cTI : un outil pour l'inférence de conditions optimales de terminaison pour Prolog

Fred Mesnard* — Ulrich Neumerkel — Etienne Payet***

* *Iremia, Université de La Réunion, France*
{fred,epayet}@univ-reunion.fr

** *Université Technique de Vienne, Autriche*
ulrich@mips.complang.tuwien.ac.at

RÉSUMÉ. La terminaison est un aspect crucial de la vérification de programmes. Cet article propose un outil pour inférer des conditions suffisantes de terminaison à partir du texte de tout programme Prolog. Nous adoptons pour cela une approche ascendante du problème pour essayer de caractériser, pour chaque relation définie dans le programme, l'ensemble des classes de requêtes qui terminent universellement pour la règle de sélection Prolog. Nous décrivons l'architecture de notre implantation nommée cTI. Nous évaluons notre système sur une soixantaine de programmes dont les tailles varient de quelques à plusieurs centaines de clauses. Enfin, nous présentons nos travaux en cours concernant la détection de l'optimalité des conditions de terminaison.

ABSTRACT. Termination is a crucial aspect of program termination. This paper describes our tool named cTI for inferring sufficient termination conditions from any Prolog program text. We adopt a bottom-up approach where we try to characterize, for each relation defined in a program, the set of universally left-terminating queries. We present the architecture of our implementation. We benchmark cTI against sixty programs which sizes range from a few to several hundred clauses. At last, we discuss our current work on checking the optimality of terminating conditions.

MOTS-CLÉS : programmation logique, terminaison, analyse statique.

KEYWORDS: logic programming, termination, static analysis.

1. Introduction

La terminaison est un aspect crucial de la vérification de programmes. L'une des particularités de la programmation logique, paradigme relationnel, tient à ce qu'*il n'existe pas* a priori *de restrictions syntaxiques sur les requêtes que l'on peut souhaiter prouver*. De nombreux chercheurs, ces quinze dernières années, se sont penchés sur le problème de la terminaison en programmation logique (voir par exemple [DES 94] pour une revue de l'état de l'art datant du milieu des années 90 et [RUG 99] pour une remarquable étude des liens entre terminaison d'une part, et correction partielle, validation et débogage d'autre part). L'immense majorité de ces efforts s'est consacrée à la terminaison *universelle*, *i.e.* la finitude de l'arbre de dérivations (indépendante de l'ordre textuel des clauses) par opposition à la terminaison *existentielle*, *i.e.* l'existence d'une réfutation (vision plus orientée démonstration automatique que programmation). L'étude de ces travaux permet de distinguer au moins deux directions de recherche : les caractérisations de la terminaison (par exemple [VAS 86, APT 93]) et l'affaiblissement de telles caractérisations (indécidables) en vue d'obtenir des conditions suffisantes implantables (par exemple [LIN 97, DEC 00]).

Cet article se place résolument dans la deuxième direction définie ci-dessus. L'innovation essentielle de notre travail est que nous tentons d'*inférer* des conditions suffisantes de terminaison à partir du texte de tout programme Prolog. Nous adoptons pour cela une approche ascendante du problème pour essayer de *caractériser, pour chaque relation définie dans le programme, l'ensemble des classes de requêtes qui terminent.* Ce point de vue est explicité à la section 2. Nous décrivons ensuite l'architecture de notre implantation nommée cTI (constraint-based Termination Inference, disponible en ligne à l'URL `http://www.complang.tuwien.ac.at/cti`). Les bases théoriques de cTI sont étudiées dans [MES 96, MES 01]. Nous évaluons notre système à la section 4, sur une soixantaine de programmes dont les tailles varient de quelques à plusieurs centaines de clauses. Plus spéculative, la section 5 présente nos travaux en cours concernant la détection de l'*optimalité* des conditions de terminaison calculées par cTI.

2. Objectif

On se donne un programme Prolog P et un atome $q(\tilde{x})$, où q est un symbole relationnel défini dans P et $\tilde{x}$ désigne une séquence de variables distinctes dont la longueur égale l'arité de q. (Un atome de la forme $q(\tilde{x})$ est qualifié de *pur*). Notre objectif est de calculer la plus grande classe de requêtes de la forme $q(\tilde{t})$ (où $\tilde{t}$ désigne une séquence de termes quelconques de longueur identique à celle de $\tilde{x}$) pour lesquelles la terminaison universelle gauche est garantie sur un processeur Prolog dont l'algorithme d'unification est muni du test d'occurrence. La définition qui suit précise le *langage* que nous utilisons pour décrire ces classes de requêtes.

Définition 1 *Soit $A := q(\tilde{x})$ un atome. Un* mode *pour A est une conjonction (éventuellement vide) de variables appartenant à $\tilde{x}$. Une* classe de requêtes *pour A est une disjonction (éventuellement vide) de modes pour A.*

Les éléments de ces langages sont considérés modulo les propriétés de la conjonction et la disjonction. On notera que tout mode est une classe de requêtes. Soient $C_q(\tilde{x})$ une classe de requêtes et $\tilde{t}$ une séquence de termes de longueur identique à $\tilde{x}$. La *valeur* booléenne de l'expression $C_q(\tilde{t})$ (où on a replacé chaque x_i par le t_i correspondant) se définit comme suit. Si l'expression est 0 (resp. 1), sa valeur est 0 (resp. 1). Si l'expression est un terme t_j, sa valeur est 1 ssi t_j est un terme sans variable (+ fini). Enfin, si l'expression est définie comme une disjonction (resp. une conjonction), sa valeur est le ou (resp. et) de ses expressions constitutives. Une classe de requêtes $C_q(\tilde{x})$ est donc une représentation finie de l'ensemble $||C_q(\tilde{x})|| = \{q(\tilde{t}) \mid \tilde{t}$ est une séquence de termes $\wedge\, C_q(\tilde{t}) = 1\}$. Nous pouvons introduire une relation d'ordre partiel sur les classes de requêtes pour $q(\tilde{x})$ définie comme suit :

Définition 2 *$C^1_q(\tilde{x})$ est* plus générale *que $C^2_q(\tilde{x})$ ssi $||C^1_q(\tilde{x})|| \supseteq ||C^2_q(\tilde{x})||$.*

Voici ce que nous entendons par condition de terminaison :

Définition 3 *Une* condition de terminaison *$CT_q(\tilde{x})$ est une classe de requêtes pour $q(\tilde{x})$ telle que toute dérivation de tout but $q(\tilde{t}) \in ||CT_q(\tilde{x})||$ utilisant la règle de sélection gauche-droite est finie.*

L'union de deux conditions de terminaison (définie comme la disjonction des termes booléens associés) est également une condition de terminaison, toujours plus générale. Donc l'union de toutes les conditions de terminaison existe, est unique et c'est la condition de terminaison la plus générale.

Définition 4 *La condition de terminaison* optimale *de l'atome pur A est l'union de toutes les conditions de terminaison de A.*

En toute généralité, il est bien sûr illusoire d'espérer disposer d'un algorithme pour le calcul de la condition optimale. De même, tester si une classe de requêtes est une condition de terminaison est indécidable. En revanche, confrontés à des exemples concrets de programmes, nous pouvons *parfois* conclure. (Il existe en effet des programmes et des classes de requêtes associées dont la terminaison reste un problème ouvert.) Tout au long de cet article, nous illustrerons notre propos au moyen de ce petit programme :

```
app([],Xs,Xs).                  nrev([],[]).                app3(Xs,Ys,Zs,Us):-
app([X|Xs],Ys,[X|Zs]):-         nrev([X|Xs],Ys):-              app(Xs,Ys,Vs),
   app(Xs,Ys,Zs).                  nrev(Xs,Zs),                app(Vs,Zs,Us).
                                   app(Zs,[X],Ys).
```

Exemple 1 *Nous avons :*

– Pour app(x, y, z), x est une condition de terminaison (car pour tout terme clos s, pour tous termes t et u, toute dérivation de app(s, t, u) termine à gauche). De même, z est une condition de terminaison. La condition de terminaison optimale est $x \vee z$. En effet, la requête Prolog `app(X,toto,Z)` *(nous faisons l'hypothèse d'un nombre infini de constantes) admet une dérivation infinie. Donc la classe y (et* a fortiori *1) n'est pas une condition de terminaison.*

– Pour $nrev(x, y)$, 0 est une condition de terminaison, l'optimale étant x.

– Nous laissons au lecteur le soin de déterminer la condition optimale de terminaison de app3(x, y, z, u).

3. Une présentation de l'architecture de cTI

Les informations purement syntaxiques sont souvent trop faibles pour raisonner sur des programmes non triviaux. Des informations d'ordre sémantique sont nécessaires. C'est pour cela que notre analyseur utilise trois structures de contraintes [JAF 87, JAF 94] : les termes de Herbrand (CLP($\mathcal{H}$)) pour le programme initial P, les entiers naturels (CLP(N)) et les booléens (CLP($\mathcal{B}$)) pour décrire les approximations de P. La correspondance entre ces structures repose sur la notion d'*approximations* [MES 96] qui sont une forme simplifiée d'interprétation abstraite [COU 77, COU 92], également appelée *compilation abstraite*. Nous illustrons notre méthode d'inférence de conditions de terminaison au moyen des prédicats `app/3`, `app3/4` et `nrev/2` présentés à la section 2.

1. Le programme Prolog initial P est projeté dans CLP(N) pour donner P^N grâce à une approximation basée sur la norme symbolique :

$$\|t\|_{\text{terme}} = \begin{cases} 1 + \sum_{i=1}^{n} \|t_i\|_{\text{terme}} & \text{si } t = f(t_1, \ldots, t_n), n > 0 \\ 0 & \text{si } t \text{ est une constante} \\ t & \text{si } t \text{ est une variable} \end{cases}$$

Par exemple, $\|f(0, 0)\|_{\text{terme}} = 1$ et $\|f(X, Y)\|_{\text{terme}} = 1 + X + Y$. Toutes les relations prédéfinies non-monotones de Prolog sont approximées par des constructions monotones, par exemple `var(x)` par `true`. Pour la correction de l'analyse, nous maintenons la propriété suivante : si un but dans P^N termine, alors tous les buts Prolog correspondants terminent. Le programme P^N s'écrit :

`app`$_N$`(0,Xs,Xs).`
`app`$_N$`(1+X+Xs,Ys,1+X+Zs):-`
`    app`$_N$`(Xs,Ys,Zs).`
	`nrev`$_N$`(0,0).`
	`nrev`$_N$`(1+X+Xs,Ys):-`
	`    nrev`$_N$`(Xs,Zs),`
	`    app`$_N$`(Zs,1+X,Ys).`
		`app3`$_N$`/4`
		comme
		`app3/4`

2. Nous calculons dans N un modèle [COU 78, BEN 97] de P^N. Le modèle décrit, au moyen d'une conjonction finie d'égalités et d'inégalités linéaires, les relations entre les arguments d'un but (relations inter-arguments nommées *post* ci-dessous). Ces relations sont valides pour toute solution. Le calcul est en fait effectué dans CLP(Q), avec l'opérateur d'élargissement de [COU 78]. Dans notre exemple, nous sommes capables de déterminer le plus petit modèle. En général, seul un modèle moins précis est calculé.

Puis, pour chaque prédicat récursif p, qui constitue donc une source potentielle de non terminaison, nous calculons (par adaptation d'une technique de programmation linéaire décrite dans [SOH 91]) une fonction de niveau (une application de la N-base de P^N dans N) appelé μ_p, strictement décroissante entre la tête et les appels (mutuellement-)récursifs du corps de la clause. Pour un prédicat non récursif p, nous posons par défaut $\mu_p = 0$. Par exemple, la signification de μ^N_{app} est : pour toute clause récursive close définissant `app`$_N$`/3`, en comparant la tête et l'atome du corps de la clause, on s'aperçoit que le premier et le troisième argument décroissent, d'où le choix $min(x, z)$ comme fonction de niveau. La nécessité d'un modèle numérique de P est uniquement justifiée par le besoin d'informations sémantiques pour le calcul de fonctions de niveau performantes. Par exemple, pour la clause $p(\tilde{t_1}) \leftarrow q(\tilde{t_2}), p(\tilde{t_3})$, la fonction de niveau pour p est calculée relativement à la clause $p(\tilde{t_1}) \leftarrow post^N_q(\tilde{t_2}), p(\tilde{t_3})$. Revenons à notre exemple initial ; nous avons :

(plus petits) modèles		fonctions de niveau	
$post^N_{app}(x, y, z)$	$\leftrightarrow z = x + y$	$\mu^N_{app}(x, y, z)$	$= min(x, z)$
$post^N_{nrev}(x, y)$	$\leftrightarrow x = y$	$\mu^N_{nrev}(x, y)$	$= x$
$post^N_{app3}(x, y, z, u)$	$\leftrightarrow u = x + y + z$	$\mu^N_{app3}(x, y, z, u)$	$= 0$

3. P^N est ensuite projeté dans CLP($\mathcal{B}$) pour donner $P^{\mathcal{B}}$. Ici, le booléen 1 signifie qu'un argument est borné par rapport à la norme considérée. Notons que le programme obtenu ne conserve plus la propriété de terminaison. Son unique intérêt est de déterminer les dépendances relatives au caractère borné des arguments au sein du programme. La simplicité et la richesse de la structure $\mathcal{B}$ permettent toujours de calculer le plus petit modèle. Pour chaque prédicat, la fonction de niveau calculée préalablement est représentée par un terme booléen.

`app`$_{\mathcal{B}}$`(1,Xs,Xs).`
`app`$_{\mathcal{B}}$`(1∧X∧Xs,Ys,1∧X∧Zs):-`
`    app`$_{\mathcal{B}}$`(Xs,Ys,Zs).`
	`nrev`$_{\mathcal{B}}$`(1,1).`
	`nrev`$_{\mathcal{B}}$`(1∧X∧Xs,Ys):-`
	`    nrev`$_{\mathcal{B}}$`(Xs,Zs),`
	`    app`$_{\mathcal{B}}$`(Zs,1∧X,Ys).`
		`app3`$_{\mathcal{B}}$`/4`
		comme
		`app3/4`

plus petits modèles		fonctions de niveau	
$post^{\mathcal{B}}_{app}(x, y, z)$	$\leftrightarrow (x \wedge y) \Leftrightarrow z$	$\mu^{\mathcal{B}}_{app}(x, y, z)$	$\leftrightarrow x \vee z$
$post^{\mathcal{B}}_{nrev}(x, y)$	$\leftrightarrow x \Leftrightarrow y$	$\mu^{\mathcal{B}}_{nrev}(x, y)$	$\leftrightarrow x$
$post^{\mathcal{B}}_{app3}(x, y, z, u)$	$\leftrightarrow (x \wedge y \wedge z) \Leftrightarrow u$	$\mu^{\mathcal{B}}_{app3}(x, y, z, u)$	$\leftrightarrow 1$

4. $P^{\mathcal{B}}$ est traduit, grâce aux informations obtenues précédemment, en un système de mu-calcul booléen. Ce système assure d'une part la décroissance effective des fonctions de niveau (pour les prédicats (mutuellement-)récursifs) et d'autre part le caractère fini des fonctions de niveau pour les prédicats qui ne font partie du cycle considéré, le tout via le flux de données gauche-droite propre à Prolog.

$$
\begin{array}{lcl}
CT_{app} & = & \nu A.\lambda(b, c, d).\left\{\begin{array}{l} b \vee d \\ \wedge \\ \forall f, g, h.[(f \wedge g \leftrightarrow b) \wedge (f \wedge h \leftrightarrow d)] \rightarrow A(g, c, h) \end{array}\right. \\
CT_{nrev} & = & \nu A.\lambda(b, c).\left\{\begin{array}{l} b \\ \wedge \\ \forall d, e, f.[d \wedge e \leftrightarrow b] \rightarrow A(e, f) \\ \wedge \\ \forall d, e, f.[(f \leftrightarrow e) \wedge (d \wedge e \leftrightarrow b)] \rightarrow CT_{app}(f, d, c) \end{array}\right. \\
CT_{app3} & = & \nu A.\lambda(b, c, d, e).\left\{\begin{array}{l} \forall f.1 \rightarrow CT_{app}(b, c, f) \\ \wedge \\ \forall f.[f \leftrightarrow c \wedge b] \rightarrow CT_{app}(f, d, e) \end{array}\right.
\end{array}
$$

La résolution de ce système (calcul du plus grand point fixe au moyen d'un μ-solveur booléen) conduit aux conditions de terminaison :

$$
\begin{array}{lcl}
CT_{app}(x, y, z) & = & x \vee z \\
CT_{nrev}(x, y) & = & x \\
CT_{app3}(x, y, z, u) & = & (x \wedge y) \vee (x \wedge u)
\end{array}
$$

Ces conditions de terminaison assurent que :

– toute preuve de `app(`t_1`,`t_2`,`t_3`)` termine à gauche si t_1 ou t_3 sont clos,

– toute preuve de `nrev(`t_1`,`t_2`)` termine à gauche si t_1 est clos,

– toute preuve de `app3(`t_1`,`t_2`,`t_3`,`t_4`)` termine à gauche si t_1 et t_2 ou bien t_1 et t_4 sont clos.

5. Enfin, la dernière phase d'analyse, réalisée par nTI et décrite à la section 5, tente de déterminer l'optimalité des conditions de terminaison calculées précédemment.

Le théorème suivant [MES 96] énonce le résultat principal sur lequel est basé cTI et assure la correction du calcul des conditions de terminaison.

Théorème 1 *Soient P un programme et p un symbole relationnel défini dans P. Supposons que p soit défini par m_p règles r_k : $p(\tilde{x}) \leftarrow c_k,\, p_{k,1}(\tilde{x}_{k,1}), \ldots\, p_{k,n_k}(\tilde{x}_{k,n_k})$. Supposons que pour chaque $q \notin \bar{p}$ (où $\bar{p}$ désigne la composante fortement connexe contenant p du graphe de dépendance de P) apparaissant dans les règles définissant $\bar{p}$, une condition de terminaison CT_q ait été calculée. Si l'ensemble des termes booléens $\{CT_p\}_{p\in\bar{p}}$ vérifie :*

$$\forall p \in \bar{p} \left\{ \begin{array}{l} CT_p(\tilde{x}) \rightarrow_{\mathcal{B}} \mu_p^{\mathcal{B}}(\tilde{x}), \\ \big[\forall 1 \leq k \leq m_p,\ \forall 1 \leq j \leq n_k, \\ \qquad\qquad \Big(CT_p(\tilde{x}) \wedge c_k^{\mathcal{B}} \wedge \bigwedge_{i=1}^{j-1} post^{\mathcal{B}}_{p_{k,i}}(\tilde{x}_{k,i})\Big) \rightarrow_{\mathcal{B}} CT_{p_{k,j}}(\tilde{x}_{k,j})\big] \end{array} \right.$$

alors $\{CT_p\}_{p\in\bar{p}}$ est une condition de terminaison pour $\bar{p}$.

4. Une évaluation des performances de cTI

Dans la première partie de cette section, nous comparons les résultats *inférés* par cTI avec les résultats *testés* (au sens où les classes de requêtes sont toujours données) par trois autres analyseurs de terminaison décrits dans les articles [DEC 00, LIN 97, SPE 97]. Pour des programmes dont la taille n'excède pas une dizaine de clauses, les temps d'exécution sont de l'ordre de la seconde. Or notre analyse de terminaison nécessite le calcul de trois points fixes. Comment se comporte cTI confronté à des programmes plus conséquents ? La seconde sous-section concerne l'analyse de douze programmes de taille moyenne (inférieure à mille clauses) et détaille les temps d'exécution des différents calculs.

4.1. *Programmes classiques pour l'analyse de terminaison de Prolog*

La table 1 présente les performances de cTI appliqué à des benchmarks classiques en analyse de terminaison de programmes Prolog (respectivement [DES 94, APT 94, PLÜ 90]). Les abréviations qu'on y trouve signifient :

– *cTI temps* : le temps requis pour l'obtention du résultat (non-compris nTI, dont le temps d'exécution pour des programmes de cette taille est négligeable en comparaison de ceux de cTI) ;

– *prédicat d'intérêt* : le symbole de prédicat auquel on s'intéresse ;

– *classe testée (autres)* : la classe de requêtes testée par les analyseurs décrits dans [DEC 00, LIN 97, SPE 97] ;

– *rés* : le meilleur (oui $>$ non) résultat obtenu par les analyseurs cités ci-dessus ;

– *classe inférée (cTI)* : la condition de terminaison inférée par cTI ;

– *nTI* : le résultat de nTI (opt ou ?) indiquant si nTI a réussi à déterminer l'optimalité de la classe calculée par cTI ;

– *note*: des remarques ou adaptations effectuées sur cTI :

- *opt* : le résultat calculé est effectivement optimal ;

- *non-opt* : le résultat calculé est effectivement non-optimal ;

- *liste* : nous avons utilisé la norme symbolique *liste* définie par

$$\|t\|_{\mathrm{liste}} = \begin{cases} 1 + \|u\|_{\mathrm{liste}} & \text{si } t = [s|u] \\ t & \text{si } t \text{ est une variable} \\ 0 & \text{sinon} \end{cases}$$

à la place de la norme *terme* (norme par défaut) ;

- *p=3* : nous avons augmenté la précision de l'interpréteur numérique abstrait de 2 (précision par défaut) à 3.

Nous notons que pour les six premiers programmes, cTI infère la condition de terminaison optimale. En d'autres termes, si le langage utilisé pour décrire les classes de requêtes est restreint à celui présenté à la section 2, cTI a *caractérisé* les classes de requêtes qui terminent. Le programme MERGESORT (idem pour MERGESORT_AP) s'appuie sur la relation `split/3` qui éclate une liste (son premier argument) en deux listes (le deuxième et le troisième argument) de longueur *sensiblement* égale :

```
split([],[],[]).
split([X|Xs],[X|Ys],Zs):- split(Xs,Zs,Ys).
```

L'emploi de la norme *terme* conduit, suivant la précision imposée à l'interpréteur abstrait, aux modèles suivants : $p \geq 2 \Rightarrow post^N_{split}(x, y, z) \leftrightarrow x = y + z$. En revanche, avec la norme *liste*, on obtient : $p \geq 3 \Rightarrow post^N_{split}(x, y, z) \leftrightarrow x = y + z \wedge 0 \leq y - z \leq 1$. Le dernier modèle (pour $p = 3$) obtenu est suffisamment précis pour montrer la terminaison du programme considéré. Pour le programme PL5.2.2, cTI passe plus de trois secondes à déterminer un modèle numérique pour la relation `turing/4` : $p \geq 3 \Rightarrow post^N_{split}(x, y, z) \leftrightarrow x = y + z \wedge 0 \leq y - z \leq 1$. Le module nTI est incapable de montrer que la condition inferée est optimale (rappelons que cette preuve est due à Alan Turing !).

4.2. *Programmes classiques en analyse de programmes Prolog*

La table 3 présente les temps d'exécution de cTI sur des benchmarks[1] classiques en analyse de programmes. Nous avons sélectionné douze programmes de taille moyenne (entre 64 et 833 lignes de code). Tous ces programmes sont décrits dans [BUE 94] exceptés CREDIT (un petit système expert), PLAN (un planificateur) et MINISSAEXP (un interpréteur pour un sous ensemble de Pascal). La table 2 présente leurs caractéristiques. Les abréviations signifient :

– *lignes* : le nombre de lignes du programme Prolog sous forme pure (pas de disjonction, par exemple), avec un symbole de prédicat par ligne et pas de ligne vide ;

1. collectés par Naomi Lindenstrauss, `www.cs.huji.ac.il/~naomil` et également disponibles à `www.complang.tuwien.ac.at/cti/bench`.

Tableau 1. *[DES 94, APT 94, PLÜ 90], cTI 0.29, Athlon 750 MHz, 256 Mo*

temps in [s] programme	cTI temps	prédicat d'intérêt	classe testée (autres)	rés	classe inférée (cTI)	nTI	note
PERMUTE	0.15	$permute(x, y)$	x	oui	x	opt	
DUPLICATE	0.05	$duplicate(x, y)$	x	oui	$x \vee y$	opt	
SUM	0.18	$sum(x, y, z)$	$x \wedge y$	oui	$x \vee y \vee z$	opt	
MERGE	0.26	$merge(x, y, z)$	$x \wedge y$	oui	$(x \wedge y) \vee z$	opt	
DIS-CON	0.24	$dis(x)$	x	oui	x	opt	
REVERSE	0.08	$reverse(x, y, z)$	$x \wedge z$	oui	x	opt	
APPEND	0.09	$append(x, y, z)$	$x \wedge y$	oui	$x \vee z$	opt	
LIST	0.01	$list(x)$	x	oui	x	opt	
FOLD	0.10	$fold(x, y, z)$	$x \wedge y$	oui	y	?	opt
LTE	0.13	$goal$	1	oui	1	opt	
MAP	0.09	$map(x, y)$	x	oui	$x \vee y$	opt	
MEMBER	0.03	$member(x, y)$	y	oui	y	opt	
MERGESORT	0.43	$mergesort(x, y)$	x	non	0	?	non-opt
MERGESORT	0.57	$mergesort(x, y)$	x	non	x	opt	liste, p=3
MERGESORT_AP	0.79	$mergesort\_ap(x, y, z)$	x	oui	z	?	non-opt
MERGESORT_AP	0.92	$mergesort\_ap(x, y, z)$	x	oui	$x \vee z$	opt	liste, p=3
NAIVE_REV	0.12	$naive\_rev(x, y)$	x	oui	x	opt	
ORDERED	0.04	$ordered(x)$	x	oui	x	opt	
OVERLAP	0.05	$overlap(x, y)$	$x \wedge y$	oui	$x \wedge y$	opt	
PERMUTATION	0.15	$permutation(x, y)$	x	oui	x	opt	
QUICKSORT	0.39	$quicksort(x, y)$	x	oui	x	opt	
SELECT	0.08	$select(x, y, z)$	y	oui	$y \vee z$	opt	
SUBSET	0.09	$subset(x, y)$	$x \wedge y$	oui	$x \wedge y$	opt	
SUBSET	0.09	$subset(x, y)$	y	non	$x \wedge y$	opt	
SUM	0.12	$sum(x, y, z)$	z	oui	$y \vee z$	opt	
PL1.2	0.16	perm(x,y)	x	oui	x	opt	
PL2.3.1	0.01	p(x,y)	x	non	0	?	opt
PL3.5.6	0.05	p(x)	1	non	x	opt	
PL3.5.6A	0.06	p(x)	1	oui	x	?	non-opt
PL3.5.6A	0.06	p(x)	1	oui	1	opt	liste
PL4.0.1	0.10	append3(x,y,z,t)	$x \wedge y \wedge z$	oui	$(x \wedge y) \vee (x \wedge t)$	opt	
PL4.4.3	0.26	merge(x,y,z)	$x \wedge y$	oui	$(x \wedge y) \vee z$	opt	
PL4.4.6A	0.12	perm(x,y)	x	oui	x	opt	
PL4.5.2	0.17	s(x,y)	x	non	0	opt	
PL4.5.3A	0.01	p(x)	x	non	0	opt	
PL5.2.2	3.41	turing(x,y,z,t)	$x \wedge y \wedge z$	non	0	?	opt
PL6.1.1	0.39	qsort(x,y)	x	oui	x	opt	
PL7.2.9	0.21	mult(x,y,z)	$x \wedge y$	oui	$x \wedge y$	?	opt
PL7.6.2A	0.14	reach(x,y,z)	$x \wedge y \wedge z$	non	0	opt	
PL7.6.2B	0.22	reach(x,y,z,t)	$x \wedge y \wedge z \wedge t$	non	0	?	opt
PL7.6.2C	0.29	reach(x,y,z,t)	$x \wedge y \wedge z \wedge t$	oui	$z \wedge t$	opt	
PL8.2.1	0.43	mergesort(x,y)	x	non	0	?	non-opt
PL8.2.1	0.58	mergesort(x,y)	x	non	x	opt	liste, p=3
PL8.2.1A	0.47	mergesort(x,y)	x	oui	x	?	opt
MERGESORT_T	0.94	mergesort(x,y)	x	oui	x	opt	
PL8.3.1	0.26	minsort(x,y)	x	non	$x \wedge y$	opt	
PL8.3.1A	0.24	minsort(x,y)	x	oui	x	opt	
PL8.4.1	0.13	even(x)	x	oui	x	opt	
PL8.4.2	0.52	e(x,y)	x	oui	x	opt	

Tableau 2. *Informations concernant les programmes analysés.*

Programme	lignes	faits	règles	cycles	longueur	vars
ANN	571	101	99	44	2	7
BID	108	24	26	20	1	4
BOYER	275	63	78	25	2	5
BROWSE	107	4	29	15	1	6
CREDIT	108	33	24	24	1	4
MINISSAEXP	833	37	223	100	5	17
PEEPHOLE	322	72	80	11	2	5
PLAN	64	12	17	16	1	4
QPLAN	403	63	87	38	3	11
RDTOK	285	7	57	12	4	12
READ	299	15	75	17	7	33
WARPLAN	304	43	68	33	3	14

– *faits* et *règles* : respectivement, le nombre de faits (clauses unitaires) et de règles (clauses non unitaires) du programme ;

– *cycles* : le nombre de composantes fortement connexes du graphe d'appel (autrement dit, le nombre de cycles de symboles de prédicats mutuellement récursifs) ;

– *longueur* : le nombre de symboles de prédicats dans le plus grand cycle ;

– *vars* : la somme des arités des symboles de prédicats dans le plus grand cycle.

Les cinq premières colonnes de la table 3 donnent les temps de calcul (minimum sur dix itérations) concernant un modèle $Post_N$, les contraintes définissant les fonctions de niveau, la génération des fonctions de niveau, le plus petit modèle booléen $Post_{\mathcal{B}}$, les conditions de terminaison. Ensuite, nous précisons le temps global (incluant diverses transformations syntaxiques), la vitesse d'analyse (le nombre moyen de lignes analysées en une seconde) et enfin la qualité de l'analyse (exprimée par le rapport en pourcentage du nombre de relations dont la condition de terminaison calculée par cTI est non-nulle par le nombre total de relations du programme).

Nous commentons à présent les résultats de la table 3. Nous avons déconnecté nTI, notre module de détection d'optimalité des conditions de terminaison. En effet, ce module n'est pas pour le moment suffisamment robuste pour s'attaquer à des programmes de cette taille. La vitesse de cTI est nettement plus faible pour PEEPHOLE que pour les autres programmes. Un examen attentif du code révèle la présence de 5 cycles de longueur 2, ce qui ralentit notablement les calculs des fonctions de niveau. Nous notons que cTI est capable de montrer que BID, CREDIT, et PLAN sont *gauche-terminants* (voir [APT 97], la preuve Prolog de tout atome clos termine universellement). Un programme gauche-terminant a au moins deux propriétés intéressantes. D'une part, sa sémantique close est décidable : la procédure de décision est Prolog, justement ! D'autre part, son opérateur T_P associé possède un unique point fixe ([APT 97], théorème 8.13), ce qui aide à prouver sa correction partielle.

Tableau 3. *[BUE 94], cTI 0.29, Athlon 750 MHz, 256Mo*

temps en [s] programme	$Post_N$	C_μ	μ	$Post_{\mathcal{B}}$	CT	global	lignes/sec	qualité %
ANN	1.07	2.62	0.17	0.46	0.13	5.43	105	48
BID	0.17	0.33	0.03	0.09	0.04	0.81	133	100
BOYER	2.55	0.36	0.03	0.25	0.05	3.91	70	84
BROWSE	0.37	1.01	0.08	0.12	0.03	1.81	59	60
CREDIT	0.12	0.18	0.04	0.07	0.03	0.61	177	100
MINISSAEXP	2.98	4.77	0.49	0.73	0.38	11.03	75	90
PEEPHOLE	1.24	8.94	0.14	0.47	0.12	11.69	28	93
PLAN	0.13	0.32	0.03	0.09	0.03	0.71	90	100
QPLAN	1.52	4.32	0.23	0.62	0.16	7.56	53	68
RDTOK	1.22	0.95	0.07	0.26	0.05	2.92	98	44
READ	1.05	5.25	0.03	0.34	0.16	7.29	41	52
WARPLAN	0.82	1.67	0.03	0.27	0.03	3.18	96	33
moyenne	23%	54%	2%	7%	2%	100%	85	73%

5. Détection de l'optimalité des conditions de terminaison

Nous décrivons à présent nos travaux en cours concernant nTI, le module de détection de l'optimalité des conditions de terminaison.

5.1. *Arguments neutres pour la terminaison*

Les prédicats contiennent fréquemment des arguments qui n'ont aucune influence sur la terminaison universelle gauche. Les structures de données codées par différences ont typiquement cette propriété : dans la plupart des cas, un des arguments de la différence est neutre pour la terminaison. Nous donnons ci-après une définition formelle de cette notion de *neutralité*.

Définition 5 *Soit $A := p(\tilde{x})$ un atome. On dit que $N_p \subseteq [1, n]$ est un* ensemble d'arguments neutres pour la terminaison de *p ssi*

$$\mathcal{A}\big(P, A\big) \textit{ est infini } \Rightarrow \forall\theta \textit{ tq } \mathrm{dom}(\theta) \subseteq \{x_i\}_{i\in N_p},\ \mathcal{A}\big(P, A\theta\big) \textit{ est infini}$$

où $\mathcal{A}\big(P, A\big)$ désigne l'arbre de dérivation de P et A associé à la règle de sélection de Prolog.

La propriété d'être neutre pour la terminaison étant indépendante d'une norme particulière, la détection de tels arguments peut être effectuée préalablement à toute autre analyse, directement sur le source du programme sous forme *homogène* (chaque tête de clause ne contient que des variables, toutes distinctes).

Exemple 2 *Considérons un extrait du programme de la section 2, sous forme homogène :*

$$\begin{array}{l|l}
\mathtt{app}(\mathtt{X}^1_{\mathtt{app}}, \mathtt{X}^2_{\mathtt{app}}, \mathtt{X}^3_{\mathtt{app}}) :- & \mathtt{nrev}(\mathtt{X}^1_{\mathtt{nrev}}, \mathtt{X}^2_{\mathtt{nrev}}) :- \\
\quad \mathtt{X}^1_{\mathtt{app}} = [\,], \mathtt{X}^2_{\mathtt{app}} = \mathtt{X}^3_{\mathtt{app}}. & \quad \mathtt{X}^1_{\mathtt{nrev}} = [\,], \mathtt{X}^2_{\mathtt{nrev}} = [\,]. \\
\mathtt{app}(\mathtt{X}^1_{\mathtt{app}}, \mathtt{X}^2_{\mathtt{app}}, \mathtt{X}^3_{\mathtt{app}}) :- & \mathtt{nrev}(\mathtt{X}^1_{\mathtt{nrev}}, \mathtt{X}^2_{\mathtt{nrev}}) :- \\
\quad \mathtt{X}^1_{\mathtt{app}} = [\mathtt{X}|\mathtt{Xs}], \mathtt{X}^3_{\mathtt{app}} = [\mathtt{X}|\mathtt{Zs}], & \quad \mathtt{X}^1_{\mathtt{nrev}} = [\mathtt{X}|\mathtt{Xs}], \mathtt{nrev}(\mathtt{Xs}, \mathtt{Zs}), \\
\quad \mathtt{app}(\mathtt{Xs}, \mathtt{X}^2_{\mathtt{app}}, \mathtt{Zs}). & \quad \mathtt{app}(\mathtt{Zs}, [\mathtt{X}], \mathtt{X}^2_{\mathtt{nrev}}).
\end{array}$$

L'ensemble des arguments neutres pour la terminaison de $\mathtt{app}/3$ *est* $N_{app} = \{2\}$*, de même* $N_{nrev} = \{2\}$.

Il existe un lien étroit entre les arguments neutres pour la terminaison et les conditions de terminaison de la section 2.

Propriété 1 *Soient* $A := p(\tilde{x})$ *un atome de* P*,* N_p *un ensemble d'arguments neutres pour la terminaison de* p *et* I *un sous-ensemble de* N_p*. S'il existe une dérivation gauche infinie pour* P *et* A*, alors le mode* $\wedge_{i \in I} x_i$ *n'est pas une condition de terminaison.*

Preuve. Nous appliquons la définition 5. La prémisse de l'implication est vraie par hypothèse. En choisissant $\theta = \{x_i/toto \mid i \in I\}$, on a bien un exemple de requête appartenant au mode $\wedge_{i \in I} x_i$ qui ne termine pas. □

5.2. *TNA : un algorithme de détection des arguments neutres pour la terminaison*

Soit P un programme logique sous forme homogène définissant un ensemble Π_P de symboles de prédicats.

$$P = \left\{ \mathcal{C}^k_p : p(\tilde{x}_p) \leftarrow \mathcal{S}^k_p \;\middle|\; p \in \Pi_P,\ 1 \le k \le n_p \right\} .$$

Voici comment détecter des arguments neutres pour la terminaison pour des prédicats de Π_P. Pour tout $p \in \Pi_P$, nous associons à chaque i-ème argument de p une variable booléenne notée B^i_p. La valeur 1 signifie que la position est neutre pour la terminaison, tandis que 0 signifie qu'*il se peut* que ce ne soit pas le cas. Nous allons lier ces variables par des contraintes booléennes dont la résolution donnera des informations sur la neutralité de certains arguments.

Les contraintes sont générées en décomposant le corps de chaque clause définissant p en une concaténation

$$\mathcal{S}^k_p \stackrel{d\acute{e}f}{=} E^k_p \star F^k_p \star G^k_p \star H^k_p$$

de séquences d'atomes possiblement vides. E^k_p contient des atomes quelconques, F^k_p contient un seul atome A tel que $\mathcal{A}\big(P, A\big)$ peut être infini, G^k_p contient des atomes

A tels que $\mathcal{A}\big(P, A\big)$ est toujours fini et H_p^k commence par un atome A tel que la dérivation de P et A échoue toujours suivi d'atomes quelconques. Si x_p^i apparaît dans E_p^k, il se peut que i ne soit pas neutre pour la terminaison de la dérivation de P et $p(\tilde{x}_p)$. On génère donc la contrainte $B_p^i = 0$. Sinon, considérons le cas où x_p^i apparaît dans $F_p^k = [q(\dots)]$ au sein d'un argument en position j. La position i peut ne pas être neutre pour la terminaison de la dérivation de P et $p(\tilde{x}_p)$ uniquement dans le cas où la position j n'est pas neutre pour la terminaison de la dérivation de P et $q(\tilde{x}_q)$. La contrainte $\neg B_q^j \Rightarrow \neg B_p^i$ assure que si B_q^j vaut 0 (*i.e.* peut influencer la terminaison), alors B_p^i vaut aussi 0. Enfin, si x_p^i n'apparaît dans aucune des deux premières séquences, nous ne générons pas de contraintes.

La mise sous forme résolue, via un solveur booléen complet, des contraintes générées aboutit à une assignation partielle. En effet, la valeur de certaines variables booléennes peut ne pas être déterminée de manière unique par ces contraintes. Chaque variable pour laquelle la forme résolue n'a pas affecté la valeur 0 est ensuite fixée à 1 (on parle parfois d'instanciation maximale).

Notons qu'il peut exister plusieurs décompositions respectant les spécifications définies plus haut. Notre preuve de correction de l'algorithme (voir l'annexe I) ne dépend que de ces spécifications. Elle est indépendante du fait que, en pratique, l'algorithme tente de générer un minimum de contraintes en remplissant autant que possible G_p^k et H_p^k. L'objectif est d'obtenir le maximum de variables affectées à 1, c'est-à-dire le plus grand ensemble d'arguments neutres pour la terminaison.

Exemple 3 *On considère le programme donné à l'exemple 2. Les décompositions et ensembles de contraintes associés sont :*

– app *clause 1 :* $[\,] \star [\,] \star [\mathtt{X}^1_{\mathtt{app}} = [\,], \mathtt{X}^2_{\mathtt{app}} = \mathtt{X}^3_{\mathtt{app}}] \star [\,] \quad \to \quad \varnothing$

– app *clause 2 :* $[\mathtt{X}^1_{\mathtt{app}} = [\mathtt{X}|\mathtt{Xs}], \mathtt{X}^3_{\mathtt{app}} = [\mathtt{X}|\mathtt{Zs}]\,] \star [\mathtt{app}(\mathtt{Xs}, \mathtt{X}^2_{\mathtt{app}}, \mathtt{Zs})] \star [\,] \star [\,]$

$$\to \{B^1_{app} = 0, B^3_{app} = 0, \neg B^2_{app} \Rightarrow \neg B^2_{app}\}$$

– nrev *clause 1 :* $[\,] \star [\,] \star [\mathtt{X}^1_{\mathtt{nrev}} = [\,], \mathtt{X}^2_{\mathtt{nrev}} = [\,]\,] \star [\,] \quad \to \quad \varnothing$

– nrev *clause 2 :* $[\mathtt{X}^1_{\mathtt{nrev}} = [\mathtt{X}|\mathtt{Xs}], \mathtt{nrev}(\mathtt{Xs}, \mathtt{Zs})] \star [\mathtt{app}(\mathtt{Zs}, [\mathtt{X}], \mathtt{X}^2_{\mathtt{nrev}})] \star [\,] \star [\,]$

$$\to \{B^1_{nrev} = 0, \neg B^3_{app} \Rightarrow \neg B^2_{nrev}\}$$

La résolution de ces contraintes conduit dans un premier temps à $\{B^1_{app} = 0, B^2_{app} = ?, B^3_{app} = 0, B^1_{nrev} = 0, B^2_{nrev} = 0\}$ *puis à :*

$$\big\{B^1_{app} = 0, B^2_{app} = 1, B^3_{app} = 0, B^1_{nrev} = 0, B^2_{nrev} = 0\big\} \;.$$

Dans l'exemple précédent, pour `app/3`, les résultats sont optimaux. Cependant, pour `nrev/2`, nous n'arrivons pas à inférer que le second argument est neutre pour la terminaison. Donc l'algorithme TNA est correct (voir Annexe I) et incomplet (on peut raisonnablement conjecturer qu'il n'existe pas d'algorithme complet pour ce problème). L'ordre de grandeur du temps d'exécution de TNA est similaire à celui de l'analyse du texte du programme via le prédicat `read/1` de SICStus Prolog écrit en C. Nous avons toujours observé que notre analyse est plus rapide qu'une double lecture du programme (l'interface web de cTI fournit une option « TNA only » qui permet d'effectuer cette comparaison).

5.3. *nTI : un algorithme de détection d'optimalité des conditions de terminaison*

Soient $A := p(x_1, \dots, x_n)$ un atome, $CT_p(x_1, \dots, x_n)$ une condition de terminaison, N_p un ensemble de d'arguments neutres pour la terminaison de p. Imaginons que nous disposons également du test $boucle(P, A)$, qui termine toujours et qui est vrai ssi il existe une dérivation gauche infinie de P et A. L'algorithme suivant permet d'affirmer, dans certains cas, que $CT_p(x_1, \dots, x_n)$ est optimale.

1: **si** $CT_p(\tilde{x}) = 1$ **alors** la condition de terminaison est optimale; **fin**;
2: $S_0 :=$ l'ensemble des modes de A;
3: $S_1 := S_0$ privé des modes moins généraux de $CT_p(\tilde{x})$ (y compris CT_p);
4: **si** $\neg boucle(P, A)$ **alors** CT_p n'est pas optimale et $CT'_p := 1$ est optimale; **fin**;
5: **si** $boucle(P, A)$ **alors**
6: $\quad$ $S_2 := S_1$ privé des modes $\{\wedge x_{i \in I} \mid I \subseteq N_p\}$;
7: $\quad$ **si** $S_2 = \varnothing$ **alors** CT_p est optimale **sinon** CT_p n'est pas optimale;

Preuve. Si la condition de terminaison est équivalente à 1, le résultat est optimal. Sinon, S_1 est l'ensemble (non-vide) des modes pour lesquels on ne sait rien. Si le test $boucle(P, A)$ est faux alors la condition calculée CT_p n'est pas optimale. Si le test est vrai, d'après la propriété 1, S_2 représente, après l'affectation de la ligne 6, l'ensemble des modes pour lesquels nous n'avons pas d'information concernant la terminaison ou la non-terminaison. Si cet ensemble est vide, alors on a bien montré l'optimalité de CT_p. □

En pratique, CT_p est calculée par cTI, N_p par TNA, le test $boucle$ est implanté par un détecteur de boucle incomplet (voir [BOL 91], par exemple) auquel nous allouons des ressources finies. Notons que nous nous intéressons uniquement à l'existence ou l'inexistence d'une boucle. Dans le cas où S_2 est non-vide à la ligne 7, nous tentons au moyen d'heuristiques, pour quelques modes M bien choisis de S_2, de générer de judicieuses instances $A' \in ||M||$ de A par la suite testées par $boucle(P, A')$. Nous arrivons souvent (mais pas toujours) à réduire l'ensemble S_2. Dans le cas où celui-ci se vide complètement, nous concluons à l'optimalité de CT_p.

Exemple 4 *Considérons une dernière fois l'exemple de la section 2.*

– Pour la relation `app`*, cTI infère la condition de terminaison* $CT_{app}(x, y, z) = x \vee z$*,* $N_{app} = \{2\}$ *et* `boucle(P,app(X,Y,Z))` *est vrai. Au début de la ligne 4,*

$S_1 = \{1, y\}$. *Après la ligne 6,* $S_2 = \emptyset$. *On conclut que la condition de terminaison est optimale.*

– Pour la relation `nrev`*, cTI infère la condition de terminaison* $CT_{nrev}(x, y) = x$. *TNA ne fournit aucune indication de neutralité et* `boucle(P,nrev(X,Y))` *est vrai. Au début de la ligne 4,* $S_1 = \{1, y\}$. *Après la ligne 6,* $S_2 = \{y\}$. *Nos heuristiques génèrent* `nrev(Xs,toto)`*, une requête concrète qui pourrait ne pas terminer. Le test* `boucle(P,nrev(Xs,toto))` *est vrai, on conclut à l'optimalité de la condition de terminaison.*

– Pour `app3`*, cTI infère la condition de terminaison* $CT_{app3}(x, y, z, u) = (x \wedge y) \vee (x \wedge u)$, $N_{app3} = \{3\}$ *et* `boucle(P,app3(X,Y,Z,U))` *est vrai. Au début de la ligne 4,* $S_1 = \{1, x, y, z, u, x \wedge z, y \wedge z, y \wedge u, z \wedge u, y \wedge z \wedge u\}$. *Après la ligne 6,* $S_2 = \{x, y, u, x \wedge z, y \wedge z, y \wedge u, z \wedge u, y \wedge z \wedge u\}$. *Le test* `boucle(P,app3(X,toto2,toto3,toto4))` *est vrai. D'où* $S_2 = \{x, x \wedge z\}$. *Enfin, le test* `boucle(P,app3([],Y,toto3,U))`*, vrai, réduit* S_2 *à* $\emptyset$. *Donc la condition* $CT_{app3}(x, y, z, u) = (x \wedge y) \vee (x \wedge u)$ *est optimale.*

6. Conclusion

Nous avons présenté l'architecture de cTI et évalué ses performances. Pour tous les programmes classiques concernant la terminaison de Prolog, notre outil est capable d'inférer des classes de requêtes incluant celles testées par les analyseurs disponibles aujourd'hui (bien qu'il nous ait fallu, à quatre reprises, régler manuellement les paramètres de cTI). Pour ces programmes, le module additionnel nTI est le plus souvent à même de montrer l'optimalité des classes inférées par cTI. Autrement dit, malgrè l'indécidabilité du problème et l'emploi de conditions suffisantes, pour ces programmes concrets, les informations calculées *caractérisent* la propriété de terminaison, relativement au langage utilisé pour décrire les classes de requêtes. Donc pour faire mieux, il conviendrait de définir un langage de description de classes de requêtes plus fin, tirant parti, par exemple, d'information sur les types des arguments. Confronté à des programmes de taille moyenne et bien que nécessitant trois calculs de points fixes, la vitesse d'analyse de cTI reste, nous semble-t-il, acceptable, d'autant que cette analyse peut être effectuée de manière incrémentale, au cours du développement des programmes. Enfin, l'analyse du programme `CHAT` (contenant un cycle de 30 relations avec une moyenne de 8 arguments par relations !) suggérée par Paul Tarau, nous a conduit à allouer des ressources finies en temps aux opérations les plus coûteuses de cTI. En cas de dépassement des ressources, ces opérations retournent des résultats qui ne nuisent pas à la correction de cTI mais affaiblissent la qualité de l'analyse. En revanche, la complexité en temps de cTI est maintenant linéaire (moyennant une imposante constante multiplicative fixée à une minute) relativement au nombre de cycles du graphe de dépendance du programme analysé !

7. Bibliographie

[APT 93] APT K. R., PEDRESCHI D., « Reasoning about termination of pure prolog programs », *Information and computation*, vol. 1, n° 106, 1993, p. 109–157.

[APT 94] APT K. R., PEDRESCHI D., « Modular termination proofs for logic and pure Prolog programs », LEVI G., Ed., *Advances in Logic Programming Theory*, p. 183–229, Oxford University Press, 1994.

[APT 97] APT K., *From Logic Programming to Prolog*, Prentice Hall, 1997.

[BEN 97] BENOY F., KING A., « Inferring Argument Size Relationships with CLP(R) », *Logic Program Synthesis and Transformation*, Springer-Verlag, 1997.

[BOL 91] BOL R., « Loop Checking in Logic Programming », PhD thesis, CWI, Amsterdam, 1991.

[BUE 94] BUENO F., GARCIA DE LA BANDA M., HERMENEGILDO M., « Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization », *Proceedings of International Symposium on Logic Programming*, 1994, p. 320–336.

[COU 77] COUSOT P., COUSOT R., « Abstract interpretation: a unifed lattice model for static analysis of programs by construction or approximation of fixpoints », *Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, , 1977, p. 238–252.

[COU 78] COUSOT P., HALBWACHS N., « Automatic discovery of linear restraints among variables of a program », *Proc. of the 5th ACM Symposium on Principles of Programming Languages*, , 1978, p. 84–96.

[COU 92] COUSOT P., COUSOT R., « Abstract interpretation and application to logic programs », *Journal of Logic Programming*, vol. 13, 1992, p. 103–179.

[DEC 00] DECORTE S., DE SCHREYE D., VANDECASTEELE H., « Constraint-based Termination Analysis of Logic Programs », *ACM Transactions of Computational Logic*, , 2000.

[DES 94] DE SCHREYE D., DECORTE S., « Termination of logic programs : the never-ending story », *Journal of Logic Programming*, vol. 19-20, 1994, p. 199–260.

[JAF 87] JAFFAR J., LASSEZ J. L., « Constraint Logic Programming », *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, 1987, p. 111–119.

[JAF 94] JAFFAR J., MAHER M. J., « Constraint Logic Programming: a survey », *Journal of Logic Programming*, vol. 19, 1994, p. 503–581.

[LIN 97] LINDENSTRAUSS N., SAGIV Y., « Automatic Termination Analysis of Logic Programs », *Proceedings of the 14th ICLP*, 1997, p. 63–77.

[MES 96] MESNARD F., « Inferring left-terminating classes of queries for constraint logic programs by means of approximations », *Proceedings of JICSLP'96*, The MIT Press, 1996, p. 7–21.

[MES 01] MESNARD F., RUGGIERI S., « On Proving Left Termination of Constraint Logic Programs », rapport, 2001, Université de La Réunion.

[PLÜ 90] PLÜMER L., « Termination proofs for Logic programs », *Lecture Notes in Artificial Intelligence*, vol. 446, 1990, Springer-Verlag.

[RUG 99] RUGGIERI S., « Verification and validation of logic programs », PhD thesis, Università di Pisa, 1999.

[SOH 91] SOHN K., VAN GELDER A., « Termination detection in logic programs using argument sizes », *Proceedings of PODS'91*, 1991, p. 216–226.

[SPE 97] SPEIRS C., SOMOGYI Z., SØNDERGAARD H., « Termination Analysis for Mercury », VAN HENTENRICK P., Ed., *Proceedings of SAS'97*, vol. 1302 de *Lecture Notes in Computer Science*, Springer, 1997.

[VAS 86] VASAK T., POTTER J., « Characterization of terminating logic programs », *Proceedings of the 1986 IEEE Symposium on Logic Programming*, 1986.

Annexe I : preuve de correction de l'algorithme TNA

Propriété 2 *L'algorithme TNA décrit à la section 5.2 est correct.*

Preuve. Supposons que pour un prédicat p, on aboutisse à la réponse

$$\forall i \in \mathrm{I}, B_p^i = 1$$

pour un ensemble I de d'arguments de p. Montrons que I est un ensemble d'arguments neutres pour la terminaison de p. Pour cela, considérons une séquence $\tilde{y}$ de variables distinctes n'apparaissant pas dans P et supposons que $\mathcal{A}\big(P, p(\tilde{y})\big)$ soit infini. Notons y^{I} l'ensemble $\{y^i \mid i \in \mathrm{I}\}$ et considérons une substitution θ telle que $\mathrm{dom}(\theta) \subseteq y^{\mathrm{I}}$. Montrons, en décrivant la structure de $\mathcal{A}\big(P, p(\tilde{y})\big)$, que $\mathcal{A}\big(P, p(\tilde{y}\theta)\big)$ est également infini.

Toute clause $\mathcal{C}_p^k$ de P a pour tête $p(\tilde{x}_p)$ qui s'unifie à $p(\tilde{y})$. L'unification de ces deux atomes conduit à un unificateur le plus général, disons σ. La résolvante de $\mathcal{C}_p^k$ et du but $\leftarrow p(\tilde{y})$ selon σ est

$$\leftarrow \big(E_p^k \star F_p^k \star G_p^k \star H_p^k\big)\sigma. \qquad (1)$$

La séquence $E_p^k\sigma$ ne contient aucune des variables de y^{I} car nous avons supposé, d'une part, que $\tilde{y}$ ne contient aucune des variables du programme et, d'autre part, que l'algorithme répond $B_p^i = 1$ pour tout i dans I.

– Si E_p^k est vide, alors F_p^k ne l'est pas, sinon, par spécification de G_p^k et H_p^k, l'arbre serait fini. Donc, d'après sa spécification, F_p^k est constituée d'un unique élément, disons $q(\sim)$. Dans $q(\sim)$, les variables de y^{I} apparaissent au sein d'un ou plusieurs arguments. Supposons que chaque variable y^i de y^{I} apparaît dans des positions d'argument de q constituant un ensemble noté $\mathrm{Pos}\big(y^i\big)$. Notons que pour chaque élément j de cet ensemble, notre algorithme a généré une contrainte de la forme $\neg B_q^j \Rightarrow \neg B_p^i$. Comme on a obtenu la réponse $B_p^i = 1$, on a forcément aussi obtenu $B_q^j = 1$.

Toute clause $\mathcal{C}_q^{k'}$ de P a pour tête $q(\tilde{x}_q)$ qui s'unifie à $q(\sim)$. L'unification de la tête de $\mathcal{C}_q^{k'}$ et de $q(\sim)$ conduit à un unificateur le plus général, disons σ. La résolvante du but de la ligne (1) et de $\mathcal{C}_q^{k'}$ selon σ' est le but

$$\leftarrow \big(E_q^{k'} \star F_q^{k'} \star G_q^{k'} \star H_q^{k'} \star G_p^k \star H_p^k\big)\sigma\sigma'.$$

Comme pour tout $i \in \mathrm{I}$ et tout $j \in \mathrm{Pos}(y^i)$ on a $B_q^j = 1$, la variable y^i n'apparaît pas dans la séquence $E_q^{k'}\sigma\sigma'$. Le traitement de ce but est donc similaire à celui de la ligne (1).

– Si E_p^k n'est pas vide, alors on pose $E_p^k = [g_p^k|\Delta_p^k]$. Si g_p^k est une unification, appelons σ' la substitution qu'elle engendre. Le traitement de cette unification conduit, à partir du but de la ligne (1), à

$$\leftarrow \left(\Delta_p^k \star F_p^k \star G_p^k \star H_p^k\right)\sigma\sigma'.$$

La séquence $\Delta_p^k\sigma\sigma'$ ne contenant aucune des variables de y^{I}, le traitement de ce but est similaire à celui de la ligne (1).

Si g_p^k n'est pas une unification, c'est un atome de la forme $q(\sim)$. Toute clause $\mathcal{C}_q^{k'}$ de P a pour tête $q(\tilde{x}_q)$ qui s'unifie à $q(\sim)$. L'unification de la tête de $\mathcal{C}_q^{k'}$ et de $q(\sim)$ conduit à un unificateur le plus général, disons σ'. La résolvante du but de la ligne (1) et de $\mathcal{C}_q^{k'}$ selon σ' est

$$\leftarrow \mathrm{Res} \star \left(\Delta_p^k \star F_p^k \star G_p^k \star H_p^k\right)\sigma\sigma'$$

où Res est la résolvante de $\leftarrow q(\sim)$ et $\mathcal{C}_q^{k'}$ selon σ'. Maintenant, ou bien Res conduit à une branche infinie, ou bien Res s'efface au fur et à mesure. Le dernier de ces deux cas engendre une substitution, disons σ''. Comme Res ne contient aucune des variables de y^{I}, celles-ci n'appartiennent ni au domaine ni à l'image de σ''. On est alors ramené au but

$$\leftarrow \left(\Delta_p^k \star F_p^k \star G_p^k \star H_p^k\right)\sigma\sigma'\sigma''.$$

où la séquence $\Delta_p^k\sigma\sigma'\sigma''$ ne contient aucune des variables de y^{I}. Le traitement de ce but est donc similaire à celui de la ligne (1).

Notons enfin qu'en remplaçant, dans l'arbre de dérivation de P et $p(\tilde{y})$ que l'on vient de décrire, les variables de y^{I} par leur valeur par θ, on obtient bien l'arbre de dérivation de P et $p(\tilde{y}\theta)$. Celui-ci est donc infini, ce qui termine notre démonstration. □