Non-Termination Inference for Constraint Logic Programs

Etienne Payet and Fred Mesnard

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

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 w.r.t. 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 to a criterion 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.

1 Introduction

Termination has been a subject of intensive research in the logic programming community for the last two decades, see the survey [4]. A more recent look on the topic, and its extension to the constraint logic programming paradigm [8, 9] is given in [14]. 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. Some of these works, *e.g.* [11, 7, 12] consider the *reverse* problem of inferring classes of queries for which universal left termination is ensured.

In contrast, the study of the *dual* problem: non-termination w.r.t. 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, *e.g.* [3, 5]. Recently we have also investigated this problem in the logic programming setting [13], where we proposed an analysis to infer non-termination.

In this paper, we study non-termination in the more general constraint logic programming framework. We rephrase our approach into this more abstract setting, which leads to a necessary and sufficient criterion 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. The analysis is fully implemented[1].

We organize the paper as follows. After the preliminaries presented in Section 2, some basic properties related to non-termination for CLP is given in Section 3. The technical machinery behind our approach is described in Section 4 and Section 5. Section 6 concludes.

2 Preliminaries

We recall some basic definitions on CLP, see [9] for more details.

2.1 Constraint Domains

In this paper, we consider a constraint logic programming language CLP($\mathcal{C}$) based on the constraint domain $\mathcal{C} := \langle \Sigma_{\mathcal{C}}, \mathcal{L}_{\mathcal{C}}, \mathcal{D}_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}}, solv_{\mathcal{C}} \rangle$.

$\Sigma_{\mathcal{C}}$ is the constraint domain *signature*, which 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 class of constraints $\mathcal{L}_{\mathcal{C}}$ is a set of first-order $\Sigma_{\mathcal{C}}$-formulas. The *domain of computation* $\mathcal{D}_{\mathcal{C}}$ is a $\Sigma_{\mathcal{C}}$-structure that is the intented interpretation of the constraints and $D_{\mathcal{C}}$ is the *domain* of $\mathcal{D}_{\mathcal{C}}$. The *constraint theory* $\mathcal{T}_{\mathcal{C}}$ is a $\Sigma_{\mathcal{C}}$-theory describing the logical semantics of the constraints. We suppose that $\mathcal{C}$ is *ideal i.e.* the *constraint solver*, $solv_{\mathcal{C}}$, is a computable function which maps each formula in $\mathcal{L}_{\mathcal{C}}$ to one of `true` or `false` indicating whether the formula is satisfiable or unsatisfiable.

We assume that the predicate symbol $=$ is in $\Sigma_{\mathcal{C}}$ and that it is interpreted as identity in $D_{\mathcal{C}}$. A *primitive constraint* is either the always satisfiable constraint *true* or the unsatisfiable constraint *false* or has the form $p(\tilde{t})$ where $p \in \Pi_{\mathcal{C}}$ and $\tilde{t}$ is a finite sequence of terms in $\Sigma_{\mathcal{C}}$. We suppose that $\mathcal{L}_{\mathcal{C}}$ contains all the primitive constraints and that it is closed under variable renaming, existential quantification and conjunction.

We suppose that $\mathcal{D}_{\mathcal{C}}$ and $\mathcal{T}_{\mathcal{C}}$ *correspond* on $\mathcal{L}_{\mathcal{C}}$ *i.e.*

- $\mathcal{D}_{\mathcal{C}}$ is a model of $\mathcal{T}_{\mathcal{C}}$ and
- for every constraint $c \in \mathcal{L}_{\mathcal{C}}$, $\mathcal{D}_{\mathcal{C}} \models \exists c$ if and only if $\mathcal{T}_{\mathcal{C}} \models \exists c$.

Moreover, we suppose that $\mathcal{T}_{\mathcal{C}}$ is *satisfaction complete* w.r.t. $\mathcal{L}_{\mathcal{C}}$ *i.e.* for every constraint $c \in \mathcal{L}_{\mathcal{C}}$, either $\mathcal{T}_{\mathcal{C}} \models \exists c$ or $\mathcal{T}_{\mathcal{C}} \models \neg\exists c$. We also assume that the theory and the solver *agree* in the sense that for every $c \in \mathcal{L}_{\mathcal{C}}$, $solv_{\mathcal{C}}(c) =$ `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 $c \in \mathcal{L}_{\mathcal{C}}$, $solv_{\mathcal{C}}(c) =$ `true` if and only if $\mathcal{D}_{\mathcal{C}} \models \exists c$.

A *valuation* is a function that maps all variables into $D_{\mathcal{C}}$. We write $O\sigma$ (instead of $\sigma(O)$) to denote the result of applying a valuation σ to an object O. If c is a constraint, we write $\mathcal{D}_{\mathcal{C}} \models c$ if for every valuation σ, $c\sigma$ is true in $\mathcal{D}_{\mathcal{C}}$ *i.e.* $\mathcal{D}_{\mathcal{C}} \models_{\sigma} c$. Hence, $\mathcal{D}_{\mathcal{C}} \models c$ is the same as $\mathcal{D}_{\mathcal{C}} \models \forall c$. Valuations are denoted by σ, η, θ, … in the sequel of this paper.

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

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 [16]. A constraint solver for $\mathcal{R}_{lin}$ always returning either `true` or `false` is described in [15]. $\square$

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 [1]. 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 $f_{Tree} : Tree^n \to Tree$ 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. $\square$

2.2 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 $p \in \Pi'_L$, $arity(p) = n$ and $\tilde{t}$ is a sequence of n terms in Σ_L. A CLP($\mathcal{C}$) *program* is a finite set of rules. A *rule* has the form $p(\tilde{x}) \leftarrow c \diamond q_1(\tilde{y}_1), \ldots, q_n(\tilde{y}_n)$ where p, q_1, $\ldots$, q_n are predicate symbols in Π'_L, c is a finite conjunction of primitive constraints and $\tilde{x}$, $\tilde{y}_1$, $\ldots$, $\tilde{y}_n$ are disjoint sequences of distinct variables. Hence, c is the conjunction of all constraints, including unifications. A *query* has the form $\langle Q \mid d \rangle$ where Q is a finite sequence of atoms and d is a finite conjunction of primitive constraints. When Q contains exactly one atom, the query is said to be *atomic*. The empty sequence of atoms is denoted by $\square$. The set of variables occurring in a syntactic object O is denoted $Var(O)$.

The examples of this paper make use of the language CLP($\mathcal{R}_{lin}$) and the language CLP(*Term*). Program and query examples are presented in `teletype` font. Program and query 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 *left derivations* from queries to queries. Let $\langle p(\tilde{t}), Q \mid d \rangle$ be a query and r be a rule. Let $r' := p(\tilde{x}) \leftarrow c \diamond \mathbf{B}$ be a variant of r variable disjoint with $\langle p(\tilde{t}), Q \mid d \rangle$ such that $solv_{\mathcal{C}}(\tilde{x} = \tilde{t} \wedge c \wedge d) =$ `true` (where $\tilde{x} = \tilde{t}$ denotes the constraint $x_1 = t_1 \wedge \cdots \wedge x_n = t_n$ with $\tilde{x} := x_1, \ldots, x_n$ and $\tilde{t} := t_1, \ldots, t_n$). Then,

$\langle p(\tilde{t}), Q \mid d\rangle \underset{r}{\Longrightarrow} \langle \mathbf{B}, Q \mid \tilde{x} = \tilde{t} \wedge c \wedge d\rangle$ is a *left derivation step* with r' as its *input rule*. We write $S \underset{P}{\overset{+}{\Longrightarrow}} S'$ to summarize a finite number (> 0) of left derivation steps from S to S' where each input rule is a variant of a rule of P. Let S_0 be a query. A maximal sequence $S_0 \underset{r_1}{\Longrightarrow} S_1 \underset{r_2}{\Longrightarrow} \cdots$ of left derivation steps is called a *left 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. A finite left derivation ends up either with a query of the form $\langle \Box \mid d\rangle$ with $\mathcal{T}_\mathcal{C} \models \exists d$ (then it is a *successful* left derivation) or with a query of the form $\langle Q \mid d\rangle$ with $Q \neq \Box$ or $\mathcal{T}_\mathcal{C} \models \neg\exists d$ (then it is a *failed* left derivation). We say S_0 *left loops* with respect to P if there exists an infinite left derivation of $P \cup \{S_0\}$.

2.3 The Binary Unfoldings of a CLP($\mathcal{C}$) Program

We say that $H \leftarrow c \diamond \mathbf{B}$ is a *binary rule* if $\mathbf{B}$ contains at most one atom. A *binary program* is a finite set of binary rules.

Now we present the main ideas about the *binary unfoldings* [6] of a program, borrowed from [2]. This technique transforms a program P into a possibly infinite set of binary rules. Intuitively, each generated binary rule $H \leftarrow c \diamond \mathbf{B}$ specifies that, with respect to the original program P, a call to $\langle H \mid d\rangle$ (or any of its instances) necessarily leads to a call to $\langle \mathbf{B} \mid c \wedge d\rangle$ (or its corresponding instance) if $c \wedge d$ is satisfiable.

More precisely, let S be an atomic query. Then, the atomic query $\langle A \mid d\rangle$ is a *call* in a left derivation of $P \cup \{S\}$ if $S \underset{P}{\overset{+}{\Longrightarrow}} \langle A, Q \mid d\rangle$. We denote by $calls_P(S)$ the set of calls which occur in the left derivations of $P \cup \{S\}$. The specialization of the goal independent semantics for call patterns for the left-to-right selection rule is given as the fixpoint of an operator T_P^β over the domain of binary rules, viewed modulo renaming. In the definition below, id denotes the set of all binary rules of the form $p(\tilde{x}) \leftarrow \tilde{x} = \tilde{y} \diamond p(\tilde{y})$ for any $p \in \Pi'_L$ and $\overline{\exists}_V c$ denotes the projection of a constraint c onto the set of variables V. Moreover, for atoms $A := p(\tilde{t})$ and $A' := p(\tilde{t}')$ we write $A = A'$ as an abbreviation for the constraint $\tilde{t} = \tilde{t}'$.

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

We define its powers as usual. It can be shown that the least fixpoint of this monotonic operator always exists and we set

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

Then, the calls that occur in the left derivations of $P \cup \{S\}$, with $S := \langle p(\tilde{t}) \mid d\rangle$, can be characterized as follows:

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

Similarly, $bin\_unf(P)$ gives a goal independent representation of the success patterns of P. But we can extract more information from the binary unfoldings of a program P: universal left termination of an atomic query S with respect to P is identical to universal termination of S with respect to $bin\_unf(P)$. Note that the selection rule is irrelevant for a binary program and an atomic query, as each subsequent query has at most one atom. The following result lies at the heart of Codish's approach to termination [2]:

Theorem 1 (Observing Termination). *Let P be a* CLP($\mathcal{C}$) *program and S be an atomic query. Then, S left loops w.r.t. P if and only if S loops w.r.t. $bin\_unf(P)$.*

Notice that $bin\_unf(P)$ is a possibly infinite set of binary rules. For this reason, in the algorithms of Section 5, we compute only the first *max* iterations of T_P^β where *max* is a parameter of the analysis. As an immediate consequence of Theorem 1 frequently used in our proofs, assume that we detect that S loops with respect to a subset of the binary rules of $T_P^\beta \uparrow i$, with $i \in N$. Then S loops with respect to $bin\_unf(P)$ hence S left loops with respect to P.

Example 3. Consider the CLP(*Term*) program P (see [10], p. 56–58):

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

Let c_1, c_2 and c_3 be the constraints in r_1, r_2 and r_3, respectively. The binary unfoldings of P are:

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

2.4 Terminology

In this paper, we design an algorithm that infers a finite set of left looping atomic queries from the text of any CLP($\mathcal{C}$) program P. First, the algorithm computes a finite subset of $bin\_unf(P)$ and then it proceeds with this subset only. For this reason, and to simplify the exposition, the theoretical results we describe below only deal with atomic queries and binary rules but can be easily extended to any form of queries or rules. Consequently, in the sequel of this paper up to Section 5, by a *query* we mean an *atomic query*, by a *rule*, we mean a *binary rule* and by a *program* we mean a *binary program.* Moreover, as mentioned above, the selection rule is irrelevant for a binary program and an atomic query, so we merely speak of *derivation step*, of *derivation* and of *loops.*

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 $p(\tilde{t}) \leftarrow p(\tilde{t'})$ such that $p(\tilde{t'})$ is more general than $p(\tilde{t})$, then the query $p(\tilde{t})$ loops with respect to P. In this section, we extend this result to the constraint logic programming framework. First, we generalize the relation "is more general than":

Definition 1 (More General Than). *Let* $S := \langle p(\tilde{t}) \mid d\rangle$ *and* $S' := \langle p(\tilde{t'}) \mid d'\rangle$ *be two queries. We say that* S' *is* more general than S *if* $\{p(\tilde{t})\eta \mid \mathcal{D}_{\mathcal{C}} \models_\eta d\} \subseteq \{p(\tilde{t'})\eta \mid \mathcal{D}_{\mathcal{C}} \models_\eta d'\}$.

Example 4. Suppose that $\mathcal{C} = \mathit{Term}$. Let $S := \langle \mathtt{p(X)} \mid \mathtt{X} = \mathtt{f(f(Y))}\rangle$ and $S' := \langle \mathtt{p(X)} \mid \mathtt{X} = \mathtt{f(Y)}\rangle$. Then, as $\{p(X)\eta \mid \mathcal{D}_{\mathcal{C}} \models_\eta (X = f(f(Y)))\} \subseteq \{p(X)\eta \mid \mathcal{D}_{\mathcal{C}} \models_\eta (X = f(Y))\}$, S' is more general than S. $\square$

This definition allows us to state a lifting result:

Theorem 2 (Lifting). *Consider a derivation step* $S \underset{r}{\Longrightarrow} S_1$*, a query* S' *that is more general than* S *and a variant* r' *of* r *variable disjoint with* S'*. Then, there exists a query* S'_1 *that is both more general than* S_1 *and such that* $S' \underset{r}{\Longrightarrow} S'_1$ *with input rule* r'.

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* $r := p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ *be a rule such that* $\mathcal{D}_{\mathcal{C}} \models \exists c$*. If* $\langle p(\tilde{y}) \mid c\rangle$ *is more general than* $\langle p(\tilde{x}) \mid c\rangle$ *then* $\langle p(\tilde{x}) \mid c\rangle$ *loops w.r.t.* $\{r\}$.

Corollary 2. *Let* $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ *be a rule from a program* P*. If* $\langle q(\tilde{y}) \mid c\rangle$ *loops w.r.t.* P *then* $\langle p(\tilde{x}) \mid c\rangle$ *loops w.r.t.* P.

Example 5. Consider the CLP(*Term*) program `APPEND`:

$$\begin{aligned} r_1 &:= \mathtt{append}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3) \leftarrow \mathtt{X}_1 = [\,] \wedge \mathtt{X}_2 = \mathtt{X}_3 \diamond \square \\ r_2 &:= \mathtt{append}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3) \leftarrow \mathtt{X}_1 = [\mathtt{A}|\mathtt{Y}_1] \wedge \mathtt{X}_2 = \mathtt{Y}_2 \wedge \mathtt{X}_3 = [\mathtt{A}|\mathtt{Y}_3] \diamond \\ &\qquad\qquad \mathtt{append}(\mathtt{Y}_1,\mathtt{Y}_2,\mathtt{Y}_3) \end{aligned}$$

Let c_2 be the constraint in the rule r_2. Then, $\mathcal{D}_{\mathit{Term}} \models \exists c_2$. Moreover, we note that $\langle \mathtt{append}(\mathtt{Y}_1,\mathtt{Y}_2,\mathtt{Y}_3) \mid \mathtt{c}_2\rangle$ is more general than $\langle \mathtt{append}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3) \mid \mathtt{c}_2\rangle$.

So, by Corollary 1, $\langle \mathtt{append}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3) \,|\, \mathtt{c}_2 \rangle$ loops w.r.t. $\{r_2\}$, hence w.r.t. APPEND. Hence, there exists an infinite derivation ξ of $\mathtt{APPEND} \cup \{\langle \mathtt{append}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3) \,|\, \mathtt{c}_2 \rangle\}$. Then, if S is a query that is more general than $\langle \mathtt{append}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3) \,|\, \mathtt{c}_2 \rangle$, by successively applying the Lifting Theorem 2 to each step of ξ, one can construct an infinite derivation of $\mathtt{APPEND} \cup \{S\}$. So, S also loops w.r.t. APPEND. □

An extended version of Corollary 1, presented in the next section, together with the above Corollary 2 will be used to design the algorithms of Section 5 which infer classes of looping queries from the text of a program.

4 Loop Inference Using Sets of Positions

A basic idea in our work lies in identifying arguments in rules which can be disregarded when unfolding a query. Such arguments are said to be *neutral.* The point is that in many cases, considering this kind of arguments allows to infer more looping queries.

Example 6 (Example 5 continued). The second argument of the predicate symbol *append* is neutral for derivation with the rule r_2: if we hold a derivation ξ of a query $\langle \mathit{append}(t_1, t_2, t_3) \,|\, c \rangle$ w.r.t. $\{r_2\}$, then for any term t there exists a derivation of $\{r_2\} \cup \{\langle \mathit{append}(t_1, t, t_3) \,|\, c \rangle\}$ whose length is the same as that of ξ. This means that we still get a looping query if we replace, in every looping query inferred in Example 5, the second argument of *append* by *any* term. □

In this section, we present a framework to describe specific arguments inside a program. Using this framework, we then give an operational definition of neutral arguments leading to a result extending Corollary 1 above. Finally, we relate the operational definition to an equivalent logical characterization and to a non-equivalent syntactic criterion. Hence, the results of this section extend those we presented in [13] where we defined, in the scope of logic programming, neutral arguments in a very syntactical way.

4.1 Sets of Positions

Definition 2 (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 $[1, \mathit{arity}(p)]$.*

Example 7. If we want to disregard the second argument of the predicate symbol *append* defined in Example 5, we set $\tau := \langle \mathit{append} \mapsto \{2\} \rangle$. □

Using a set of positions τ, one can *restrict* any atom by "erasing" the arguments whose position is distinguished by τ:

Definition 3 (Restriction). *Let τ be a set of positions.*

– *Let $p \in \Pi'_L$ be a predicate symbol of arity n. The* restriction of p w.r.t. τ *is the predicate symbol p_τ. Its arity equals the number of elements of $[1, n] \setminus \tau(p)$.*

– *Let $A := p(t_1, \dots, t_n)$ be an atom. The* restriction of A w.r.t. τ, *denoted by A_τ, is the atom $p_\tau(t_{i_1}, \dots, t_{i_m})$ where $\{i_1, \dots, i_m\} = [1, n] \setminus \tau(p)$ and $i_1 \leq \dots \leq i_m$.*
– *Let $S := \langle A \mid d \rangle$ be a query. The* restriction of S w.r.t. τ, *denoted by S_τ, is the query $\langle A_\tau \mid d \rangle$.*

Example 8 (Example 7 continued). The restriction of the query

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

w.r.t. τ is the query $\langle \mathtt{append}_\tau\mathtt{(X,Z)} \mid \mathtt{X} = \mathtt{[A|B]} \wedge \mathtt{Y} = \mathtt{a} \wedge \mathtt{Z} = \mathtt{[A|C]} \rangle$. $\square$

Sets of positions, together with the restriction they induce, lead to a generalization of the relation "is more general than":

Definition 4 (τ-More General). *Let τ be a set of positions and S and S' be two queries. Then, S' is* τ-more general than S *if S'_τ is more general than S_τ.*

Example 9 (Example 7 continued). Since $\tau = \langle \mathit{append} \mapsto \{2\} \rangle$, we do not care what happens to the second argument of *append*. So $\langle \mathtt{append(X,a,Z)} \mid \mathtt{true} \rangle$ is τ-more general than $\langle \mathtt{append(X,Y,Z)} \mid \mathtt{true} \rangle$ because $\{\mathit{append}_\tau(X, Z)\eta \mid \mathcal{D}_\mathcal{C} \models_\eta \mathit{true}\} \subseteq \{\mathit{append}_\tau(X, Z)\eta \mid \mathcal{D}_\mathcal{C} \models_\eta \mathit{true}\}$. $\square$

4.2 Derivation Neutral Sets of Positions

Now we give a precise operational definition of the kind of arguments we are interested in. The name "derivation neutral" stems from the fact that τ-arguments do not play any rôle in the derivation process.

Definition 5 (Derivation Neutral). *Let r be a rule and τ be a set of positions. We say that τ is* DN *for r if for each derivation step $S \underset{r}{\Longrightarrow} S_1$, for each query S' that is τ-more general than S and for each variant r' of r variable disjoint with S', there exists a query S'_1 that is τ-more general than S_1 and such that $S' \underset{r}{\Longrightarrow} S'_1$ with input rule r'. This definition is extended to programs: τ is* DN *for P if it is DN for each rule of P.*

Therefore, while lifting a derivation, we can safely ignore derivation neutral arguments which can be instantiated to any term. As a consequence, we get the following extended version of Corollary 1:

Proposition 1. *Let $r := p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ be a rule such that $\mathcal{D}_\mathcal{C} \models \exists c$. Let τ be a set of positions that is DN for r. If $\langle p(\tilde{y}) \mid c \rangle$ is τ-more general than $\langle p(\tilde{x}) \mid c \rangle$ then $\langle p(\tilde{x}) \mid c \rangle$ loops w.r.t. $\{r\}$.*

Finding out neutral arguments from the text of a program is not an easy task if we use the definition above. The next subsections present a logical and a syntactic characterization that can be used (see Section 5.2) to compute neutral arguments that appear inside a given program.

4.3 A Logical Characterization

We distinguish the following sets of variables that appear within a rule:

Definition 6. *Let $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ be a rule and τ be a set of positions.*

1. *Let $\tilde{x} := x_1, \ldots, x_h$. The set of variables of the head of r that are distinguished by τ is $vars\_head(r, \tau) := \{x_i \in \tilde{x} \mid i \in \tau(p)\}$.*
2. *The set of local variables of r is denoted by $local\_vars(r)$ and defined as: $local\_vars(r) := Var(c) \setminus (\tilde{x} \cup \tilde{y})$.*
3. *Let $\tilde{y} := y_1, \ldots, y_b$. The set of variables of the body of r that are distinguished by τ is $vars\_body(r, \tau) := \{y_i \in \tilde{y} \mid i \in \tau(q)\}$.*

Example 10 (Example 7 continued). Consider the rule

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

We have: $vars\_head(r, \tau) = \{X_2\}$, $local\_vars(r) = \{A, B\}$ and $vars\_body(r, \tau) = \{Y_2\}$. □

Now we give a logical definition of derivation neutrality. As we will see below, this definition is equivalent to the operational one we stated above.

Definition 7 (Logical Derivation Neutral). *Let $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ be a rule and τ be a set of positions. We say that τ is* DNlog *for r if $\mathcal{D}_{\mathcal{C}} \models (c \rightarrow \forall_{\mathcal{X}} \exists_{\mathcal{Y}} c)$ where $\mathcal{X} = vars\_head(r, \tau)$ and $\mathcal{Y} = local\_vars(r) \cup vars\_body(r, \tau)$.*

So, τ is DNlog for r if for any valuation σ such that $\mathcal{D}_{\mathcal{C}} \models_\sigma c$, if one changes the value of $x\sigma$ where $x \in vars\_head(r, \tau)$ into *any* value, then there exists a corresponding value for each $y\sigma$, where y is in $local\_vars(r)$ or in $vars\_body(r, \tau)$, such that c still holds.

Example 11 (Example 10 continued). The set of positions τ is DNlog for the rule r because $\mathcal{X} = \{X_2\}$, $\mathcal{Y} = \{A, B, Y_2\}$, c is the constraint

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

and for every valuation σ, if $\mathcal{D}_{\mathcal{C}} \models_\sigma c$ then $\mathcal{D}_{\mathcal{C}} \models_\sigma \forall X_2\ \exists B\ \exists Y_2\ c$ hence $\mathcal{D}_{\mathcal{C}} \models_\sigma \forall_{\mathcal{X}} \exists_{\mathcal{Y}} c$. □

Theorem 3. *Let r be a rule and τ be a set of positions. Then, τ is DNlog for r if and only if τ is DN for r.*

Example 12. Consider the rule $r := p(X) \leftarrow Y = f(X) \diamond p(Y)$. Let $\tau := \langle p \mapsto \{1\} \rangle$. We have $\mathcal{X} = \{X\}$ and $\mathcal{Y} = \{Y\}$. As the formula $\forall X\ \exists Y\ Y = f(X)$ is true in $Term$, so is $\forall X, Y[Y = f(X) \rightarrow \forall X\ \exists Y\ Y = f(X)]$. Hence τ is DN for the rule r. □

4.4 A Syntactic Characterization

In [13], 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 8 (Flat Rule). *A rule* $r := p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ *is said to be* flat *if* c *has the form* $(\tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t})$ *for some sequences of terms* $\tilde{s}$ *and* $\tilde{t}$ *such that* $Var(\tilde{s}, \tilde{t}) \subseteq local\_vars(r)$.

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 $\mathcal{D}_{\mathcal{C}} \models \left[(\exists_{local\_vars(r)} c) \leftrightarrow (\exists_{local\_vars(r')} c')\right]$ (take for instance $r := \mathtt{p(X)} \leftarrow \mathtt{X} > \mathtt{0} \diamond \mathtt{p(Y)}$ in $\mathcal{R}_{lin}$.)

Next, we consider universal terms:

Definition 9 (Universal Term). *A term* t *in* $\Sigma_{\mathcal{C}}$ *is said to be* universal *if for a variable* x *not occurring in* t *we have:* $\mathcal{D}_{\mathcal{C}} \models \forall_x \exists_{Var(t)} (x = t)$.

Hence, a term t is universal if it can take any value in $D_{\mathcal{C}}$ *i.e.* if for any value a in $D_{\mathcal{C}}$, there exists a valuation σ such that $\mathcal{D}_{\mathcal{C}} \models (a = t\sigma)$.

Example 13. A term t in Σ_{Term} is universal if and only if t is a variable. If x is a variable, then x, $x + 0$, $x + 1 + (-1)$, ... and $x + 1$ or $2 * x$ are universal terms in $\Sigma_{\mathcal{R}_{lin}}$. □

Now, we can define syntactic derivaration neutrality:

Definition 10 (Syntactic Derivation Neutral). *Consider a flat rule* $r := p(\tilde{x}) \leftarrow (\tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t}) \diamond q(\tilde{y})$ *with* $\tilde{s} := s_1, \dots, s_h$ *and* $\tilde{t} := t_1, \dots, t_b$ *(*h *and* b *are the arity of* p *and* q *respectively). Let* τ *be a set of positions. We say that* τ *is* DNsyn *for* r *if:*

$$\forall i \in \tau(p), \begin{cases} \textbf{(C1)}\ s_i \textit{ is a universal term and} \\ \textbf{(C2)}\ \forall j \in [1,h] \setminus \{i\},\ Var(s_i) \cap Var(s_j) = \varnothing \textit{ and} \\ \textbf{(C3)}\ \forall j \in [1,b] \setminus \tau(q),\ Var(s_i) \cap Var(t_j) = \varnothing \end{cases}$$

Example 14. The rule $\mathtt{p(X_1)} \leftarrow \mathtt{X_1} = \mathtt{Z} \wedge \mathtt{Y_1} = \mathtt{Z} \diamond \mathtt{p(Y_1)}$ is flat and the set of positions $\langle p \mapsto \{1\} \rangle$ is DNsyn for it. The rule $\mathtt{p(X)} \leftarrow \mathtt{X} > \mathtt{0} \diamond \mathtt{p(Y)}$ has no DNsyn set of positions in $\mathcal{R}_{lin}$. □

Proposition 2. *Let* r *be a flat rule and* τ *be a set of positions. If* τ *is DNsyn for* r *then* τ *is DN for* r. *If* τ *is DN for* r *then* **(C1)** *of Definition 10 holds.*

Notice that a DN set of positions is not necessarily DNsyn because **(C2)** or **(C3)** of Definition 10 may not hold:

Example 15. Let $\mathcal{C} := \mathcal{R}_{lin}$.

– Let $r_1 := \mathtt{p_1(X_1, X_2)} \leftarrow \mathtt{X_1 = A} \wedge \mathtt{X_2 = A + B} \diamond \square$. The set of positions $\tau_1 := \langle p_1 \mapsto \{1, 2\}\rangle$ is DNlog for r_1, so τ_1 is DN for r_1. But τ_1 is not DNsyn for r_1 because, as the terms A and $A + B$ share the variable A, **(C2)** does not hold.
– Let $r_2 := \mathtt{p_2(X_1, X_2)} \leftarrow \mathtt{X_1 = A} \wedge \mathtt{X_2 = 0} \wedge \mathtt{Y_2 = A - A} \diamond \mathtt{p_2(Y_1, Y_2)}$. The set of positions $\tau_2 := \langle p_2 \mapsto \{1\}\rangle$ is DNlog for r_2, so τ_2 is DN for r_2. But τ_2 is not DNsyn for r_2 because, as the terms A and $A - A$ share the variable A, **(C3)** does not hold. □

In the special case of logic programming, we have an equivalence:

Theorem 4 (Logic Programming). *Suppose that $\mathcal{C} = Term$. Let r be a flat rule and τ be a set of positions. Then, τ is DNsyn for r if and only if τ is DN for r*

Every rule $p(\tilde{s}) \leftarrow q(\tilde{t})$ in logic programming can be easily translated to a rule $p(\tilde{x}) \leftarrow (\tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t}) \diamond q(\tilde{y})$ in flat form. As the only universal terms in Σ_{Term} are the variables, Definition 10 is equivalent to that we gave in [13] for Derivation Neutral. Therefore, Theorem 4 states that in the case of logic programming, we have a form of completeness because we cannot get a better syntactic criterion than that of [13] (by "better", we mean a criterion allowing to distinguish at least the same positions).

5 Algorithms

In this section, we describe a set of correct algorithms that allow to infer classes of left looping atomic queries from the text of a (non necessary binary) given program P. Using the operator T_P^β, our technique first computes a finite subset of $bin\_unf(P)$ which is then analysed using DN sets of positions and a data structure called *loop dictionary*.

5.1 Loop Dictionaries

Definition 11 (Looping Pair, Loop Dictionary). *A* looping pair *has the form $(BinSeq, \tau)$ where $BinSeq$ is a finite ordered sequence of binary rules, τ is a set of positions that is DN for $BinSeq$ and*

– *either $BinSeq = [p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})]$ where $\mathcal{D}_\mathcal{C} \models \exists c$ and $\langle p(\tilde{y}) \,|\, c\rangle$ is τ-more general than $\langle p(\tilde{x}) \,|\, c\rangle$*
– *or $BinSeq = [p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1) | BinSeq']$ and there exists a set of positions τ' which is such that $([p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1) | BinSeq'], \tau')$ is a looping pair and $\langle q(\tilde{y}) \,|\, c\rangle$ is τ'-more general than $\langle p_1(\tilde{x}_1) \,|\, c_1\rangle$.*

A loop dictionary *is a finite set of looping pairs.*

Example 16. In the constraint domain $\mathcal{R}_{lin}$, the pair $(BinSeq, \tau)$ where $BinSeq := [\mathtt{p(X)} \leftarrow \mathtt{X > 0} \wedge \mathtt{X = Y} \diamond \mathtt{q(Y)}, \mathtt{q(X)} \leftarrow \mathtt{Y = 2 * X} \diamond \mathtt{q(Y)}]$ and τ is the set of positions $\langle p \mapsto \varnothing, q \mapsto \{1\}\rangle$, is a looping one because:

– as τ is DNlog for *BinSeq*, by Theorem 3 it is DN for *BinSeq*,
– $([\mathtt{q(X)} \leftarrow \mathtt{Y} = \mathtt{2} * \mathtt{X} \diamond \mathtt{q(Y)}], \tau')$, where $\tau' := \langle q \mapsto \{1\}\rangle$, is a looping pair because τ' is DN for $[\mathtt{q(X)} \leftarrow \mathtt{Y} = \mathtt{2} * \mathtt{X} \diamond \mathtt{q(Y)}]$ (because it is DNlog for that program), $\mathcal{D}_{\mathcal{R}_{lin}} \models \exists(Y = 2 * X)$ and $\langle \mathtt{q(Y)} \mid \mathtt{Y} = \mathtt{2} * \mathtt{X}\rangle$ is τ'-more general than $\langle \mathtt{q(X)} \mid \mathtt{Y} = \mathtt{2} * \mathtt{X}\rangle$,
– $\langle \mathtt{q(Y)} \mid \mathtt{X} > \mathtt{0} \wedge \mathtt{X} = \mathtt{Y}\rangle$ is τ'-more general than $\langle \mathtt{q(X)} \mid \mathtt{Y} = \mathtt{2} * \mathtt{X}\rangle$. □

One motivation for introducing this definition is that a looping pair immediately provides a looping atomic query:

Proposition 3. *Let $([p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}) | BinSeq], \tau)$ be a looping pair. Then, $\langle p(\tilde{x}) \mid c\rangle$ loops w.r.t. $[p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}) | BinSeq]$.*

Proof. By induction on the length of *BinSeq*, using Proposition 1 and Corollary 2. □

A second motivation for using loop dictionaries is that they can be built incrementally by simple algorithms as those described below.

5.2 Getting a Loop Dictionary from a Binary Program

The most simple form of a looping pair is $([p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})], \tau)$ where τ is a set of positions that is DN for $[p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})]$, where $\mathcal{D}_{\mathcal{C}} \models \exists c$ and $\langle p(\tilde{y}) \mid c\rangle$ is τ-more general than $\langle p(\tilde{x}) \mid c\rangle$. So, given a binary rule $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ such that $\mathcal{D}_{\mathcal{C}} \models \exists c$, if we hold a set of positions τ that is DN for $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$, it suffices to test if $\langle p(\tilde{y}) \mid c\rangle$ is τ-more general than $\langle p(\tilde{x}) \mid c\rangle$. If so, we have a looping pair $([p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})], \tau)$. This is how the following function works.

```
unit_loop($p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$, $Dict$):
   in:  $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$: a binary rule
        $Dict$: a loop dictionary
  out:  $Dict'$: a loop dictionary
   1:  $Dict' := Dict$
   2:  if $\mathcal{D}_{\mathcal{C}} \models \exists c$ then
   3:      $\tau :=$ a DN set of positions for $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$
   4:      if $\langle p(\tilde{y}) \mid c\rangle$ is $\tau$-more general than $\langle p(\tilde{x}) \mid c\rangle$ then
   5:          $Dict' := Dict' \cup \{([p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})], \tau)\}$
   6:  return $Dict'$
```

Termination of `unit_loop` is straightforward, provided that at line 3 we use a terminating algorithm to compute τ. Partial correctness is deduced from the following theorem.

Theorem 5 (Partial Correctness of `unit_loop`). *If $p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})$ is a binary rule and Dict a loop dictionary, then* `unit_loop`*$(p(\tilde{x}) \leftarrow c \diamond p(\tilde{y}), Dict)$ is a loop dictionary, every element $(BinSeq, \tau)$ of which is such that $(BinSeq, \tau) \in Dict$ or $BinSeq = [p(\tilde{x}) \leftarrow c \diamond p(\tilde{y})]$.*

Now suppose we hold a loop dictionary *Dict* and a rule $p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$. Then we may get some more looping pairs: it suffices to take the elements $\big([p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1)|BinSeq'], \tau'\big)$ of *Dict* such that $\langle q(\tilde{y}) \,|\, c\rangle$ is τ'-more general than $\langle p_1(\tilde{x}_1) \,|\, c_1\rangle$ and to compute a set of positions τ that is DN for $[p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1)|BinSeq']$. Then $([p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1)|BinSeq'], \tau)$ is a looping pair. The following function works this way.

```
loops_from_dict($p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$, $Dict$):
 in:  $p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$: a binary rule
      $Dict$: a loop dictionary
out:  $Dict'$: a loop dictionary
  1:  $Dict' := Dict$
  2:  for each $\big([p_1(\tilde{x}_1) \leftarrow c_1 \diamond q_1(\tilde{y}_1)|BinSeq'], \tau'\big) \in Dict$ do
  3:    if $\langle q(\tilde{y}) \,|\, c\rangle$ is $\tau'$-more general than $\langle p_1(\tilde{x}_1) \,|\, c_1\rangle$ 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:      $\tau :=$ a DN set of positions for $BinSeq$
  6:      $Dict' := Dict' \cup \{(BinSeq, \tau)\}$
  7:  return $Dict'$
```

Termination of `loops_from_dict` follows from finiteness of *Dict* (because *Dict* is a loop dictionary), provided that we use a terminating algorithm to compute τ at line 5. Partial correctness follows from the result below.

Theorem 6 (Partial Correctness of `loops_from_dict`). *Suppose that* $p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})$ *is a binary rule and Dict is a loop dictionary. Then,* `loops_from_dict`$(p(\tilde{x}) \leftarrow c \diamond q(\tilde{y}), Dict)$ *is a loop dictionary, every element* $(BinSeq, \tau)$ *of which is such that* $(BinSeq, \tau) \in Dict$ *or* $BinSeq = [p(\tilde{x}) \leftarrow c \diamond q(\tilde{y})|BinSeq']$ *for some* $(BinSeq', \tau')$ *in Dict.*

Finally, here is the top-level function for inferring loop dictionaries from a finite set of binary rules.

```
infer_loop_dict($BinProg$):
 in:  $BinProg$: a finite set of binary rules
out:  a loop dictionary
  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 $Dict$
```

Theorem 7 (Correctness of `infer_loop_dict`). *Let BinProg be a finite set of binary rules. Then,* `infer_loop_dict`$(BinProg)$ *terminates and returns a loop dictionary, every element* $(BinSeq, \tau)$ *of which is such that* $BinSeq \subseteq BinProg$.

Proof. By Theorem 5 and Theorem 6. □

5.3 Inferring Looping Conditions

Finally, we present an algorithm which infers classes of left looping atomic queries from the text of a given program. The classes we consider are defined by a pair (S, τ) which finitely denotes the possibly infinite set $[S]^\tau$:

Definition 12. *Let S be an atomic query and τ be a set of positions. Then $[S]^\tau$ denotes the class of atomic queries defined as:*

$$[S]^\tau \overset{def}{=} \{S' : \textit{ an atomic query} \mid S' \textit{ is } \tau\textit{-more general than } S\} .$$

Once each element of $[S]^\tau$ left loops w.r.t. a CLP($\mathcal{C}$) program, we get a *looping condition* for that program:

Definition 13 (Looping Condition). *Let P be a* CLP($\mathcal{C}$) *program.* A looping condition for *P is a pair (S, τ) such that each element of $[S]^\tau$ left loops w.r.t. P.*

Looping conditions can be easily infered from a loop dictionary. It suffices to consider the property of looping pairs stated by Proposition 3. The following function computes a finite set of looping conditions for any given CLP($\mathcal{C}$) program.

```
infer_loop_cond($P$, $max$):
   in:  $P$: a CLP($\mathcal{C}$) program
        $max$: a non-negative integer
  out:  a finite set of looping conditions 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:       $L := L \cup \{(\langle p(\tilde{x}) \mid c \rangle, \tau)\}$
   5:   return $L$
```

A call to `infer_loop_cond`(P, max) terminates for any program P and any non-negative integer max because, as $T_P^\beta \uparrow max$ is finite, at line 2 the call to `infer_loop_dict` terminates and the loop at line 3 has a finite number of iterations (because, by correctness of `infer_loop_dict`, *Dict* is finite.) From some preliminary experiments we made over 50 logic programs, we find that the maximum value for *max* is 4. Partial correctness of `infer_loop_cond` follows from the next theorem.

Theorem 8 (Partial Correctness of `infer_loop_cond`). *If P is a program and max a non-negative integer, then* `infer_loop_cond`(P, max) *is a finite set of looping conditions for P.*

Proof. By Proposition 3, Theorem 7 and the Observing Termination Theorem 1. □

We point out that correctness of `infer_loop_cond` is independent of whether the predicate symbols are analysed according to a topological sort of the strongly connected components of the call graph of P. However, inference of looping classes is much more efficient if predicate symbols are processed bottom-up. Precision issues could be dealt with by comparing non-termination analysis with termination analysis, as in [13].

Example 17. Consider the CLP($\mathcal{R}_{lin}$) program $\mathtt{SUM}$:

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

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

The set $T^{\beta}_{\mathtt{SUM}} \uparrow 1$ includes:

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

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

A call to `unit_loop`$(br_2, \varnothing)$ returns $Dict_1 := \{([br_2], \tau_2)\}$ where $\tau_2 = \langle pow2 \mapsto \{2,3\}\rangle$. A call to `loops_from_dict`$(br_1, Dict_1)$ returns $Dict_1 \cup \{([br_1, br_2], \tau_1)\}$ where $\tau_1 = \langle sum \mapsto \{2\}, pow2 \mapsto \{2,3\}\rangle$. Hence, a call to `infer_loop_cond(SUM`, 1) returns the looping conditions $(\langle \mathtt{sum}(\mathtt{X}_1,\mathtt{X}_2) \mid c_1\rangle, \tau_1)$ and $(\langle \mathtt{pow2}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3) \mid c_2\rangle, \tau_2)$ where c_1 and c_2 are the constraints of br_1 and br_2 respectively. □

6 Conclusion

We have proposed a self contained framework for non-termination analysis of constraint logic programs. As usual [9], we were able to give simpler definitions and proofs than in the logic programming setting. Also, starting from an operational definition of *derivation neutrality*, we have given a new *equivalent* logical definition. Then, by reexamining the syntactic criterion of derivation neutrality that we proposed in [13], we have proved that this syntactic criterion can be considered as a correct and complete implementation of derivation neutrality.

Acknowledgements

We thank Mike Codish for some useful comments on a previous version of this paper.

References

1. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.
2. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
3. 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.
4. D. De Schreye and S. Decorte. Termination of logic programs : the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.
5. 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.
6. 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.
7. S. Genaim and M. Codish. Inferring termination condition for logic programs using backwards analysis. In *Proceedings of Logic for Programming, Artificial intelligence and Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2001.
8. 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.
9. 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.
10. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
11. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In M. J. Maher, editor, *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, pages 7–21. MIT Press, 1996.
12. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In P. Cousot, editor, *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 93–110. Springer-Verlag, Berlin, 2001.
13. 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.
14. F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. *ACM Transactions on Computational Logic*, pages 207–259, 2003.
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. J. Shoenfield. *Mathematical Logic.* Addison Wesley, Reading, 1967.