y.?(Z))&leng(?(Z),?(T))
```

```
   predicat associe:
esG(X,Y,Z,T)

   clauses collectees:

esG([],Y,Z,T):-esG1(Y,Z,T).
esG([U|X],Y,Z,T):-esG(X,Y,W,V),esG2(X,Y.W,V,U,Z,T).
esG1(Y,Y,T):-esH(Y,T).
esH(Y,T):-leng(Y,T).
esG2(X,Y,W,V,U,[U|Z],T):-esI (X,Y,W,V,Z,U,T).
es_I(X,Y,Z,V,Z,U,T):-es_J(Z,V,U,T).
es_J(Z,V,u,T):-es_K(Z,V,T)
es_K(Z.T.T).

>>>>>>>>>> REGLE: depl_es.

   but prouve:
(G) app(X.Y.?(Z))&leng(?(Z),?(T))

   predicat associe:
esG(X,Y,Z,T)

   clauses d'entree-sortie apres depliages:
esG([],Y,Y,T):-leng(Y,T).
esG([U|X],Y,[U|Z],s(T)):-esG(X,Y,Z,T).
```

On peut enfin tronquer le programme obtenu par rapport à certains de ses arguments (les variables apparaissant plusieurs fois dans le but initial sont d'abord proposées, car les variables intermédiaires vérifient toujours ce critère) :

```
>>>> Voulez-vous tronquer es_G (o/n)? o.
>>>> par rapport a l'argument numero 3 (o/n/ofin/nfin)? ofin.

   clauses d'entree-sortie tronquees:
esG([],Y,T):-leng(Y,T).
esG([U|X],Y,s(T)):-esG(X,Y,T).
```

Induction par point fixe

Le même exemple peut être traité par induction par point fixe. Le but initial est légèrement différent, puisque cette forme d'induction nécessite une hypothèse atomique.

```
>>>>>>>>>> REGLE: init.
```

```
>>>>>>>>>> BUT:

|: app(X,Y,Z)=>leng(Z,?(T)).

   but initial:
(G) app(X,Y,Z)=>leng(Z,?(T))

La règle d'induction par point fixe donne deux lemmes:
>>>>>>>>>> REGLE: comput.

   nouveaux buts:

(G1) leng(Y,?(T))
(G2) leng(Z,V)=>leng([U|Z],?(T))

   clauses associees:
esG([],Y,Y,T):-esG1(Y,T).
esG([U|X].Y,[U|Z],T):-esG(X,Y,Z,V),esG2(Z,V,U,T).
```

La suite est similaire à la preuve par induction structurelle présentée plus haut, et le programme final est identique.

Bibliographie

[AFQ88] Francis Alexandre, Jean-Pierre Finance et Alain Quéré. « SPES : un système de transformation de programmes logiques ». In : *SPLT'88, 7ème Séminaire Programmation en Logique, 25-27 mai 1988, Trégastel, France*. Sous la dir. de Serge Bourgault et Mehmet Dincbas. 1988, p. 69-84.

[BD77] Rod M. Burstall et John Darlington. « A Transformation System for Developing Recursive Programs ». In : *J. ACM* 24.1 (1977), p. 44-67. doi : 10.1145/321992.321996. url : https://doi.org/10.1145/321992.321996.

[Bur69] Rod M. Burstall. « Proving Properties of Programs by Structural Induction ». In : *Comput. J.* 12.1 (1969), p. 41-48. doi : 10.1093/COMJNL/12.1.41. url : https://doi.org/10.1093/comjnl/12.1.41.

[Fri90] Laurent Fribourg. « Extracting Logic Programs from Proofs that Use Extended Prolog Execution and Induction ». In : *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, June 18-20, 1990*. Sous la dir. de David H. D. Warren et Péter Szeredi. MIT Press, 1990, p. 685-699.

[KS86] Tadashi Kanamori et Hirohisa Seki. « Verification of Prolog Programs Using an Extension of Execution ». In : *Third International Conference on Logic Programming, Imperial College of Science and Technology, London, United Kingdom, July 14-18, 1986, Proceedings*. Sous la dir. d'Ehud Shapiro. T. 225. Lecture Notes in Computer Science. Springer, 1986, p. 475-489. doi : 10.1007/3-540-16492-8_96. url : https://doi.org/10.1007/3-540-16492-8%5C_96.