Inférence de non-terminaison pour les programmes logiques avec contraintes

Etienne Payet, Frédéric Mesnard

Inférence de non-terminaison pour les programmes logiques avec contraintes

ETIENNE PAYET et FRED MESNARD

IREMIA - université de La Réunion, France
email : {epayet, fred}@univ-reunion.fr

RÉSUMÉ. *L'analyse de terminaison des programmes logiques a été sujette à une recherche intensive durant les deux dernières décennies. La majorité des travaux s'est intéressée à la terminaison universelle gauche d'une classe donnée de requêtes, c'est-à-dire au fait que toutes les dérivations des requêtes de cette classe produites par un moteur Prolog sont finies. En revanche, l'étude du problème* dual *: la non-terminaison par rapport à la règle de sélection gauche,* i.e *l'existence d'une requête dans une classe donnée qui admet une dérivation gauche infinie, a fait l'objet de peu d'articles. Dans ce papier, nous étudions la non-terminaison dans le contexte de la programmation logique avec contraintes. Nous reformulons, dans ce cadre plus abstrait, les concepts que nous avions définis pour la programmation logique, ce qui nous donne des critères nécessaires et suffisants exprimés de façon logique ainsi que des preuves plus simples. Par ailleurs, en reconsidérant nos travaux précédents, nous démontrons que dans un certain sens, nous détenions déjà le meilleur critère syntaxique dans le cas de la programmation logique. Enfin, nous décrivons un ensemble d'algorithmes corrects pour l'inférence de non-terminaison des programmes CLP.*

ABSTRACT. *Termination has been a subject of intensive research in the logic programming community for the last two decades. Most works deal with proving universal left termination of a given class of queries,* i.e. *finiteness of all the possible derivations produced by a Prolog engine from any query in that class. In contrast, the study of the* dual *problem: non-termination relativement à the left selection rule* i.e *the existence of one query in a given class of queries which admits an infinite left derivation, has given rise to only a few papers. In this article, we study non-termination in the more general constraint logic programming framework. We rephrase our previous logic programming approach into this more abstract setting, which leads a necessary and sufficient criteria expressed in a logical way and simpler proofs, as expected. Also, by reconsidering our previous work, we now prove that in some sense, we already had the best syntactic criterion for logic programming. Last but not least, we offer a set of correct algorithms for inferring non-termination for CLP.*

MOTS-CLÉS : *programmation logique avec contraintes, analyse statique, non-terminaison*

KEYWORDS: *constraint logic programming, static analysis, non-termination*

1. Introduction

L'analyse de terminaison des programmes logiques a été sujette à une recherche intensive durant les deux dernières décennies, voir [De 94]. Une étude plus récente du sujet et son extension à la programmation logique avec contraintes [JAF 87, JAF 98] est effectuée dans [MES 03]. La majorité des travaux s'est intéressée à la terminaison universelle gauche d'une classe donnée de requêtes, c'est-à-dire au fait que toutes les dérivations des requêtes de cette classe produites par un moteur Prolog sont finies. Certains de ces travaux, par exemple [MES 96, GEN 01, MES 01], considèrent le problème *inverse* consistant à inférer des classes de requêtes pour lesquelles la terminaison universelle gauche est assurée.

En revanche, l'étude du problème *dual* : la non-terminaison par rapport à la règle de sélection gauche, *i.e* l'existence d'une requête dans une classe donnée qui admet une dérivation gauche infinie, a fait l'objet de peu d'articles, par exemple [De 89, De 90]. Récemment, nous avons nous aussi étudié ce problème dans le cadre de la programmation logique [MES 02], où nous avons proposé une analyse permettant l'inférence de non-terminaison.

Dans ce papier, nous étudions la non-terminaison dans le contexte de la programmation logique avec contraintes. Nous reformulons, dans ce cadre plus abstrait, les concepts que nous avions définis pour la programmation logique, ce qui nous donne des critères nécessaires et suffisants exprimés de façon logique ainsi que des preuves plus simples. Par ailleurs, en reconsidérant nos travaux précédents, nous démontrons que dans un certain sens, nous détenions déjà le meilleur critère syntaxique dans le cas de la programmation logique. Enfin, nous décrivons un ensemble d'algorithmes corrects pour l'inférence de non-terminaison des programmes CLP. Notre analyse est complètement implantée[1].

Le papier est organisé de la façon suivante. Après les préliminaires de la section 2, nous décrivons, dans la section 3, des propriétés de base concernant la non-terminaison de programmes logiques avec contraintes. L'arsenal technique sous-jascent à notre approche est présenté dans les sections 4 et 5. Enfin, la section 6 conclut l'article.

2. Préliminaires

Nous rappelons au lecteur quelques définitions de base concernant la programmation logique avec contraintes, voir [JAF 98] pour plus de détails.

2.1. *Domaines de contraintes*

Nous considérons un langage de programmation logique avec contraintes CLP($\mathcal{C}$) basé sur le domaine de contraintes $\mathcal{C} := \langle \Sigma_{\mathcal{C}}, \mathcal{L}_{\mathcal{C}}, \mathcal{D}_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}}, solv_{\mathcal{C}} \rangle$.

1. `http ://www.univ-reunion.fr/~gcc/`

$\Sigma_{\mathcal{C}}$ est la *signature* du domaine de contraintes, c'est-à-dire un couple $\langle F_{\mathcal{C}}, \Pi_{\mathcal{C}} \rangle$ où $F_{\mathcal{C}}$ est un ensemble de symboles de fonctions et $\Pi_{\mathcal{C}}$ est un ensemble de symboles de prédicats. La classe de contraintes $\mathcal{L}_{\mathcal{C}}$ est un ensemble de $\Sigma_{\mathcal{C}}$-formules du premier ordre. Le *domaine of calcul* $\mathcal{D}_{\mathcal{C}}$ est une $\Sigma_{\mathcal{C}}$-structure qui est l'interprétation désirée des contraintes et $D_{\mathcal{C}}$ est le *domaine* de $\mathcal{D}_{\mathcal{C}}$. La *théorie des contraintes* $\mathcal{T}_{\mathcal{C}}$ est une $\Sigma_{\mathcal{C}}$-théorie qui décrit la sémantique logique des contraintes. On suppose que $\mathcal{C}$ est *idéal i.e.* le *solveur de contraintes*, $solv_{\mathcal{C}}$, est une fonction calculable qui associe `true` ou `false` à chaque formule de $\mathcal{L}_{\mathcal{C}}$ ce qui indique si la formule est satisfiable ou pas.

On convient que le symbole de prédicat $=$ est un élément de $\Sigma_{\mathcal{C}}$ et qu'il est interprété comme l'identité dans $D_{\mathcal{C}}$. Une *contrainte primitive* est soit la contrainte toujours satisfiable $true$, soit la contrainte insatisfiable $false$ ou alors est de la forme $p(\tilde{t})$ où $p \in \Pi_{\mathcal{C}}$ et $\tilde{t}$ est une séquence finie de termes de $\Sigma_{\mathcal{C}}$. On suppose que $\mathcal{L}_{\mathcal{C}}$ contient toutes les contraintes primitives et que cet ensemble est clos par renommage de variable, quantification existentielle et conjonction.

On suppose que $\mathcal{D}_{\mathcal{C}}$ et $\mathcal{T}_{\mathcal{C}}$ *correspondent* sur $\mathcal{L}_{\mathcal{C}}$ *i.e.*

– $\mathcal{D}_{\mathcal{C}}$ est un modèle de $\mathcal{T}_{\mathcal{C}}$ et

– pour toute contrainte $c \in \mathcal{L}_{\mathcal{C}}$, $\mathcal{D}_{\mathcal{C}} \models \exists c$ si et seulement si $\mathcal{T}_{\mathcal{C}} \models \exists c$.

De plus, on suppose que $\mathcal{T}_{\mathcal{C}}$ est *complet pour la satisfaction* par rapport à $\mathcal{L}_{\mathcal{C}}$ *i.e.* pour toute contrainte $c \in \mathcal{L}_{\mathcal{C}}$, ou bien $\mathcal{T}_{\mathcal{C}} \models \exists c$ ou bien $\mathcal{T}_{\mathcal{C}} \models \neg\exists c$. On suppose également que la théorie et le solveur s'*accordent* dans le sens où pour tout $c \in \mathcal{L}_{\mathcal{C}}$, $solv_{\mathcal{C}}(c) = \texttt{true}$ si et seulement si $\mathcal{T}_{\mathcal{C}} \models \exists c$. Par conséquent, comme $\mathcal{D}_{\mathcal{C}}$ et $\mathcal{T}_{\mathcal{C}}$ correspondent sur $\mathcal{L}_{\mathcal{C}}$, on a, pour tout $c \in \mathcal{L}_{\mathcal{C}}$, $solv_{\mathcal{C}}(c) = \texttt{true}$ si et seulement si $\mathcal{D}_{\mathcal{C}} \models \exists c$.

Une *valuation* est une fonction qui associe à toute variable une valeur de $D_{\mathcal{C}}$. On écrit $O\sigma$ (au lieu de $\sigma(O)$) pour dénoter le résultat de l'application d'une valuation σ à un objet O. Si c est une contrainte, on écrit $\mathcal{D}_{\mathcal{C}} \models c$ si pour toute valuation σ, $c\sigma$ est vraie dans $\mathcal{D}_{\mathcal{C}}$ *i.e.* $\mathcal{D}_{\mathcal{C}} \models_{\sigma} c$. Donc, $\mathcal{D}_{\mathcal{C}} \models c$ est la même chose que $\mathcal{D}_{\mathcal{C}} \models \forall c$. Les valuations sont dénotées par σ, η, θ, ... dans la suite de ce papier.

Exemple 1 ($\mathcal{R}_{lin}$) *Le domaine de contraintes $\mathcal{R}_{lin}$ a $<$, $\leq$, $=$, $\geq$ et $>$ pour symboles de prédicats, $+$, $-$, $*$, $/$ pour symboles de fonctions et les séquences de chiffres (avec éventuellement un point décimal) pour symboles de constantes. Seules les contraintes linéaires sont admises. Le domaine de calcul est une structure dont l'ensemble des nombres réels est le domaine et dont les symboles de fonctions et de prédicats sont interprétés par les fonctions et relations usuelles sur les réels. La théorie $\mathcal{T}_{\mathcal{R}_{lin}}$ est celle décrite dans [SHO 67]. Un solveur de contraintes pour $\mathcal{R}_{lin}$ renvoyant soit* `true` *soit* `false` *est décrit dans [REF 96].*

Exemple 2 (Programmation Logique) *Le domaine de contraintes $Term$ a comme symbole de prédicat le symbole $=$ et comme symboles de fonctions les chaînes de caractères alphanumériques. Le domaine de calcul de $Term$ est l'ensemble des* arbres finis *(ou, de façon équivalente, des termes finis), $Tree$, tandis que la théorie $\mathcal{T}_{Term}$ est*

la théorie égalitaire de Clark [CLA 78]. L'interprétation d'une constante est un arbre avec un seul nœud étiqueté par la constante. L'interprétation d'un symbole de fonction n-aire f est la fonction $f_{Tree} : Tree^n \rightarrow Tree$ associant aux arbres T_1, ..., T_n un nouvel arbre dont la racine, étiquetée par f, a pour fils T_1, ..., T_n. Un solveur de contraintes renvoyant soit `true` *soit* `false` *est fourni par l'algorithme d'*unification. *CLP($Term$) coïncide alors avec la programmation logique.*

2.2. *Sémantique opérationnelle*

La signature servant à écrire les programmes et requêtes considérés est $\Sigma_L := \langle F_L, \Pi_L \rangle$ avec $F_L := F_{\mathcal{C}}$ et $\Pi_L := \Pi_{\mathcal{C}} \cup \Pi'_L$ où Π'_L, l'ensemble des symboles de prédicats pouvant être définis dans les programmes, est disjoint de $\Pi_{\mathcal{C}}$. On suppose que chaque symbole de prédicat p de Π_L a une arité unique dénotée par $arity(p)$.

Un *atome* a la forme $p(\tilde{t})$ où $p \in \Pi'_L$, $arity(p) = n$ et $\tilde{t}$ est une séquence de n termes de Σ_L. Un *programme* de CLP($\mathcal{C}$) est un ensemble fini de règles. Une *règle* a la forme $p(\tilde{x}) \leftarrow c \diamond q_1(\tilde{y}_1), \ldots, q_n(\tilde{y}_n)$ où p, q_1, ..., q_n sont des symboles de prédicats de Π'_L, c est une conjonction finie de contraintes primitives et $\tilde{x}$, $\tilde{y}_1$, ..., $\tilde{y}_n$ sont des séquences disjointes de variables distinctes. Ainsi, c est la conjonction de toutes les contraintes, unifications comprises. Une *requête* a la forme $\langle Q \mid d \rangle$ où Q est une séquence finie d'atomes et d est une conjonction finie de contraintes primitives. Lorsque Q contient exactement un atome, la requête est dite *atomique*. La séquence vide d'atomes est dénotée par $\Box$. L'ensemble des variables apparaissant dans un objet syntaxique est dénoté par $Var(O)$.

Les exemples de ce papier utilisent les langages CLP($\mathcal{R}_{lin}$) et CLP($Term$). Les exemples de programmes et de requêtes sont rédigés en police `courrier`. Les variables apparaissant dans les programmes et les requêtes commencent par une majuscule, $[Head|Tail]$ dénote une liste dont la tête est $Head$ et la queue est $Tail$, et $[\,]$ dénote la liste vide.

On considère la sémantique opérationnelle suivante définie en termes de *dérivation gauche* d'une requête en une autre requête. Soit $\langle p(\tilde{t}), Q \mid d \rangle$ une requête et r une règle. Soit $r' := p(\tilde{x}) \leftarrow c \diamond \mathbf{B}$ une variante de r variable disjointe de $\langle p(\tilde{t}), Q \mid d \rangle$ et telle que $solv_{\mathcal{C}}(\tilde{x} = \tilde{t} \wedge c \wedge d) =$ `true` (où $\tilde{x} = \tilde{t}$ dénote la contrainte $x_1 = t_1 \wedge \cdots \wedge x_n = t_n$ avec $\tilde{x} := x_1, \ldots, x_n$ et $\tilde{t} := t_1, \ldots, t_n$). Alors,

$$\langle p(\tilde{t}), Q \mid d \rangle \underset{r}{\Longrightarrow} \langle \mathbf{B}, Q \mid \tilde{x} = \tilde{t} \wedge c \wedge d \rangle$$

est un *pas de dérivation gauche* avec r' comme *règle d'entrée*. On écrit $S \overset{+}{\underset{P}{\Longrightarrow}} S'$ pour résumer un nombre fini (> 0) de pas de dérivation gauche de S à S' où chaque règle d'entrée est une variante d'une règle de P. Soit S_0 une requête. Une séquence maximale $S_0 \underset{r_1}{\Longrightarrow} S_1 \underset{r_2}{\Longrightarrow} \cdots$ de pas de dérivation gauche est appelée *dérivation gauche* de $P \cup \{S_0\}$ si r_1, r_2, ... sont des règles de P et si la condition de *standardisation* est vérifiée, *i.e.* chaque règle d'entrée utilisée est variable disjointe de la requête initiale

S_0 et des règles d'entrée utilisées lors des pas de dérivation antérieurs. Une dérivation gauche finie se termine soit par une requête de la forme $\langle \Box \mid d \rangle$ avec $\mathcal{T}_\mathcal{C} \models \exists d$ (dans ce cas, c'est une dérivation gauche *qui réussit*) ou par une requête de la forme $\langle Q \mid d \rangle$ avec $Q \neq \Box$ ou $\mathcal{T}_\mathcal{C} \models \neg\exists d$ (dans ce cas, c'est une dérivation gauche *qui échoue*). On dit que S_0 *boucle à gauche* par rapport à P s'il existe une dérivation gauche infinie de $P \cup \{S_0\}$.

2.3. *Les dépliages binaires d'un programme CLP(C)*

On dit que $H \leftarrow c \diamond \mathbf{B}$ est une *règle binaire* si $\mathbf{B}$ contient au plus un atome. Un *programme binaire* est un ensemble fini de règles binaires.

Voici à présent les idées principales sous-jascentes au concept de *dépliages binaires* [GAB 94] d'un programme (ces idées sont tirées de [COD 99].) Cette technique transforme un programme P en un ensemble éventuellement infini de règles binaires. De façon intuitive, chaque règle binaire générée $H \leftarrow c \diamond \mathbf{B}$ spécifie que, par rapport au programme original P, un appel à $\langle H \mid d \rangle$ (ou à l'une de ses instances) mène nécessairement à un appel à $\langle \mathbf{B} \mid c \wedge d \rangle$ (ou à l'instance correspondante) si $c \wedge d$ est satisfiable.

De façon plus précise, soit S une requête atomique. Alors, la requête atomique $\langle A \mid d \rangle$ est un *appel* dans une dérivation gauche de $P \cup \{S\}$ si $S \xRightarrow[P]{+} \langle A, Q \mid d \rangle$. On dénote par $calls_P(S)$ l'ensemble des appels qui apparaissent dans les dérivations gauches de $P \cup \{S\}$. La spécialisation aux modèles d'appels de la sémantique indépendante des buts dans le cas de la règle de sélection gauche-droite est donnée par le point fixe d'un opérateur T_P^β sur le domaine des règles binaires considérées modulo le renommage de variables. Dans la définition ci-dessous, id dénote l'ensemble des règles binaires de la forme $p(\tilde{x}) \leftarrow \tilde{x} = \tilde{y} \diamond p(\tilde{y})$ pour tout $p \in \Pi'_L$ et $\overline{\exists}_V c$ dénote la projection d'une contrainte c sur l'ensemble des variables V. De plus, étant donnés les atomes $A := p(\tilde{t})$ et $A' := p(\tilde{t'})$, $A = A'$ représente la contrainte $\tilde{t} = \tilde{t'}$.

$$
\begin{aligned}
T_P^\beta(X) \quad = \quad & \big\{ H \leftarrow c \diamond \mathbf{B} \,\big|\, H \leftarrow c \diamond \mathbf{B} \in P,\, \mathcal{D}_\mathcal{C} \models \exists c,\, \mathbf{B} = \Box \big\} \bigcup \\
& \left\{ H \leftarrow c \diamond \mathbf{B} \;\middle|\; \begin{array}{l} r := H \leftarrow c_0 \diamond B_1, \ldots, B_m \in P,\ i \in [1, m] \\ \langle H_j \leftarrow c_j \diamond \Box \rangle_{j=1}^{i-1} \in X \text{, renommé, var. disj. de } r \\ H_i \leftarrow c_i \diamond \mathbf{B} \in X \cup id \text{, renommé, var. disj. de } r \\ i < m \Rightarrow \mathbf{B} \neq \Box \\ c = \overline{\exists}_{Var(H,\mathbf{B})} \big[c_0 \wedge \bigwedge_{j=1}^{i} (c_j \wedge \{B_j = H_j\}) \big] \\ \mathcal{D}_\mathcal{C} \models \exists c \end{array} \right\}
\end{aligned}
$$

Ses puissances sont définies de façon usuelle. On peut montrer que le plus petit point fixe de cet opérateur monotone existe toujours et on pose

$$bin\_unf(P) := lfp(T_P^\beta).$$

Alors, les appels qui apparaissent dans les dérivations gauches de $P \cup \{S\}$, avec $S := \langle p(\tilde{t}) \mid d \rangle$, peuvent être caractérisés comme suit :

$$calls_P(S) = \left\{ \langle \mathbf{B} \mid \tilde{t} = \tilde{t'} \wedge c \wedge d \rangle \;\middle|\; \begin{array}{l} p(\tilde{t'}) \leftarrow c \diamond \mathbf{B} \in bin\_unf(P) \\ \mathcal{D}_\mathcal{C} \models \exists(\tilde{t} = \tilde{t'} \wedge c \wedge d) \end{array} \right\}$$

C'est cette dernière propriété qui fut l'une des principales motivations initiales de la sémantique abstraite proposée pour optimiser les programmes logiques. De façon similaire, $bin\_unf(P)$ donne une représentation indépendante des buts des modèles de succès de P.

Il est possible d'extraire encore plus d'information des dépliages binaires d'un programme P : la terminaison universelle gauche d'une requête atomique S par rapport P est identique à la terminaison universelle gauche de S par rapport à $bin\_unf(P)$. On peut noter que la règle de sélection n'a pas d'importance lorsqu'on a affaire à un programme binaire et à une requête atomique car toutes les requêtes dérivées n'ont alors, au plus, qu'un atome. Le résultat suivant est au cœur de l'approche de Codish concernant la terminaison [COD 99] :

Théorème 1 (Observation de la terminaison) *Soit P un programme* CLP($\mathcal{C}$) *et S une requête atomique. Alors, S boucle à gauche par rapport à P si et seulement si S boucle par rapport à $bin\_unf(P)$.*

On peut noter que $bin\_unf(P)$ est un ensemble de règles binaires qui peut être infini. C'est pourquoi, dans les algorithmes de la section 5, nous calculons seulement les max premières itérations de T_P^β où max est un paramètre de l'analyse. Comme conséquence immédiate du théorème 1 que nous utilisons fréquemment dans nos preuves, supposons que nous détections que S boucle par rapport à un sous-ensemble de $T_P^\beta \uparrow i$, avec $i \in N$. Alors, S boucle par rapport à $bin\_unf(P)$, donc S boucle à gauche par rapport à P.

Exemple 3 *Considérons ce programme P de CLP($Term$) (voir [LLO 87], p. 56–58) :*

$$\begin{array}{lcl} r_1 := \mathtt{q}(\mathtt{X}_1, \mathtt{X}_2) & \leftarrow & \mathtt{X}_1 = \mathtt{a} \wedge \mathtt{X}_2 = \mathtt{b} \diamond \Box \\ r_2 := \mathtt{p}(\mathtt{X}_1, \mathtt{X}_2) & \leftarrow & \mathtt{X}_1 = \mathtt{X}_2 \diamond \Box \\ r_3 := \mathtt{p}(\mathtt{X}_1, \mathtt{X}_2) & \leftarrow & \mathtt{Y}_1 = \mathtt{Z}_2 \wedge \mathtt{Y}_2 = \mathtt{X}_2 \wedge \mathtt{Z}_1 = \mathtt{X}_1 \diamond \mathtt{p}(\mathtt{Y}_1, \mathtt{Y}_2), \mathtt{q}(\mathtt{Z}_1, \mathtt{Z}_2) \end{array}$$

Soit c_1, c_2 et c_3 les contraintes de r_1, r_2 et r_3, respectivement. Les dépliages binaires de P sont :

$$\begin{array}{lcl} T_P^\beta \uparrow 0 & = & \varnothing \\ T_P^\beta \uparrow 1 & = & \{r_1, r_2, p(x_1, x_2) \leftarrow c_3 \diamond p(y_1, y_2)\} \cup T_P^\beta \uparrow 0 \\ T_P^\beta \uparrow 2 & = & \{p(x_1, x_2) \leftarrow x_1 = a \wedge x_2 = b \diamond \Box, \\ & & \; p(x_1, x_2) \leftarrow x_1 = z_1 \wedge x_2 = z_2 \diamond q(z_1, z_2)\} \cup T_P^\beta \uparrow 1 \\ T_P^\beta \uparrow 3 & = & \{p(x_1, x_2) \leftarrow x_1 = z_1 \wedge x_2 = b \wedge z_2 = a \diamond q(z_1, z_2), \\ & & \; p(x_1, x_2) \leftarrow x_2 = z_2 \diamond q(z_1, z_2)\} \cup T_P^\beta \uparrow 2 \\ T_P^\beta \uparrow 4 & = & \{p(x_1, x_2) \leftarrow x_2 = b \wedge z_2 = a \diamond q(z_1, z_2)\} \cup T_P^\beta \uparrow 3 \\ T_P^\beta \uparrow 5 & = & T_P^\beta \uparrow 4 = bin\_unf(P) \end{array}$$

2.4. *Terminologie*

Dans cet article, nous présentons un algorithme qui infère un ensemble fini de requêtes atomiques qui bouclent à gauche à partir du code de n'importe quel programme P de CLP($\mathcal{C}$). Dans un premier temps, notre algorithme calcule un sous-ensemble fini de $bin\_unf(P)$, puis il travaille sur ce sous-ensemble uniquement. Pour cette raison, et pour simplifier notre exposé, nous décrivons ci-après des résultats théoriques qui concernent seulement des requêtes atomiques et des règles binaires. Néanmoins, ces résultats peuvent être facilement étendus à toute forme de requête ou de règle. Par conséquent, dans la suite de ce papier jusqu'à la section 5, par *requête* nous entendons *requête atomique*, par *règle* nous entendons *règle binaire* et par *programme* nous entendons *programme binaire*. De plus, comme mentionné ci-dessus, la règle de sélection est sans importance lorsqu'on a affaire à un programme binaire et à une requête atomique, ce qui nous conduit à simplement écrire *pas de dérivation*, *dérivation* et *boucle* sans mentionner de règle de sélection particulière.

3. Inférence de boucles en présence de contraintes

En programmation logique, le test de subsumption fournit un moyen simple pour inférer des requêtes qui bouclent : si, dans un programme logique P, il existe une règle $p(\tilde{t}) \leftarrow p(\tilde{t'})$ telle que $p(\tilde{t'})$ est plus générale que $p(\tilde{t})$, alors la requête $p(\tilde{t})$ boucle par rapport à P. Dans cette section, nous étendons ce résultat aux programmes logiques avec contraintes. Dans un premier temps, nous généralisons la relation "est plus général que" :

Définition 1 (Plus général que) *Soit* $S := \langle p(\tilde{t}) \mid d \rangle$ *et* $S' := \langle p(\tilde{t'}) \mid d' \rangle$ *deux requêtes.* S' *est* plus générale que S *si* $\{p(\tilde{t})\eta \mid \mathcal{D}_\mathcal{C} \models_\eta d\} \subseteq \{p(\tilde{t'})\eta \mid \mathcal{D}_\mathcal{C} \models_\eta d'\}$.

Exemple 4 *Supposons que* $\mathcal{C} = Term$. *Soit* $S := \langle \mathtt{p(X)} \mid \mathtt{X = f(f(Y))} \rangle$ *et* $S' := \langle \mathtt{p(X)} \mid \mathtt{X = f(Y)} \rangle$. *Alors, comme*

$$\big\{p(X)\eta \mid \mathcal{D}_\mathcal{C} \models_\eta (X = f(f(Y)))\big\} \subseteq \big\{p(X)\eta \mid \mathcal{D}_\mathcal{C} \models_\eta (X = f(Y))\big\}$$

S' *est plus générale que* S.

Théorème 2 (Lifting) *Considérons un pas de dérivation* $S \underset{r}{\Longrightarrow} S_1$, *une requête* S' *plus générale que* S *et une variante* r' *de* r *variable disjointe de* S'. *Alors, il existe une requête* S'_1 *qui est plus générale que* S_1 *et telle que* $S' \underset{r}{\Longrightarrow} S'_1$ *avec* r' *comme règle d'entrée.*

De ce théorème, nous déduisons deux corollaires qui peuvent être utilisés pour inférer des requêtes qui bouclent juste à partir du code d'un programme CLP($\mathcal{C}$) :

Corollaire 1 *Soit $r := p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ une règle telle que $\mathcal{D}_\mathcal{C} \models \exists c$. Si $\langle p(\tilde{y}) \mid c \rangle$ est plus générale que $\langle p(\tilde{x}) \mid c \rangle$ alors $\langle p(\tilde{x}) \mid c \rangle$ boucle par rapport à $\{r\}$.*

Corollaire 2 *Soit $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ une règle d'un programme P. Si $\langle q(\tilde{y}) \mid c \rangle$ boucle par rapport à P alors $\langle p(\tilde{x}) \mid c \rangle$ boucle par rapport à P.*

Exemple 5 *Considérons le programme `APPEND` de CLP($Term$) :*

$$\begin{array}{lcll} r_1 & := & \mathtt{append}(\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}) & \leftarrow \quad \mathtt{X_1} = [\,] \wedge \mathtt{X_2} = \mathtt{X_3} \diamond \Box \\ r_2 & := & \mathtt{append}(\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}) & \leftarrow \quad \mathtt{X_1} = [\mathtt{A}|\mathtt{Y_1}] \wedge \mathtt{X_2} = \mathtt{Y_2} \wedge \mathtt{X_3} = [\mathtt{A}|\mathtt{Y_3}] \diamond \\ & & & \qquad\quad \mathtt{append}(\mathtt{Y_1}, \mathtt{Y_2}, \mathtt{Y_3}) \end{array}$$

En notant c_2 la contrainte de la règle r_2, on a $\mathcal{D}_{Term} \models \exists c_2$. De plus, la requête $\langle \mathtt{append}(\mathtt{Y_1}, \mathtt{Y_2}, \mathtt{Y_3}) \mid \mathtt{c_2} \rangle$ est plus générale que la requête $\langle \mathtt{append}(\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}) \mid \mathtt{c_2} \rangle$. Donc, d'après le corollaire 1, $\langle \mathtt{append}(\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}) \mid \mathtt{c_2} \rangle$ boucle par rapport à $\{r_2\}$, donc par rapport à `APPEND`*. Par conséquent, il existe une dérivation infinie ξ de $\mathtt{APPEND} \cup \{\langle \mathtt{append}(\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}) \mid \mathtt{c_2} \rangle\}$. Alors, si S est une requête qui est plus générale que $\langle \mathtt{append}(\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}) \mid \mathtt{c_2} \rangle$, en appliquant successivement le théorème 2 à chaque pas de ξ, on peut construire une dérivation infinie de $\mathtt{APPEND} \cup \{S\}$. Donc, S boucle aussi par rapport à* `APPEND`*.*

Une version étendue du corollaire 1 présentée dans la section suivante ainsi que le corollaire 2 seront utilisés pour construire les algorithmes de la section 5 qui infèrent de manière incrémentale, à partir du code d'un programme, des classes de requêtes qui bouclent.

4. Inférence de boucles et ensemble de positions

Une idée essentielle dans notre travail consiste à identifier des arguments qui peuvent être ignorés lors d'un dépliage. De tels arguments sont appelés *neutres*. Il s'avère que dans de nombreux cas, cela permet d'augmenter significativement le nombre de requêtes bouclantes.

Exemple 6 (Suite de l'exemple 5) *Le deuxième argument du symbole de prédicat $append$ est neutre pour la dérivation relativement à la règle r_2. En effet, si nous tenons une dérivation ξ d'une requête $\langle append(t_1, t_2, t_3) \mid c \rangle$ n'utilisant que $\{r_2\}$, alors pour tout terme t, il existe une dérivation de $\{r_2\} \cup \{\langle append(t_1, t, t_3) \mid c \rangle\}$ dont la longueur est identique à celle de ξ. Ceci signifie que pour chaque requête bouclante de l'exemple 5, le second argument de $append$ peut être remplacé par n'importe quel terme sans que la requête perde son caractère bouclant.*

Dans cette section, nous présentons un cadre pour décrire des arguments spécifiques à l'intérieur d'un programme. Nous donnons ensuite une définition opérationnelle des

arguments neutres permettant d'étendre le corollaire 1. Enfin, nous présentons une caractérisation logique et un critère syntaxique non-équivalent pour les arguments neutres. Notons que les résultats ci-dessous étendent ceux présentés dans [MES 02] où nous définissions, dans le cadre de la programmation logique, les arguments neutres d'une manière purement syntaxique.

4.1. *Ensemble de positions*

Définition 2 (Ensemble de positions) *Un* ensemble de positions, *dénoté τ, est une application qui associe à chaque symbole de prédicat $p \in \Pi'_L$ un sous-ensemble* $[1, arity(p)]$.

Exemple 7 *Si nous souhaitons ignorer le deuxième argument du symbole de prédicat $append$ défini à l'exemple 5, nous posons $\tau := \langle append \mapsto \{2\}\rangle$.*

Via l'emploi d'un ensemble de positions τ, nous pouvons restreindre tout atome en effaçant les arguments dont la position est distinguée par τ :

Définition 3 (Restriction) *Soit τ un ensemble de positions.*

– Soit $p \in \Pi'_L$ un symbole de prédicat d'arité n. La restriction de p relativement à *τ est le symbole de prédicat p_τ dont l'arité est le cardinal de $[1, n] \setminus \tau(p)$.*

– Soit $A := p(t_1, \ldots, t_n)$ un atome. La restriction de A relativement à *τ, dénotée A_τ, est l'atome $p_\tau(t_{i_1}, \ldots, t_{i_m})$ où $\{i_1, \ldots, i_m\} = [1, n] \setminus \tau(p)$ et $i_1 \leq \cdots \leq i_m$.*

– Soit $S := \langle A \mid d\rangle$ une requête. La restriction de S relativement à *τ, dénotée S_τ, est la requête $\langle A_\tau \mid d\rangle$.*

Exemple 8 (Suite de l'exemple 7) *La restriction de la requête*

$$\langle \mathtt{append}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) \mid \mathtt{X} = [\mathtt{A}|\mathtt{B}] \wedge \mathtt{Y} = \mathtt{a} \wedge \mathtt{Z} = [\mathtt{A}|\mathtt{C}]\rangle$$

relativement à τ est $\langle \mathtt{append}_\tau(\mathtt{X}, \mathtt{Z}) \mid \mathtt{X} = [\mathtt{A}|\mathtt{B}] \wedge \mathtt{Y} = \mathtt{a} \wedge \mathtt{Z} = [\mathtt{A}|\mathtt{C}]\rangle$.

Les ensembles de positions, et ce concept de restriction conduisent à une généralisation de la relation "plus général" :

Définition 4 (τ-plus général) *Soient τ un ensemble de positions, S et S' deux requêtes. S' est* τ-plus générale que *S si S'_τ est plus générale que S_τ.*

Exemple 9 (Suite de l'exemple 7) *La requête $\langle \mathtt{append}(\mathtt{X}, \mathtt{a}, \mathtt{Z}) \mid \mathtt{true}\rangle$, comme $\tau = \langle append \mapsto \{2\}\rangle$, est τ-plus générale que $\langle \mathtt{append}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) \mid \mathtt{true}\rangle$ car :*

$$\{append_\tau(X, Z)\eta \mid \mathcal{D}_\mathcal{C} \models_\eta true\} \subseteq \{append_\tau(X, Z)\eta \mid \mathcal{D}_\mathcal{C} \models_\eta true\}$$

4.2. *Ensembles de positions neutres pour la dérivation*

Nous donnons à présent une définition opérationnelle précise du type d'arguments auquel nous nous intéressons. Le terme "neutre pour la dérivation" vient du fait que les τ-arguments ne jouent aucun rôle lors d'un dépliage.

Définition 5 (Neutre pour la dérivation) *Soient r une règle et τ un ensemble de positions. τ est* DN *pour r si pour tout pas de dérivation $S \underset{r}{\Longrightarrow} S_1$, pour toute requête S' τ-plus générale que S et pour toute variante r' de r sans variable commune avec S', il existe une requête S'_1 qui est τ-plus générale que S_1 et telle que $S' \underset{r}{\Longrightarrow} S'_1$ avec r' comme règle d'entrée. Cette définition est étendue aux programmes : τ est* DN *pour P si il est DN pour chaque règle de P.*

Par conséquent, pour tout lifting d'une dérivation, nous pouvons ignorer les arguments DN qui peuvent être instanciés à n'importe quel terme. Nous obtenons ainsi une version étendue du corollaire 1 (prendre $\tau := \langle p \mapsto \varnothing \rangle$ pour tout p) :

Proposition 1 *Soient $r := p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ une règle telle que $\mathcal{D}_\mathcal{C} \models \exists c$ et τ un ensemble de positions DN pour r. Si $\langle p(\tilde{y}) \mid c \rangle$ est τ-plus générale que $\langle p(\tilde{x}) \mid c \rangle$ alors $\langle p(\tilde{x}) \mid c \rangle$ boucle relativement à $\{r\}$.*

Déterminer les arguments neutres à partir du texte d'un programme n'est pas chose aisée si nous tentons d'appliquer la définition ci-dessus. Les sous-sections qui suivent proposent une caractérisation logique et syntaxique qui ont toutes deux été implantées pour calculer des arguments neutres.

4.3. *Une caractérisation logique*

Au sein d'une règle, nous distinguons les ensembles suivants de variables :

Définition 6 *Soient $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ une règle et τ un ensemble de positions.*

1) Soit $\tilde{x} := x_1, \ldots, x_h$. L'ensemble des variables de la tête de r distinguées par τ est $vars\_head(r, \tau) := \{x_i \in \tilde{x} \mid i \in \tau(p)\}$.

2) L'ensemble des variables locales de r, noté $local\_vars(r)$, est défini comme suit : $local\_vars(r) := Var(c) \setminus (\tilde{x} \cup \tilde{y})$.

3) Soit $\tilde{y} := y_1, \ldots, y_b$. L'ensemble des variables du corps de r distinguées par τ est $vars\_body(r, \tau) := \{y_i \in \tilde{y} \mid i \in \tau(q)\}$.

Exemple 10 (Suite de l'exemple 7) *Considérons la règle*

$$r := \mathtt{append}(\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}) \quad \leftarrow \quad \mathtt{X_1} = [\mathtt{A}|\mathtt{Y_1}] \wedge \mathtt{X_2} = \mathtt{B} \wedge \mathtt{B} = \mathtt{Y_2} \wedge \mathtt{X_3} = [\mathtt{A}|\mathtt{Y_3}] \diamond \mathtt{append}(\mathtt{Y_1}, \mathtt{Y_2}, \mathtt{Y_3}).$$

Nous avons : $vars\_head(r,\tau) = \{X_2\}$, $local\_vars(r) = \{A, B\}$ *et* $vars\_body(r,\tau) = \{Y_2\}$.

Nous donnons à présent une définition logique des arguments neutres qui s'avère équivalente à la définition opérationnelle définie ci-dessus.

Définition 7 (DNlog) *Soient* $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ *une règle et* τ *un ensemble de positions.* τ *est* DNlog *pour* r *si* $\mathcal{D}_\mathcal{C} \models (c \rightarrow \forall_\mathcal{X} \exists_\mathcal{Y} c)$ *où* $\mathcal{X} = vars\_head(r,\tau)$ *et* $\mathcal{Y} = local\_vars(r) \cup vars\_body(r,\tau)$. *Cette définition est étendue aux programmes :* τ *est* DNlog *pour* P *si il est DNlog pour chaque règle de* P.

Ainsi τ est DNlog pour r si pour toute valuation σ telle que $\mathcal{D}_\mathcal{C} \models_\sigma c$, quand on change la valeur de $x\sigma$ où $x \in vars\_head(r,\tau)$ en *n'importe quelle* valeur, alors il existe une valeur correspondante pour chaque $y\sigma$ où y appartient soit à $local\_vars(r)$ soit à $vars\_body(r,\tau)$, tel que c reste valide.

Exemple 11 (Suite de l'exemple 10) *L'ensemble de positions* τ *est DNlog pour la règle* r *car* $\mathcal{X} = \{X_2\}$, $\mathcal{Y} = \{A, B, Y_2\}$, c *est la contrainte*

$$(X_1 = [A|Y_1]) \wedge (X_2 = B) \wedge (B = Y_2) \wedge (X_3 = [A|Y_3])$$

et pour toute valuation σ, *si* $\mathcal{D}_\mathcal{C} \models_\sigma c$ *alors* $\mathcal{D}_\mathcal{C} \models_\sigma \forall X_2\ \exists B\ \exists Y_2\ c$ *donc* $\mathcal{D}_\mathcal{C} \models_\sigma \forall_\mathcal{X} \exists_\mathcal{Y} c$.

Théorème 3 *Soient* r *une règle et* τ *un ensemble de positions.* τ *est DNlog pour* r *si et seulement si* τ *est DN pour* r.

4.4. *Une caractérisation syntaxique*

Dans [MES 02], nous donnions une définition syntaxique des arguments pour la programmation logique. Nous étendons dans cette section ce critère au cadre plus générale de la programmation logique avec contraintes. Précisons d'abord ce que nous entendons par règle plate.

Définition 8 (Règle plate) *Une règle* $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ *est qualifiée de* plate *si* c *est de la forme* $(\tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t})$ *pour des séquences de termes* $\tilde{s}$ *et* $\tilde{t}$ *telles que* $Var(\tilde{s}, \tilde{t}) \subseteq local\_vars(r)$.

Notons qu'il existe des règles $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ pour lesquelles il n'existe pas de règle "équivalente" plate. Plus exactement, il n'existe pas de règle $r' := p(\tilde{x}) \leftarrow c' \diamond q(\tilde{y})$ telle que $\mathcal{D}_\mathcal{C} \models \left[(\exists_{local\_vars(r)} c) \leftrightarrow (\exists_{local\_vars(r')} c')\right]$ (considérer par exemple $r := \mathtt{p(X)} \leftarrow \mathtt{X > 0} \diamond \mathtt{p(Y)}$ in $\mathcal{R}_{lin}$).

Introduisons à présent les termes universels :

Définition 9 (Terme universel) *Un terme t in $\Sigma_{\mathcal{C}}$ est dit* universel *si pour une variable x n'apparaissant pas dans t, nous avons : $\mathcal{D}_{\mathcal{C}} \models \forall_x \exists_{Var(t)}(x = t)$.*

Donc un terme t est universel s'il peut prendre n'importe quelle valeur de $D_{\mathcal{C}}$.

Exemple 12 *Un terme t de Σ_{Term} est universel si et seulement si t est une variable. Si x est une variable, alors x, $x + 0$, $x + 1 + (-1)$, ... ainsi que $x + 1$ et $2 * x$ sont des termes universels de $\Sigma_{\mathcal{R}_{lin}}$.*

Nous pouvons maintenant définir :

Définition 10 (Syntaxiquement neutre pour la dérivation) *Considérons une règle plate $r := p(\tilde{x}) \leftarrow (\tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t}) \diamond q(\tilde{y})$ avec $\tilde{s} := s_1, \ldots, s_h$ et $\tilde{t} := t_1, \ldots, t_b$. Soit τ un ensemble de positions. τ est* DNsyn *pour r si :*

$$\forall i \in \tau(p), \begin{cases} (\mathbf{C1}) & s_i \text{ est un terme universel} \\ (\mathbf{C2}) & \forall j \in [1,h] \setminus \{i\},\ Var(s_i) \cap Var(s_j) = \varnothing \\ (\mathbf{C3}) & \forall j \in [1,b] \setminus \tau(q),\ Var(s_i) \cap Var(t_j) = \varnothing \end{cases}$$

Exemple 13 *La règle $\mathtt{p}(\mathtt{X_1}) \leftarrow \mathtt{X_1} = \mathtt{Z} \wedge \mathtt{Y_1} = \mathtt{Z} \diamond \mathtt{p}(\mathtt{Y_1})$ est plate et l'ensemble de positions $\langle p \mapsto \{1\}\rangle$ est DNsyn pour cette règle.*

Proposition 2 *Soient r une règle plate et τ un ensemble de positions. Si τ est DNsyn pour r alors τ est DN pour r. Si τ est DN pour r alors la condition* **(C1)** *de la définition 10 est vérifiée.*

L'exemple suivant montre qu'un ensemble de positions DN n'est pas nécessairement DNsyn car les conditions **(C2)** ou **(C3)** de la définition 10 peuvent ne pas être vérifiées.

Exemple 14 *Plaçons nous dans $\mathcal{R}_{lin}$.*

– Soit $r_1 := \mathtt{p_1}(\mathtt{X_1}, \mathtt{X_2}) \leftarrow \mathtt{X_1} = \mathtt{A} \wedge \mathtt{X_2} = \mathtt{A} + \mathtt{B} \diamond \Box$. L'ensemble de positions $\tau_1 := \langle p_1 \mapsto \{1,2\}\rangle$ est DNlog pour r_1, donc τ_1 est DN pour r_1. Mais τ_1 n'est pas DNsyn pour r_1 : comme les termes A et $A + B$ partagent la variable A, ***(C2)*** *n'est pas vérifiée.*

– Soit $r_2 := \mathtt{p_2}(\mathtt{X_1}, \mathtt{X_2}) \leftarrow \mathtt{X_1} = \mathtt{A} \wedge \mathtt{X_2} = \mathtt{0} \wedge \mathtt{Y_2} = \mathtt{A} - \mathtt{A} \diamond \mathtt{p_2}(\mathtt{Y_1}, \mathtt{Y_2})$. L'ensemble de positions $\tau_2 := \langle p_2 \mapsto \{1\}\rangle$ est DNlog pour r_2, donc τ_2 est DN for r_2. Mais τ_2 n'est pas DNsyn pour r_2 : comme les terms A and $A - A$ partagent la variable A, ***(C3)*** *n'est pas vérifiée.*

Dans le cas particulier de la programmation logique, les deux notions sont équivalentes.

Théorème 4 (Programmation logique) *Posons $\mathcal{C} = Term$. Soient r une règle plate et τ un ensemble de positions. Alors τ est DNsyn pour r si et seulement si τ est DN pour r.*

Toute règle $p(\tilde{s}) \leftarrow q(\tilde{t})$ du domaine $Term$ peut se ré-écrire comme règle plate $p(\tilde{x}) \leftarrow (\tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t}) \diamond q(\tilde{y})$. Comme les seuls termes universels de Σ_{Term} sont les variables, la définition 10 est équivalente à celle que nous donnions dans [MES 02]. En conséquence, le théorème 4 établit que pour le cas de la programmation logique, nous ne pouvons pas obtenir de meilleur critère syntaxique que celui présenté dans [MES 02] (par "meilleur", nous entendons un critère permettant de distinguer au moins les mêmes positions).

5. Algorithmes

Dans cette section, nous décrivons un ensemble d'algorithmes corrects pour inférer des classes de requêtes atomiques bouclantes à partir du texte d'un programme (non nécessairement binaire) P. Via l'emploi de l'opérateur T_P^β, notre technique calcule d'abord un sous-ensemble fini de $bin\_unf(P)$ qui est ensuite analysé. Nous utilisons également une structure de données appelée *dictionnaire de boucles*.

5.1. *Les dictionnaires de boucles*

Définition 11 (Paire bouclante, dictionnaire de boucles) *Une* paire bouclante *est un couple $(BinSeq, \tau)$ où $BinSeq$ est séquence ordonnée finie de règles binaires et τ un ensemble de positions DN pour $BinSeq$ vérifiant :*

– ou bien $BinSeq = [p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})]$ où $\mathcal{D}_\mathcal{C} \models \exists c$ et $\langle p(\tilde{y}) \mid c \rangle$ est τ-plus générale than $\langle p(\tilde{x}) \mid c \rangle$

– ou bien $BinSeq = [p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1) | BinSeq']$ et il existe un ensemble de positions τ' tel que $([p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1) | BinSeq'], \tau')$ est une paire bouclante avec $\langle q(\tilde{y}) \mid c \rangle$ τ'-plus général que $\langle p_1(\tilde{x}_1) \mid c_1 \rangle$.

Un dictionnaire de boucles *est un ensemble fini de paires bouclantes.*

Exemple 15 *Pour le domaine $\mathcal{R}_{lin}$, la paire $(BinSeq, \tau)$ où $BinSeq := [\mathtt{p(X)} \leftarrow \mathtt{X} > 0 \wedge \mathtt{X} = \mathtt{Y} \diamond \mathtt{q(Y)}, \mathtt{q(X)} \leftarrow \mathtt{Y} = 2 * \mathtt{X} \diamond \mathtt{q(Y)}]$ et $\tau := \langle p \mapsto \varnothing, q \mapsto \{1\} \rangle$, est une paire bouclante car :*

– comme τ est DNlog pour $BinSeq$, d'après le théorème 3, il est DN pour $BinSeq$,

– $([\mathtt{q(X)} \leftarrow \mathtt{Y} = 2 * \mathtt{X} \diamond \mathtt{q(Y)}], \tau')$, où $\tau' := \langle q \mapsto \{1\} \rangle$, est une paire bouclante puisque τ' est DN pour $[\mathtt{q(X)} \leftarrow \mathtt{Y} = 2 * \mathtt{X} \diamond \mathtt{q(Y)}]$ (il est DNlog pour ce programme), $\mathcal{D}_{\mathcal{R}_{lin}} \models \exists (Y = 2 * X)$ et $\langle \mathtt{q(Y)} \mid \mathtt{Y} = 2 * \mathtt{X} \rangle$ est τ'-plus générale que $\langle \mathtt{q(X)} \mid \mathtt{Y} = 2 * \mathtt{X} \rangle$,

– $\langle \mathtt{q(Y)} \mid \mathtt{X} > 0 \wedge \mathtt{X} = \mathtt{Y} \rangle$ est τ'-plus générale que $\langle \mathtt{q(X)} \mid \mathtt{Y} = 2 * \mathtt{X} \rangle$.

Une première motivation concernant cette définition vient de ce qu'une paire bouclante génère une requête atomique bouclante :

Proposition 3 *Soit* $([p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}) | BinSeq], \tau)$ *une paire bouclante. Alors* $\langle p(\tilde{x}) \mid c \rangle$ *boucle relativement à* $[p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}) | BinSeq]$.

Notre deuxième motivation concerne la construction incrémentale des dictionnaires de boucles.

5.2. *Calculer un dictionnaire de boucles à partir d'un programme binaire*

La forme la plus simple d'un dictionnaire de boucles est $([p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})], \tau)$ où τ est DN pour $[p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})]$, avec $\mathcal{D}_\mathcal{C} \models \exists c$ et $\langle p(\tilde{y}) \mid c \rangle$ τ-plus générale que $\langle p(\tilde{x}) \mid c \rangle$. Ainsi, étant donnée une règle binaire $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ où $\mathcal{D}_\mathcal{C} \models \exists c$, si τ est DN pour $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$, il suffit de tester si $\langle p(\tilde{y}) \mid c \rangle$ est τ-plus générale que $\langle p(\tilde{x}) \mid c \rangle$. Dans l'affirmative, $([p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})], \tau)$ est une paire bouclante. C'est le principe de la fonction qui suit.

`unit_loop`($p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$, $Dict$) :

in : $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$: une règle binaire
$Dict$: un dictionnaire de boucles
out : $Dict'$: un dictionnaire de boucles

1 : $Dict' := Dict$
2 : **if** $\mathcal{D}_\mathcal{C} \models \exists c$ **then**
3 : $\quad \tau :=$ un ensemble de positions DN pour $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$
4 : $\quad$ **if** $\langle p(\tilde{y}) \mid c \rangle$ est τ-plus générale que $\langle p(\tilde{x}) \mid c \rangle$ **then**
5 : $\quad\quad Dict' := Dict' \cup \{([p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})], \tau)\}$
6 : **return** $Dict'$

La terminaison de `unit_loop` est immédiate. Sa correction partielle se déduit du résultat suivant :

Théorème 5 (Correction partielle de `unit_loop`) *Si* $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ *est une règle binaire et* $Dict$ *un dictionnaire de boucles, alors* `unit_loop`$(p(\tilde{x}) \leftarrow c \diamond p(\tilde{y}), Dict)$ *est un dictionnaire de boucles dont chaque élément* $(BinSeq, \tau)$ *appartient à* $Dict$ *ou bien vérifie* $BinSeq = [p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})]$.

Considérons maintenant un dictionnaire de boucles $Dict$ et une règle $p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$. Nous pouvons tenter d'obtenir des paires bouclantes supplémentaires comme suit. Nous prenons des éléments $\left([p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1) | BinSeq'], \tau'\right)$ de $Dict$ tels que $\langle q(\tilde{y}) \mid c \rangle$ soit τ'-plus générale que $\langle p_1(\tilde{x}_1) \mid c_1 \rangle$ et nous calculons τ DN pour $[p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1) | BinSeq']$. Alors $([p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1) | BinSeq'], \tau)$ est une paire bouclante. D'où la fonction :

`loops_from_dict`($p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$, $Dict$) :

in : $p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$: une règle binaire
$Dict$: un dictionnaire de boucles
out : $Dict'$: un dictionnaire de boucles

1 : $Dict' := Dict$
2 : **for each** $\left([p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1)|BinSeq'], \tau'\right) \in Dict$ **do**
3 : **if** $\langle q(\tilde{y}) \,|\, c\rangle$ est τ'-plus générale que $\langle p_1(\tilde{x}_1) \,|\, c_1\rangle$ **then**
4 : $BinSeq := [p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1)|BinSeq']$
5 : $\tau :=$ un ensemble de positions DN pour $BinSeq$
6 : $Dict' := Dict' \cup \{(BinSeq, \tau)\}$
7 : **return** $Dict'$

La terminaison de `loops_from_dict` est conséquence du caractère fini de $Dict$ (car $Dict$ est un dictionnaire de boucles), sa correction partielle de :

Théorème 6 (Correction partielle de `loops_from_dict`) *Soient $p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ une règle binaire et $Dict$ un dictionnaire de boucles. Alors* `loops_from_dict`$(p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), Dict)$ *est un dictionnaire de boucles dont chaque élément $(BinSeq, \tau)$ appartient à $Dict$ ou bien vérifie $BinSeq = [p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})|BinSeq']$ pour une paire $(BinSeq', \tau')$ de $Dict$.*

Enfin, voici la fonction principale pour calculer un dictionnaire de boucles à partir d'un ensemble fini de règles binaires :

`infer_loop_dict`($BinProg$) :

in : $BinProg$: un ensemble fini de règles binaires
out : un dictionnaire de boucles

1 : $Dict := \varnothing$
2 : **for each** $p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}) \in BinProg$ **do**
3 : **if** $q = p$ **then**
4 : $Dict :=$ `unit_loop`$(p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), Dict)$
5 : $Dict :=$ `loops_from_dict`$(p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), Dict)$
6 : **return** $Dict$

Nous pouvons énoncer :

Théorème 7 (Correction de `infer_loop_dict`) *Soit $BinProg$ un ensemble fini de règles binaires. Alors* `infer_loop_dict`$(BinProg)$ *termine et renvoie un dictionnaire de boucles dont chaque élément $(BinSeq, \tau)$ vérifie $BinSeq \subseteq BinProg$.*

5.3. *Inférer des conditions de bouclage*

Nous sommes à présent en mesure de présenter un algorithme qui infère à partir du texte d'un programme des classes de requêtes atomiques qui bouclent à gauche. Une classe est définie par une paire (S, τ) qui résume l'ensemble infini $[S]^\tau$:

Définition 12 *Soient S une requête atomique et τ un ensemble de positions. $[S]^\tau$ denote la classe des requêtes atomiques :*

$$[S]^\tau \stackrel{def}{=} \{S' : \textit{ une requête atomique} \mid S' \textit{ est } \tau\textit{-plus générale que } S\} .$$

Lorsque chaque élément de $[S]^\tau$ boucle à gauche relativement à un programme PLC($\mathcal{C}$), nous obtenons une *condition de bouclage* pour ce programme :

Définition 13 (Condition de bouclage) *Soit P un programme de PLC($\mathcal{C}$). Une* condition de bouclage pour P *est une paire (S, τ) telle que chaque élément de $[S]^\tau$ boucle à gauche relativement à P.*

Inférer des conditions de bouclage à partir d'un dictionnaire de boucles est chose aisée : il suffit de considérer la propriété des paires bouclantes établie pour la proposition 3. La fonction qui suit calcule un ensemble fini de conditions de bouclage pour tout programme PLC($\mathcal{C}$).

`infer_loop_cond`(P, max) :

in : P : un programme PLC($\mathcal{C}$)
max : un entier naturel
out : un ensemble fini de condition de bouclage for P

1 : $L := \varnothing$
2 : $Dict :=$ `infer_loop_dict`$(T_P^\beta \uparrow max)$
3 : **for each** $([p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}) | BinSeq], \tau) \in Dict$ **do**
4 : $\quad L := L \cup \{(\langle p(\tilde{x}) \mid c \rangle, \tau)\}$
5 : **return** L

Pour tout programme P et tout entier max, l'évaluation de `infer_loop_cond`(P, max) termine. En effet, $T_P^\beta \uparrow max$ est fini et à la ligne 2, `infer_loop_dict`$(T_P^\beta \uparrow max)$ termine et la boucle de la ligne 3 effectue un nombre fini d'itérations La correction partielle de `infer_loop_cond` est une conséquence du résultat qui suit.

Théorème 8 (Correction partielle de `infer_loop_cond`) *Si P est un programme et max un entier naturel alors* `infer_loop_cond`(P, max) *est un ensemble fini de conditions de bouclage pour P.*

Remarquons que la correction de `infer_loop_cond` est indépendante de l'ordre d'examen des prédicats. Cependant, l'inférence est bien plus efficace lorsque les prédicats sont traités de manière ascendante sur un tri topologique des composantes fortement connexes du graphe d'appel du programme analysé.

Exemple 16 *Considerons le programme PLC($\mathcal{R}_{lin}$)* `SUM` *:*

$$\mathtt{sum}(\mathtt{X}_1, \mathtt{X}_2) \leftarrow \mathtt{X}_1 > 0 \wedge \mathtt{Y}_1 = \mathtt{X}_1 \wedge \mathtt{Y}_2 = 1 \wedge \mathtt{Z}_1 = \mathtt{X}_1 - 1 \wedge \mathtt{X}_2 = \mathtt{Y}_3 + \mathtt{Z}_2 \diamond \mathtt{pow2}(\mathtt{Y}_1, \mathtt{Y}_2, \mathtt{Y}_3), \mathtt{sum}(\mathtt{Z}_1, \mathtt{Z}_2)$$

$$\mathtt{pow2}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3) \leftarrow \mathtt{X}_1 \leq 0 \wedge \mathtt{X}_2 = \mathtt{X}_3 \diamond \square$$
$$\mathtt{pow2}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3) \leftarrow \mathtt{X}_1 > 0 \wedge \mathtt{Y}_1 = \mathtt{X}_1 - 1 \wedge \mathtt{Y}_2 = 2 * \mathtt{X}_2 \wedge \mathtt{Y}_3 = \mathtt{X}_3 \diamond \mathtt{pow2}(\mathtt{Y}_1, \mathtt{Y}_2, \mathtt{Y}_3)$$

L'ensemble $T^{\beta}_{\mathtt{SUM}} \uparrow 1$ *comprend :*

$$br_1 := \mathtt{sum}(\mathtt{X}_1, \mathtt{X}_2) \leftarrow \mathtt{X}_1 > 0 \wedge \mathtt{Y}_1 = \mathtt{X}_1 \wedge \mathtt{Y}_2 = 1 \wedge \mathtt{Z}_1 = \mathtt{X}_1 - 1 \wedge \mathtt{X}_2 = \mathtt{Y}_3 + \mathtt{Z}_2 \diamond \mathtt{pow2}(\mathtt{Y}_1, \mathtt{Y}_2, \mathtt{Y}_3)$$

$$br_2 := \mathtt{pow2}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3) \leftarrow \mathtt{X}_1 > 0 \wedge \mathtt{Y}_1 = \mathtt{X}_1 - 1 \wedge \mathtt{Y}_2 = 2 * \mathtt{X}_2 \wedge \mathtt{Y}_3 = \mathtt{X}_3 \diamond \mathtt{pow2}(\mathtt{Y}_1, \mathtt{Y}_2, \mathtt{Y}_3)$$

Un appel à `unit_loop`$(br_2, \varnothing)$ *renvoie* $Dict_1 := \{([br_2], \tau_2)\}$ *où* $\tau_2 = \langle pow2 \mapsto \{2, 3\}\rangle$. *Puis* `loops_from_dict`$(br_1, Dict_1)$ *renvoie* $Dict_1 \cup \{([br_1, br_2], \tau_1)\}$ *où* $\tau_1 = \langle sum \mapsto \{2\}, pow2 \mapsto \{2, 3\}\rangle$. *Enfin un appel à* `infer_loop_cond`$(\mathtt{SUM}, 1)$ *calcule les paires* $(\langle \mathtt{sum}(\mathtt{X}_1, \mathtt{X}_2) \,|\, c_1\rangle, \tau_1)$ *et* $(\langle \mathtt{pow2}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3) \,|\, c_2\rangle, \tau_2)$ *où* c_1 *et* c_2 *sont les contraintes respectives de* br_1 *et* br_2.

6. Conclusion

Nous avons présenté dans cet article un ensemble de résultats et d'algorithmes permettant l'inférence de classes de requêtes non-terminantes pour les programmes logiques avec contraintes. Le passage à la PLC nous a permis de proposer à la fois des définitions plus générales ainsi que des preuves plus simples, phénomène déjà remarqué par les auteurs de [JAF 98]. Enfin, partant d'une définition opérationnelle du concept de neutralité pour la dérivation, nous avons présenté une définition logique équivalente. Du coup, en ré-examinant le critère syntaxique utilisé dans [MES 02], nous avons montré que ce critère syntaxique peut être considéré comme une implantation correcte et complète de la neutralité pour la dérivation.

Parmi les questions ouvertes soulevées par l'analyse de non-terminaison, nous aimerions définir une approche descendante de cette analyse.

7. Bibliographie

[CLA 78] CLARK K. L., « Negation as Failure », GALLAIRE H., MINKER J., Eds., *Logic and Databases*, p. 293-322, Plenum Press, New York, 1978.

[COD 99] CODISH M., TABOCH C., « A semantic basis for the termination analysis of logic programs », *Journal of Logic Programming*, vol. 41, n° 1, 1999, p. 103-123.

[De 89] DE SCHREYE D., BRUYNOOGHE M., VERSCHAETSE K., « On the existence of non-terminating queries for a restricted class of Prolog-clauses », *Artificial Intelligence*, vol. 41, 1989, p. 237–248.

[De 90] DE SCHREYE D., VERSCHAETSE K., BRUYNOOGHE M., « A practical technique for detecting non-terminating queries for a restricted class of Horn clauses, using directed, weighted graphs », *Proc. of ICLP'90*, The MIT Press, 1990, p. 649–663.

[De 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.

[GAB 94] GABBRIELLI M., GIACOBAZZI R., « Goal independency and call patterns in the analysis of logic programs », *Proceedings of the ACM Symposium on applied computing*, ACM Press, 1994, p. 394–399.

[GEN 01] GENAIM S., CODISH M., « Inferring Termination Condition for Logic Programs using Backwards Analysis », *Proceedings of Logic for Programming, Artificial intelligence and Reasoning*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001.

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

[JAF 98] JAFFAR J., MAHER M. J., MARRIOTT K., STUCKEY P. J., « The Semantics of Constraint Logic Programs », *Journal of Logic Programming*, vol. 37, n° 1-3, 1998, p. 1-46.

[LLO 87] LLOYD J. W., *Foundations of Logic Programming*, Springer-Verlag, 1987.

[MES 96] MESNARD F., « Inferring left-terminating classes of queries for constraint logic programs by means of approximations », MAHER M. J., Ed., *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, MIT Press, 1996, p. 7–21.

[MES 01] MESNARD F., NEUMERKEL U., « Applying static analysis techniques for inferring termination conditions of logic programs », COUSOT P., Ed., *Static Analysis Symposium*, vol. 2126 de *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 2001, p. 93–110.

[MES 02] MESNARD F., PAYET E., NEUMERKEL U., « Detecting optimal termination conditions of logic programs », HERMENEGILDO M., PUEBLA G., Eds., *Proc. of the 9th International Symposium on Static Analysis*, vol. 2477 de *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 2002, p. 509–525.

[MES 03] MESNARD F., RUGGIERI S., « On proving left termination of constraint logic programs », *ACM Transactions on Computational Logic*, , 2003, p. 207–259.

[REF 96] REFALO P., HENTENRYCK P. V., « CLP ($\mathcal{R}_{lin}$) Revised », MAHER M., Ed., *Proc. of the Joint International Conf. and Symposium on Logic Programming*, The MIT Press, 1996, p. 22–36.

[SHO 67] SHOENFIELD J., *Mathematical logic*, Addison Wesdley, Reading, 1967.