Une transformation de programme basée sur la notion d'équations entre termes *

Ulrich Neumerkel

Institut für Computersprachen
Technische Universität Wien
ulrich@mips.complang.tuwien.ac.at

Résumé

Nous décrivons une nouvelle approche destinée à transformer des programmes Prolog en programmes plus efficaces tout en gardant l'équivalence opérationnelle. Notre approche repose sur l'introduction d'équations entre termes, ce qui permet une représentation plus efficace des buts. Notre optimisation ne peut pas être obtenue par les méthodes courantes : en particulier, elle élimine toutes les variables existentielles des listes par différence dans les grammaires. Nous comparerons ensuite notre transformation avec les transformations courantes basées sur le pliage et le dépliage. Si nous obtenons des résultats comparables, notre transformation est plus simple à mettre en œuvre. De plus, elle se révélera utile comme première étape avant le pliage/dépliage. En ce qui concerne la machine abstraite de Warren (WAM), l'élimination des variables existentielles d'un programme avec notre transformation a pour conséquence une allocation de registres entre les procédures. Nous montrerons l'utilité de nos transformations pour les programmes en Prolog binaire [14] et pour les systèmes Prolog sans pile d'environnement [6].

Mots-clés : Prolog binaire, transformation, continuations, grammaires de métamorphose

Plan. Cet article comprend 6 chapitres. Le premier chapitre présente un exemple préliminaire. Le deuxième est consacré à l'étude du noyau de transformation EBC (*equality based continuation transformation*) développant la notion de continuation. Le 3ème décrit un schéma d'implantation des grammaires de métamorphose avec notre méthode, suivi d'une analyse de la consommation mémoire. On trouvera dans le chapitre 4 une analyse plus générale des différentes machines virtuelles. Le chapitre 5 compare la transformation EBC et la technique de pliage et dépliage : quoique de nature différente, ces transformations présentent des cas communs. Certaines applications s'écartant du schéma décrit dans le chapitre 2 nous affinerons les transformations dans le chapitre 6.
Conventions pour la présentation des programmes : l'argument de la structure principale contenant la continuation est souligné et les **parties nouvelles** sont écrites en caractères gras.

*Secondes journées francophones sur la programmation en logique, Nîmes-Avingnon, (JFPL'93).

1 Un exemple démonstratif

Nous commençons par la dérivation en cinq étapes d'un programme très simple contenant une variable existentielle. Le prédicat ≪ expression/3 ≫ décrit la relation entre une liste en différence de symboles terminaux et l'arbre syntaxique correspondant. On remarquera que même pour cette grammaire très simple les approches traditionnelles sont incapables d'éliminer la variable existentielle ≪ Xs1 ≫.

```
expression(terminal(T)) ⟶
    [terminal(T)].
expression(nœud(Gauche,Droite)) ⟶
    [opérateur],
    expression(Gauche),
    expression(Droite).
```

```
expression(terminal(T), [terminal(T)|Xs],Xs).
expression(nœud(Gauche,Droite), [opérateur|Xs0],Xs) ←
    expression(Gauche, Xs0,Xs1),
    expression(Droite, Xs1,Xs).
```

1ère étape : passage à la forme binaire. Tout d'abord, le programme est converti en forme binaire grâce à la transformation décrite par Tarau [15]. Dans cette forme un nouvel argument est ajouté pour représenter de manière explicite la continuation. Dans le corps, les appels de prédicats qui suivent le premier sont codés comme des termes fonctionnels placés en argument du premier but. Ils correspondent à la notion de fermeture dans les langages fonctionnels [1].

```
expression(terminal(T), [terminal(T)|Xs],Xs, Cont) ←
    demo(Cont).
expression(nœud(Gauche,Droite), [opérateur|Xs0],Xs, Cont) ←
    expression(Gauche, Xs0,Xs1, expression(Droite, Xs1,Xs, Cont)).
```

2ème étape : séparation de l'argument de sortie. L'équation des termes ci-dessus introduit deux nouvelles structures. Les fonctions ≪ expression/4 ≫ sont remplacées par une représentation simplifiée. Un nouveau symbole ≪ expression/4 ≫ sert de jonction entre les prédicats externes et notre prédicat transformé ≪ expression/3 ≫.

```
expression(T, Xs0,Xs, Cont) ≐ expression(T, Xs0, reliquat(Xs, Cont)).

expression(T, Xs0,Xs, Cont) ←
    expression(T, Xs0, reliquat(Xs, Cont)).

expression(terminal(T), [terminal(T)|Xs], reliquat(Xs, Cont)) ←
    demo(Cont).
expression(nœud(Gauche,Droite), [opérateur|Xs0], reliquat(Xs, Cont)) ←
    expression(Gauche, Xs0, reliquat(Xs1, expression(Droite, Xs1, reliquat(Xs, Cont)))).
```

3ème étape : simplification de la continuation. La structure ≪ reliquat/2 ≫ est source de quelques redondances dans la règle. La structure ≪ reliquat(Xs,Cont) ≫ est éliminée parce qu'elle n'apporte rien: ni échec, ni création de nouvelle structure, ni liaison. La continuation de la règle peut donc être simplifiée.

```
expression(T, Xs0,Xs, Cont) ←
    expression(T, Xs0, reliquat(Xs, Cont)).

expression(terminal(T), [terminal(T)|Xs], reliquat(Xs, Cont)) ←
    demo(Cont).
expression(nœud(Gauche,Droite), [opérateur|Xs0], ReliquatXsCont) ←
    expression(Gauche, Xs0, reliquat(Xs1, expression(Droite, Xs1, ReliquatXsCont))).
```

4ème étape : définition d'un prédicat séparé pour l'exécution de la continuation. Afin de simplifier et d'augmenter l'efficacité du programme, on combine l'appel au prédicat « demo/1 » et l'unification de la structure de continuation « reliquat/2 » en créant dans le nouveau prédicat « démontre_reliquat/2 ».

```
expression(T, Xs0,Xs, Cont) ←
    expression(T, Xs0, reliquat(Xs, Cont)).

expression(terminal(T), [terminal(T)|Xs], ReliquatXsCont) ←
    démontre_reliquat(ReliquatXsCont, Xs).
expression(nœud(Gauche,Droite), [opérateur|Xs0], ReliquatXsCont) ←
    expression(Gauche, Xs0, reliquat(Xs1, expression(Droite, Xs1, ReliquatXsCont))).

démontre_reliquat(reliquat(Xs, Cont), Xs) ←
    demo(Cont).
```

5ème étape : élimination des variables existentielles. La première équation simplifie les structures et réduit la variable existentielle « Xs1 » à une variable vide. La deuxième équation ne contient la variable « Xs1 » qu'une fois. Elle n'est donc pas valide pour les termes généraux mais seulement pour les variables vides dans les continuations qui ne sont lues qu'une fois. La structure extérieure « reliquat/2 » est lue dans « démontre_reliquat/2 ». Nous devons donc réécrire ce prédicat pour prendre en compte la représentation modifiée. L'introduction d'une nouvelle règle ne crée pas de nouveaux points de choix car un but ne peut pas avoir une variable comme continuation.

```
reliquat(Xs1, expression(Droite, Xs1, ReliquatXsCont)) ≐ reliquat_expression(Droite, Xs1, ReliquatXsCont).
reliquat_expression(Droite, Xs1, ReliquatXsCont) ≐ reliquat_expression(Droite, ReliquatXsCont).

expression(T, Xs0,Xs, Cont) ←
    expression(T, Xs0, reliquat(Xs, Cont)).

expression(terminal(T), [terminal(T)|Xs], ReliquatXsCont) ←
    démontre_reliquat(ReliquatXsCont, Xs).
expression(nœud(Gauche,Droite), [opérateur|Xs0], ReliquatXsCont) ←
    expression(Gauche, Xs0, reliquat_expression(Droite, ReliquatXsCont)).

démontre_reliquat(reliquat(Xs, Cont), Xs) ←
    demo(Cont).
démontre_reliquat(reliquat_expression(Droite, ReliquatXsCont), Xs) ←
    expression(Droite, Xs, ReliquatXsCont).
```

Le programme analyse les premières 2551 phrases 67% plus vite avec BinProlog 1.39 et 30% plus vite avec le compilateur SICStus Prolog 2.1. Les mesures ont été déterminées dans SPARCstation ELC, (unité central 33MHz Cypress, 8Mo RAM).

2 La transformation EBC

Afin de varier la représentation du programme nous avons besoin d'un formalisme qui permet d'exprimer les changements. Notamment nous ne nous intéresserons qu'aux changements gardant la sémantique opérationnelle du programme existant. En Prolog,

l'unification exprime l'égalité entre termes. Avec des équations $\ll s \doteq t \gg$ nous exprimons qu'un terme s possède une représentation alternative t. Par exemple, dans un programme P_0 ayant la fonction $\ll$ f/2 $\gg$, et n'ayant pas $\ll$ g/2 $\gg$ et $\ll$ h/1 $\gg$ l'équation $\ll$ f(A,B) $\doteq$ g(B,h(A)) $\gg$ permet d'utiliser soit $\ll$ f(A,B) $\gg$ soit $\ll$ g(B,h(A)) $\gg$ comme représentations équivalentes dans un programme nouveau P_1 contenant les nouvelles fonctions. Nous notons que notre approche à l'unification est bien différente à l'approche des contraintes. Le langage à optimiser est pour nous Prolog lui-même sans aucune extension. Cependant, les contraintes sont les extensions visibles pour l'utilisateur.

2.1 Extension d'unification syntaxique

Nous utilisons surtout des ensembles E contenants les équations $\ll s \doteq t \gg$ et définissants l'unification equationelle $=_E$. Au début, on a le programme original P_0 et l'unification syntaxique ($E_0 = \{\}$). Pendant chaque étape nous raffinons la représentation du programme P_i et les équations E_i obtenant un programme P_{i+1} avec E_{i+1}, $E_i \subseteq E_{i+1}$, comme suit :

1. Dans l'ensemble $E_{i+1} - E_i$ on a les nouvelles équations de la forme $t_i \doteq t_{i+1}$.

   t_i est construite avec les termes de P_i;

   t_{i+1} est construite avec les termes de P_{i+1}.

2. La relation de l'inégalité est préservée : Pour tous les termes s, t de P_i : $s \neq_{E_i} t$ implique $s \neq_{E_{i+1}} t$. Donc, le comportement de l'unification ne change pas à cause de nouvelles équations. Pour le cas général nous devrions démontrer que les équations $E_{i+1} - E_i$ ne changent pas le comportement de l'unification.

3. $s =_{E_{i+1}} t$ est décidable.

L'unification avec des équations additionnelles nécessite un appareillage technique nettement plus lourd que celui employé pour l'unification syntaxique. L'effacement des buts peut être ralenti par l'algorithme de l'unification plus complexe. Ceci rendrait notre transformation inutilisable. Le remède serait pire que le mal. Par la suite, nous nous limiterons aux programmes pour lesquels on peut implanter les équations avec un coût raisonnable utilisant l'unification syntaxique.

2.2 La continuation

Nous développons une notion de continuation pouvant être utilisée pour l'analyse de programme. Pour nous, une continuation est un terme identifiable (par exemple, par la position d'argument, par le nom de fonction etc.) qui est d'une structure restreinte et qui n'est utilisé que d'une manière restreinte.

Termes clos. Pour les termes clos l'algorithme de l'unification avec des équations convergentes est une simple réécriture des termes. Une source d'inefficacité reste quand même : Cet algorithme s'implante par un prédicat récursif.

Continuations closes. Les continuations closes sont les termes clos qui ne sont qu'unifiés, au fur et à mesure que l'effacement se déroule, avec des autres termes t d'une forme comme suit: t ne contient qu'une seulle occurence d'une variable et t est d'une

taille constante. Il est donc impossible de raisonner dans le programme à transformer sur la continuation avec l'unification générale. Par exemple, il est impossible de comparer deux continuations en utilisant l'unification elle-même. Pour notre propos il suffit d'exiger qu'une continuation n'apparaisse pas dans une variable qui se trouve deux fois dans la tête d'une règle.

Continuations génerales. Un programme avec les continuations génerales correspond au programme avec les continuations closes par une projection négligeant les arguments sans continuations. Par exemple, un programme binaire P correspond au programme Q avec une continuation close par la projection $f(a_1, \ldots, a_n, c) \mapsto f(c)$. pour chaque fonction f.

L'analyse des continuations. Dans les programmes binaires l'analyse des continuations n'est nécessaire que pour les continuations réalisées par le programmeur (Cf. l'interface *make/next/done* [10] ou les monades [16]). Nous avons implanté une analyse très simple détectant quelques fonctions et arguments de continuation. L'analyse manquant de points de programme est très imprécise : Les listes ne sont jamais détectées comme continuation (Cf. « dcg_spécialisée/3 ») parce qu'elles servent à représenter les structures de données différentes. Nous pensons qu'il est préférable de renommer les symboles fonctionnels représentant les continuations, plutot que de faire une analyse plus complexe et coûteuse. Afin de traduire des programmes logiques en forme binaire il suffit de générer les continuations avec des noms internes. Pour les continuations programmées d'une manière explicite un système de modules basés sur les noms obtient le même résultat. Cela encourage en passant l'utilisation de la notion du type abstrait.

2.3 Compilation des équations

Nous ne faisons aucune différence entre les symboles fonctionnels et les formules atomiques. On peut s'imaginer que toutes les clauses soient représentées avec une fermeture définie (Cf. [15], définition 9). Un seul prédicat binaire reste à transformer.

Compilation des buts. Les buts créant de nouvelles structures ont le choix entre les représentations différentes. L'équation $s \doteq t$ peut réécrire un but b, si pour sous-terme u :

- $s\theta = u\theta, b = b\theta$, pour $\mathrm{VAR}(s) = \mathrm{VAR}(t)$.
- $s = u$, pour $\mathrm{VAR}(t) \subset \mathrm{VAR}(s)$, et les variables $v \in \mathrm{VAR}(s) - \mathrm{VAR}(t)$ sont vides.

Par exemple, avec « a(X, b(X, Cont)) ≐ ab(Cont). » nous pouvons réécrire « ← p(a(X, b(X, Cont))). » à « ← p(ab(Cont)). ». Mais on ne peut pas réécrire « ← p(a(X, b(Y, Cont))). ».

Compilation des têtes. Les têtes lisent les continuations. Elles sont responsables de reconnaître toutes les représentations créées par les buts. Pour une règle r nous créons pour chaque réécriture de tête une règle alternative r_i. Notons, que de nouvelles liaisons sont introduites. Une réécriture est invalide, si l'effacement peut sélecter deux règles différentes r_j et r_k. Pour les cas pratiques, il suffit d'analyser si les règles se distinguent déjà dans les positions des continuations ou sous-continuations.

Il faut remarquer que notre unification compilée peut agrandir un programme à la taille exponentielle. Pour illustrer ce phénomène dans le pire des cas, supposons le fait

« pire($\underline{s^n(0)}$). » et l'équation « s($\underline{X}$) $\doteq$ t($\underline{X}$) ». La réécriture engendre 2^n faits « pire/1 » équivalents. Pour nos besoins, ce cas n'existe pas parce que nous ne considérons que les continuations dans les programmes binaires. Ici, nous avons un seul terme structuré dans la tête des règles d'un prédicat unique (*meta-call*) fournissant l'effacement du premier but de la continuation.

Simplification des sous-continuations par généralisation. La réécriture à l'aide des équations introduit des redondances structurelles qu'on peut éliminer ensuite : a) La sous-continuation s apparaissant dans la tête et dans le but est inutile si leurs variables VAR(s) ne se trouvent que dans s et si s est créé par une équation. b) On peut partager les occurrences d'une variable en plusieurs variables si les variables sont produites par une équation redoublant la variable.

Raccourcis. Afin de raccourcir la dérivation nous utiliserons $r \doteq t$ en place de $r \doteq s$ et $s \doteq t$. Pour éviter le redoublement des clauses, nous introduisons une règle échangeant la représentation ancienne contre la représentation nouvelle. Par exemple, ayant l'équation « p($\underline{\text{Cont}}$) $\doteq$ q($\underline{\text{Cont}}$) » le paquet des clauses du prédicat « p/1 » est échangé contre le paquet du « q/1 » en y ajoutant une règle en raccourci « p($\underline{\text{Cont}}$) $\leftarrow$ q($\underline{\text{Cont}}$). »

2.4 Schémas des équations utiles

Les types d'équations $s \doteq t$ suivants ont été conçus de façon à pouvoir être utilisée pour l'allocation de registres et la restitution de valeurs (Cf. 6.2). La position de la (sous-) continuation est occupée par une variable apparaissant exactement une fois dans chaque côté de l'équation. Les points « … » sont partout des variables.

1. VAR(s) = VAR(t).

   (a) $s \doteq u(\ldots, \underline{g(\ldots)}, \ldots)$.
   u n'apparaît que dans une seule équation. g n'apparaît que dans les équations du même type d'étape actuelle. L'argument de continuation dans g se trouve dans toutes les équations sur la même position. Nous utiliserons ces équations pour passer des sous-termes aux continuations. Avec ces équations nous diminuons les arguments d'un prédicat.
   « expression(T, Xs0,Xs, $\underline{\text{Cont}}$) $\doteq$ expression(T, Xs0, $\underline{\text{reliquat(Xs, Cont)}}$). »

   (b) $f(\ldots, \underline{g(\ldots)}, \ldots) \doteq f(\ldots, \underline{h(\ldots, g(\ldots), \ldots)}, \ldots)$.
   f et g sont les fonctions existantes. h n'apparaît que dans les équations du même type. Nous utiliserons ces équations pour exprimer les dépendances de contexte, indispensables pour la restitution de valeurs.
   « factorielle(N0, $\underline{\text{résmult(N,RCont)}}$) $\doteq$ factorielle(N0, $\underline{\text{aux(N0, résmult(N,RCont))}}$). »

   (c) $t = u(a_1, \ldots, a_n), \quad a_i \in \text{VAR}(s)$.
   Ces équations serviront à contracter des termes redondants.
   « aux(N, $\underline{\text{résmult(s(N), RCont)}}$) $\doteq$ auxrésmult(N,$\underline{\text{RCont}}$). »

2. VAR(s) $\supset$ VAR(t). $f(a_1, \ldots, a_n) \doteq g(b_1, \ldots, b_m), \quad a_i \in \text{VAR}(s), b_j \in \text{VAR}(s)$.
   Cette équation élimine les variables vides dans les buts.

3 La traduction des grammaires avec EBC

Les grammaires de métamorphose ont été conçues par Colmerauer [4]. Elles sont bien connues depuis leur vulgarisation par Pereira et Warren sous le nom de grammaire à clauses définies (DCG) [11]. Les grammaires bénéficient particulièrement de l'effet de EBC. Avant la présentation du cas général, nous présenterons la traduction de DCG.

3.1 Grammaires à clauses définies

Nous décrivons d'abord les grammaires sous la forme d'un méta-interpréteur dans le but d'exposer la transformation essentielle. Le programme à interpréter est représenté à l'aide d'une collection de faits définissant le prédicat « dcg_règle/2 ».

```
dcg_phrase([NT|NTs1], Cs0,Cs) ←
    not prédéfini(NT),
    dcg_règle(NT, MTs),
    conc(MTs, NTs1, NTs0),
    dcg_phrase(NTs0, Cs0,Cs).
dcg_phrase([{G}|NTs], Cs0,Cs) ←
    G,
    dcg_phrase(NTs, Cs0,Cs).
dcg_phrase([Terminaux|NTs], Cs0,Cs) ←
    conc(Terminaux, Cs1,Cs0),
    dcg_phrase(NTs, Cs1,Cs).
dcg_phrase([], Cs,Cs).
```

La première règle efface un non-terminal. Ce méta-interpréteur contient plusieurs variables existentielles qu'on peut éliminer avec les transformations traditionnelles. Par exemple le logiciel appelé $\mathcal{T}$ (*big-Tau*) créé par Neumann [8], étant un générateur de compilateurs acceptant un interpréteur annoté, est déjà capable d'éliminer toutes les variables existentielles générant un programme qui utilise un argument ayant le rôle d'une continuation :

```
dcg_spécialisée([], Cs,Cs).
dcg_spécialisée([expression(Arbre)|NTs], Cs0,Cs) ←
    dcg_expression(Arbre, Cs0,Cs, NTs).

dcg_expression(terminal(T), [terminal(T)|Cs0],Cs, NTs) ←
    dcg_spécialisée(NTs, Cs0,Cs).
dcg_expression(nœud(Gauche,Droite), [opérateur|Cs0],Cs, NTs) ←
    dcg_expression(Gauche, Cs0,Cs, [expression(Droite)|NTs]).
```

Les différences entre ce programme et notre spécialisation sont très faibles : Le programme utilise une continuation plus épaisse au niveau de l'utilisateur. La liste en différence résiste à la transformation.

Schéma de la traduction. Chaque règle de la grammaire est traduite dans une seule règle (Fig. 1). De plus, un seul prédicat auxiliaire est créé pour faciliter l'appel aux continuations transformées :

```
démontre(g(A1, ..., AN, Cont), L) ←
    dcg_g(A1, ..., AN, L, Cont).
```

Les prédicats prédéfinis ainsi que les opérations d'arithmétique sont intégrés aux règles comme suit :

$\ll$ phrase(g(A1, ..., AN), L0,L) $\gg$ $\ll$ dcg_g(A1, ..., AN, L0, reliquat(L, Cont)) $\gg$
$\ll$ phrase(g(A1, ..., AN), L0,[]) $\gg$ $\ll$ dcg_g(A1, ..., AN, L0, reliquat(Cont)) $\gg$
$\ll$ g(A1, ..., AN) $\longrightarrow$ ... $\gg$ $\ll$ dcg_g(A1, ..., AN, L,Cont) $\leftarrow$ ... $\gg$
$\ll$ ... $\longrightarrow$ ..., g(A1, ..., AN), ... $\gg$ $\ll$ ... g(A1, ..., AN, Cont) $\gg$
$\ll$ ... $\longrightarrow$ {p(E1, ..., EN)}. $\gg$ $\ll$ ... $\leftarrow$ p(E1, ..., EN, démontre(Cont, L)). $\gg$
$\ll$ ... $\longrightarrow$ {p(E1, ..., EN)}, g(A1, ..., AN), ... $\gg$ $\ll$ ... $\leftarrow$ p(E1, ..., EN, dcg_g(A1, ..., AN, L, Cont)). $\gg$
$\ll$ ... $\longrightarrow$ ..., {p(E1, ..., EN)}, ... $\gg$ $\ll$ ... $\leftarrow$ ..... reliquat(L, p(E1, ..., EN, Cont)).... $\gg$

Figure 1: Schéma de traduction

```
bip_multiplication(R, A, B) ⟶
    {R is A * B}.

dcg_bib_multiplication(R, A, B, L, Cont) ←
    R is A * B,
    démontre(Cont, L).
```

Estimation de l'économie en mémoire. Nous avons estimé le coût d'entassement pour les grammaires et pour les formalismes similaires qui utilisent n états implicites correspondant aux $2n$ arguments de l'utilisateur. Notons que n est égal à 1 pour les grammaires. Donc, dans certains cas le gain disparaît, notamment pour les utilisations de prédicats en dehors de la grammaire. Ce qui n'est pas une surprise puisque ces prédicats interrompent l'utilisation de registres allouées globalement. Pour les cas non mentionnés par exemple $\ll$ p $\longrightarrow$ [x]. $\gg$ l'entassement est le même. Nous supposons que l'entassement des continuations coûterait une cellule pour le symbole fonctionnel, une cellule pour la continuation et pour chaque argument d'utilisateur. Pour les systèmes qui allouent les variables séparément (par exemple MALI [13]) le gain est plus grand. Les parenthèses indiquent le nombre de structures créées.

Détériorations à envisager. Dans les systèmes allouant les variables logiques séparément, les grammaires traduites peuvent diminuer la vitesse d'effacement lorsque la grammaire est très ambiguë. Avec notre transformation les variables liées à la queue de la liste à générer sont créées le plus tard possible. Ayant beaucoup de points de choix, nous créons plus souvent des variables que dans le cas de la version originale. Si alors le coût de la création des variables est plus grand que la somme des coûts de la gestion de la traînée et l'accès aux variables par des références, les équations éliminant les variables vides sont à éviter. Tout de même, EBC est capable de réduire la taille des continuations.

L'allocation globale de registres. Avec une convention très simple la variable de la liste est allouée dans un seul registre : Pendant la traduction cette variable est mise, disons,

1 orig. EBC	$\ll \leftarrow \text{phrase}(g,x_1,\ldots,x_n,\ y_1,\ldots,y_n\ ).\gg$ 0 $(2+n)$	$-(2+n)$
2 orig. EBC	$\ll \leftarrow \text{phrase}(g,x_1,\ldots,x_n,[],\ldots,[]).\gg$ 0 (2)	-2
3 orig. EBC	$\ll \leftarrow \ldots, \text{phrase}(g,x_1,\ldots,x_n,\ y_1,\ldots,y_n\ ).\gg$ $(2+n_g+2n)$ $(2+n_g+n)+(2+n)$	-2
4 orig. EBC	$\ll \leftarrow \ldots, \text{phrase}(g,x_1,\ldots,x_n,[],\ldots,[]).\gg$ $(2+n_g+2n)$ $(2+n_g+n)+(2)$	$n-2$
5 orig. EBC	$\ll \ldots \longrightarrow \ldots, g, \ldots \gg$ $(2+n_g+2n)$ $(2+n_g)$	$2n$
6 orig. EBC	$\ll \ldots \longrightarrow \{\ldots\}.\gg$ $(2+2n)$ $(2+n)$	n
7 orig. EBC	$\ll \ldots \longrightarrow \{\ldots\}, g, \ldots \gg$ $(2+n_g+2n)$ $(2+n_g+n)$	n
8 orig. EBC	$\ll \ldots \longrightarrow \ldots, g, \{\ldots\}.\gg$ $(2+n_g+2n)+(2+2n)$ $(2+n_g)+(2+n)+(2+n)$	$2n-2$
9 orig. EBC	$\ll \ldots \longrightarrow \ldots, g, \{\ldots\}, h, \ldots \gg$ $(2+n_g+2n)+(2+n_h+2n)$ $(2+n_g)+(2+n)+(2+n_h+n)$	$2n-2$

Figure 2: Comparaison de tailles de continuation

au deuxième argument.

3.2 Implantation de grammaires de métamorphose complètes

Colmerauer remarquait que la forme la plus générale des règles des grammaires de métamorphose est peu fréquemment utilisée. Néanmoins, nous présenterons le méta-interpréteur « gm_phrase/3 » qui réalise une implantation alternative des grammaires de métamorphose. Les non-terminaux situés à gauche de la flèche sont interprétés dans le sens inverse. « gm_retour » produit la liste négative. Cette inversion élimine tous les enregistrements dans la traînée pour engendrer la liste dans la tête de la règle de grammaire. La traduction correspondante produit deux règles pour chaque production.

```
gm_phrase(NT, Cs0,Cs) ←
    gm_tour([NT], Cs0,Cs).

gm_tour([NT|NTs1], Cs0,Cs) ←
    gm_règle(NT, XTs, MTs),
    renverser(XTs, XTRs),
    conc(MTs, [changer|XTRs], MTs_c_XTRs),
    conc(MTs_c_XTRs, [changer|NTs1], NTs0),
    gm_tour(NTs0, Cs0,Cs).
gm_tour([Terminaux|NTs], Cs0,Cs) ←
    conc(Terminaux, Cs1,Cs0),
    gm_tour(NTs, Cs1,Cs).
gm_tour([changer|NTs], Cs0,Cs) ←
    gm_retour(NTs, Cs0,Cs).
gm_tour([], Cs,Cs).

gm_retour([NT|NTs1], Cs0,Cs) ←
    gm_règle(NT, XTs, MTs),
    renverser(MTs, MTRs),
    conc(MTRs, [changer|XTs], MTRs_c_XTs),
    conc(MTRs_c_XTs, [changer|NTs1], NTs0),
    gm_retour(NTs0, Cs0,Cs).
gm_retour([Terminaux|NTs], Cs0,Cs) ←
    conc(Terminaux, Cs0,Cs1),
    gm_retour(NTs, Cs1,Cs).
gm_retour([changer|NTs], Cs0,Cs) ←
    gm_tour(NTs, Cs0,Cs).
```

4 Particularités des machines virtuelles en regard de EBC

Dans cette section nous allons comparer l'effet de notre transformation sur les machines virtuelles suivantes : La première machine de Warren (PLM) [17], la machine ZIP [3], la deuxième machine de Warren, la Machine Abstraite de Warren (WAM) [18], la Machine Abstraite de Vienne (VAM) [7], et la Machine Abstraite de Warren Binaire (BINWAM) [14]. La PLM utilise une représentation des termes par partage de structures. La ZIP et la WAM utilisent une représentation par recopie de structures. La VAM emploie toutes les deux méthodes. La transmission de paramètres est mise en œuvre par le partage de structures. La recopie de structures sera utilisée pour les termes permanents.

Afin d'évaluer l'efficacité de nos transformations il suffit d'examiner la mise en œuvre de transmission des arguments d'un prédicat au niveau de machines virtuelles. En particulier la transmission des variables logiques. Nous distinguons deux cas de la transmission des variables logiques : l'un concerne les variables existentielles, tandis que l'autre concerne les variables qui apparaissent exactement une fois dans la tête d'une règle et qui n'apparaissent que dans le premier but.

1. Dans les machines PLM, ZIP, et WAM toutes les variables dans un but à effacer doivent être initialisées par l'appelante. Dans la PLM elles sont déjà initialisées après la tête de règle appelante. Cependant, la VAM diffère l'initialisation des variables grâce à l'entrelacement de l'initialisation des arguments d'un but avec le code de la tête d'une règle.

2. Dans les machines PLM, ZIP, et VAM, le passage de ces variables est réalisé par au moins une instruction transmettant la variable dans la mémoire (p.ex. pour la ZIP on a une instruction pour la lecture d'argument dans la pile d'arguments et une pour empiler la variable). Cependant, la WAM n'utilise point la mémoire grâce à son interface par registres. De plus, l'instruction peut être éliminée. Dans la plupart des implantations de la WAM il suffit de laisser les variables dans la même position d'argument des prédicats. La WAM classifie ces variables comme temporaires.

Notre transformation soulève quelques problèmes liés à la gestion des variables. C'est notamment le premier cas qui est réduit en échange du deuxième cas. En conclusion, la machine WAM profite le plus de notre transformation au regard de la gestion des variables. La BINWAM correspond à une simplification d'une variation de la WAM classique [18] utilisant l'architecture de la zone de travail selon le *split stack model* (dans [2] cette variation

est appelée classique). La pile locale (d'environnements ET) n'existe plus car la BINWAM n'efface que les clauses avec un seul but (clauses binaires). Les continuations ET sont mises en œuvre avec les structures grâce à la transformation de Tarau [15]. L'EBC commençant avec cette transformation, est donc particulièrement intéressant pour la BINWAM. En résumé, un couple de variables est optimisé comme suit:

points de reprise : réduction de 2 registres à un registre

la traînée : pas d'usage pour les variables de ce couple

continuations, environnement : réduction de 2 arguments à zéro.

instructions : toutes les instructions attachées à la transmission du couple sont éliminées

5 Comparaison EBC et pliage/dépliage

La transformation classique de pliage et dépliage présente une méthode proche du déroulement de la résolution de Prolog [12]. Le dépliage est comparable à l'effacement d'un but. Le pliage ne correspond pas directement à une opération effectuée pendant la résolution. Mais elle est définie comme l'opération inverse.

L'autre méthode fondamentale est la définition des nouveaux prédicats, qui ajoute une seule règle d'un nouveau prédicat. La tête de règle ne contient que des variables libres. Dans ce noyau il n'est pas possible d'introduire de nouvelles représentations des symboles fonctionnels ce qui est l'essentiel de notre transformation.

Néanmoins, il y a aussi des cas communs entre les deux approches. Pour les illustrer, prenons comme exemple le but $\ll \leftarrow a, b \gg$. Supposons que les deux prédicats ne puissent pas être optimisés. Afin d'éviter l'écriture de la continuation pour le but $\ll b \gg$ nous définissons un nouveau prédicat $\ll ab \gg$. Après l'effacement du but $\ll a \gg$ nous effaçons l'autre but sans aucune écriture d'une continuation.

Pliage/dépliage.

```
a.
a ←
    f,
    a.

ab ←
    a,
    b.
```

```
a.
a ←
    f,
    a.

ab ←
    b.
ab ←
    f,
    a,
    b.
```

```
a.
a ←
    f,
    a.

ab ←
    b.
ab ←
    f,
    ab.
```

EBC.

```
p(Cont) ←
    a(b(Cont)).

a(Cont) ←
    Cont.
a(Cont) ←
    f(a(Cont)).
```

```
a(b(Cont)) ≐ ab(Cont).

p(Cont) ←
    ab(Cont).

a(Cont) ←
    Cont.
a(Cont) ←
    f(a(Cont)).

ab(Cont) ←
    b(Cont).
ab(Cont) ←
    f(a(b(Cont))).
```

```
p(Cont) ←
    ab(Cont).

a(Cont) ←
    Cont.
a(Cont) ←
    f(a(Cont)).

ab(Cont) ←
    b(Cont).
ab(Cont) ←
    f(ab(Cont)).
```

6 Transformations affinées

6.1 Allouer les registres

La transformation EBC élimine le goulet d'étranglement principal de la WAM : Les variables existentielles et donc les registres de courte de durée. On peut profiter en plus de l'EBC avec plusieurs transformations par pliage/dépliage comme le montre « tri_rapide ».

```
tri_rapide([]) ⟶
    [].
tri_rapide([E|Es]) ⟶
    {coupe(Es, E, As, Bs)},
    tri_rapide(As),
    [E],
    tri_rapide(Bs).
```

Nous montrerons plusieurs optimisations possibles : Le renversement du prédicat « coupe » pour éviter l'entassement des variables; le pliage complet du corps; l'indexation dans « coupe ». L'entassement du programme « tri_rapide_plat » est comparable avec l'empilement d'un programme impératif, utilisant la transmission de paramètres par des registres entre les procédures appelante et appelée. La procédure appelée tient compte de la sauvegarde des registres : Pour le corps original avec « As = [] » aucune continuation n'est entassée. Ce qui correspond bien au programme impératif. La continuation elle-même se réduit de $4 + 3 + 3 = 10$ à 4 ou zéro pour chaque inférence récursive accomplie.

```
tri_rapide_plat([]) ⟶
    [].
tri_rapide_plat([E|Es]) ⟶
    coupe(Es, E, [], []).

tri_rapide([], X, Bs) ⟶
    tri_rapide(X, Bs).
tri_rapide([E|Es], X, Bs) ⟶
    coupe(Es, E, [], []),
    tri_rapide(X, Bs).

tri_rapide(X, Bs) ⟶
    [X],
    tri_rapide_plat(Bs).
```

```
coupe([], X, As, Bs) ⟶
    tri_rapide(As, X, Bs).
coupe([E|Es], X, As, Bs) ⟶
    {compare(R, E, X)},
    coupe(R, X, As, Bs, E, Es).

coupe(=, X, As, Bs, E, Es) ⟶
    coupe(Es, X, [E|As], Bs).
coupe(<, X, As, Bs, E, Es) ⟶
    coupe(Es, X, [E|As], Bs).
coupe(>, X, As, Bs, E, Es) ⟶
    coupe(Es, X, As, [E|Bs]).
```

6.2 Restitution de valeurs

Pour terminer, nous montrerons comment établir des transformations plus avancées dans le noyau EBC. Prenons l'exemple bien connu du calcul de la factorielle. Le traitement des nombres entiers est contourné avec l'arithmétique de successeur.

```
factorielle(0,1).
factorielle(N0, R0) ←
    N0 > 0,
    N1 is N0 - 1,
    factorielle(N1, R1),
    R0 is N0 * R1.
```

```
factorielle(0, 1, Cont) ←
    demo(Cont).
factorielle(N0, R0, Cont) ←
    s(N1) = N0,
    factorielle(N1, R1, mult(R0, N0, R1, Cont)).

mult(R0, N0, R1, Cont) ←
    R0 is N0 * R1,
    demo(Cont).
```

Le programme ci-dessous, obtenu par plusieurs transformations, rend explicite ce qui se passe pendant le calcul : Dans la première boucle « factorielle/2 » on empile successivement les valeurs n, $n-1$, ... Arrivé à zéro on change la boucle restaurant les valeurs à l'aide de « mult/3 » commençant avec 1, 2, ... La première boucle est donc complètement inutile, puisque la seconde peut restituer les valeurs en passant.

```
factorielle(N, R, Cont) ≐ factorielle(N, résultat(R, Cont)).
mult(R0, N0, R1, Cont) ≐ mult(N0, R1, résultat(R0, Cont)).
résultat(R1, mult(N0, R1, R0Cont)) ≐ résmult(N0, R0Cont).
```

```
factorielle(N, R, Cont) ←
    factorielle(N, résultat(R, Cont)).

factorielle(0, résultat(1, Cont)) ←
    demo(Cont).
factorielle(0, résmult(N, R0Cont)) ←
    mult(N, 1, R0Cont).
factorielle(s(N1), R0Cont) ←
    factorielle(N1, résmult(s(N1), R0Cont)).

mult(N0, R1, résultat(R0, Cont)) ←
    R0 is N0 * R1,
    demo(Cont).
mult(N0, R1, résmult(N, R0Cont)) ←
    R0 is N0 * R1,
    mult(N, R0, R0Cont).
```

Les équations ci-dessous contiennent les symboles « factorielle/2 », « mult/3 » et « résmult/2 » dans leurs deux côtés. La raison est l'intention d'exprimer une *dépendance de contexte* entre la continuation et le premier argument de « factorielle/2 ». Néanmoins, avec ces équations nous n'introduisons qu'un ensemble fini de représentations équivalentes. Chaque structure « factorielle(N0, résmult(N, RCont)) » possède exactement une représentation alternative « factorielle(N0, aux(N0, résmult(N, RCont))) ». Les symboles « aux/2 » et « résmult/2 » bloquent les réécritures infinies.

factorielle(N0, résmult(N, RCont)) $\doteq$ factorielle(N0, aux(N0, résmult(N, RCont))).
mult(N0, R, résmult(N, RCont)) $\doteq$ mult(N0, R, aux(N0, résmult(N, RCont))).

factorielle(N, R, Cont) ←
    factorielle(N, résultat(R, Cont)).

factorielle(0, résultat(1, Cont)) ←
    demo(Cont).
factorielle(0, **aux(0, résmult(N, R0Cont))**) ←
    mult(N, 1, R0Cont).
factorielle(s(N1), R0Cont) ←
    factorielle(N1, **aux(N1, résmult(s(N1), R0Cont))**).

mult(N0, R1, résultat(R0, Cont)) ←
    R0 is N0 * R1,
    demo(Cont).
mult(N0, R1, **aux(N0, résmult(N, R0Cont))**) ←
    R0 is N0 * R1,
    mult(N, R0, R0Cont).

aux(N, résmult(s(N), RCont)) $\doteq$ auxrésmult(RCont).

factorielle(N, R, Cont) ←
    factorielle(N, résultat(R, Cont)).

factorielle(0, résultat(1, Cont)) ←
    demo(Cont).
factorielle(0, **auxrésmult(R0Cont)**) ←
    mult(**1**, 1, R0Cont).
factorielle(s(N1), R0Cont) ←
    factorielle(N1, **auxrésmult(R0Cont)**).

mult(N0, R1, résultat(R0, Cont)) ←
    R0 is N0 * R1,
    demo(Cont).
mult(N0, R1, **auxrésmult(R0Cont)**) ←
    R0 is N0 * R1,
    mult(**s(N0)**, R0, R0Cont).

Malheureusement, EBC ne suffit pas à dériver le programme « factorielle_ebc/2 » ci-dessous. L'élimination de l'empilage nécessite un raisonnement plus fort. D'une part le raisonnement d'équations doit s'éloigner des termes (presque) clos jusqu'aux variables. D'autre part nous avons besoin de contracter la première boucle « factorielle/2 » qui ne serve qu'à tester si l'argument est un nombre entier. Couramment, nous conjecturons que notre technique des buts partiellement statiques [9] puisse être appliquée. Voici le programme 'final' désiré, mais pas encore dérivé :

factorielle_ebc(N, R) ←
    factorielle_ebc(0, N, 1, R).

factorielle_ebc(N, N, R, R).
factorielle_ebc(N0, N, R0, R) ←
    N0 < N,
    N1 is N0 + 1,
    R1 is R0 * N1,
    factorielle_ebc(N1, N, R1, R).

Comparaison avec les transformations courantes. Les transformations basées sur le pliage et le dépliage utilisant l'associativité dépendent des informations externes. De plus elles peuvent changer la structure des programmes d'une manière complètement imprévue par l'utilisateur. L'usage de ces transformations pouvait être un jeu de hasard. De plus, les opérations ne sont pas toujours associatives comme par exemple les opérations d'arithmétique en virgule flottante. Ci-dessous le prédicat proposé par ailleurs en vertu des transformations algébriques.

```
factorielle_dp(N, R) ←
    factorielle_dp(N, 1, R).

factorielle_dp(0, R, R).
factorielle_dp(N0, P0, R) ←
    N0 > 0,
    P1 is P0 * N0,
    N1 is N0 - 1,
    factorielle_dp(N1, P1, R).
```

Le calcul de la factorielle de 2000 avec le programme original dit inefficace prend 2,32 secondes autant que la version « factorielle_ebc/2 » qui n'entasse aucune continuation et qui est donc plus efficace pour les nombres plus petits. La version obtenue utilisant l'associativité « factorielle_dp/2 », prend 2,68 secondes.

Que les transformations proposées pour cet exemple bien aimé rendent pire, l'effacement de ce programme, ne manque pas d'une certaine ironie. Les mesures ont été déterminées avec SICStus-Prolog 2.1. Nous présumons que l'application hâtive de l'associativité est la raison de cette détérioration.

Remerciements. Je tiens à remercier Jean-François et Joëlle Pique ainsi que Müberra Manasur et finalement Johannes Gehringer de leur aide precieuse. Les fautes qui restent sont, sans doute, les miennes. Ma reconnaissance va aussi à M. Brockhaus qui soutient mon travail.

Références

[1] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992. ISBN 0-521-41695-7.

[2] P. Boizumault. *Prolog, l'implantation.* Masson, Paris, 1988.

[3] D. Bowen, L. Byrd, and W. Clocksin. A portable Prolog compiler. In *Proceedings of the Logic Programming Workshop*, Albufeira, Portugal, 1983.

[4] A. Colmerauer. Les grammaires de métamorphose. Technical report, Groupe d'Intelligence Artificielle, Université de Marseille II, Nov. 1975.

[5] P. Deransart and J. Małuszyński, editors. *2nd International Symposium, PLILP 90*, volume 456 of *Lecture Notes in Computer Science*, Linköping, Sweden, Aug. 1990. Springer-Verlag.

[6] S. L. Huitouze. *Mise en œvre de PrologII/MALI.* PhD thesis, Université de Rennes 1, 1988.

[7] A. Krall and U. Neumerkel. The Vienna Abstract Machine. In Deransart and Małuszyński [5], pages 121–135.

[8] G. Neumann. Meta-interpreter directed compilation of logic programs into Prolog. Research Report RC 12113 (No. 54357), IBM, Yorktown Heights, New York, 1986.

[9] U. Neumerkel. Specialization of Prolog programs with partially static goals and binarization, Dissertation. Bericht TR 1851-1992-12, Institut für Computersprachen, Technische Universität Wien, 1992.

[10] R. O'Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[11] F. Pereira and D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *AI*, 13:231–278, 1980.

[12] M. Proietti and A. Pettorossi. Unfolding-definition-folding in this order, for avoiding unnecessary variables in logic programs. In J. Małuszyński and M. Wirsing, editors, *Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 347–358, Passau, Germany, Aug. 1991. Springer-Verlag.

[13] O. Ridoux. Mali v06 — tutorial and reference manual. Technical report, IRISA, Université RENNES, June 1992.

[14] P. Tarau. A simplified abstract machine for the execution of binary metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, Sept. 1991.

[15] P. Tarau and M. Boyer. Elementary logic programs. In Deransart and Małuszyński [5], pages 159–173.

[16] P. Wadler. The essence of functional programming. In *Conference Record of the 19'th Annual ACM Symposium on Principles of Programming Languages*, Jan. 1992.

[17] D. H. Warren. Implementing Prolog – compiling predicate logic programs, vol. 1 & 2. Technical Report 39-40, D.A.I., May 1977.

[18] D. H. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, 1983.