Inferring and Compiling Termination for Constraint Logic Programs [1]

Sébastien Hoarau, Fred Mesnard

IREMIA, Université de La Réunion
BP 7151 - 97715 Saint-Denis Messag. Cedex 9, FRANCE
E-mail : {Sebastien.Hoarau, fred}@univ-reunion.fr
URL : www.univ-reunion.fr/~gcc

Abstract - This paper presents an automated method that deals with termination of constraint logic programs in two steps. First, the method *infers* a set of potentially terminating classes (using approximation techniques and boolean mu-calculus). By "potentially", we mean that for each of these classes, one can find a static order over the literals of the clauses of the program to ensure termination.
Then, given a terminating class among those computed at the first step, the second step consists of a "compilation" of the original program to another one by *reordering literals*. For this new program, the universal left-termination of any query of the considered class is guaranteed. The method has been implemented.

1 Introduction

Many research works have been devoted to termination analysis of (constraint) logic programs in recent years, as shown by the survey [3]. For most researchers, the main problem is universal left-termination of a *given* class of queries. This is a somehow restricted view of logic programming. In [7], we addressed a broader question: find the classes of queries for which universal left-termination is guaranteed. Here we go one step further: automatically infer (larger) classes of queries such that there exists a static reordering of the bodies of the clauses which insures universal left-termination.

Example 1.1 Let us consider the following CLP($\mathcal{N}$) program:

$$\begin{array}{ll} \text{rule } r_1 : & \mathtt{p(X,Y)\ :-\ s(X,Z), s(Z,Y).} \\ \text{rule } r_2 : & \mathtt{s(0,0).} \\ \text{rule } r_3 : & \mathtt{s(X+1, Y+2X+1)\ :-\ s(X,Y).} \end{array}$$

The method described in [7] can infer that "$\leftarrow$X *bounded*$, p(X,Y)$" is a left-terminating query. Nevertheless, the resolution of the query $\leftarrow Y \leq 16, p(X,Y)$ could terminate, if in the clause defining the predicate p, we prove $\leftarrow Y \leq 16, s(Z,Y)$ before $s(X,Z)$. We would like to conclude that there are two terminating classes of queries: $\leftarrow X$ *bounded*$, p(X,Y)$ and $\leftarrow Y$ *bounded*$, p(X,Y)$. The new method we present in this abstract does this job. $\triangleleft$

[1]Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

The rest of the paper is organized as follows: section 2 recalls some basic notions about CLP(χ) and approximations between two CLP systems. Section 3 summarizes our results for the inference of left-termination conditions. Then, sections 4 and 5 present the two points of the new method: the inference of larger classes of queries and the reordering of the literals inside clauses. Implementation issues are slightly discussed in section 6.

2 Preliminaries

2.1 CLP(χ)

Let us remind some notations and conventions about CLP introduced in [5]. For concision, we simplify the notations when we consider there is no ambiguity. Moreover, in concrete program examples, we use the Edinburgh syntax.
In the following, we consider ideal CLP systems[2] without limit element. $\tilde{t}$ (resp. $\tilde{x}$) represents a sequence of terms (resp. distinct variables). Let o be a CLP(χ) object. $var(o)$ denotes the set of variables of o and $o(\tilde{x})$ means o *where* $var(o) = \tilde{x}$. If o_1 and o_2 are two objects, $o_1 \equiv o_2$ means o_1 and o_2 are *syntactically equal*. A *χ-constraint* is a constraint from the structure χ. The χ-constraint c is *χ-solvable* and θ is a *χ-solution* of c if θ is a χ- *valuation* s.t. $\chi \models c\theta$. Otherwise c is χ- *unsolvable*. Let c_1 and c_2 be two χ-constraints. We write $c_1 \rightarrow_\chi c_2$ (resp. $c_1 \leftrightarrow_\chi c_2$) as a shorthand for $\chi \models \forall[c_1 \rightarrow c_2]$ (resp. $\chi \models \forall[c_1 \leftrightarrow c_2]$).
We use the following structures (where the symbols have their usual interpretation):

- $\mathcal{B} = \langle\{true, false\}; \{0, 1, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}; \{\Rightarrow, =\}\rangle$,
- $\mathcal{N} = \langle\mathbb{N}; \{0, 1, +\}; \{\geq, =\}\rangle$ where $\mathbb{N}$ is the set of natural numbers.

In $\mathcal{N}$, for $n \in \mathbb{N}$, $n \geq 1$, we write n (resp. nx) as an abbreviation for $1 + \ldots + 1$ (resp. $x + \ldots + x$), where 1 (resp. x) appears n times. Let $c(\tilde{x})$ a $\mathcal{N}$-constraint, $y \in \tilde{x}$ and $m \in \mathbb{N}$. We say that y is *bounded* (resp. *bounded by* m) in c if there exists $n \in \mathbb{N}$ such that $c(\tilde{x}) \rightarrow_\mathcal{N} n \geq y$ (resp. $c(\tilde{x}) \rightarrow_\mathcal{N} m \geq y$).

On the set Π of predicate symbols defined by a program P, we define the binary relation $p \rightarrow q$ if P contains a rule of the form $p(\tilde{t}_1) \leftarrow \ldots, q(\tilde{t}_2), \ldots$. Let $\rightarrow^*$ denotes the reflexive transitive closure of $\rightarrow$. The relation $p \sim q$ if $p \rightarrow^* q \wedge q \rightarrow^* p$ is an equivalence relation. We note $\bar{p}$ the equivalence class including p.
A rule $p(\tilde{x}) \leftarrow c, \tilde{B}$ is *recursive* if there is a predicate symbol $q \in \bar{p}$ s.t. $\tilde{B} = \ldots, q(\tilde{y}), \ldots$. A predicate symbol p is *directly recursive* if $\bar{p} = \{p\} \wedge (p \rightarrow p)$. The predicate symbols $\{p_1, \ldots, p_n\}$ $(n \geq 2)$ are *mutually recursive* if $\bar{p_1} = \ldots = \bar{p_n} = \{p_1, \ldots, p_n\}$. A predicate p is *recursive* if it is either directly recursive or mutually recursive. Otherwise p is *non-recursive*.

2.2 From CLP(χ) to CLP($\mathcal{B}ool$)

An *approximation* $\mathcal{A}_\chi^\psi$ from χ to ψ consists in a pair of functions $\langle\mathcal{A}_{sx}, \mathcal{A}_{sm}\rangle$ with some properties (see [7] for details). The main approximation we use is $\mathcal{A}_\chi^\mathcal{B}$ which turns CLP(χ) entities into boolean entities and which is the composition of two approximations:

1. $\mathcal{A}_\chi^\mathcal{N}$: which, informally, replace all data structures by their *sizes* (by example lists by their length, trees by the number of nodes, ...),

[2] i.e. systems which embody a decision procedure for the problem: $\models \exists\, c$ (where c is any constraint).

2. $\mathcal{A}_{\mathcal{N}}^{\mathcal{B}}$: $\mathcal{A}_{sx}(0) = 1$, $\mathcal{A}_{sx}(1) = 1$, $\mathcal{A}_{sx}(+) = \wedge$, $\mathcal{A}_{sx}(\geq) = \Rightarrow$, $\mathcal{A}_{sm}(n) = true$.

Example 2.1 Let $\mathcal{L}(\chi) = \langle D_\chi^*; \{[\,], [\_|\_]\}; \{=\}\rangle$ be the lists structure over χ, where as usual, the constant $[\,]$ denotes the empty list and the operator $[\_|\_]$ denotes the list constructor. We define the approximation $\mathcal{A}_{\mathcal{L}(\chi)}^{\mathcal{N}}$ from $\mathcal{L}(\chi)$ to $\mathcal{N}$: $\mathcal{A}_{sx}([\,]) = 0$, $\mathcal{A}_{sx}([x|y]) = 1 + \mathcal{A}_{sx}(y)$ and $\mathcal{A}_{sm}([e_1, \ldots, e_n]) = n$. Let us approximate the well-known APPEND CLP($\mathcal{L}(\chi)$) program:

CLP($\mathcal{L}(\chi)$) version	CLP($\mathcal{N}$) version	CLP($\mathcal{B}$) version
$\mathtt{app([\,], Ys, Ys).}$ $\mathtt{app([X\|Xs], Ys, [X\|Zs]) :-}$ $\mathtt{app(Xs, Ys, Zs).}$	$\mathtt{app(0, Ys, Ys).}$ $\mathtt{app(1 + Xs, Ys, 1 + Zs) :-}$ $\mathtt{app(Xs, Ys, Zs).}$	$\mathtt{app(1, Ys, Ys).}$ $\mathtt{app(1 \wedge Xs, Ys, 1 \wedge Zs) :-}$ $\mathtt{app(Xs, Ys, Zs).}$

$\triangleleft$

Now, we give some results related to approximations. Let $\mathcal{A}_\chi^\psi$ be an approximation, P be a CLP(χ) program and S_P^χ its semantics [4]. Then, the image of the semantics of P is included in the semantics of the image of P:

Theorem 2.2 *[7]* $\quad \mathcal{A}_\chi^\psi(S_P^\chi) \subseteq S^\psi_{\mathcal{A}_\chi^\psi(P)}$

Here is a very useful property about the boolean approximation $\mathcal{A}_{\mathcal{N}}^{\mathcal{B}}$:

Proposition 2.3 *(see [7] for details and examples) Let c_1 be a $\mathcal{N}$-constraint, c_2 a $\mathcal{B}$-constraint s.t. $\mathcal{A}_{\mathcal{N}}^{\mathcal{B}}(c_1) \rightarrow_{\mathcal{B}} c_2$ and $t \equiv \vee_{j \in J}(\wedge_{i \in I_j} x_i)$ a boolean term. If $c_2 \rightarrow_{\mathcal{B}} t$ then $\exists j \in J, \forall i \in I_j, x_i$ is bounded in c_1.*

Moreover, we can compute the *boolean model* of a CLP(χ) program and:

Remark 2.4 *Let P be a CLP(χ) program. We can always assume that $S^{\mathcal{B}}_{\mathcal{A}_\chi^{\mathcal{B}}(P)}$, the model of $\mathcal{A}_\chi^{\mathcal{B}}(P)$, contains exactly one formula $p(\tilde{x}) \leftrightarrow Post_p(\tilde{x})$ for each predicate symbol p of P.*

We note $Post_p$ the boolean model of a predicate p because of the obvious link with the post condition of De Schreye and Decorte [3].

Example 2.5 If we consider one more time the APPEND program, then we have:

$$Post_{app}(x, y, z) = x \wedge (y \Leftrightarrow z).$$

Informally, it means that when a proof of a goal $\leftarrow app(X, Y, Z)$ terminates in CLP($\mathcal{L}(\chi)$), on one hand the length of X is bounded and, on the other hand, the length of Y is bounded if and only if the length of Z is bounded. $\triangleleft$

Remark 2.6 *From now on, in the following sections, we consider only CLP($\mathcal{N}$) programs and we denote by Π the set of predicate symbols of a program. For another structure, we switch to CLP($\mathcal{N}$) by an appropriated approximation. For any numerical constraint c, we write $c^{\mathcal{B}}$ its boolean version.*

3 Inferring left-termination conditions: the previous approach

In [7], for each predicate symbol p of a program P, a boolean term Pre_p is computed. This boolean term is a condition of left-termination such that for any query $\leftarrow c, p(\tilde{x})$ if $c^{\mathcal{B}}(\tilde{x}) \rightarrow_{\mathcal{B}} Pre_p(\tilde{x})$ then the derivation of the query universally left-terminates. Let us summarize this result.

Definition 3.1 A *linear measure* μ_p for a predicate symbol $p \in \bar{p}$ of arity n is a mapping defined by:

$$\begin{array}{cccc} \mu_p : & \mathbb{N}^n & \rightarrow & \mathbb{N} \\ & (x_1, \ldots, x_n) & \mapsto & \sum_{i=1}^{n} a_i x_i \end{array}$$

s.t. a_i's are integers (we note I_{μ_p} the non empty set s.t. $i \in I_{\mu_p}$ implies $a_i \neq 0$) and for each rule $p(\tilde{x}) \leftarrow c, \tilde{B}$ defining p, for each solution θ of c, for each atom $q(\tilde{y})$ appearing in $\tilde{B}$ with $q \in \bar{p}$ we have: $\mu_p(\tilde{x}\theta) \geq 1 + \mu_q(\tilde{y}\theta)$ and $\mu_q(\tilde{y}\theta) \geq 0$.

Example 3.2 If we consider the CLP($\mathcal{N}$) version of APPEND program then, $\mu^1(x, y, z) = x$ and $\mu^2(x, y, z) = z$ are two linear measures. $\triangleleft$

The work of K. Sohn and A. Van Gelder [8] shows that there exists a complete procedure for deciding the existence of a linear measure. This procedure can be adapted to compute the coefficients of μ.

Definition 3.3 Let P be a CLP($\mathcal{N}$) program. Let $p \in \bar{p}$ be a recursive predicate. Its set of *maximal* measures is always finite (see [7]). We write q_p the number of maximal measures for a predicate p and $\Gamma_{\bar{p}} = \{\mu_p^j \mid p \in \bar{p}, 1 \leq j \leq q_p\}$ the set of associated maximal linear measures for $\bar{p}$. For each $p \in \bar{p}$, we compute a boolean term called its *boolean measure* defined by:

$$\gamma_p(\tilde{x}) = \bigvee_{1 \leq j \leq q_p} \left(\bigwedge_{i \in I_{\mu_p^j}} x_i \right)$$

If p is a non-recursive predicate, we set $\gamma_p(\tilde{x}) = 1$.

Example 3.4 If we continue our example 3.2, we have $\bar{p} = \{app\}$, $q_{app} = 2$ and $\Gamma_{\overline{app}} = \{\mu^1, \mu^2\}$ and the boolean measure is $\gamma_{app}(x, y, z) = x \vee z$. $\triangleleft$

Definition 3.5 A boolean term $Pre_p(\tilde{x})$ is a *left-termination condition* for $p(\tilde{x})$ if, for every constraint c, $c^{\mathcal{B}} \rightarrow_{\mathcal{B}} Pre_p(\tilde{x})$ implies $\leftarrow c, p(\tilde{x})$ left-terminates.

Let P be a CLP($\mathcal{N}$) program, p a predicate symbol of P defined by m_p rules. Let $P^{\mathcal{B}}$ be the boolean version of P and the kth-rule r_k defining p in $P^{\mathcal{B}}$ be:

$$p(\tilde{x}) \leftarrow \ c_{k,0}^{\mathcal{B}}, p_{k,1}(\tilde{x}_{k,1}), \ldots, p_{k,j_k}(\tilde{x}_{k,j_k})$$

Theorem 3.6 *Assume that for each predicate $q \notin \bar{p}$ appearing in the rules defining $\bar{p}$ we have a left-termination condition $Pre_q(\tilde{y})$. If $\{Pre_p(\tilde{x})\}_{p\in\bar{p}}$ verifies:*

$$\forall p \in \bar{p} \quad \begin{cases} Pre_p(\tilde{x}) \;\rightarrow_{\mathcal{B}}\; \gamma_p(\tilde{x}), \\ \forall 1 \le k \le m_p, \forall 1 \le i \le j_k, \; \left(Pre_p(\tilde{x}) \wedge c^{\mathcal{B}}_{k,0} \bigwedge_{j=1}^{i-1} Post_{p_j}(\tilde{x}_j)\right) \rightarrow_{\mathcal{B}} Pre_{p_{k,i}}(\tilde{x}_{k,i}) \end{cases}$$

then $\{Pre_p(\tilde{x})\}_{p\in\bar{p}}$ is a left-termination condition for $\bar{p}$.

Note that the above system of boolean formulae can be used not only to check that some relations Pre_p's are correct left-termination conditions but also to *compute* the Pre_p's by means of boolean mu-calculus [2].

Example 3.7 Let us consider the CLP($\mathcal{N}$) program, POWER-4, of the introduction. We compute: $\gamma_s(x,y) = x \vee y$, $Post_s(x,y) = x \wedge y$, $\gamma_p(x,y) = 1$ and $Post_p(x,y) = x \wedge y$. For left-termination we find by the above method: $Pre_s(x,y) = x \vee y$ and $Pre_p(x,y) = x$. Informally, these relations give us the following informations:

- for predicate s: for any query $\leftarrow c, s(x,y)$, if x or y is bounded in c, then the considered query left-terminates (Pre_s) and after the proof x and y are bounded ($Post_s$).
- for predicate p: $\leftarrow c, p(x,y)$, if x is bounded in c, then the considered query left-terminates and after the proof x and y are bounded.

But explained in the introduction, a query $\leftarrow c, p(x,y)$ s.t. y is bounded in c terminates if we prove $s(z,y)$ before $s(x,z)$ and then, for the predicate p, we would like to infer $Pre_p(x,y) = x \vee y$. $\triangleleft$

4 Inferring extended left-termination conditions

This section presents an important improvement of the previous method in order to infer larger termination conditions. The idea consists in the introduction of the notion of order inside clauses.

Definition 4.1 A boolean term $Pre_p(\tilde{x})$ is an *extended left-termination condition* for $p(\tilde{x})$ if, for every constraint c, $c^{\mathcal{B}} \rightarrow_{\mathcal{B}} Pre_p(\tilde{x})$ implies there exists a static reordering of literals of the clauses such that $\leftarrow c, p(\tilde{x})$ left-terminates.

The following definition presents the idea that allows us to deal with the notion of order inside clauses.

Definition 4.2 Let P be a program. For each rule of P: $p(\tilde{x}) \leftarrow c, \tilde{B}$ where the body $\tilde{B}$ contains the literals: $p_1, \ldots, p_n$ (but we do not know the order of them), we can associate a sequence of $n(n-1)/2$ boolean variables $(b_{ij})_{1\le i<j\le n}$, called *order variables*, with the following constraints (which express transitivity of the order relation):

$$\forall 1 \le i < j < k \le n, \begin{cases} b_{ij} \wedge b_{jk} \rightarrow b_{ik}, & b_{jk} \wedge \neg b_{ik} \rightarrow \neg b_{ij}, \\ b_{ik} \wedge \neg b_{jk} \rightarrow b_{ij}, & \neg b_{ik} \wedge b_{ij} \rightarrow \neg b_{jk}, \\ \neg b_{ij} \wedge b_{ik} \rightarrow b_{jk}, & \neg b_{jk} \wedge \neg b_{ij} \rightarrow \neg b_{ik} \end{cases}$$

The semantics of these variables is: $\forall\ 1 \le i < j \le n, \begin{cases} b_{ij} & = 1 \text{ if } p_i \text{ appears before } p_j, \\ & = 0 \text{ if } p_j \text{ appears before } p_i \end{cases}$

Let P be a CLP($\mathcal{N}$) program, p a predicate symbol of P defined by m_p rules. Let $P^{\mathcal{B}}$ be the boolean version of P and the kth-rule r_k defining p in $P^{\mathcal{B}}$ be:

$$p(\tilde{x}) \leftarrow\ c^{\mathcal{B}}_{k,0}, p_{k,1}(\tilde{x}_{k,1}), \ldots, p_{k,j_k}(\tilde{x}_{k,j_k})$$

Theorem 4.3 *Assume that for each $q \notin \bar{p}$ and appearing in the rules defining $\bar{p}$, an extended left-termination condition Pre_q has been computed. If the set of terms $\{Pre_p\}_{p\in\bar{p}}$ verifies:*

$$\forall p \in \bar{p} \begin{cases} Pre_p(\tilde{x}) \rightarrow_{\mathcal{B}} \gamma_p(\tilde{x}), \\ \forall 1 \leq k \leq m_p\ \exists(b^k_{eh})_{1\leq e<h\leq j_k}\ \bigwedge_{j=1}^{j=j_k} \Big[\Big(Pre_p(\tilde{x}) \wedge c^{\mathcal{B}}_{k,0} \wedge \bigwedge_{i=1}^{j-1}(\neg b^k_{ij} \vee Post_{p_{k,i}}(\tilde{x}_{k,i})) \\ \qquad\qquad \wedge\ \bigwedge_{i=j+1}^{j_k}(b^k_{ji} \vee Post_{p_{k,i}}(\tilde{x}_{k,i}))\Big) \rightarrow_{\mathcal{B}} Pre_{p_{k,j}}(\tilde{x}_{k,j})\Big] \\ \end{cases}$$

then $\{Pre_p\}_{p\in\bar{p}}$ is an extended left-termination condition for $\bar{p}$.

In the same way of theorem 3.6, the above system of boolean formulae is really used in our system to compute the terms Pre_p (here again by a fixpoint calculus using mu-calculus, see the table of results of section 6). A left-termination condition can be written as a disjunction of conjunctions of variables and in the following, we call *class of queries* such a conjunction.

5 Reordering to left-terminate

We did the first part of the work: compute for each predicate its extended left-termination condition. Now, if the user gives a query that entails the corresponding extended left-termination condition, we must compile the program into another one by reordering the literals to ensure left-termination of any proof of the query.

Definition 5.1 Let $\{1, \ldots, n\}$ be a set of natural numbers. A *permutation* σ over $\{1, \ldots, n\}$ is a bijection from $\{1, \ldots, n\}$ to $\{1, \ldots, n\}$. We note σ_i the image of i by the permutation σ and σ^{-1} the reverse permutation.

Definition 5.2 We define an *environment* as a tuple $\langle P, \{Pre_p \mid p \in \Pi\}, \phi\rangle$ where:

- P is a CLP($\mathcal{N}$) program with Π as the set of its predicate symbols,
- $Pre_p = \bigvee C^i_p$ is the extended left-termination condition of the predicate p,
- ϕ is a function that maps a rule $r_k : p \leftarrow c, p_{k,1}, \ldots, p_{k,j_k}$ and a class C^i_p to a permutation over $\{1, \ldots, j_k\}$.

In concrete terms, after inferring the pre-conditions Pre_p for each predicate p (theorem 4.3), we compute a permutation for each class C^l_p of Pre_p and for each rule r_k defining p. To this aim, we compute with a boolean solver a sequence $(b^k_{eh})_{1\leq e<h\leq j_k}$ of booleans s.t. the following formulae are true:

$$\bigwedge_{j\leq j_k} \Big[\Big(C^l_p(\tilde{x}) \wedge c^{\mathcal{B}}_{k,0} \wedge \bigwedge_{i=1}^{j-1}(\neg b^k_{ij} \vee Post_{p_{k,i}}(\tilde{x}_{k,i}))\wedge \\ \bigwedge_{i=j+1}^{j_k}(b^k_{ji} \vee Post_{p_{k,i}}(\tilde{x}_{k,i}))\Big) \rightarrow_{\mathcal{B}} Pre_{p_{k,j}}(\tilde{x}_{k,j})\Big]$$

We get a (partial) instantiation of order variables b^k_{ij}'s that we turn into a total instantiation corresponding to a permutation σ. We define $\phi(r_k, C^l_p) = \sigma$. At this point, we have an environment for the program P. Then to each pair $\langle$ predicate p; class $C^l_p\rangle$ we associate a new predicate symbol p^l:

Definition 5.3 Let $\langle P, \{Pre_p \mid p \in \Pi\}, \phi\rangle$ be an environment. We define a *renaming table* T as a mapping that associates an unique new symbol of predicate p^l to a pair $\langle p, C_p^l\rangle$ where $p \in \Pi$ and C_p^l is a class of the extended left-termination condition of p.

Now, we have a program P, $\langle P, \{Pre_p \mid p \in \Pi\}, \phi\rangle$ its computed environment, T the associated renaming table and $\leftarrow c, p(\tilde{x})$ a user query related to P s.t. $c^{\mathcal{B}}$ (the boolean version of c) verifies: $\exists C_p \in Pre_p, c^{\mathcal{B}} \rightarrow_{\mathcal{B}} C_p(\tilde{x})$. The three points below outline a procedure[3] to construct a new program P' by renaming and reordering literals:

1. We compute C the class of queries corresponding to the current predicate q with its current calling context. We begin with $q \equiv p$ and $C \equiv C_p$.
2. We copy each rule r defining q in P, we change q by q' ($q' = T(C, q)$) and we reorder literals using the permutation $\phi(r, C)$. Then we add this new rule in program P'.
3. For each rule of P', for each predicate appearing in this rule but not defined in P' we compute its calling context. Then, we deduce its class of queries and we apply points 1 and 2, seen above, for this predicate. And so on. The procedure ends when all predicates appearing in P' are defined in P'.

The new program P' verifies two essential properties:

Proposition 5.4 *For all $p' \in \Pi'$ there exists $p \in \Pi$ and $C \in Pre_p$ s.t. $\langle p, C, p'\rangle \in E$. If we rename each symbol p' by its corresponding symbol p then we have: $S_{P'}^{\mathcal{N}} = S_P^{\mathcal{N}}$*

Theorem 5.5 *The proof of the query $\leftarrow c(\tilde{y}), p'(\tilde{x})$ related to P' left-terminates.*

We end this abstract by the example of the introduction.

Example 5.6 When we apply our method on the program POWER-4 (examples 1.1 and 3.7). We find: $Pre_s(x, y) = x \vee y$, $Pre_p(x, y) = x \vee y$, $\phi(r_3, x) = \omega$ (defined by $\omega_1 = 1$, $\omega_2 = 2$), $\phi(r_3, y) = \omega'$ ($\omega'_1 = 2$, $\omega'_2 = 1$).
Now let us consider the query of the introduction: $\leftarrow Y \leq 16, p(X, Y)$. It corresponds to the class "Y bounded" i.e. $Y = 1$ in the boolean version. The new program P' is:

rule r'_1 :	$\mathtt{p'(X,Y)\ :-\ s'(Z,Y), s'(X,Z).}$
rule r'_2 :	$\mathtt{s'(0,0).}$
rule r'_3 :	$\mathtt{s'(X+1, Y+2X+1)\ :-\ s'(X,Y).}$

The call $\leftarrow Y \leq 16, p'(X, Y)$ left-terminates with answers: $\{X = 0\}, \{X = 1\}, \{X = 2\}$. $\triangleleft$

6 Conclusion

Let us first summarize our approach. From our previous work on the inference of left-terminating classes of queries, we are able to get rid of any particular order of literals to compute the set of terminating classes of queries. The main idea is to embody all the orderings by introducing sequences of new boolean variables. Then, for any terminating

[3]the detailed algorithm is available in the extended version of this abstract.

class, we can statically compile the control into Prolog. In other words, we can produce a Prolog program such that for any query of this class, universal left-termination is guaranteed. The approach described in [7] is now completely implemented in SICStus Prolog. For the work presented here, we have implemented the computation of $\{Pre_p\}_{p\in\bar{p}}$ as defined in the point 1 of the method of section 4. In the following table, we present some results on program tests found in N. Lindenstrauss et Y. Sagiv's works [4] (execution times are given in seconds).

Name of		Hoarau/Mesnard		Lindenstrauss/Sagiv	
program	main predicate	inferred classes	time	checked classes	time
`append`	$append(x,y,z)$	$\{x \vee z\}$	0,47	$\{x \vee z\}$	0,51
`reverse`	$reverse(x,y,z)$	$\{(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)\}$	0,89	$\{x \wedge z\}$	0,17
`permute`	$permute(x,y)$	$\{x \vee y\}$	1,31	$\{x\}$	0,69
`hanoiapp`	$hanoi(w,x,y,z,t)$	$\{(w \wedge x \wedge y \wedge z) \vee t\}$	53,2	$\{w \wedge x \wedge y \wedge z\}$	62,5
`fib_t`	$fib(x,y)$	$\{x\}$	7,3	$\{x\}$	0,79
`occur`	$occurall(x,y,z)$	$\{x \wedge y\}$	4,2	$\{x \wedge y\}$	2,45
`money`	$money(a,b,c,d,e,f,g,h)$	$\{1\}$	3,65	$\{1\}$	27,5
`zebra`	$zebra(a,b,c,d,e,f,g)$	$\{1\}$	3,26	$\{1\}$	0,58
Machines		Ultra 1 SUN Station, 128MB		Super SPARC 51, 128MB	

Let us point out that on these examples, the set of classes of queries that we *infer* is at least as large as the set of classes *checked* by Lindenstrauss et Sagiv. We cannot expect the timings given by [6] or [9] but we address a much more general problem. Note also that there is room for trying to *optimize* the control of a given logic with respect to a given terminating class of queries. While we construct the function ϕ (see definition 5.2), among the orderings which ensure termination, we may choose an ordering which leads to a "small" search tree. This is work in progress.

Thanks to Antoine Rauzy for sharing with us its knowledge about the boolean mu-calculus.

References

[1] K.R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In *Proceedings Esprit symposium on computational logic*, pages 150–176. Springer-Verlag, 1990.

[2] S. Colin, F. Mesnard, and A. Rauzy. Constraint logic programming and mu-calculus. *ERCIM/COMPULOG Workshop on Constraints*, 1997.

[3] D. De Schreye and S. Decorte. Termination of logic programs : the never-ending story. *The Journal of Logic Programming*, 12:1–66, 1993.

[4] M. Gabbrielli and G. Levi. Modelling answer constraints in constraint logic programs. In MIT Press, editor, *Proc. of ICLP'91*, pages 238–252, 1991.

[5] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *J. Logic Programming*, 19:503–581, 1994.

[6] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. *Proc. of the 14th ICLP*, pages 63–77, 1997.

[4] in the full version of [6] available at: www.cs.huji.ac.il/~naomil.

[7] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In *Proc. of JICSLP'96*, pages 7–21. MIT Press, 1996.

[8] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. *Proc. of PODS'91*, pages 216–226, 1991.

[9] C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. *Proc. of SAS'97*, 1997.