An Improved Non-Termination Criterion for Binary Constraint Logic Programs

Etienne Payet and Fred Mesnard

IREMIA - Université de La Réunion, France
email: epayet@univ-reunion.fr

Abstract. On one hand, termination analysis of logic programs is now a fairly established research topic within the logic programming community. On the other hand, non-termination analysis seems to remain a much less attractive subject. If we divide this line of research into two kinds of approaches: dynamic versus static analysis, this paper belongs to the latter. It proposes a criterion for detecting non-terminating atomic queries with respect to binary CLP clauses, which strictly generalizes our previous works on this subject. We give a generic operational definition and a logical form of this criterion. Then we show that the logical form is correct and complete with respect to the operational definition.

1 Introduction

On one hand, termination analysis of logic programs is a fairly established research topic within the logic programming community, see the surveys [5, 12]. For Prolog, various tools are now available via web interfaces and we note that the Mercury compiler, designed with industrial goals in mind by its implementors, has included two termination analyzers (see [18] and [7]) for a few years.

On the other hand, non-termination analysis seems to remain a much less attractive subject. We can divide this line of research into two kinds of approaches: dynamic versus static analysis. In the former one, [1] sets up some solid foundations for loop checking, while some recent work is presented in [16]. The main idea is to prune at runtime at least all infinite derivations, and possibly some finite ones. In the latter approach, which includes the work we present in this article, [4, 6] present an algorithm for detecting non-terminating atomic queries with respect to a binary clause of the type $p(\tilde{s}) \leftarrow p(\tilde{t})$. The condition is described in terms of rational trees, while we aim at generalizing non-termination analysis for the generic CLP(X) framework.

Our analysis shares with some work on termination analysis [3] a key component: the binary unfoldings of a logic program [8], which transforms a finite set of definite clauses into a possibly infinite set of facts and binary definite clauses. While some termination analyses begin with the analysis of the recursive binary clauses of an upper approximation of the binary unfoldings of an abstract CLP(N) version of the original program, we start from a finite subset of the binary unfoldings of the concrete program P (a larger subset may increase the

precision of the analysis, see [13] for some experimental evidence). First we detect patterns of non-terminating atomic queries for binary recursive clauses and then propagate this non-termination information to compute classes of atomic queries for which we have a finite proof that there exists at least one infinite derivation with respect to the subset of the binary unfoldings of P.

The equivalence of termination for a program or its binary unfoldings given in [3] is a corner stone of both analyses. It allows us to conclude that any atomic query belonging to the identified above classes of queries admits an infinite left derivation with respect to P. So in this paper, we deliberately choose to restrict the analysis to binary CLP clauses and atomic CLP queries as the result we obtain can be directly lifted to full CLP.

Our initial motivation, see [11], is to complement termination analysis with non-termination inside the logic programming paradigm in order to detect optimal termination conditions expressed in a language describing classes of queries. We started from a generalization of the lifting lemma where we may ignore some arguments. For instance, from the clause $p(f(X), Y) \leftarrow p(X, g(Y))$, we can conclude that the atomic query $p(X, t)$ loops for any term t, thus ignoring the second argument. Then we have extended the approach, see [13] which gives the full picture of the non-termination analysis, an extensive experimental evaluation, and a detailed comparison with related works. For instance, from the clause $p(f(X), g(Y)) \leftarrow p(X, g(b))$, and with the help of the criterion designed in [13] we can now conclude that $p(X, t)$ loops for any term t which is an instance of $g(X)$.

Although we obtained interesting experimental results from such a criterion, the overall approach remains quite syntactic, with an *ad hoc* flavor and tight links to some basic logic programming machinery such as the unification algorithm. So we moved to the constraint logic programming scheme: in [14], we started from a generic definition of the generalization of the lifting lemma we were looking for. Such a definition was practically useless but we were able to give a sufficient condition expressed as a logical formula related to the constraint binary clause $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ under consideration. For some constraint domains, we showed that the condition is also necessary. Depending on the constraint theory, the validity of such a condition can be automatically decided. Moreover, we showed that the syntactic criterion we used in [11] was actually equivalent to the logical criterion and could be considered as a correct and complete implementation specialized for the algebra of finite trees *Term*.

The main contribution of this article consists in a strict generalization of the logical criterion defined in [14] which allows us to reconstruct the syntactic approaches described in [11] and [13]. We emphasize the improvement with respect to [14] in Sect. 5 (see Example 21).

The paper is organized as follows. First, in Sect. 2, we introduce some preliminary definitions. Then, in Sect. 3, we recall, using CLP terms, the subsumption test to detect looping queries. In Sect. 4, we present our generalized criterion for detecting looping queries, whilst in Sect. 5 we consider the connections with the results of [14].

2 Preliminaries

For any non-negative integer n, $[1, n]$ denotes the set $\{1, \ldots, n\}$. If $n = 0$, then $[1, n] = \varnothing$.

2.1 First Order Formulas

Throughout this paper, we consider a fixed, infinite and denumerable set of variables $\mathcal{V}$.

A *signature* defines a set of function and predicate symbols and associates an *arity* with each symbol. If ϕ is a first order formula on a signature Σ and $W := \{X_1, \ldots, X_n\}$ is a set of variables, then $\exists_W \phi$ (resp. $\forall_W \phi$) denotes the formula $\exists X_1 \ldots \exists X_n \phi$ (resp. $\forall X_1 \ldots \forall X_n \phi$). We let $\exists \phi$ (resp. $\forall \phi$) denote the existential (resp. universal) closure of ϕ. A *Σ-structure* $\mathcal{D}$ is an interpretation of the symbols in the signature Σ. It is a pair $(D, [\cdot])$ where D is a set called the *domain* of $\mathcal{D}$ and $[\cdot]$ maps:

- each function symbol f of arity n in Σ to a function $[f] : D^n \to D$,
- each predicate symbol p of arity n in Σ to a boolean function $[p] : D^n \to \{0, 1\}$.

A *$\mathcal{D}$-valuation* (or simply a *valuation* if the Σ-structure $\mathcal{D}$ is understood) is a mapping $v : \mathcal{V} \to D$. Every $\mathcal{D}$-valuation v extends (by morphism) to terms:

- $v(f(t_1, \ldots, t_n)) := [f](v(t_1), \ldots, v(t_n))$ if $f(t_1, \ldots, t_n)$ is a term.

A $\mathcal{D}$-valuation v induces a valuation $[\cdot]_v$ of formulas to $\{0, 1\}$:

- $[p(t_1, \ldots, t_n)]_v := [p](v(t_1), \ldots, v(t_n))$ if $p(t_1, \ldots, t_n)$ is an atomic proposition,
- if ϕ_1 and ϕ_2 are formulas and $\circ \in \{\wedge, \vee, \to, \leftrightarrow\}$, $[\neg\phi_1]_v$ and $[\phi_1 \circ \phi_2]_v$ are deduced from $[\phi_1]_v$, $[\phi_2]_v$ and the truth table of $\neg$ and $\circ$,
- if X is a variable and ϕ is a formula, $[\exists X \phi]_v = 1$ if and only if there exists a valuation v' such that $[\phi]_{v'} = 1$ and for each variable Y distinct from X, $v'(Y) = v(Y)$,
- if X is a variable and ϕ is a formula, $[\forall X \phi]_v = 1$ if and only if $[\phi]_{v'} = 1$ for every valuation v' such that for each variable Y distinct from X, $v'(Y) = v(Y)$.

Given a formula ϕ, we write $\mathcal{D} \models_v \phi$ if $[\phi]_v = 1$ and $\mathcal{D} \not\models_v \phi$ if $[\phi]_v = 0$. We write $\mathcal{D} \models \phi$ if and only if for every $\mathcal{D}$-valuation v, we have $\mathcal{D} \models_v \phi$. Notice that $\mathcal{D} \models \forall\phi$ if and only if $\mathcal{D} \models \phi$, that $\mathcal{D} \models \exists\phi$ if and only if there exists a $\mathcal{D}$-valuation v such that $\mathcal{D} \models_v \phi$, and that $\mathcal{D} \models \neg\exists\phi$ if and only if $\mathcal{D} \models \neg\phi$.

Given a Σ-structure $\mathcal{D}$, we say that a Σ-formula ϕ is *satisfiable* (resp. *unsatisfiable*) in $\mathcal{D}$ if $\mathcal{D} \models \exists\phi$ (resp. $\mathcal{D} \models \neg\phi$). We say that $\mathcal{D}$ is a *model* of a set S of Σ-formulas if for each element ϕ of S we have $\mathcal{D} \models \phi$. Given two sets S and T of Σ-formulas, we say that S *semantically implies* T, written $S \models T$, if every model of S is also a model of T.

2.2 Sequences

Sequences of distinct variables are denoted by $\tilde{X}$, $\tilde{Y}$, $\tilde{Z}$, $\tilde{U}$, ... and sequences of (not necessarily distinct) terms are denoted by $\tilde{s}$, $\tilde{t}$, ... Given two sequences of n terms $\tilde{s} := (s_1, \ldots, s_n)$ and $\tilde{t} := (t_1, \ldots, t_n)$, we write $\tilde{s} = \tilde{t}$ either to denote the formula $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$ or as a shorthand for "$s_1 = t_1$ and ... and $s_n = t_n$". Moreover, given a valuation v, we write $v(\tilde{s})$ to denote the sequence $(v(s_1), \ldots, v(s_n))$. Finally, given a sequence $\tilde{X} := (X_1, \ldots, X_n)$ of distinct variables and given a formula ϕ, we write $\exists_{\tilde{X}}\phi$ (resp. $\forall_{\tilde{X}}\phi$) to denote the formula $\exists X_1 \ldots \exists X_n \phi$ (resp $\forall X_1 \ldots \forall X_n \phi$).

2.3 Constraint Domains

We recall some basic definitions about CLP, see [10] for more details. In this paper, we consider a constraint logic programming language CLP($\mathcal{C}$) based on the constraint domain $\mathcal{C} := \langle \Sigma_\mathcal{C}, \mathcal{L}_\mathcal{C}, \mathcal{D}_\mathcal{C}, \mathcal{T}_\mathcal{C}, \mathit{solv}_\mathcal{C} \rangle$. The constraint domain signature $\Sigma_\mathcal{C}$ is a pair $\langle F_\mathcal{C}, \Pi_\mathcal{C} \rangle$ where $F_\mathcal{C}$ is a set of function symbols and $\Pi_\mathcal{C}$ is a set of predicate symbols. The domain of computation $\mathcal{D}_\mathcal{C}$ is a $\Sigma_\mathcal{C}$-structure $(D_\mathcal{C}, [\cdot]_\mathcal{C})$ that is the intended interpretation of the constraints. We assume the following:

- $\mathcal{C}$ is ideal,
- the predicate symbol $=$ is in $\Sigma_\mathcal{C}$ and is interpreted as identity in $D_\mathcal{C}$,
- $\mathcal{D}_\mathcal{C}$ and $\mathcal{T}_\mathcal{C}$ correspond on $\mathcal{L}_\mathcal{C}$,
- $\mathcal{T}_\mathcal{C}$ is satisfaction complete with respect to $\mathcal{L}_\mathcal{C}$,
- the theory and the solver agree in the sense that for every $c \in \mathcal{L}_\mathcal{C}$, $\mathit{solv}_\mathcal{C}(c) =$ `true` if and only if $\mathcal{T}_\mathcal{C} \models \exists c$. Consequently, as $\mathcal{D}_\mathcal{C}$ and $\mathcal{T}_\mathcal{C}$ correspond on $\mathcal{L}_\mathcal{C}$, we have, for every $c \in \mathcal{L}_\mathcal{C}$, $\mathit{solv}_\mathcal{C}(c) =$ `true` if and only if $\mathcal{D}_\mathcal{C} \models \exists c$.

Example 1 ($\mathcal{R}_{lin}$). The constraint domain $\mathcal{R}_{lin}$ has $<$, $\leq$, $=$, $\geq$ and $>$ as predicate symbols, $+$, $-$, $*$, / as function symbols and sequences of digits (possibly with a decimal point) as constant symbols. Only linear constraints are admitted. The domain of computation is the structure with reals as domain and where the predicate symbols and the function symbols are interpreted as the usual relations and functions over reals. The theory $\mathcal{T}_{\mathcal{R}_{lin}}$ is the theory of real closed fields [17]. A constraint solver for $\mathcal{R}_{lin}$ always returning either `true` or `false` is described in [15].

Example 2 (Logic Programming). The constraint domain *Term* has $=$ as predicate symbol and strings of alphanumeric characters as function symbols. The domain of computation of *Term* is the set of *finite trees* (or, equivalently, of finite terms), *Tree*, while the theory $\mathcal{T}_{Term}$ is Clark's equality theory [2]. The interpretation of a constant is a tree with a single node labeled with the constant. The interpretation of an n-ary function symbol f is the function $f_{Tree} : \mathit{Tree}^n \to \mathit{Tree}$ mapping the trees $T_1, \ldots, T_n$ to a new tree with root labeled with f and with $T_1, \ldots, T_n$ as child nodes. A constraint solver always returning either `true` or `false` is provided by the *unification* algorithm. CLP(*Term*) coincides then with logic programming.

2.4 Operational Semantics

The signature in which all programs and queries under consideration are included is $\Sigma_L := \langle F_L, \Pi_L \rangle$ with $F_L := F_{\mathcal{C}}$ and $\Pi_L := \Pi_{\mathcal{C}} \cup \Pi'_L$ where Π'_L, the set of predicate symbols that can be defined in programs, is disjoint from $\Pi_{\mathcal{C}}$. We assume that each predicate symbol p in Π_L has a unique arity denoted by $arity(p)$.

An *atom* has the form $p(\tilde{t})$ where $p \in \Pi'_L$ and $\tilde{t}$ is a sequence of $arity(p)$ Σ_L-terms. Throughout this paper, when we write $p(\tilde{t})$, we implicitly assume that $\tilde{t}$ contains $arity(p)$ terms. A CLP($\mathcal{C}$) *program* is a finite set of rules. A *rule* has the form $H \leftarrow c \diamond B$ where H and B are atoms and c is a finite conjunction of primitive constraints such that $\mathcal{D}_{\mathcal{C}} \models \exists c$. A *query* has the form $\langle A \mid d \rangle$ where A is an atom and d is a finite conjunction of primitive constraints. Given an atom $A := p(\tilde{t})$, we write $rel(A)$ to denote the predicate symbol p. Given a query $S := \langle A \mid d \rangle$, we write $rel(S)$ to denote the predicate symbol $rel(A)$. The set of variables occurring in some syntactic objects $O_1, \ldots, O_n$ is denoted $Var(O_1, \ldots, O_n)$.

The examples of this paper make use of the language CLP($\mathcal{R}_{lin}$) and the language CLP(*Term*). In program and query examples, variables begin with an upper-case letter, $[Head|Tail]$ denotes a list with head *Head* and tail *Tail*, and $[]$ denotes an empty list.

We consider the following operational semantics given in terms of *derivations* from queries to queries. Let $\langle p(\tilde{u}) \mid d \rangle$ be a query and $r := p(\tilde{s}) \leftarrow c \diamond q(\tilde{t})$ be a rule. Let $r' := p(\tilde{s}') \leftarrow c' \diamond q(\tilde{t}')$ be a variant of r variable disjoint with $\langle p(\tilde{u}) \mid d \rangle$ such that $solv_{\mathcal{C}}(\tilde{s}' = \tilde{u} \wedge c' \wedge d) = \texttt{true}$. Then, $\langle p(\tilde{u}) \mid d \rangle \underset{r}{\Longrightarrow} \langle q(\tilde{t}') \mid \tilde{s}' = \tilde{u} \wedge c' \wedge d \rangle$ is a *derivation step* of $\langle p(\tilde{u}) \mid d \rangle$ with respect to r with r' as its *input rule*. We write $S \overset{+}{\underset{P}{\Longrightarrow}} S'$ to summarize a finite number (> 0) of derivation steps from S to S' where each input rule is a variant of a rule from program P. Let S_0 be a query. A sequence of derivation steps $S_0 \underset{r_1}{\Longrightarrow} S_1 \underset{r_2}{\Longrightarrow} \cdots$ of maximal length is called a *derivation* of $P \cup \{S_0\}$ if r_1, r_2, ... are rules from P and if the *standardization apart* condition holds, *i.e.* each input rule used is variable disjoint from the initial query S_0 and from the input rules used at earlier steps. We say S_0 *loops* with respect to P if there exists an infinite derivation of $P \cup \{S_0\}$.

3 Loop Inference with Constraints

In the logic programming framework, the subsumption test provides a simple way to infer looping queries: if, in a logic program P, there is a rule $p(\tilde{s}) \leftarrow p(\tilde{t})$ such that $p(\tilde{t})$ is more general than $p(\tilde{s})$, then the query $p(\tilde{s})$ loops with respect to P. In this section, we extend this result to the constraint logic programming framework.

3.1 A "More General Than" Relation

A query can be viewed as a finite description of a possibly infinite set of atoms, the arguments of which are values from $D_{\mathcal{C}}$.

Example 3. Suppose that $\mathcal{C} = \mathcal{R}_{lin}$.

- The query $\langle p(2 * X) \mid X \geq -1 \rangle$ describes those atoms $p(x)$ where x is a real and the term $2 * X$ can be made equal to x while the constraint $X \geq -1$ is satisfied.
- The query $\langle q(X, Y) \mid Y \leq X + 2 \rangle$ describes those atoms $q(x, y)$ where x and y are reals and X and Y can be made equal to x and y respectively while the constraint $Y \leq X + 2$ is satisfied.

In order to capture this intuition, we introduce the following definition.

Definition 1 (Set Described by a Query). *The set of atoms that is described by a query $S := \langle p(\tilde{t}) \mid d \rangle$ is denoted by $Set(S)$ and is defined as: $Set(S) = \{p(v(\tilde{t})) \mid \mathcal{D}_\mathcal{C} \models_v d\}$.*

Clearly, $Set(\langle p(\tilde{t}) \mid d \rangle) = \varnothing$ if and only if d is unsatisfiable in $\mathcal{D}_\mathcal{C}$. Moreover, two variants describe the same set. Notice that the operational semantics we introduced above can be expressed using sets described by queries:

Lemma 1. *Let S be a query and $r := H \leftarrow c \diamond B$ be a rule. There exists a derivation step of S with respect to r if and only if $Set(S) \cap Set(\langle H \mid c \rangle) \neq \varnothing$.*

The "more general than" relation we consider is defined as follows:

Definition 2 (More General). *We say that a query S' is* more general than *a query S if $Set(S) \subseteq Set(S')$.*

Example 4.

- In any constraint domain, $\langle p(X) \mid true \rangle$ is more general than any query S verifying $rel(S) = p$;
- In the constraint domain *Term*, the query $\langle p(Y) \mid Y = f(X) \rangle$ is more general than the query $\langle p(Y) \mid Y = f(f(X)) \rangle$;
- In the constraint domain $\mathcal{R}_{lin}$, the query $\langle q(X, Y) \mid Y \leq X + 2 \rangle$ is more general than the query $\langle q(X, Y) \mid Y \leq X + 1 \rangle$.

3.2 Loop Inference

Suppose we have a derivation step $S \underset{r}{\Longrightarrow} T$ where $r := H \leftarrow c \diamond B$. Then, by Lemma 1, $Set(S) \cap Set(\langle H \mid c \rangle) \neq \varnothing$. Hence, if S' is a query that is more general than S, as $Set(S) \subseteq Set(S')$, we have $Set(S') \cap Set(\langle H \mid c \rangle) \neq \varnothing$. So, by Lemma 1, there exists a query T' such that $S' \underset{r}{\Longrightarrow} T'$. The following lifting result says that, moreover, T' is more general than T:

Theorem 1 (Lifting). *Consider a derivation step $S \underset{r}{\Longrightarrow} T$ and a query S' that is more general than S. Then, there exists a derivation step $S' \underset{r}{\Longrightarrow} T'$ where T' is more general than T.*

From this theorem, we derive two corollaries that can be used to infer looping queries just from the text of a CLP($\mathcal{C}$) program:

Corollary 1. *Let $r := H \leftarrow c \diamond B$ be a rule. If $\langle B \,|\, c\rangle$ is more general than $\langle H \,|\, c\rangle$ then $\langle H \,|\, c\rangle$ loops with respect to $\{r\}$.*

Corollary 2. *Let $r := H \leftarrow c \diamond B$ be a rule from a program P. If $\langle B \,|\, c\rangle$ loops with respect to P then $\langle H \,|\, c\rangle$ loops with respect to P.*

Example 5. Consider the CLP(*Term*) rule r:

$$append([X|Xs],\, Ys, [X|Zs]) \leftarrow true \diamond append(Xs,\, Ys, Zs)$$

We note that the query $\langle append(Xs,\, Ys, Zs) \,|\, true\rangle$ is more general than the query $S := \langle append([X|Xs],\, Ys, [X|Zs]) \,|\, true\rangle$. So, by Corollary 1, S loops with respect to $\{r\}$. Therefore, there exists an infinite derivation ξ of $\{r\} \cup \{S\}$. Then, if S' is a query that is more general than S, by successively applying the Lifting Theorem 1 to each step of ξ, one can construct an infinite derivation of $\{r\} \cup \{S'\}$. So, S' also loops with respect to $\{r\}$.

4 Loop Inference Using Filters

The condition provided by Corollary 1 is rather weak because it fails at inferring looping queries in some simple cases. This is illustrated by the following example.

Example 6. Assume $\mathcal{C} = \mathcal{R}_{lin}$. Let

$$r := p(X, Y) \leftarrow X \geq 0 \wedge Y \leq 10 \diamond p(X + 1, Y + 1)\ .$$

We have the infinite derivation:

$$\begin{aligned} \langle p(X,Y) \,|\, c\rangle &\underset{r}{\Longrightarrow} \langle p(X_1 + 1, Y_1 + 1) \,|\, c \wedge c_1\rangle \\ &\underset{r}{\Longrightarrow} \langle p(X_2 + 1, Y_2 + 1) \,|\, c \wedge c_1 \ \wedge c_2\rangle \\ &\ \ \vdots \end{aligned}$$

where:

c is the constraint $X \geq 0 \wedge Y \leq 10$,
c_1 is the constraint $X_1 = X \wedge Y_1 = Y \wedge X_1 \geq 0 \wedge Y_1 \leq 10$ and
c_2 is the constraint $X_2 = X_1 + 1 \wedge Y_2 = Y_1 + 1 \wedge X_2 \geq 0 \wedge Y_2 \leq 10$.

But as in r, $\langle p(X + 1, Y + 1) \,|\, c\rangle$ is not more general than $\langle p(X, Y) \,|\, c\rangle$, Corollary 1 does not allow to infer that $\langle p(X, Y) \,|\, c\rangle$ loops with respect to $\{r\}$.

In this section, we extend the relation "is more general". Instead of comparing atoms in all positions using the "more general" relation, we distinguish some predicate argument positions for which we just require that a certain property must hold, while for the other positions we use the "more general" relation as before. Doing so, we aim at inferring more looping queries.

Example 7 (Example 6 continued). Let us consider argument position 1 of predicate symbol p. In the rule r, the argument of $p(X, Y)$ in position 1 is X and the argument of $p(X + 1, Y + 1)$ in position 1 is $X + 1$. Notice that the condition on X in c is $X \geq 0$ and that if $X \geq 0$ then $X + 1 \geq 0$. Hence, let us define the condition δ as: a query satisfies δ if it has the form $\langle p(t_1, t_2) \,|\, d\rangle$ where t_1 and t_2 are some terms and $\{v(t_1) \mid \mathcal{D}_\mathcal{C} \models_v d\}$ is included in the set of positive real numbers. Then, both $S := \langle p(X, Y) \,|\, c\rangle$ and $T := \langle p(X + 1, Y + 1) \,|\, c\rangle$ satisfy δ.

So, if we consider a "more general than" relation where we "filter" queries using δ, as S and T both satisfy δ and as the "piece" $\langle p(Y + 1) \,|\, c\rangle$ of T is more general than the "piece" $\langle p(Y) \,|\, c\rangle$ of S, by an extended version of Corollary 1 we could infer that S loops with respect to $\{r\}$.

4.1 Sets of Positions

A basic idea in Example 7 lies in identifying argument positions of predicate symbols. Below, we introduce a formalism to do so.

Definition 3 (Set of Positions). *A* set of positions*, denoted by τ, is a function that maps each predicate symbol $p \in \Pi'_L$ to a subset of $[1, arity(p)]$.*

Example 8. If we want to distinguish the first argument position of the predicate symbol p defined in Example 6, we set $\tau := \langle p \mapsto \{1\}\rangle$.

Definition 4. *Let τ be a set of positions. Then, $\overline{\tau}$ is the set of positions defined as: for each predicate symbol $p \in \Pi'_L$, $\overline{\tau}(p) = [1, arity(p)] \setminus \tau(p)$.*

Example 9 (Example 8 continued). We have $\overline{\tau} = \langle p \mapsto \{2\}\rangle$.

Using a set of positions τ, one can *project* syntactic objects:

Definition 5 (Projection). *Let τ be a set of positions.*

- *Let $p \in \Pi'_L$ be a predicate symbol. The* projection of p on τ *is the predicate symbol denoted by p_τ. Its arity equals the number of elements of $\tau(p)$.*
- *Let $p \in \Pi'_L$ be a predicate symbol of arity n and $\tilde{t} := (t_1, \ldots, t_n)$ be a sequence of n terms. The* projection of $\tilde{t}$ on τ*, denoted by $\tilde{t}_\tau$ is the sequence $(t_{i_1}, \ldots, t_{i_m})$ where $\{i_1, \ldots, i_m\} = \tau(p)$ and $i_1 \leq \cdots \leq i_m$.*
- *Let $A := p(\tilde{t})$ be an atom. The* projection of A on τ*, denoted by A_τ, is the atom $p_\tau(\tilde{t}_\tau)$.*
- *The* projection of a query $\langle A \,|\, d\rangle$ on τ*, denoted by $\langle A \,|\, d\rangle_\tau$, is the query $\langle A_\tau \,|\, d\rangle$.*

Example 10 (Example 6 and Example 8 continued). The projection of the query $\langle p(X, Y) \,|\, c\rangle$ on τ is the query $\langle p_\tau(X) \,|\, c\rangle$.

4.2 Filters

According to the intuitions described in Example 7 above, we define a filter as follows.

Definition 6 (Filter). *A* filter*, denoted by Δ, is a pair (τ, δ) where τ is a set of positions and δ is a function that maps each predicate symbol $p \in \Pi'_L$ to a query of the form $\langle p_\tau(\tilde{u}) \,|\, d\rangle$ where $\mathcal{D}_\mathcal{C} \models \exists d$ and $\tilde{u}$ is a sequence of arity(p_τ) terms.*

Example 11 (Example 6 and Example 7 continued). Let δ be the function defined as $\delta := \langle\, p \mapsto \langle p_\tau(X) \,|\, X \geq 0\rangle \,\rangle$. Then, $\Delta := (\tau, \delta)$ is a filter.

Example 12. Suppose that $\mathcal{C} = Term$. Let $p \in \Pi'_L$ be a predicate symbol whose arity is 1. Let $\tau := \langle p \mapsto \{1\}\rangle$ and $\delta := \langle\, p \mapsto \langle p_\tau(f(X)) \,|\, true\rangle \,\rangle$. Then, $\Delta := (\tau, \delta)$ is a filter.

The function δ is used to "filter" queries as indicated by the next definition.

Definition 7 (Satisfies). *Let $\Delta := (\tau, \delta)$ be a filter and S be a query. Let $p := rel(S)$. We say that S* satisfies *Δ if $Set(S_\tau) \subseteq Set(\delta(p))$.*

Now we come to the extension of the relation "more general than". Intuitively, $\langle p(\tilde{t'}) \,|\, d'\rangle$ is Δ-more general than $\langle p(\tilde{t}) \,|\, d\rangle$ if the "more general than" relation holds for the elements of $\tilde{t}$ and $\tilde{t'}$ whose position is not in τ while the elements of $\tilde{t'}$ whose position is in τ satisfy δ. More formally:

Definition 8 (Δ-More General). *Let $\Delta := (\tau, \delta)$ be a filter and S and S' be two queries. We say that S' is* Δ-more general than *S if $S'_{\overline{\tau}}$ is more general than $S_{\overline{\tau}}$ and S' satisfies Δ.*

Example 13.

- In the context of Example 11, $\langle p(X+1, Y+1) \,|\, X \geq 0 \wedge Y \leq 10\rangle$ is Δ-more general than $\langle p(X, Y) \,|\, X \geq 0 \wedge Y \leq 10\rangle$.
- In the context of Example 12, $\langle p(f(f(X))) \,|\, true\rangle$ is Δ-more general than $\langle p(f(X)) \,|\, true\rangle$.

Notice that for any filter $\Delta := (\tau, \delta)$ and any query S, we have that $S_{\overline{\tau}}$ is more general than itself (because the "more general than" relation is reflexive), but S may not satisfy Δ. Hence, the "Δ-more general than" relation is not always reflexive.

Example 14 (Example 12 continued). $S := \langle p(g(X)) \,|\, true\rangle$ is not Δ-more general than itself because, as $Set(S_\tau) = \{p_\tau(g(t)) \mid t \text{ is a term}\}$ and $Set(\delta(p)) = \{p_\tau(f(t)) \mid t \text{ is a term}\}$, we have $Set(S_\tau) \cap Set(\delta(p)) = \varnothing$. Hence, S does not satisfy Δ.

The fact that reflexivity does not always hold is an expected property. Indeed, suppose that a filter $\Delta := (\tau, \delta)$ induces a "Δ-more general than" relation that is reflexive. Then for any queries S and S', we have that S' is Δ-more general than S if and only if $S'_{\overline{\tau}}$ is more general than $S_{\overline{\tau}}$ (because, as S' is Δ-more general than itself, S' necessarily satisfies Δ). Hence, δ is useless in the sense that it "does not filter anything". Filters equipped with such a δ are studied in Sect. 5 and were introduced in [14] where for any predicate symbol p, $\delta(p)$ is $\langle p_\tau(\tilde{X}) \mid true \rangle$. In this paper, we aim at generalizing the approach of [14]. Hence, we also consider functions δ that really filter queries.

4.3 Derivation Neutral Filters: an Operational Definition

In the sequel of this paper, we focus on "derivation neutral" filters. The name "derivation neutral" stems from the fact that if, in a derivation of a query S, we replace S by S' that satisfies the filter, then we get a "similar" derivation.

Definition 9 (Derivation Neutral). *Let r be a rule and Δ be a filter. We say that Δ is* DN *for r if for each derivation step $S \underset{r}{\Longrightarrow} T$ and each query S' that is Δ-more general than S, there exists a derivation step $S' \underset{r}{\Longrightarrow} T'$ where T' is Δ-more general than T. This definition is extended to programs: Δ is* DN *for P if it is DN for each rule of P.*

Derivation neutral filters lead to the following extended version of Corollary 1 (to get Corollary 1, take $\Delta := (\tau, \delta)$ such that $\tau(p) = \varnothing$ for any p).

Proposition 1. *Let $r := H \leftarrow c \diamond B$ be a rule. Let Δ be a filter that is DN for r. If $\langle B \mid c \rangle$ is Δ-more general than $\langle H \mid c \rangle$ then $\langle H \mid c \rangle$ loops with respect to $\{r\}$.*

Example 15 (Example 7 continued). Suppose that Δ is DN for r. Now we can deduce that the query $\langle p(X, Y) \mid X \geq 0 \wedge Y \leq 10 \rangle$ loops with respect to r because the query $\langle p(X + 1, Y + 1) \mid X \geq 0 \wedge Y \leq 10 \rangle$ is Δ-more general than the query $\langle p(X, Y) \mid X \geq 0 \wedge Y \leq 10 \rangle$.

Computing a neutral filter from the text of a program is not that easy if we use the definition above. The next subsections present a logical and a syntactic characterization that can be used to compute a filter that is DN for a given program.

4.4 A Logical Characterization of Derivation Neutral Filters

From now on, we suppose that, without loss of generality, a rule has the form $p(\tilde{X}) \leftarrow c \diamond q(\tilde{Y})$ where $\tilde{X}$ and $\tilde{Y}$ are disjoint sequences of distinct variables. Hence, c is the conjunction of all the constraints, including unifications. We distinguish the following set of variables that appear inside such a rule.

Definition 10 (Local Variables). *Let $r := p(\tilde{X}) \leftarrow c \diamond q(\tilde{Y})$ be a rule. The set of* local variables *of r is denoted by $local\_var(r)$ and is defined as: $local\_var(r) := Var(c) \setminus (Var(\tilde{X}) \cup Var(\tilde{Y}))$.*

In this section, we aim at characterizing DN filters in a logical way. To this end, we define:

Definition 11 (sat). *Let $S := \langle p(\tilde{u}) \,|\, d\rangle$ be a query and $\tilde{s}$ be a sequence of $arity(p)$ terms. Then, $sat(\tilde{s}, S)$ denotes a formula of the form $\exists_{Var(S')}(\tilde{s} = \tilde{u}' \wedge d')$ where $S' := \langle p(\tilde{u}') \,|\, d'\rangle$ is any variant of S variable disjoint with $\tilde{s}$.*

Clearly, the satisfiability of $sat(\tilde{s}, S)$ does not depend on the choice of the variant of S. Now we give a logical definition of derivation neutrality. As we will see below, under certain circumstances, this definition is equivalent to the operational one we gave above.

Definition 12 (Logical Derivation Neutral). *We say that a filter $\Delta := (\tau, \delta)$ is* DNlog *for a rule $r := p(\tilde{X}) \leftarrow c \diamond q(\tilde{Y})$ if*

$$\mathcal{D}_\mathcal{C} \models c \rightarrow \forall_{\tilde{X}_\tau} \big[sat(\tilde{X}_\tau, \delta(p)) \rightarrow \exists_\mathcal{Y} [sat(\tilde{Y}_\tau, \delta(q)) \wedge c] \big]$$

where $\mathcal{Y} := Var(\tilde{Y}_\tau) \cup local\_var(r)$.

Intuitively, the formula in Definition 12 has the following meaning. If one holds a solution v for constraint c, then, changing the value given to the variables of $\tilde{X}$ distinguished by τ to some value satisfying $\delta(p)$, there exists a value for the local variables and the variables of $\tilde{Y}$ distinguished by τ such that c is still satisfied.

Example 16. Suppose that $\mathcal{C} = \mathcal{R}_{lin}$. Consider the rule $r := p(X_1, X_2) \leftarrow c \diamond p(Y_1, Y_2)$ where c is the constraint $X_1 = A + B \wedge A \geq 0 \wedge B \geq 0 \wedge X_2 \leq 10 \wedge Y_1 = X_1 + 1 \wedge Y_2 = X_2 + 1$. Then, the local variables of r are A and B. Any filter $\Delta := (\tau, \delta)$ where $\tau(p) = \{1\}$ and $\delta(p) = \langle p_\tau(X) \,|\, X \geq 0\rangle$ is DNlog for r. Indeed, $\tilde{X}_\tau = X_1$, $\mathcal{Y} = \{Y_1, A, B\}$ and $sat(t, \delta(p))$ is true if and only if $t \geq 0$. So the formula of Definition 12 turns into $\mathcal{D}_\mathcal{C} \models c \rightarrow \forall X_1 \big[X_1 \geq 0 \rightarrow \exists_{\{Y_1, A, B\}} [Y_1 \geq 0 \wedge c] \big]$, which is true.

Example 17. Suppose that $\mathcal{C} = Term$. Consider the rule $r := p(X) \leftarrow c \diamond p(Y)$ where c is the constraint $X = f(A) \wedge Y = f(f(A))$. Then, the only local variable of r is A. Any filter $\Delta := (\tau, \delta)$ where $\tau(p) = \{1\}$ and $\delta(p) = \langle p_\tau(X) \,|\, X = f(A)\rangle$ is DNlog for r. Indeed, $\tilde{X}_\tau = X$, $\mathcal{Y} = \{Y, A\}$ and $sat(t, \delta(p))$ is true if and only if t has the form $f(\cdots)$. So the formula of Definition 12 turns into

$$\begin{aligned} \mathcal{D}_\mathcal{C} \models c \rightarrow \forall X \big[& \text{ } X \text{ has the form } f(\cdots) \\ & \rightarrow \exists_{\{Y, A\}} [Y \text{ has the form } f(\cdots) \wedge c] \big], \end{aligned}$$

which is true.

The logical definition of derivation neutrality implies the operational one:

Proposition 2. *Let r be a rule and Δ be a filter. If Δ is DNlog for r then Δ is DN for r.*

The reverse implication does not always hold. But when considering a special case of the (SC_1) condition of *solution compactness* given in [9], we get:

Theorem 2. *Let r be a rule and Δ be a filter. Assume $\mathcal{C}$ enjoys the following property: for each $\alpha \in D_{\mathcal{C}}$, there exists a ground $\Sigma_{\mathcal{C}}$-term a such that $[a] = \alpha$. Then, Δ is DN for r if and only if Δ is DNlog for r.*

Proof (Sketch). We show how the (SC_1) condition is used to get this result.

By Proposition 2, we just have to establish that DN $\Rightarrow$ DNlog. Let $(\tau, \delta) := \Delta$ and $p(\tilde{X}) \leftarrow c \diamond q(\tilde{Y}) := r$. Suppose that Δ is DN for r. We have to prove that then, the formula of Definition 12 holds. Assume that v is a valuation such that

$$\mathcal{D}_{\mathcal{C}} \models_v c\,. \tag{1}$$

By property of $\mathcal{C}$, we can consider the query $S := \langle p(\tilde{a}) \,|\, true\rangle$ where $\tilde{a}$ is a sequence of ground terms such that $[\tilde{a}] = v(\tilde{X})$. As r and S are variable disjoint, we have $S \underset{r}{\Longrightarrow} T$ where T is the query $\langle q(\tilde{Y}) \,|\, c \wedge \tilde{X} = \tilde{a}\rangle$.

As we assumed (1), we have to establish that $\mathcal{D}_{\mathcal{C}} \models_v \forall_{\tilde{X}_\tau}\big[sat(\tilde{X}_\tau, \delta(p)) \rightarrow \exists_{\mathcal{Y}}[sat(\tilde{Y}_\tau, \delta(q)) \wedge c]\big]$ holds. Assume v_1 is a valuation such that

$$\mathcal{D}_{\mathcal{C}} \models_{v_1} sat(\tilde{X}_\tau, \delta(p)) \tag{2}$$

and for each variable $X \notin \tilde{X}_\tau$, $v(X) = v_1(X)$. By property of $\mathcal{C}$, we can consider the query $S' := \langle p(\tilde{b}) \,|\, true\rangle$ where $\tilde{b}_{\overline{\tau}} = \tilde{a}_{\overline{\tau}}$ and $\tilde{b}_\tau$ is a sequence of ground terms such that $[\tilde{b}_\tau] = v_1(\tilde{X}_\tau)$.

It can be noticed that S' is Δ-more general than S. As Δ is DN for r, there exists a query T' that is Δ-more general than T and such that $S' \underset{r}{\Longrightarrow} T'$. Necessarily, $T' = \langle q(\tilde{Y}') \,|\, c' \wedge \tilde{X}' = \tilde{b}\rangle$ where $p(\tilde{X}') \leftarrow c \diamond q(\tilde{Y}')$ is a variant of r variable disjoint with S'.

As we assumed (2), we now have to establish that $\mathcal{D}_{\mathcal{C}} \models_{v_1} \exists_{\mathcal{Y}}[sat(\tilde{Y}_\tau, \delta(q)) \wedge c]$ holds. This is done using the fact that T' is Δ-more general than T and that $Set(T') = Set(\langle q(\tilde{Y}) \,|\, c \wedge \tilde{X} = \tilde{b}\rangle)$. □

Example 18. In the constraint domain *Term*, DN is equivalent to DNlog.

4.5 A Syntactic Characterization of Derivation Neutral Filters

In [11], we gave, in the scope of logic programming, a syntactic definition of neutral arguments. Now we extend this syntactic criterion to the more general framework of constraint logic programming. First, we need rules in flat form:

Definition 13 (Flat Rule). *A rule $r := p(\tilde{X}) \leftarrow c \diamond q(\tilde{Y})$ is said to be* flat *if c has the form $(\tilde{X} = \tilde{s} \wedge \tilde{Y} = \tilde{t})$ where $\tilde{s}$ is a sequence of arity(p) terms and $\tilde{t}$ is a sequence of arity(q) terms such that $Var(\tilde{s}, \tilde{t}) \subseteq local\_var(r)$.*

Notice that there are some rules $r := p(\tilde{X}) \leftarrow c \diamond q(\tilde{Y})$ for which there exists no "equivalent" rule in flat form. More precisely, there exists no rule $r' := p(\tilde{X}) \leftarrow c' \diamond q(\tilde{Y})$ verifying $\mathcal{D}_{\mathcal{C}} \models \exists_{local\_var(r)} c \leftrightarrow \exists_{local\_var(r')} c'$ (take for instance $r := p(X) \leftarrow X > 0 \diamond p(Y)$ in $\mathcal{R}_{lin}$.)

Syntactic derivation neutrality is defined that way:

Definition 14 (Syntactic Derivation Neutral). *Let $\Delta := (\tau, \delta)$ be a filter and $r := p(\tilde{X}) \leftarrow (\tilde{X} = \tilde{s} \wedge \tilde{Y} = \tilde{t}) \diamond q(\tilde{Y})$ be a flat rule. We say that Δ is* DNsyn *for r if*

- **(DNsyn1)** *$\langle p(\tilde{s}) \mid true \rangle_\tau$ is more general than $\delta(p)$,*
- **(DNsyn2)** *$\delta(q)$ is more general than $\langle q(\tilde{t}) \mid true \rangle_\tau$,*
- **(DNsyn3)** *$Var(\tilde{s}_\tau) \cap Var(\tilde{s}_{\overline{\tau}}) = \varnothing$,*
- **(DNsyn4)** *$Var(\tilde{s}_\tau) \cap Var(\tilde{t}_{\overline{\tau}}) = \varnothing$.*

Example 19. In Example 17, the rule r is flat. Moreover, the filter Δ is DNsyn for r.

A connection between DN, DNsyn and DNlog is as follows:

Proposition 3. *Let r be a flat rule and Δ be a filter. If Δ is DNsyn for r then Δ is DNlog for r hence (by Proposition 2) Δ is DN for r. If Δ is DNlog for r then* **(DNsyn1)** *holds.*

Notice that a DNlog filter is not necessarily DNsyn because one of **(DNsyn2–4)** may not hold:

Example 20. In $\mathcal{R}_{lin}$, consider the flat rule r:

$$p(X_1, X_2) \leftarrow X_1 = A \wedge Y_1 = A \wedge X_2 = A - A \wedge Y_2 = A - A \diamond p(Y_1, Y_2)\ .$$

Let $\Delta := (\tau, \delta)$ where $\tau(p) = \{1\}$ and $\delta(p) = \langle p_\tau(X) \mid X \geq 0 \rangle$. Then, Δ is DNlog for r, but none of **(DNsyn2–4)** hold.

However, in the special case of logic programming, we have:

Proposition 4 (Logic Programming). *Suppose that $\mathcal{C} = Term$. Let r be a flat rule and Δ be filter. If Δ is DNlog for r then* **(DNsyn3)** *and* **(DNsyn4)** *hold.*

5 Connections with Earlier Results

The results of [14] can be easily obtained within the framework presented above. It suffices to consider the following special kind of filter:

Definition 15 (Open Filter). *We say that $\Delta := (\tau, \delta)$ is an* open filter *if for all $p \in \Pi'_L$, $\delta(p)$ has the form $\langle p_\tau(\tilde{Z}) \mid true \rangle$ where $\tilde{Z}$ is a sequence of distinct variables.*

In an open filter, the function δ "does not filter anything":

Lemma 2. *Let $\Delta := (\tau, \delta)$ be an open filter. Then, a query S' is Δ-more general than a query S if and only if $S'_{\overline{\tau}}$ is more general than $S_{\overline{\tau}}$.*

Consequently, an open filter is uniquely determined by its set of positions. When reconsidering the definitions and results of the preceding section within such a context, we exactly get what we presented in [14]. In particular, Definition 12 can be rephrased as:

Definition 16 (Logical Derivation Neutral). *A set of positions τ is* DNlog *for a rule $r := p(\tilde{X}) \leftarrow c \diamond q(\tilde{Y})$ if $\mathcal{D}_\mathcal{C} \models c \rightarrow \forall_{\tilde{X}_\tau} \exists_\mathcal{Y} c$ where $\mathcal{Y} := Var(\tilde{Y}_\tau) \cup local\_var(r)$.*

As stated in Sect 1, the framework presented in this paper is a strict generalization of that of [14]. This is illustrated by the following example.

Example 21 (Example 17 continued). First, notice that, as $\langle p(Y) \,|\, c\rangle$ is not more general than $\langle p(X) \,|\, c\rangle$, Corollary 1 does not allow to infer that $\langle p(X) \,|\, c\rangle$ loops with respect to $\{r\}$.

Let us try to use Definition 16 to prove that the argument of p is "irrelevant". We let $\tau(p) = \{1\}$. Hence, $\tilde{X}_\tau = X$, $\tilde{Y}_\tau = Y$ and $local\_var(r) = \{A\}$. Let us consider a valuation v such that $v(X) = f(a)$, $v(Y) = f(f(a))$ and $v(A) = a$. So, we have $\mathcal{D}_\mathcal{C} \models_v c$. But we do not have $\mathcal{D}_\mathcal{C} \models_v \forall_{\tilde{X}_\tau} \exists_\mathcal{Y} c$. For instance, if we consider v_1 such that $v_1(X) = a$ and $v_1(Z) = v(Z)$ for each variable Z distinct from X, we do not have $\mathcal{D}_\mathcal{C} \models_{v_1} \exists_\mathcal{Y} c$ as the subformula $X = f(A)$ of c cannot hold, whatever value is assign to A. Consequently, we do not have $\mathcal{D}_\mathcal{C} \models_v c \rightarrow \forall_{\tilde{X}_\tau} \exists_\mathcal{Y} c$, so τ is not DNlog for r. As $\mathcal{C} = Term$, by Theorem 2 τ is not DN for r. Therefore, using open filters with Proposition 1 we are not able to prove that $\langle p(X) \,|\, c\rangle$ loops with respect to $\{r\}$.

However, in Example 17, we noticed that any filter $\Delta := (\tau, \delta)$ where $\tau(p) = \{1\}$ and $\delta(p) = \langle p_\tau(X) \,|\, X = f(A)\rangle$ is DNlog, hence DN, for r. Moreover, for such a filter, $\langle p(Y) \,|\, c\rangle$ is Δ-more general than $\langle p(X) \,|\, c\rangle$. Consequently, by Proposition 1, $\langle p(X) \,|\, c\rangle$ loops with respect to $\{r\}$.

6 Conclusion

We have presented a criterion to detect non-terminating atomic queries with respect to a binary CLP clause. This criterion generalizes our previous papers in the CLP settings and allows us to reconstruct the work we did in the LP framework. However, when switching from LP to CLP, we lose the ability to compute, given a binary clause, a useful filter. We plan to work on this and try to define some conditions on the constraint domain which enable the computation of such filters. Moreover, as pointed out by an anonymous referee, DNsyn and DNlog seem to be independent notions which we proved to coincide only for open filters with the specific constraint domain *Term*. In Theorem 2 we investigate the relationship between DNlog and DN while Proposition 3 and Proposition 4 essentially establish some connections between DNsyn and DNlog. The study of relationship between DNsyn and DN is still missing and we intend to work on this shortly.

References

1. R. N. Bol, K. R. Apt, and J. W. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86:35–79, 1991.

2. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.
3. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
4. D. De Schreye, M. Bruynooghe, and K. Verschaetse. On the existence of non-terminating queries for a restricted class of Prolog-clauses. *Artificial Intelligence*, 41:237–248, 1989.
5. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.
6. D. De Schreye, K. Verschaetse, and M. Bruynooghe. A practical technique for detecting non-terminating queries for a restricted class of Horn clauses, using directed, weighted graphs. In *Proc. of ICLP'90*, pages 649–663. The MIT Press, 1990.
7. J. Fischer. Termination analysis for Mercury using convex constraints. Master's thesis, The University of Melbourne, Department of Computer Science and Software Engineering, 2002.
8. M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the ACM Symposium on applied computing*, pages 394–399. ACM Press, 1994.
9. J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM Press, 1987.
10. J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
11. F. Mesnard, E. Payet, and U. Neumerkel. Detecting optimal termination conditions of logic programs. In M. Hermenegildo and G. Puebla, editors, *Proc. of the 9th International Symposium on Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 509–525. Springer-Verlag, Berlin, 2002.
12. F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. *ACM Transactions on Computational Logic*, pages 207–259, 2003.
13. E. Payet and F. Mesnard. Non-termination inference of logic programs. *ACM Transactions on Programming Languages and Systems*. Accepted for publication. Preliminary version available at `http://www2.univ-reunion.fr/~gcc/papers.htm`.
14. E. Payet and F. Mesnard. Non-termination inference for constraint logic programs. In Roberto Giacobazzi, editor, *Proc. of the 11th International Symposium on Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 377–392. Springer-Verlag, Berlin, 2004.
15. P. Refalo and P. Van Hentenryck. CLP ($\mathcal{R}_{lin}$) revised. In M. Maher, editor, *Proc. of the Joint International Conf. and Symposium on Logic Programming*, pages 22–36. The MIT Press, 1996.
16. Y-D. Shen, L-Y. Yuan, and J-H. You. Loops checks for logic programs with functions. *Theoretical Computer Science*, 266(1-2):441–461, 2001.
17. J. Shoenfield. *Mathematical Logic*. Addison Wesley, Reading, 1967.
18. C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. van Hentenrick, editor, *Proc. of the 1997 Intl. Symp. on Static Analysis*, volume 1302 of *LNCS*. Springer-Verlag, 1997.