Etat de l'Art sur la Spécialisation de Programmes

Tristan Cazenave[1]

Résumé. La spécialisation de programmes est aussi appelée l'évaluation partielle. C'est une technique de transformation de programmes qui consiste à évaluer une partie d'un programme générique sur certaines de ses entrées de façon à ce qu'il s'exécute ensuite plus rapidement pour ces entrées particulières. Nous verrons ses applications actuelles et son intérêt potentiel pour l'Intelligence Artificielle.

1 INTRODUCTION

La spécialisation de programmes est aussi appelée l'évaluation partielle. C'est une technique de transformation de programmes qui consiste à évaluer une partie d'un programme générique sur certaines de ses entrées de façon à ce qu'il s'exécute ensuite plus rapidement pour ces entrées particulières. Cette technique a été utilisée avec succès sur de nombreux langages, des langages fonctionnels (Scheme, Escher…), logiques (Prolog, Goedel) et impératifs (C, Pascal, Fortran).
Nous commençons par replacer, dans la section 2, la spécialisation de programme dans le cadre plus général des transformations de programmes, de nombreux exemples sont tiré de l'excellent papier de Alberto Pettorossi et Maurizio Proietti [Pettorossi et al. 98] et aussi de [Pettorossi et al. 96]. La section 3 décrit l'évaluation partielle. La section 4 décrit son application au langage C. Nous concluons avec la section 5 qui donne des pistes de l'utilité potentielle de la spécialisation de programmes pour l'Intelligence Artificielle.

2 TRANSFORMATIONS DE PROGRAMMES LOGIQUES

2.1 Transformations de base

Les transformations de base utilisées sont :

- Le dépliage
- Le pliage
- La définition
- La généralisation

Le dépliage [Tamaki et al. 84] correspond au remplacement d'un but dans le corps d'une clause par sa définition.

Dépliage : Soit P un programme logique contenant la clause C de la forme H:-F,A,G où A est un littéral positif et F et G des buts pouvant être vides. On suppose :

- $D_1,\ldots,D_n$, est l'ensemble de toutes les clauses du programme P telles que A est unifiable avec les Tete(D_i) en utilisant les unificateurs les plus généraux $\theta_1,\ldots,\theta_n$.
- Ci est la clause (H:-F, Corps(D_i),G) θ_i, pour i=1,…,n.

Le dépliage de A dans la clause C dans le programme P consiste à remplacer la clause C par les clauses $C_1,\ldots,C_n$.

Soit P le programme suivant :

---

[1] Laboratoire d'Intelligence Artificielle, Université Paris 8, 2 Rue de la Liberté, 93 Saint Denis, France. cazenave@ai.univ-paris8.fr

```
p(X) :- q(X), r(X).
q(a).
r(a).
r(b).
```

Spécialiser le programme P en le dépliant permet d'obtenir le programme :

```
p(a).
q(a).
r(a).
r(b).
```

Soit le programme suivant :

```
grandmere(X,Y) :- mere(X,Z), parent(Z,Y).
grandpere(X,Y) :- pere(X,Z), parent(Z,Y).
parent(X,Y) :- mere(X,Y).
parent(X,Y) :- pere(X,Y).
mere(alexandra,pierre).
mere(alexandra,mathilde).
mere(chantal,alexandra).
```

Lorsqu'on spécialise ce programme sur le but grandmere(X,pierre) en dépliant toujours l'atome le plus à gauche, et en ne dépliant pas le prédicat 'pere' dont les instances ne sont pas connues lors de la spécialisation, on obtient :

```
grandmere(alexandra,pierre) :- pere(pierre, pierre).
grandmere(alexandra,pierre) :- pere(mathilde, pierre).
grandmere(chantal,pierre).
grandmere(chantal,pierre) :- pere(alexandra, pierre).
```

Le pliage est l'opération inverse du dépliage. Elle correspond donc au remplacement d'un ensemble d'atome dans le corps d'une clause par la un seul atome correspondant au prédicat plié.

Pliage : Soit P un programme logique contenant les clauses $C_1,\ldots,C_n$ ainsi que les clauses $D_1,\ldots,D_n$. Supposons qu'il existe un atome A et deux buts F et G tels que pour tout i compris entre 1 et n, il existe une substitution θ_i qui satisfait :

- C_i est une variante de la clause H:-F, Corps(D_i) θ_i,G.
- A= Tete(D_i) θ_i
- Pour chaque clause D de P qui n'est pas dans $D_1,\ldots,D_n$, Tete(D) n'est pas unifiable avec A
- Pour toute variable X dans l'ensemble vars (Di)-vars(Tete(Di)), on a :
  - -$X\theta_i$ est une variable qui n'apparaît pas dans (H,F,G)
  - -La variable $X\theta_i$ n'apparaît pas dans le terme $Y\theta_i$, pour toute variable Y de Corps(D_i) et différente de X.

Le pliage des clauses $C_1,\ldots,C_n$ en utilisant les définitions $D_1,\ldots,D_n$ dans P consiste à remplacer les clauses Ci par la clause C = H:-F,A,G.

Définition : On définit une clause newp(X_1, …, X_m) :- B_1, …, B_n. où newp est un nouveau nom de prédicat, où les B_j sont des atomes dont les prédicats appartiennent au programme initial et où les X_i sont un sous ensemble des variables des B_j, en ajoutant la clause au programme.

Soit le programme Palyndrome qui teste (inefficacement) si une liste est un palyndrome :

```
1. pal([]).
2. pal([H]).
3. pal([H|T]) :- append(Y,[H],T), pal(Y).
4. append([],Y,Y).
5. append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

On définit le prédicat newp :

```
newp (L,T) :- append(Y,L,T), pal(Y).
```

Plier le prédicat newp dans le programme Palyndrome consiste à remplacer la ligne 3 par la ligne 3f :

```
3f. pal([H|T]) :- newp([H],T).
```

Généralisation :

La généralisation de deux termes t_1 et t_2 consiste à trouver un terme t et deux substitutions σ_1 et σ_2, t le plus précis possible, tels que $t\sigma_1=t_1$ et $t\sigma_2=t_2$, on dit que t est la généralisation la plus précise des termes t_1 et t_2.

Cela revient à trouver une formule qui généralise deux formules données. La généralisation est toujours possible, ne serait ce qu'avec le terme t correspondant à une variable X et avec $\sigma_1=(X,t_1)$ et $\sigma_2=(X,t_2)$.

Exemples d'applications de l'algorithme de généralisation à des couples de termes :

f(g(a,b),X) et f(a,g(b,X)) donnent $\sigma_1=\{(Y,g(a,b)),(Z,X)\}$ et $\sigma_2=\{(Y,a)),(Z,g(b,X))\}$ et t=f(Y,Z)

f(a,a) et f(g(b,b),g(b,b)) donnent $\sigma_1=\{(X,a)\}$ et $\sigma_2=\{(X,b)\}$ et t=f(X,X)

f(a,g(a,b)) et f(b,g(b,b)) donnent $\sigma_1=\{(X,a)\}$ et $\sigma_2=\{(X,b)\}$ et t=f(X,g(X,b))

2.2 Le tupling

Le tupling est un stratégie de transformation de programmes qui consiste à ne pas faire faire au programme plusieurs fois le même travail. Elle consiste a regrouper les sous buts communs à plusieurs atomes

Soit le programme suivant qui traverse deux fois une liste, une fois pour la construire, une fois pour calculer sa longueur :

```
1. append ( [], L, L).
2. append([H|L1], L, [H|L2]) :- append(L1, L, L2).
3. length([] , 0).
4. length([T|L], X) :- length (L,Y), X is Y+1.
5. append_length(L1, L2, L3, X) :- append(L1,L2,L3), length(L3,X).
```

On obtient par dépliage de append :

```
6. append_length ( [], L, L, X) :- length (L,X).
7. append_length ( [T|R], L1, [T|L2], X) :- append (R, L1, L2), length
([T|L2], X).
```

Puis par dépliage de length :

```
8. append_length ( [T|R], L1, [T|L2], X) :- append (R, L1, L2), length (L2,
Y), X is Y+1.
```

Enfin par pliage de (5) dans (8)

```
9. append_length ( [T|R], L1, [T|L2], X) :- append_length (R, L1, L2, Y), X
is Y+1.
```

D'où le programme final qui ne traverse qu'une fois la liste :

```
1. length([] , 0).
2. length([T|L], X) :- length (L,Y), X is Y+1.
3. append_length ( [], L, L, X) :- length (L,X).
4. append_length ( [T|R], L1, [T|L2], X) :- append_length (R, L1, L2, Y), X
   is Y+1.
```

La transformation que nous avons effectuée sur le programme append_length s'appelle le tupling, elle consiste à remplacer deux parcours de liste par un seul parcours qui fait les deux opérations à la fois quand c'est possible.

Tupling : Soit une clause C de la forme

H :- $A_1,\ldots,A_m,B_1,\ldots,B_n$.

On définit le prédicat newp en insérant la clause T :

newp($X_1,\ldots,X_k$) :- $A_1,\ldots,A_n$.

Où les arguments $X_1,\ldots,X_n$ sont les éléments de vars($A_1,\ldots,A_m$) $\cap$vars(H, $B_1,\ldots,B_n$). On cherche alors à trouver une définition récursive du prédicat newp par des pliages et des dépliages utilisant la clause T.
Le tupling est souvent utilisé quand $A_1,\ldots,A_m$ ont des variables communes. Les améliorations dues à cette transformation viennent de ce que l'on n'évalue qu'une seule fois des buts qui sont communs aux atomes $A_1,\ldots,A_m$. Le tupling permet aussi d'éviter de visiter plusieurs fois des structures de données et d'éviter la construction de données intermédiaires.

On peut aussi appliquer la stratégie du tupling au programme palyndrome donné précédemment. pal (Y) et append(Y,[H],T) traverse la même liste Y. Le tupling va chercher a éviter cette duplication de la visite de la liste. On définit pour cela le prédicat newp :

```
newp(L,T) :- append(Y,L,T),pal(Y).
```

(l'argument [H] dans le corps de la clause 3 a été généralisé à L), et on insère cette clause comme la clause 6 dans le programme.On plie alors la définition de 6 dans la clause 3, on obtient :

```
3f. pal([H|T]) :- newp([H],T).
```

La double visite n'est toujours pas évitée en remplacant 3 par 3f. On doit pour cela trouver une définition récursive de newp. Pour cela, on commence par déplier pal(Y) dans la clause 6, on obtient:

```
7. newp(L,T) :- append([],L,T).
8. newp(L,T) :- append([H],L,T).
9. newp(L,T) :- append([H|Y],L,T), append(R,[H],Y), pal(R).
```

On déplie alors append dans les clauses 7, 8 et 9 et on obtient :

```
10. newp(L,L).
11. newp(L,[H|L]).
12. newp(L,[H|U]) :- append(Y,L,U), append(R,[H],Y), pal(R).
```

Maintenant pour avoir une définition récursive de newp on va chercher à plier 6 dans 12. Pour cela on commence par remplacer append(Y,L,U), append(R,[H],Y) par append(R,[H|L],U), on obtient :

```
13. newp(L,[H|U]) :- append(R,[H|L],U), pal(R).
```

puis on peut alors plier 6 dans 13f :

```
13f. newp(L,[H|U]) :- newp([H|L],U).
```

Le programme transformé par tupling permet ainsi de ne parcourir qu'une seule fois la liste de caractères à tester.

2.3 La déforestation

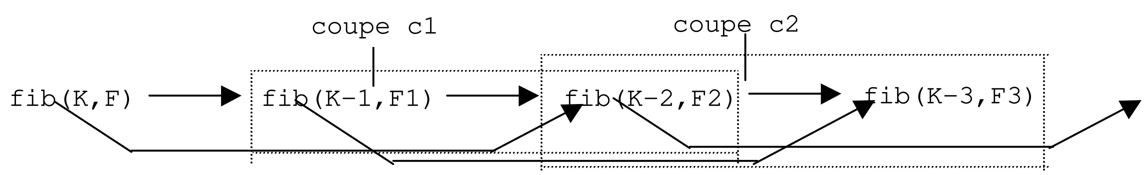
La Déforestation : La déforestation est une stratégie de transformation de programmes qui a pour but d'éliminer les structures de données intermédiaires dans les programmes en introduisant de nouvelles définitions qui sont égales à la composition de prédicats déjà définis.

On veut transformer le programme suivant qui calcule les nombres de Fibonacci :

```
1. fib(0,1).
2. fib(1,1).
3. fib(N,F) :- N1 is N-1,fib(N1,F1),N2 is N1-1,fib(N2,F2),F is F1+F2.
```

Pour un nombre donné, le programme calcule un nombre exponentiel de sommes. Complexité = $O(2^n)$ pour le nième nombre de Fibonacci.

Lorsqu'on représente sur un même schéma pour les nombres de Fibonacci d'indices k, k-1 et k-2 les instances de prédicats auxquels il font appel. On observe les régularités suivantes :



Les régularités sont représentées par les coupes c1, c2 … On introduit un nouveau prédicat t(N,F) pour prendre en compte ces régularités, puis on applique la stratégie du tupling en dépliant cette définition et en repliant pour trouver une définition récursive qui sera appelée pour calculer fib(N,F).

On introduit la définition suivante :

```
4. t(N,[F1,F]) :- N1 is N+1, fib(N1,F1), fib(N,F).
```

On repliant la définition dans fib(N,F) on obtient :

```
3a. fib(N,F) :- N2 is N-2, t(N2,[F1,F2]), F is F1+F2.
```

En dépliant fib(N1,F1), on obtient :

```
5. t(-1,[1,F]) :- fib(-1,F).
6. t(0,[1,1]).
7. t(N,[F1,F]) :- N1 is N+1, N2 is N1-1, fib(N2,F2), N3 is N2-1, fib(N3,F3),
F1 is F2+F3, fib(N,F).
```

Qui devient apres les simplification sur les affectations :

```
7a. t(N,[F1,F]):- fib(N,F2), N3 is N-1, fib(N3,F3), F1 is F2+F3, fib(N,F).
```

Qui devient après qu'on ait remarqué que a un N donné ne correspond qu'un seul nombre de Fibonacci, ce qui permet d'unifier F et F2 :

```
7b. t(N,[F1,F]):- N3 is N-1, fib(N3,F3), fib(N,F), F1 is F+F3.
```

On peut rajouter N is N3+1, dans cette clause puisque on a N3 is N-1, En repliant t, on obtient alors :

```
8. t(N,[F1,F]):- N3 is N-1, t(N3,[F,F3]), F1 is F+F3.
```

Le programme final est alors :

```
fib(0,1).
fib(1,1).
fib(N,F) :- N2 is N-2, t(N2,[F1,F2]), F is F1+F2.
t(-1,[1,F]) :- fib(-1,F).
t(0,[1,1]).
t(N,[F1,F]):- N3 is N-1, t(N3,[F,F3]), F1 is F+F3.
```

Il a une complexité en O(n) pour le nième nombre de Fibonacci.

2.4 Exemple d'application : la compilation du contrôle

La stratégie de Prolog pour contrôler la SLD résolution consiste à évaluer les clauses dans l'ordre dans lequel elles ont été données, de gauche à droite. Cette stratégie simple ne donne pas toujours l'efficacité voulue. Une solution pour améliorer la stratégie de contrôle est de définir une stratégie de contrôle améliorée puis de transformer les programmes utilisant cette stratégie améliorée en des programmes équivalents utilisant la stratégie simple.

Les sous séquences communes : Une séquence est représentée par une liste d'éléments. Pour deux séquences X et Y, on a subseq(X,Y) si et seulement si X est une sous séquence de Y, c'est à dire que X peut être obtenue à partir de Y en lui enlevant des éléments.

Le programme suivant permet de vérifier que X est une sous séquence commune aux deux séquences Y et Z. Pour cela il commence par vérifier que X est une sous séquence de Y, puis il vérifie que X est une sous séquence de Z.

```
1. csub(X,Y,Z) :-subseq(X,Y),subseq(X,Z).
2. subseq([],X).
3. subseq([A|X],[A|Y]) :-subseq(X,Y).
4. subseq([A|X],[B|Y]) :-subseq([A|X],Y).
```

On veut appliquer à ce programme la stratégie de résolution S suivante :

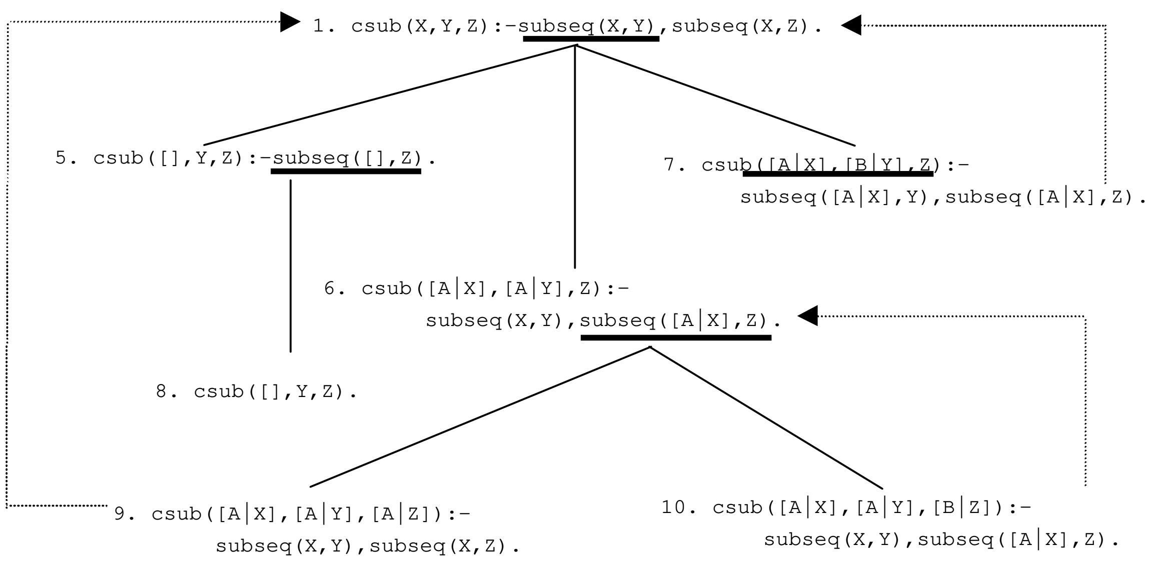
Si le but est ‘subseq(w,x),subseq(y,z)’ et que w est un sous terme de y
Alors choisir de résoudre l’atome subseq(y,z)
Sinon choisir de résoudre l’atome le plus à gauche dans le but.

L’utilisation de cette stratégie est à priori plus efficace que la stratégie de gauche à droite de Prolog parce que la deuxième occurence de subseq est choisie dès qu’elle est suffisamment instanciée par l’évaluation de la première occurence. L’utilisation de cette stratégie permettra donc à un but de la forme ‘subseq(...),subseq(...)’ d’échouer même si la première occurence n’a pas été complètement évaluée.

On définit une stratégie de dépliage Sd qui corresponde à la stratégie de résolution S définie précédemment :

Si le corps de la clause à déplier est ‘subseq(w,x),subseq(y,z)’ et que w est un sous terme de y
Alors choisir de déplier l’atome subseq(y,z)
Sinon choisir de déplier l’atome le plus a gauche dans le corps de la clause.

On déplie alors les prédicats subseq dans la clause 1 en utilisant la stratégie de dépliage définie précédemment :



On définit alors le prédicat newcsub et on plie les clauses définissant csub qui peuvent l’être.

```
11. newcsub(A,X,Y,Z) :-subseq(X,Y),subseq([A|X],Z).
```

On peut plier newcsub dans la clause 6 :

```
6f. csub([A|X],[A|Y],Z) :-newcsub(A,X,Y,Z).
```

On peut aussi plier la clause 7 en utilisant la clause 1

```
7f. csub([A|X],[B|Y],Z) :-csub([A|X],Y,Z).
```

On peut déplier de nouveau la clause newcsub en utilisant la stratégie de dépliage Sd définie.

On applique à la clause 11, les dépliages correspondants à ceux qui amène de la clause 6 aux clauses 9 et 10. On trouve alors les clauses 12 et 13 :

```
12. newcsub(A,X,Y,[A|Z]) :-subseq(X,Y),subseq(X,Z).
13. newcsub(A,X,Y,[B|Z]) :-subseq(X,Y),subseq([A|X],Z).
```

En pliant les clauses newcsub en utilisant les définitions présentes dans le programme, on obtient :

```
12f. newcsub(A,X,Y,[A|Z]) :-csub(X,Y,Z).
13f. newcsub(A,X,Y,[B|Z]) :-newcsub(A,X,Y,Z).
```

Ce qui donne le programme final ainsi obtenu.

```
8. csub([],Y,Z).
6f. csub([A|X],[A|Y],Z) :-newcsub(A,X,Y,Z).
7f. csub([A|X],[B|Y],Z) :-csub([A|X],Y,Z).
12f. newcsub(A,X,Y,[A|Z]) :-csub(X,Y,Z).
13f. newcsub(A,X,Y,[B|Z]) :-newcsub(A,X,Y,Z).
```

Ce programme est plus efficace que l'original. La stratégie de contrôle a été compilée dans le programme original pour donner le programme final.

2.5 Autre exemple : la composition de programmes

Une technique courante en programmation est de diviser un problème en sous problèmes plus faciles, d'écrire des sous programmes pour ces sous problèmes pour finalement composer les sous programmes en un programme qui résout le problème. Toutefois cette façon de procéder amène souvent à des programmes inefficaces parce que les différents sous programmes ne partagent pas des calculs qu'ils pourraient partager. Pour résoudre les problèmes dus à ce type de programmation, deux types de techniques sont utilisées en métaprogrammation :

- l'amélioration de l'évaluateur en utilisant par exemple un ramasse-miettes, le memoing, une lazy évaluation ou le coroutinage.
- la transformation du programme en un autre programme qui peut être efficacement évalué par un évaluateur standard.

Nous allons voir comment cela est possible sur le programme suivant qui ôte d'une liste X de nombres positifs toutes les occurrences de son maximum :

```
deletemax(Xs,Ys) :- maximal(Xs,M),delete(M,Xs,Ys).
maximal([],0).
maximal([X|Xs],M) :- maximal(Xs,N), M is max(N,X).
delete(M,[],[]).
delete(M,[M|Xs],Ys) :- delete(M,Xs,Ys).
delete(M,[X|Xs],[X|Ys]) :- M=\=X,delete(M,Xs,Ys).
```

On veut obtenir un programme qui ne traverse la liste Xs qu'une seule fois. Pour cela, on applique la stratégie du tupling aux prédicat maximal et delete qui partagent l'argument Xs.

En dépliant delete dans la clause deletemax, on obtient :

```
7. deletemax([],[]).
8. deletemax([M|Xs],Ys) :-maximal(Xs,N), M is max(N,M), delete(M,Xs,Ys).
9. deletemax([X|Xs],[X|Ys]) :-maximal(Xs,N),M is
max(N,X),M=\=X,delete(M,Xs,Ys).
```

A ce stade, le pliage n'est pas possible, on va donc donner une nouvelle définition qui va permettre le pliage en utilisant la généralisation. On définit un nouveau prédicat gen(Xs,P,Q,Ys) qui généralise les deux buts pour permettre le pliage. On généralise les deux buts 'maximal(Xs,N),delete(M,Xs,Ys)' et 'maximal(Xs,M),delete(M,Xs,Ys)' ce qui donne :

```
10. gen(Xs,P,Q,Ys) :-maximal(Xs,P),delete(Q,Xs,Ys).
```

En appliquant la stratégie du tupling en pliant le prédicat défini dans la clause 1, on obtient :

```
1f. deletemax(Xs,Ys) :-gen(Xs,M,M,Ys).
```

On trouve alors une définition récursive du prédicat nouvellement défini en dépliant le prédicat delete puis en pliant avec la définition du nouveau prédicat :

```
11. gen([],0,Q,[]).
12. gen([X|Xs],P,X,Ys) :-gen(Xs,N,X,Ys),P is max(N,X).
13. gen(([X|Xs],P,X,[X|Ys]) :-gen(Xs,N,Q,Ys),Q=\=X,P is max(N,X).
```

Le programme ainsi obtenu ne parcoure qu'une seule fois la liste comme on le voulait.

2.6 Le changement de représentation des données

Le choix de structures de données appropriées au problème est généralement très important pour écrire des programmes efficaces, on a même introduit une équation qui résume ceci : 'Algorithmes + Structures de Données = Programme'.
Toutefois, il est parfois difficile de connaître les structures de données adaptées à un problème avant d'avoir écrit un programme pour résoudre ce problème. La transformation de programme peut permettre de transformer les structures de données d'un programme pour le rendre plus efficace.
Par exemple, nous allons transformer un programme qui utilise des listes en un programme qui utilise les différence-listes. Une différence-liste est une paire de listes L-R, telle qu'il existe une troisième liste telle que lorsqu'on ajoute X et R, on retrouve L. Une liste peut être représentée par de nombreuses difference-listes.
L'utilisation des difference-listes en transformation de programme se fait souvent en introduisant une définition de la forme :

```
diffp(X,L-R):-p(X,Y),append(Y,R,L).
```

et en trouvant ensuite une définition récursive de diffp qui ne dépend ni de p ni de append.

Nous allons appliquer cette méthode au programme suivant qui retourne une liste :

```
1. reverse([],[]).
2. reverse ([H|T],R) :- reverse(T,V), append(V,[H],R).
3. append([],L,L).
4. append ([H|T],L,[H|S]) :- append(T,L,S).
```

Pour une liste de longueur n, la complexité du programme est en $O(n^2)$. On veut obtenir un programme qui ne traverse la liste Xs qu'une seule fois. Pour cela, on applique la stratégie du tupling aux prédicat maximal et delete qui partagent l'argument Xs.

Ce programme peut être amélioré en utilisant une difference-liste pour le second argument de reverse. Ceci permet d'améliorer le programme parce que la concaténation de difference-listes peut être beaucoup plus efficace que la concaténation de listes.
On introduit pour cela la définition :

```
5. diffrev(X,L-R):-reverse(X,Y),append(Y,R,L).
```

En dépliant diffrev, puis en utilisant sa définition, on trouve une définition récursive de diffrev ne contenant ni reverse ni append.

On déplie reverse :

```
6. diffrev([],L-R) :-append([],R,L).
7. diffrev([H|T],L-R):-reverse(T,V),append(V,[H],Y),append(Y,R,L).
```

On déplie la clause 6, on a :

```
8. diffrev([],R-R).
```

Par pliages et dépliages on peut prouver le remplacement suivant :

```
append(V,[H],Y),append(Y,R,L) ⇔ append(V,[H|R],L)
```

La clause 7 devient alors :

```
9. diffrev([H|T],L-R):-reverse(T,V),append(V,[H|R],L).
```

En pliant la clause 9 avec la clause 5 on obtient :

```
10. diffrev([H|T],L-R):- diffrev (T,L-[H|R]).
```

On définit reverse à l'aide de diffrev :

```
11. reverse(X,Y) :-diffrev(X,Y-[]).
```

Ce qui donne le programme final de complexité O(n) qui utilise les difference-listes:

```
11. reverse(X,Y) :-diffrev(X,Y-[]).
```

```
8. diffrev([],R-R).
10. diffrev([H|T],L-R):- diffrev (T,L-[H|R]).
```

2.7 Conclusion sur les transformations de programmes

Nous avons vu sur quelques exemples qu'à partir de quelques transformations de base assez simples, on peut améliorer de façon significative des programmes. Toutefois ces transformations très générales ne sont pas encore automatisables et font appel à ce qu'on appelle des prédicats 'euréka' qui doivent être donnés au programme pour qu'il arrive à effectuer des transformations intéressantes. L'outil est puissant mais encore mal maîtrisé. On peut toutefois l'utiliser dans un cadre plus restreint qui est celui de la spécialisation de programmes proprement dite. Cette restriction permet alors de créer des métaprogrammes de spécialisation entièrement indépendants. C'est ce que nous allons voir avec l'évaluation partielle.

3 L'EVALUATION PARTIELLE

3.1 Introduction

Le but de l'évaluation partielle est de transformer les programmes de façon à ôter les niveaux d'interprétations. Beaucoup de programmes qui ne sont pas des interpréteurs ont tout de même des niveaux d'interprétation, par exemple tous les programme dont le comportement est régit par des données extérieures. Ainsi les programmes suivants peuvent gagner à être évalué partiellement :

Reconnaissance de motifs : un programme général de reconnaissance de motifs peut être spécialisé sur une entrée particulière et reconnaître plus rapidement cette entrée.

Synthèse d'images : un programme de 'Ray tracing' recalcule souvent la façon dont les rayons de lumière traversent une scène donnée depuis plusieurs origines et dans différentes directions. Spécialiser un programme de 'Ray tracing' général sur une scène donnée pour le transforme en un 'ray tracer' spécialisé sur cette scène qui ne fonctionne que pour cette scène, donne un algorithme beaucoup plus rapide. De plus, la spécialisation ne demande que quelque minutes et rend le programme en moyenne 2 fois plus efficace, alors que le programme général peut demander des heures pour calculer une scène.

Bases de données : l'évaluation partielle peut permettre de compiler des requêtes vers des programmes spécialisés de recherche, dont la seule tâche est de répondre à la requête. Une autre application de l'évaluation partielle aux bases de données est la spécialisation d'un programme général de vérification de contraintes d'intégrité sur une base de données particulière.

Réseaux de neurones : le temps d'apprentissage est généralement long. On peut améliorer ce temps d'apprentissage en spécialisant un simulateur général de réseaux de neurones sur une topologie de réseau donnée.

Calculs scientifiques : On peut spécialiser des programmes généraux de calcul scientifique dont on connaît déjà certaines données d'entrée. On peut par exemple spécialiser la FFT sur un nombre de bits donné, ou spécialiser un programme de calcul d'orbites planétaires sur une système planétaire particulier, ou encore spécialiser une simulation générale de circuits électriques sur un circuit particulier.

Une propriété de l'évaluation partielle est qu'elle est automatique, bien que l'on se permette, notamment pour les langages impératifs d'annoter le code pour diriger l'évaluateur partiel.

On définit aussi l'évaluation partielle dans la cadre de la programmation logique :

> **Evaluation Partielle** : Soit P un programme et G un but. L'évaluation partielle produit un nouveau programme P', qui est P-spécialisé pour le but G. L'objectif de cette transformation est que le but G ait la même solution pour les programmes P et P', et que G s'exécute plus rapidement pour P' que pour P.

Lorsque l'évaluation partielle s'applique à des programmes logiques purs (sans cut, sans effets de bords, sans prédicats non-logiques) elle est aussi appelée **déduction partielle**. Nous étudierons principalement la déduction partielle dans le cadre des programmes logiques et l'évaluation partielle dans le cadre des programmes C.

3.2 Un exemple en C

Nous considérons une fonction C qui affiche un texte sous un certain format. Nous appelons cette fonction mini_printf parce qu'elle fait une partie du travail de la fonction printf.

```
void mini_printf(char *chaine,int *valeurs)
{
  int i;
  for (i=0; *chaine != '\0'; chaine++)
     if(*chaine != '%')
       putchar(*chaine);
     else
       switch(*++chaine)
       {
         case 'd': putint(valeur[i++]); break;
         case '%': putchar('%'); break;
         default : error();
       }
}
```

On se propose de spécialiser cette fonction sur la chaîne "n = %d".

Avant de spécialiser, on analyse dans le programme ce qui dépend des données sur lesquelles on spécialise. On notera avec des underscore ('_') les instructions, variables et expressions dont on ne connaît pas la valeur lors de la spécialisation dans le programme ci-dessus.

```
void _mini_printf(char *chaine,int *_valeurs)
{
  int i;
  for (i=0; *chaine != '\0'; chaine++)
     if(*chaine != '%')
       _putchar(*chaine);
     else
       switch(*++chaine)
       {
         case 'd': _putint(_valeur[i++]); break;
         case '%': _putchar('%'); break;
         default : _error();
       }
}
```

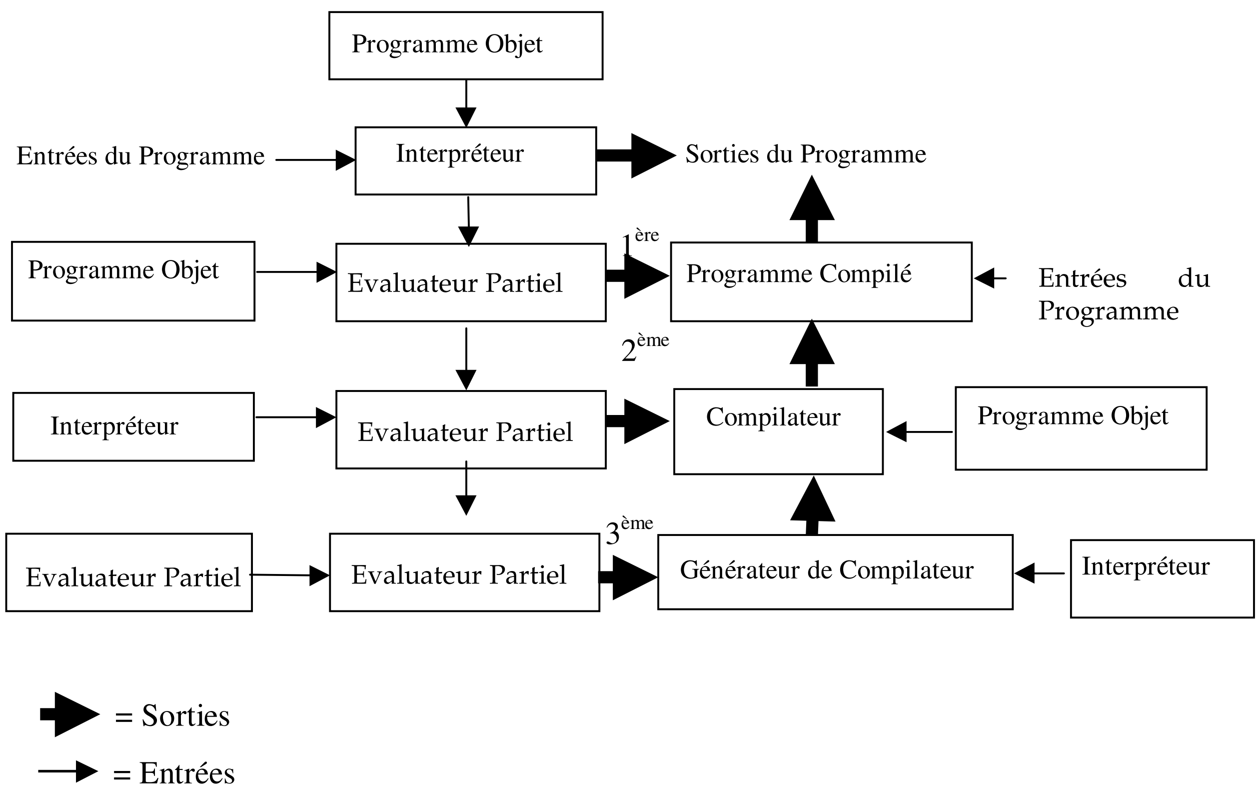
On spécialise alors la fonction mini_printf sur la chaine de caractères "n = %d" :

```
void mini_printf_chaine(int *valeurs)
{
  putchar('n');
  putchar(' ');
  putchar('=');
  putchar(' ');
  putchar(value[0]);
}
```

Le programme spécialisé s'éxecute plus rapidement que le programme original qui interprétait la chaîne à chaque appel.

3.3 Les équations de Futamura

Un évaluateur partiel [Consel et al. 93] [Gallagher 93] [Lloyd et al. 91] ou un système de déduction est dit auto-applicable s'il est capable de se spécialiser lui-même efficacement. Les intérêts d'une telle propriété sont multiples. Les plus connus sont les deuxièmes et troisièmes projections de Futamura [Futamura 71]. Les trois projections de Futamura sont données de façon graphique dans la figure suivante :



La première projection de Futamura consiste à spécialiser un interpréteur pour un programme objet particulier, produisant ainsi une version spécialisée de l'interpréteur qui peut être vue comme une compilation du programme objet.

Si l'évaluateur partiel peut s'appliquer à lui-même, on peut alors spécialiser l'évaluateur partiel sur l'interpréteur afin qu'il soit spécialisé dans la première projection de Futamura. On obtient ainsi un compilateur correspondant à l'interpréteur donné en entrée.

La troisième projection de Futamura consiste à spécialiser l'évaluateur partiel pour faire la deuxième projection de Futamura. En spécialisant l'évaluateur partiel sur l'évaluateur partiel, on obtient alors un générateur de compilateur. C'est à dire un programme qui transforme un interpréteur en compilateur.

L'idée principale qui permet l'auto-application est de séparer la spécialisation en deux phases distinctes :

- en premier, une analyse du programme à spécialiser qui détermine les valeurs qui pourront être spécialisées, cette phase est appelée Binding-Time Analysis(BTA).
- en second, une phase simplifiées de spécialisation, guidée par les résultats de la BTA.

Cette approche de l'évaluation partielle est dite off-line, parce que les décision de contrôle sont prises avant la spécialisation. Elle est mise en contraste par l'approche on-line qui décide au fur et à mesure de la spécialisation ce qui va être spécialisé. Les méthodes de spécialisation on-line sont plus difficiles à auto-appliquer.
Dans le cadre de la programmation logique, le langage Goëdel [Hill et al. 94], [Hill et al. 98] a été développé pour faciliter la métaprogrammation en logique[Barklund 94]. Les trois projections de Futamura ont été effectuées pour le langage Goëdel avec l'évaluateur partiel SAGE [Gurr 93].

3.4 Application au Pattern Matching

Soit le programme suivant qui vérifie si la chaîne de caractères P est présente dans la chaîne S :

```
1. match(P,S):-aux(P,S,P,S).
2. aux([],X,Y,Z).
3. aux([A|Ps],[A|Ss],P,S):-aux(Ps,Ss,P,S).
4. aux([A|Ps],[B|Ss],P,[C|S]):-A=\=B,aux(P,S,P,S).
```

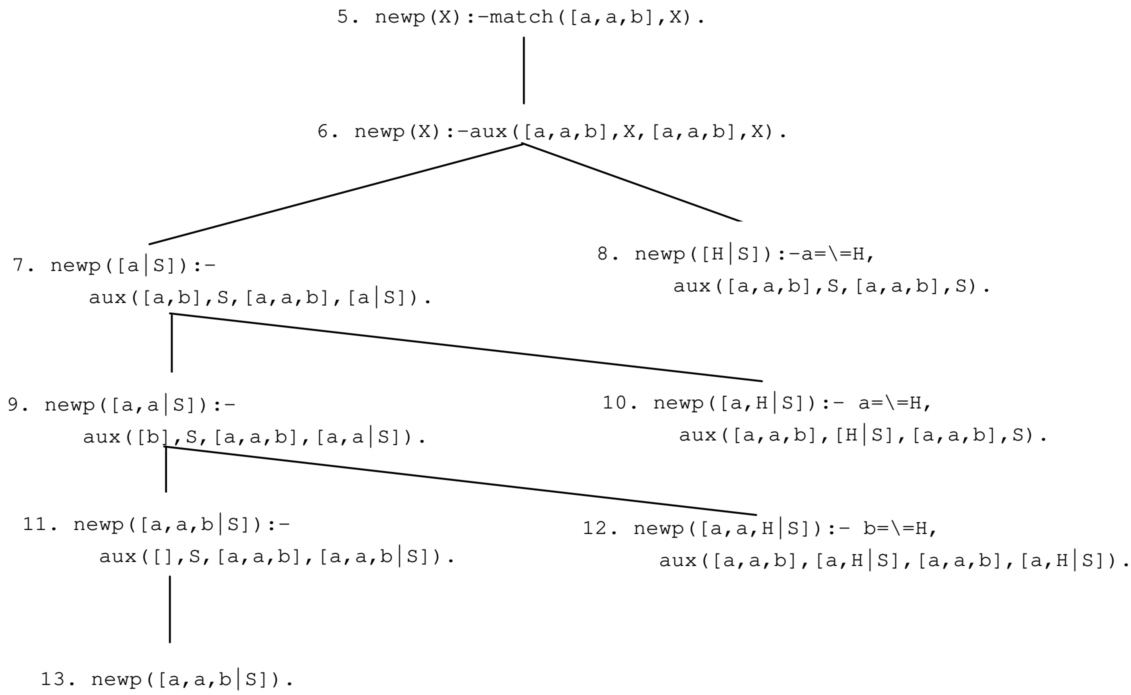
Par exemple, la chaîne [a,b] apparaît dans la chaîne [c,a,b] mais pas dans la chaîne [a,c,b].
Nous allons maintenant évaluer partiellement ce programme sur le but match([a,a,b],X). Pour cela, on commence par introduire une nouvelle définition :

```
5. newp(X):-match([a,a,b],X).
```

On va maintenant déplier ce prédicat newp avec la stratégie de dépliage suivante :

- déplier dans les clauses tous les atomes de la forme match(…) ou aux(…).
- ne pas déplier une clause dont le corps contient une instance d'un atome qui apparaît dans le corps d'une clause parente dans l'arbre de dépliage.

On obtient l'arbre de dépliage suivant :



Les atomes des clauses 8, 10 et 12 sont des instances du corps de la clause 6. Le dépliage est donc arrêté. On définit alors le prédicat newq qui permet de rendre les clauses récursives terminales :

```
14. newq(S):-aux([a,a,b],S,[a,a,b],S).
```

En pliant la clause 6 avec la clause 14, on obtient :

```
6f. newp(X):-newq(X).
```

Quand on déplie le programme sur la clause newq, On obtient le même arbre que précédemment avec les newp remplacés par des newq :

```
13q. newq([a,a,b|S]).
12q. newq([a,a,H|S]):-b=\=H,aux([a,a,b],[a,H|S],[a,a,b],[a,H|S]).
10q. newq([a,H|S]):-a=\=H,aux([a,a,b],[H|S],[a,a,b],[H|S]).
8q. newq([a,H|S]):-a=\=H,aux([a,a,b],S,[a,a,b],S).
```

Ce qui permet d'obtenir lorsqu'on plie les clauses du programme ainsi obtenu avec la définition de newq.

```
6f. newp(X):-newq(X).
13qf. newq([a,a,b|S]).
12qf. newq([a,a,H|S]):-b=\=H,newq([a,H|S]).
10qf. newq([a,H|S]):-a=\=H, newq ([H|S]).
8qf. newq([a,H|S]):-a=\=H, newq (S).
```

C'est le programme qui est obtenu avec Mixtus [Sahlin 91], l'évaluateur partiel standard pour Prolog.

3.5 Déduction partielle d'un parser

Soit le programme suivant qui parse les mots d'un langage non contextuel :

```
1. parse(Grammar,[Symb],[Symb|X]-X):-terminal(Symb).
2. parse(Grammar,[Symb],Word):-nonterminal(Symb),
                              member(Symb->Symbs,Grammar),
                              parse(Grammar,Symbs,Word).
3. parse(Grammar,[Symb1,Symb2|Symbs],WordX-X):-
                              parse(Grammar,[Symb1],WordX-Y).
                              parse(Grammar,[Symb2],Y-X).
4. terminal(0).
5. terminal(1).
6. nonterminal(s).
7. nonterminal(u).
```

Le premier argument de parse est une grammaire représentée par une liste de productions de la forme Symb→Symbs, où Symb est un symbole non terminal et Symbs est une séquence de symboles terminaux et non terminaux. Le second argument de parse est la chaîne générée par parse représentée par une liste. Le troisième élément est le mot à parser.

On va spécialiser ce programme général sur la grammaire suivante :

```
s→0u
u→1
u→0uu
u→1s
```

On veut donc spécialiser le programme parse sur le but

```
parse([s→[0,u],u→[1],u→[0,u,u],u→[1,s]],[s],X-[]).
```

on notera désormais Γ la liste [s→[0,u],u→[1],u→[0,u,u],u→[1,s]].

En appliquant la stratégie de dépliage définie au paragraphe précédent, On obtient :

```
parse(Γ,[s],[0|Z]-Y):-parse(Γ,[u],Z-Y).
parse(Γ,[u],[1|Y]-Y).
parse(Γ,[u],[0|Z]-Y):-parse(Γ,[u],Z-V),parse(Γ,[u],V-Y).
parse(Γ,[u],[1|Z]-Y):-parse(Γ,[s],Z-Y).
```

La spécialisation du parser général sur une grammaire particulière permet de le rendre bien plus efficace.

4 SPECIALISATION DE PROGRAMMES C

Il existe actuellement deux évaluateurs partiels pour le langage C : C-Mix a DIKU au Danemark et Tempo à l'IRISA à Rennes. L'évaluation partielle de langages impératifs est aussi décrite dans [Gomard et al. 93]. C-Mix a été appliqué avec succès à la spécialisation d'un ray tracer sur une scène donnée. Il est actuellement en accès libre et peut être utilisé pour un grand nombre de programmes C. Une autre de ses applications a été la spécialisation de fonctions mathématiques sur certaines données d'entrées. Par exemple la spécialisation d'une fonction calculant une FFT sur un nombre d'entrées égal à 512, ou encore la spécialisation du calcul de l'intégrale de Romberg ou le calcul des polynômes de Tchebyshev.
Tempo quant à lui a été appliqué avec succès à la spécialisation de la couche RPC du système Unix ainsi

qu'à d'autres types de programmes C.

5 APPLICATIONS POUR L'INTELLIGENCE ARTIFICIELLE

Le parallèle entre l'évaluation partielle et l'apprentissage par explication a été effectué par Franck Van Harmelen [Van Harmelen et al. 88]. L'apprentissage par explication [Mitchell et al. 86] est une spécialisation dynamique de programme qui tend à rendre les programmes plus efficace lorsqu'elle est utilisée convenablement, on peut par exemple apprendre à atteindre des buts aux échecs [Pitrat 76] ou au jeu de Go [Cazenave 98].

Il existe aussi des liens importants entre spécialisation de programmes, métaprogrammation et métaconnaissances [Pitrat 90]. D'une façon générale, la spécialisation de programmes peut être utilisée par les chercheurs en Intelligence Artificielle pour engendrer des programmes efficaces à partir d'un programme générique et simple.

Il peut y avoir des applications en langage naturel comme le montre l'exemple de la section 3 sur la déduction partielle d'un parser, mais aussi en planification. Ainsi Augustsson utilise l'évaluation partielle pour accélérer la planification des équipages d'avions [Augustsson 97]. Une autre utilisation déjà citée est la spécialisation d'un simulateur de réseaux de neurones sur une architecture de réseau particulière.

On peut aussi noter la similarité des idées sur lesquelles est basée l'évaluation partielle avec des idées qui ont donné des résultats particulièrement bons en Intelligence Artificielle. Ainsi, le programme de K. Thompson de calcul des bases de données de finales d'Echecs, de même que le programme de J. Schaeffer pour les finales de Checkers peuvent être vus comme des évaluations partielles de la définition du gain aux Echecs ou aux Checkers en utilisant les règles du jeu. Ces programmes précalculent des positions au lieu d'avoir à les recalculer à chaque fois, ce qui leur fait gagner du temps.

6 BIBLIOGRAPHIE

[Augustsson 97] Augustsson L. 1997. *Partial Evaluation in aircraft crew planning*. http://www.carlstedt.se/~augustss.

[Barklund 94] Barklund J. 1994. *Metaprogramming in Logic*. UPMAIL Technical Report N° 80, Uppsala, Sweden, 1994.

[Consel et al. 93] Consel C., Danvy O. 1993. *Tutorial Notes on Partial Evaluation*. Proceedings of the ACM SIGPLAN Symposium on POPL'93. Charleston, 1993.

[Cazenave 98] Cazenave T. 1998. *Metaprogramming Forced Moves*, Proceedings ECAI-98, pp. 645-649. Brighton, 1998.

[Futamura 71] Futamura Y. 1971. *Partial Evaluation of computation process – an approach to a compiler compiler.* Systems, Computers, Controls 2 (5), 45-50.

[Gallagher 93] Gallagher J. 1993. *Specialization of Logic Programs.* Proceedings of the ACM SIGPLAN Symposium on PEPM'93, Ed. David Schmidt, ACM Press, Copenhagen, Danemark, 1993.

[Gomard et al. 93] Gomard, Jones, Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall 1993

[Gurr 93] Gurr C. 1993. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Dept. of Computer Science, University of Bristol.

[Hill et al. 94] Hill P. M. and Lloyd J. W. 1994. *The Gödel Programming Language.* MIT Press, Cambridge, Mass., 1994.

[Hill et al. 98] Hill P. M. and Gallagher J. G. 1998. *Meta-Programming in Logic Programming*. Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, Oxford University Press, 1998.

[Lloyd et al. 91] Lloyd J. W. and Shepherdson J. C. 1991. *Partial Evaluation in Logic Programming.* J. Logic Programming, 11 :217-242., 1991.

[Mitchell et al. 86] Mitchell, T. M.; Keller, R. M. and Kedar-Kabelli S. T. 1986. Explanation-based Generalization : A unifying view. *Machine Learning* 1 (1), 1986.

[Pitrat 76] Pitrat J. 1976. Realization of a Program Learning to Find Combinations at Chess. Computer Oriented Learning Processes, J. C. Simon editor. NATO Advanced Study Institutes Series. Series E: Applied Science - N° 14. Noordhoff, Leyden, 1976.

[Pitrat 90] Pitrat, J. 1990. *Métaconnaissance - Futur de l'Intelligence Artificielle*. Hermès, Paris.

[Pettorossi et al. 96] Pettorossi A., Proietti M. 1996. *A Comparative Revisitation of Some Program Transformation Techniques*. LNCS 1110, pp. 355-385.

[Pettorossi et al. 98] Pettorossi A., Proietti M. 1998. *Transformation of Logic Programs*. Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, Oxford University Press, 1998, pp. 697-787.

[Sahlin 91] Sahlin D. 1991. An Automatic Partial Evaluator for Full Prolog. PhD thesis, Dept. of Telecommunication and Computer Systems, Royal Institute of Technology, Stockholm, Sweden.

[Tamaki et al. 84] Tamaki H. and Sato T. 1984. *Unfold/Fold Transformations of Logic Programs*. Proc. 2nd Intl. Logic Programming Conf., Uppsala Univ., 1984.

[Van Harmelen et al. 88] Van Harmelen F. and Bundy A. 1988. *Explanation based generalisation = partial evaluation*. Artificial Intelligence 36:401-412, 1988.