Non-Termination Inference of Logic Programs

ETIENNE PAYET and FRED MESNARD
IREMIA, Université de La Réunion

We present a static analysis technique for non-termination inference of logic programs. Our framework relies on an extension of the subsumption test, where some specific argument positions can be instantiated while others are generalized. We give syntactic criteria to statically identify such argument positions from the text of a program. Atomic left looping queries are generated bottom-up from selected subsets of the binary unfoldings of the program of interest. We propose a set of correct algorithms for automating the approach. Then, non-termination inference is tailored to attempt proofs of optimality of left termination conditions computed by a termination inference tool. An experimental evaluation is reported and the analysers can be tried online at `http://www.univ-reunion.fr/~gcc`. When termination and non-termination analysis produce complementary results for a logic procedure, then with respect to the leftmost selection rule and the language used to describe sets of atomic queries, each analysis is optimal and together, they induce a *characterization* of the operational behavior of the logic procedure.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Validation

General Terms: Languages, Verification

Additional Key Words and Phrases: Logic programming, static analysis, non-termination analysis, optimal termination condition

1. INTRODUCTION

Since the work of N. Lindenstrauss on TermiLog [Lindenstrauss 1997; Dershowitz et al. 2001], several automatic tools for termination checking (e.g. TALP [Arts and Zantema 1996]) or termination inference (e.g. cTI [Mesnard and Neumerkel 2000; 2001] or TerminWeb [Genaim and Codish 2001]) are now available to the logic programmer. As the halting problem is undecidable for logic programs, such analyzers compute sufficient termination conditions implying left termination. In most works, only universal left termination is considered and termination conditions rely on a language for describing classes of atomic queries. The search tree associated to *any* (concrete) query satisfying a termination condition is guaranteed to be finite. When terms are abstracted using the *term-size* norm, the termination conditions are (disjunctions of) conjunctions of conditions of the form "the i-th argument is ground". Let us call this language $\mathcal{L}_{term}$.

In this report, we present the first approach to non-termination inference tailored to attempt proofs of *optimality* of termination conditions at *verification time* for pure logic programs. The aim is to ensure the existence, for each class of atomic

Author's address: Etienne Payet, IREMIA, Université de La Réunion, 15 avenue René Cassin, BP7151, 97715 Saint Denis messageries CEDEX 9, France; email: epayet@univ-reunion.fr.
This article is based on an earlier conference paper [Mesnard et al. 2002].

queries not covered by a termination condition, of *one* query from this class which leads to an infinite search tree when such a query is proved using any standard Prolog engine. We shall first present an analysis which computes classes of left looping queries, where any atomic query from such a class is guaranteed to lead to at least one infinite derivation under the usual left-to-right selection rule. Intuitively, we begin by computing looping queries from recursive binary clauses of the form $p(\ldots) \leftarrow p(\ldots)$. Then we try to add binary clauses of the form $q(\ldots) \leftarrow p(\ldots)$ to increase the set of looping queries. Finally by combining the result of non-termination inference with termination inference, for each predicate, we compute the set of modes for which the overall verification system has no information.

The main contributions of this work are:

—A new application of binary unfoldings to left loop inference. [Gabbrielli and Giacobazzi 1994] introduced the binary unfoldings of a logic program P as a goal independent technique to transform P into a possibly infinite set of binary clauses, which preserves the termination property [Codish and Taboch 1999] while abstracting the standard operational semantics. We present a correct algorithm to *construct* left looping classes of atomic goals, where such classes are computed bottom-up from selected subsets of the binary unfoldings of the analyzed program.

—A correct algorithm which, when combined with termination inference [Mesnard 1996], may detect *optimal* left termination conditions expressed in $\mathcal{L}_{term}$ for logic programs. When termination and non-termination analysis produce complementary results for a logic procedure, then with respect to the leftmost selection rule and the language used to describe sets of atomic queries, each analysis is optimal and together, they induce a *characterization* of the operational behavior of the logic procedure.

—A report on the experimental evaluation we conduct. We have fully implemented termination and non-termination inference for logic programs, which can be tried online at `http://www.univ-reunion.fr/~gcc`. We have run the couple of analyzers on a set of classical logic programs, the sizes of which range from 2 to 177 clauses. The results of this experiment should help the reader to appreciate the value of the approach.

We organize the paper as follows: Section 2 presents the notations. In Section 3 we study loop inference for binary programs. We offer a full set of correct algorithms for non-termination inference in Section 4 and optimality proofs of termination conditions in Section 5. Finally, in Section 6, we discuss related works. The detailed proofs of the results we present here can be found in the long version of this paper which is available as a CoRR[1] archive.

2. PRELIMINARIES

2.1 Functions

Let E and F be two sets. Then, $f : E \to F$ denotes that f is a partial function from E to F and $f : E \rightarrowtail F$ denotes that f is a function from E to F. The

[1]http://arxiv.org/archive/cs/intro.html – Paper ID is `cs.PL/0406041` (submitted to CoRR on 22 June 2004.)

domain of a partial function f from E to F is denoted by $Dom(f)$ and is defined as: $Dom(f) = \{e \mid e \in E,\ f(e) \text{ exists}\}$. Thus, if f is a function from E to F, then $Dom(f) = E$. Finally, if $f : E \to F$ is a partial function and E' is a set, then $f|E'$ is the function from $Dom(f) \cap E'$ to F such that for each $e \in Dom(f) \cap E'$, $f|E'$ maps e to $f(e)$.

2.2 Logic Programming

We strictly adhere to the notations, definitions, and results presented in [Apt 1997].

N denotes the set of non-negative integers and for any $n \in N$, $[1, n]$ denotes the set $\{1, \ldots, n\}$. If $n = 0$ then $[1, n] = \varnothing$.

From now on, we fix a language $\mathcal{L}$ of programs. We assume that $\mathcal{L}$ contains an infinite number of constant symbols. $TU_{\mathcal{L}}$ denotes the set of all (ground and non ground) terms of $\mathcal{L}$. The set of relation symbols of $\mathcal{L}$ is Π and we assume that each relation symbol p has a *unique* arity, denoted $arity(p)$. An *atom* is a construct of the form $p(t_1, \ldots, t_n)$ where p is an n-ary relation symbol and $t_1, \ldots, t_n$ are terms. $TB_{\mathcal{L}}$ denotes the set of all (ground and non ground) atoms of $\mathcal{L}$. A *query* is a finite sequence of atoms $A_1, \ldots, A_n$ (where $n \geq 0$). When $n = 1$, we say that the query is *atomic*. A *clause* is a construct a the form $H \leftarrow \mathbf{B}$ where H is an atom and $\mathbf{B}$ is a query; H is called its *head* and $\mathbf{B}$ its *body*. Throughout this article, the variables of $\mathcal{L}$ are denoted by $X, Y, Z, \ldots$, the constant symbols by $a, b, \ldots$, the function symbols by $f, g, h, \ldots$, the relation symbols by $p, q, r, \ldots$, the atoms by $A, B, \ldots$, the queries by Q, Q', ... or by $\mathbf{A}$, $\mathbf{B}$, ... and the clauses by c, c', ...

Let t be a term. Then $Var(t)$ denotes the set of variables occurring in t. This notation is extended to atoms, queries and clauses. Let $\theta := \{X_1/t_1, \ldots, X_n/t_n\}$ be a substitution. We denote by $Dom(\theta)$ the set of variables $\{X_1, \ldots, X_n\}$ and by $Ran(\theta)$ the set of variables appearing in $t_1, \ldots, t_n$. We define $Var(\theta) = Dom(\theta) \cup Ran(\theta)$. Given a set of variables V, $\theta|V$ denotes the substitution obtained from θ by restricting its domain to V.

Let t be a term and θ be a substitution. Then, the term $t\theta$ is called an *instance* of t. If θ is a *renaming* (*i.e.* a substitution that is a 1-1 and onto mapping from its domain to itself), then $t\theta$ is called a *variant* of t. Finally, t is called *more general than* t' if t' is an instance of t.

A *logic program* is a finite set of clauses. In program examples, we use the ISO-Prolog syntax. Let P be a logic program. Then Π_P denotes the set of relation symbols appearing in P. In this paper, we only focus on left derivations *i.e.* we only consider the leftmost selection rule. Consider a non-empty query $B, \mathbf{C}$ and a clause c. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint with $B, \mathbf{C}$ and assume that B and H unify. Let θ be an mgu of B and H. Then $B, \mathbf{C} \underset{c}{\overset{\theta}{\Longrightarrow}} (\mathbf{B}, \mathbf{C})\theta$ is a *left derivation step* with $H \leftarrow \mathbf{B}$ as its *input clause*. If the substitution θ or the clause c is irrelevant, we drop a reference to it.

Let Q_0 be a query. A maximal sequence $Q_0 \underset{c_1}{\overset{\theta_1}{\Longrightarrow}} Q_1 \underset{c_2}{\overset{\theta_2}{\Longrightarrow}} \cdots$ of left derivation steps is called a *left derivation* of $P \cup \{Q_0\}$ if c_1, c_2, ... are clauses of P and if the *standardization apart* condition holds, *i.e.* each input clause used is variable disjoint from the initial query Q_0 and from the mgu's and input clauses used at earlier steps. A finite left derivation may end up either with the empty query (then it is a *successful* left derivation) or with a non-empty query (then it is a *failed* left

derivation). We say Q_0 *left loops* with respect to (w.r.t.) P if there exists an infinite left derivation of $P \cup \{Q_0\}$. We write $Q \overset{+}{\underset{P}{\Longrightarrow}} Q'$ if there exists a finite non-empty prefix ending at Q' of a left derivation of $P \cup \{Q\}$.

2.3 The Binary Unfoldings of a Logic Program

Let us present the main ideas about the *binary unfoldings* [Gabbrielli and Giacobazzi 1994] of a logic program, borrowed from [Codish and Taboch 1999]. This technique transforms a logic program P into a possibly infinite set of binary clauses. Intuitively, each generated *binary clause* $H \leftarrow B$ (where B is either an atom or the atom *true* which denotes the empty query) specifies that, with respect to the original program P, a call to H (or any of its instances) necessarily leads to a call to B (or its corresponding instance).

More precisely, let Q be an atomic query. Then A is a *call* in a left derivation of $P \cup \{Q\}$ if $Q \overset{+}{\underset{P}{\Longrightarrow}} A, \mathbf{B}$. We denote by $calls_P(Q)$ the set of calls which occur in the left derivations of $P \cup \{Q\}$. 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 clauses, viewed modulo renaming. In the definition below, *id* denotes the set of all binary clauses of the form $true \leftarrow true$ or $p(X_1, \ldots, X_n) \leftarrow p(X_1, \ldots, X_n)$ for any $p \in \Pi_P$, where $arity(p) = n$.

$$T_P^\beta(X) = \left\{ (H \leftarrow B)\theta \;\middle|\; \begin{array}{l} c := H \leftarrow B_1, \ldots, B_m \in P,\ i \in [1, m], \\ \langle H_j \leftarrow true \rangle_{j=1}^{i-1} \in X \text{ renamed with fresh variables}, \\ H_i \leftarrow B \in X \cup id \text{ renamed with fresh variables}, \\ i < m \Rightarrow B \neq true \\ \theta = mgu(\langle B_1, \ldots, B_i \rangle, \langle H_1, \ldots, H_i \rangle) \end{array} \right\}$$

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 $P \cup \{Q\}$ can be characterized as follows: $calls_P(Q) = \{B\theta | H \leftarrow B \in bin\_unf(P), \theta = mgu(Q, H)\}$. This last property was one of the main initial motivations of the proposed abstract semantics, enabling logic programs optimizations. 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 Q with respect to P is identical to universal termination of Q 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:

THEOREM 2.1. *[Codish and Taboch 1999] Let P be a program and Q an atomic query. Then Q left loops with respect to P iff Q loops with respect to $bin\_unf(P)$.*

Notice that $bin\_unf(P)$ is a possibly infinite set of binary clauses. For this reason, in the algorithms of Section 4, 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 2.1, assume that we detect that Q loops with respect to a subset of the binary clauses

of $T_P^\beta \uparrow i$, with $i \in N$. Then Q loops with respect to $bin\_unf(P)$ hence Q left loops with respect to P.

Example 2.2. Consider the following program P (see [Lloyd 1987], p. 56–58):

```
p(X,Z) :- p(Y,Z),q(X,Y).     p(X,X).     q(a,b).
```

The binary unfoldings of P are:

$$
\begin{aligned}
T_P^\beta \uparrow 0 &= \varnothing \\
T_P^\beta \uparrow 1 &= \{p(X,Z) \leftarrow p(Y,Z), p(X,X) \leftarrow true, q(a,b) \leftarrow true\} \cup T_P^\beta \uparrow 0 \\
T_P^\beta \uparrow 2 &= \{p(a,b) \leftarrow true, p(X,Y) \leftarrow q(X,Y)\} \cup T_P^\beta \uparrow 1 \\
T_P^\beta \uparrow 3 &= \{p(X,b) \leftarrow q(X,a), p(X,Z) \leftarrow q(Y,Z)\} \cup T_P^\beta \uparrow 2 \\
T_P^\beta \uparrow 4 &= \{p(X,b) \leftarrow q(Y,a)\} \cup T_P^\beta \uparrow 3 \\
T_P^\beta \uparrow 5 &= T_P^\beta \uparrow 4 = bin\_unf(P)
\end{aligned}
$$

Let $Q := p(X,b)$. Note that Q loops w.r.t. $T_P^\beta \uparrow 1$, hence it loops w.r.t. $bin\_unf(P)$. So Q left loops w.r.t. P. □

3. LOOP INFERENCE USING FILTERS

In this paper, we propose a mechanism that, given a logic program P, generates *at verification time* classes of atomic queries that left loop w.r.t. P. Our approach is completely based on the binary unfoldings of P and relies on Theorem 2.1. It consists in computing a finite subset *BinProg* of $bin\_unf(P)$ and then in inferring a set of atomic queries that loop w.r.t. *BinProg*. By Theorem 2.1, these queries left loop w.r.t. P.

Hence, we reduce the problem of inferring looping atomic queries w.r.t. a logic program to that of inferring looping atomic queries w.r.t. a binary program. This is why in the sequel, our definitions, results and discussions mainly concentrate on binary programs only.

The central point of our method is the subsumption test, as the following lifting lemma, specialized for the leftmost selection rule, holds:

LEMMA 3.1. *(One Step Lifting, [Apt 1997]) Let $Q \underset{c}{\Longrightarrow} Q_1$ be a left derivation step, Q' be a query that is more general than Q and c' be a variant of c variable disjoint with Q'. Then, there exists a query Q'_1 that is more general than Q_1 and such that $Q' \underset{c}{\Longrightarrow} Q'_1$ with input clause c'.*

From this result, we derive:

COROLLARY 3.2. *Let $c := H \leftarrow B$ be a binary clause. If B is more general than H then H loops w.r.t. $\{c\}$.*

COROLLARY 3.3. *Let $c := H \leftarrow B$ be a clause from a binary program BinProg. If B loops w.r.t. BinProg then H loops w.r.t. BinProg.*

These corollaries provide two sufficient conditions that can be used to design an incremental bottom-up mechanism that infers looping atomic queries. Given a binary program *BinProg*, it suffices to build the set $\mathcal{Q}$ of atomic queries consisting of the heads of the clauses whose body is more general than the head. By Corollary 3.2, the elements of $\mathcal{Q}$ loop w.r.t. *BinProg*. Then, by Corollary 3.3, the head of the

clauses whose body is more general than an element of $\mathcal{Q}$ can safely been added to $\mathcal{Q}$ while retaining the property that every query in $\mathcal{Q}$ loops w.r.t. *BinProg*.

Notice that using this technique, we may not detect some looping queries. In [Devienne et al. 1993], the authors show that there is no algorithm that, when given a right-linear binary recursive clause (*i.e.* a binary clause $p(\cdots) \leftarrow p(\cdots)$ such that all variables occur at most once in the body) and given an atomic query, always decides in a finite number of steps whether or not the resolution stops. In the case of a linear atomic query (*i.e.* an atomic query such that all variables occur at most once) however, the halting problem of derivations w.r.t. one binary clause is decidable [Schmidt-Schauss 1988; Devienne 1988; 1990].

It can be argued that the condition provided by Corollary 3.2 is rather weak because it fails at inferring looping queries in some simple cases. This is illustrated by the following example.

Example 3.4. Let c be the clause $p(X) \leftarrow p(f(X))$. We have the infinite derivation: $p(X) \underset{c}{\Longrightarrow} p(f(X)) \underset{c}{\Longrightarrow} p(f(f(X))) \underset{c}{\Longrightarrow} p(f(f(f(X)))) \cdots$ But, since the body of c is not more general than its head, Corollary 3.2 does not allow to infer that $p(X)$ loops w.r.t. $\{c\}$. □

In this section, we distinguish a special kind of argument positions that are "neutral" for derivation. Our goal is to extend the relation "is more general than" by, roughly, disregarding the predicate arguments whose position has been identified as neutral. Doing so, we aim at inferring more looping queries.

Intuitively, a set of predicate argument positions Δ is "Derivation Neutral" (DN for short) for a binary clause c when the following holds. Let Q be an atomic query and Q' be a query obtained by replacing by any terms the predicate arguments in Q whose position is in Δ. If $Q \underset{c}{\Longrightarrow} Q_1$ then $Q' \underset{c}{\Longrightarrow} Q'_1$ where Q'_1 is more general than Q_1 up to the arguments whose position is in Δ.

Example 3.5. (Example 3.4 continued) The predicate p has only one argument position, so let us consider $\Delta := \langle p \mapsto \{1\} \rangle$ which distinguishes position 1 for predicate p. For any derivation step $p(s) \underset{c}{\Longrightarrow} p(s_1)$ if we replace s by any term t then there exists a derivation step $p(t) \underset{c}{\Longrightarrow} p(t_1)$. Notice that $p(t_1)$ is more general than $p(s_1)$ up to the argument of p. So, by the intuition described above, Δ is DN for c. Consequently, as in c the body $p(f(X))$ is more general than the head $p(X)$ up to the argument of p which is neutral, by an extended version of Corollary 3.2 there exists an infinite derivation of $\{c\} \cup \{p(X)\}$. □

Let us give some more concrete examples of DN positions.

Example 3.6. The second argument position of the relation symbol *append* in the program `APPEND`:

```
append([],Ys,Ys).                              % C1
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).  % C2
```

is DN for `C2`. Notice that a very common programming technique called *accumulator passing* (see for instance e.g. [O'Keefe 1990], p. 21–25) always produces DN

positions. A classical example of the accumulator passing technique is the following program REVERSE.

```
reverse(L,R) :- rev(L,[],R).                    % C1
rev([],R,R).                                    % C2
rev([X|Xs],R0,R) :- rev(Xs,[X|R0],R).           % C3
```

Concerning termination, we may ignore the second and the third argument of *rev* in the recursive clause C3 while unfolding a query with this clause. Only the first argument can stop the unfolding. □

But we can be even more precise. Instead of only identifying positions that can be totaly disregarded as in the above examples, we can try to identify positions where we can place any terms for which a given condition holds.

Example 3.7. Consider the clause $c := p(f(X)) \leftarrow p(f(f(X)))$. If we mean by a DN position a position where we can place *any* terms, then the argument position of p is not DN for c. This is because, for example, we have the derivation step $p(X) \underset{c}{\Longrightarrow} p(f(f(X_1)))$ but if we replace X by $g(X)$ then there is no derivation step of $\{c\} \cup \{p(g(X))\}$. However, if we mean by a DN position a position where we can place any instances of $f(X)$, then the argument position of p is DN for c. □

In the sequel of the section, we define more precisely DN positions as positions where we can place any terms satisfying certain conditions identified by "filters". We use filters to present an extension of the relation "is more general than" and we propose an extended version of Corollary 3.2. We offer two syntactic conditions of increasing power for easily identifying DN positions from mere inspection of the text of a logic program. The practical impact of such filters will be tackled in Section 5.

3.1 Filters

Let us first introduce the notion of a *filter*. We use filters in order to distinguish atoms, some arguments of which satisfy a given condition. A condition upon atom arguments, *i.e.* terms, can be defined as a function in the following way.

Definition 3.8. (Term-condition) A *term-condition* is a function from the set of terms $TU_{\mathcal{L}}$ to $\{\texttt{true}, \texttt{false}\}$.

Example 3.9. The following functions are term-conditions.

$$\begin{aligned} f_{true} : TU_{\mathcal{L}} &\rightarrowtail \{\texttt{true}, \texttt{false}\} \\ t &\mapsto \texttt{true} \end{aligned}$$

$$\begin{aligned} f_1 : TU_{\mathcal{L}} &\rightarrowtail \{\texttt{true}, \texttt{false}\} \\ t &\mapsto \texttt{true} \text{ iff } t \text{ is an instance of } [X|Y] \end{aligned}$$

$$\begin{aligned} f_2 : TU_{\mathcal{L}} &\rightarrowtail \{\texttt{true}, \texttt{false}\} \\ t &\mapsto \texttt{true} \text{ iff } t \text{ unifies with } h(a, X) \end{aligned}$$

□

Notice that a term-condition might give distinct results for two terms which are equal modulo renaming. For instance $f_2(X) = \texttt{false}$ and $f_2(Y) = \texttt{true}$. However, in Definition 3.12 below, we will only consider *variant independent* term-conditions.

Definition 3.10. (Variant Independent Term-Condition) A term-condition f is *variant independent* if, for every term t, $f(t) = \texttt{true}$ implies that $f(t') = \texttt{true}$ for every variant t' of t.

Example 3.11. (Example 3.9 continued) f_{true} and f_1 are variant independent while f_2 is not. □

We restrict the class of term-conditions to that of variant independent ones because we want to extend the relation "is more general than" so that if an atom A is linked to an atom B by the extended relation, then every variant of A is also linked to B (see Proposition 3.16 below). This will be essential to establish the forthcoming main Proposition 3.19 which is an extension of Corollary 3.2. Now we can define what we exactly mean by a filter.

Definition 3.12. (Filter) A *filter*, denoted by Δ, is a function from Π such that: for each $p \in \Pi$, $\Delta(p)$ is a partial function from $[1, arity(p)]$ to the set of variant independent term-conditions.

Example 3.13. (Example 3.9 continued) Let p be a relation symbol whose arity equals 3. The filter Δ which maps p to the function $\langle 1 \mapsto f_{true},\ 2 \mapsto f_1 \rangle$ and any $q \in \Pi \setminus \{p\}$ to $\langle\rangle$ is noted $\Delta := \langle\, p \mapsto \langle 1 \mapsto f_{true},\ 2 \mapsto f_1 \rangle\, \rangle$. □

3.2 Extension of the Relation "Is More General Than"

Given a filter Δ, the relation "is more general than" can be extended in the following way: an atom $A := p(\cdots)$ is Δ-more general than $B := p(\cdots)$ if the "is more general than" requirement holds for those arguments of A whose position is not in the domain of $\Delta(p)$ while the other arguments satisfy their associated term-condition.

Definition 3.14. (Δ-more general) Let Δ be a filter and A and B be two atoms.

—Let η be a substitution. Then A *is* Δ*-more general than* B *for* η if:

$$\begin{cases} A = p(s_1, \ldots, s_n) \\ B = p(t_1, \ldots, t_n) \\ \forall i \in [1, n] \setminus Dom(\Delta(p)),\ t_i = s_i\eta \\ \forall i \in Dom(\Delta(p)),\ \Delta(p)(i)(s_i) = \texttt{true}\,. \end{cases}$$

—A *is* Δ*-more general than* B if there exists a substitution η s.t. A is Δ-more general than B for η.

An atomic query Q is Δ*-more general* than an atomic query Q' if either Q and Q' are both empty or Q contains the atom A, Q' contains the atom B and A is Δ-more general than B.

Example 3.15. (Example 3.13 continued) Let

$$\begin{array}{l} A := p(\ b\ ,\ \ X\ \ ,\ h(a, X)\ )\\ B := p(\ a\ ,\ [a|b]\ ,\ \ \ X\ \ \ )\\ C := p(\ a\ ,\ [a|b]\ ,\ h(Y, b)\ )\,. \end{array}$$

Then, A is not Δ-more general than B and C because, for instance, its second argument X is not an instance of $[X|Y]$ as required by f_1. On the other hand, B is Δ-more general than A for the substitution $\{X/h(a,X)\}$ and B is Δ-more general than C for the substitution $\{X/h(Y,b)\}$. Finally, C is not Δ-more general than A because $h(Y,b)$ is not more general than $h(a,X)$ and C is not Δ-more general than B because $h(Y,b)$ is not more general than X. □

As in a filter the term-conditions are variant independent, we get the following proposition.

PROPOSITION 3.16. *Let Δ be a filter and A and B be two atoms. If A is Δ-more general than B then every variant of A is Δ-more general than B.*

3.3 Derivation Neutral Filters: Operational Definition

In the sequel of this paper, we focus on "derivation neutral" filters. The name "derivation neutral" stems from the fact that in any derivation of an atomic query Q, the arguments of Q whose position is distinguished by such a filter can be safely replaced by any terms satisfying the associated term-condition. Such a replacement does not modify the derivation process.

Definition 3.17. (Derivation Neutral) Let Δ be a filter and c be a binary clause. We say that Δ *is DN for* c if for each derivation step $Q \underset{c}{\Longrightarrow} Q_1$ where Q is an atomic query, for each Q' that is Δ-more general than Q and for each variant c' of c variable disjoint with Q', there exists a query Q'_1 that is Δ-more general than Q_1 and such that $Q' \underset{c}{\Longrightarrow} Q'_1$ with input clause c'. This definition is extended to binary programs: Δ *is DN for* P if it is DN for each clause of P.

Example 3.18. The following examples illustrate the previous definition.

—Let us reconsider the program `APPEND` from Example 3.6 with the term-condition f_{true} defined in Example 3.9 and the filter $\Delta := \langle append \mapsto \langle 2 \mapsto f_{true}\rangle\rangle$. Δ is DN for `C2`. However, Δ is not DN for `APPEND` because it is not DN for `C1`.

—Consider the following clause:

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

The filter $\langle merge \mapsto \langle 2 \mapsto f_1\rangle\rangle$, where the term-condition f_1 is defined in Example 3.9, is DN for this clause.

In the next subsection, we present some syntactic criteria for identifying correct DN filters. For proving that the above filters are indeed DN, we will just check that they actually fulfill these syntactic criteria that are sufficient conditions. □

Derivation neutral filters lead to the following extended version of Corollary 3.2 (take Δ such that for any p, $\Delta(p)$ is a function whose domain is empty):

PROPOSITION 3.19. *Let $c := H \leftarrow B$ be a binary clause and Δ be a filter that is DN for c. If B is Δ-more general than H then H loops w.r.t. $\{c\}$.*

We point out that our non-termination inference technique remains valid when the program under consideration is restricted to its set of clauses used in the derivation steps. For instance, although the filter Δ of Example 3.18 is not DN for `APPEND`,

it will help us to construct queries which loop w.r.t. `C2`. Such queries also loop w.r.t. `APPEND`.

Finally, notice that lifting lemmas are used in the literature to prove completeness of SLD-resolution. As Definition 3.17 corresponds to an extended version of the One Step Lifting Lemma 3.1, it may be worth to investigate its consequences from the model theoretic point of view.

First of all, a filter may be used to "expand" atoms by replacing every argument whose position is distinguished by any term that satisfies the associated term-condition.

Definition 3.20. Let Δ be a filter and A be an atom. *The expansion of A w.r.t.* Δ, denoted $A_{\uparrow\Delta}$, is the set defined as

$$A_{\uparrow\Delta} \stackrel{def}{=} \{A\} \cup \{B \in TB_{\mathcal{L}} \mid B \text{ is } \Delta\text{-more general than } A \text{ for } \epsilon\}$$

where ϵ denotes the empty substitution.

Notice that in this definition, we do not necessary have the inclusion

$$\{A\} \subseteq \{B \in TB_{\mathcal{L}} \mid B \text{ is } \Delta\text{-more general than A for } \epsilon\} \ .$$

For instance, suppose that $A := p(f(X))$ and that Δ maps p to the function $\langle 1 \mapsto f \rangle$ where f is the term-condition mapping any term t to `true` iff t is an instance of $g(X)$. Then

$$\{B \in TB_{\mathcal{L}} \mid B \text{ is } \Delta\text{-more general than A}\} = \{p(t) \mid t \text{ is an instance of } g(X)\}$$

with $A \notin \{p(t) \mid t \text{ is an instance of } g(X)\}$.

Term interpretations in the context of logic programming were first introduced in [Clark 1979] and further investigated in [Deransart and Ferrand 1987] and then in [M. Falaschi and Palamidessi 1993]. A term interpretation for $\mathcal{L}$ is identified with a (possibly empty) subset of the term base $TB_{\mathcal{L}}$. So, as for atoms, a term interpretation can be expanded by a filter.

Definition 3.21. Let Δ be a filter and I be a term interpretation for $\mathcal{L}$. Then $I_{\uparrow\Delta}$ is the term interpretation for $\mathcal{L}$ defined as:

$$I_{\uparrow\Delta} \stackrel{def}{=} \bigcup_{A \in I} A_{\uparrow\Delta} \ .$$

For any logic program P, we denote by $\mathcal{C}(P)$ its least term model.

THEOREM 3.22. *Let P be a binary program and Δ be a DN filter for P. Then $\mathcal{C}(P)_{\uparrow\Delta} = \mathcal{C}(P)$.*

PROOF. The inclusion $\mathcal{C}(P) \subseteq \mathcal{C}(P)_{\uparrow\Delta}$ is straightforward so let us concentrate on the other one *i.e.* $\mathcal{C}(P)_{\uparrow\Delta} \subseteq \mathcal{C}(P)$. Let $A' \in \mathcal{C}(P)_{\uparrow\Delta}$. Then there exists $A \in \mathcal{C}(P)$ such that $A' \in A_{\uparrow\Delta}$. A well known result states:

$$\mathcal{C}(P) = \{B \in TB_{\mathcal{L}} \mid \text{there exists a successful derivation of } P \cup \{B\}\} \qquad (1)$$

Consequently, there exists a successful derivation ξ of $P \cup \{A\}$. Therefore, by successively applying Definition 3.17 to each step of ξ, one constructs a successful derivation of A'. So by (1) $A' \in \mathcal{C}(P)$. □

3.4 Some Particular DN Filters

In this section, we provide two sufficient syntactic conditions for identifying DN filters.

3.4.1 *DN Sets of Positions.* The first instance we consider corresponds to filters, the associated term-conditions of which are all equal to f_{true} (see Example 3.9). Within such a context, as the term-conditions are fixed, each filter Δ is uniquely determined by the domains of the partial functions $\Delta(p)$ for $p \in \Pi$. Hence the following definition.

Definition 3.23. (Set of Positions) A *set of positions*, denoted by τ, is a function from Π to 2^N such that: for each $p \in \Pi$, $\tau(p)$ is a subset of $[1, arity(p)]$.

Example 3.24. Let *append* and *append3* be two relation symbols. Let us assume that $arity(append) = 3$ and $arity(append3) = 4$. Then

$$\tau := \langle\ append \mapsto \{2\},\ append3 \mapsto \{2, 3, 4\}\ \rangle$$

is a set of positions. □

Not surprisingly, the filter that is generated by a set of positions is defined as follows.

Definition 3.25. (Associated Filter) Let τ be a set of positions and f_{true} be the term-condition defined in Example 3.9. The filter $\Delta[\tau]$ defined as:

$$\text{for each } p \in \Pi,\ \Delta[\tau](p) \text{ is the function from } \tau(p) \text{ to } \{f_{true}\}$$

is called *the filter associated to* τ.

Example 3.26. (Example 3.24 continued) The filter associated to τ is

$$\Delta[\tau] := \langle append \mapsto \langle 2 \mapsto f_{true}\rangle, append3 \mapsto \langle 2 \mapsto f_{true}, 3 \mapsto f_{true}, 4 \mapsto f_{true}\rangle\rangle.$$

□

Now we define a particular kind of sets of positions. These are named after "DN" because, as stated by Theorem 3.29 below, they generate DN filters.

Definition 3.27. (DN Set of Positions) Let τ be a set of positions. We say that τ *is DN for a binary clause* $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_m)$ if:

$$\forall i \in \tau(p),\ \begin{cases} s_i \text{ is a variable} \\ s_i \text{ occurs only once in } p(s_1, \ldots, s_n) \\ \forall j \in [1, m],\ s_i \in Var(t_j) \Rightarrow j \in \tau(q)\ . \end{cases}$$

A set of positions is *DN for a binary program* P if it is DN for each clause of P.

The intuition of Definition 3.27 is the following. If for instance we have a clause $c := p(X, Y, f(Z)) \leftarrow p(g(Y, Z), X, Z)$ then in the first two positions of p we can put any terms and get a derivation step w.r.t. c because the first two arguments of the head of c are variables that appear exactly once in the head. Moreover, X and Y of the head reappear in the body but again only in the first two positions of p. So, if we have a derivation step $p(s_1, s_2, s_3) \underset{c}{\Longrightarrow} p(t_1, t_2, t_3)$, we can replace s_1 and s_2 by any terms s'_1 and s'_2 and get another derivation step $p(s'_1, s'_2, s_3) \underset{c}{\Longrightarrow} p(t'_1, t'_2, t'_3)$ where t'_3 is the same as t_3 up to variable names.

Example 3.28. (Example 3.24 continued) τ is DN for the program:

```
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
append3(Xs,Ys,Zs,Ts) :- append(Xs,Ys,Us).
```

which is a subset of the binary unfoldings of the program `APPEND3`:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
append3(Xs,Ys,Zs,Ts) :- append(Xs,Ys,Us), append(Us,Zs,Ts).
```

□

DN sets of positions generate DN filters.

THEOREM 3.29. *Let τ be a DN set of positions for a binary program P. Then $\Delta[\tau]$ is DN for P.*

PROOF. As we will see in Section 3.4.2, this theorem is a particular case of Theorem 3.38. □

Notice that the set of DN sets of positions of any binary program P is not empty because, by Definition 3.27, $\tau_0 := \langle p \mapsto \varnothing \mid p \in \Pi \rangle$ is DN for P. Moreover, an atom A is $\Delta[\tau_0]$-more general than an atom B iff A is more general than B.

3.4.2 *DN Sets of Positions with Associated Terms.* Now we consider another instance of Definition 3.17. As we will see, it is more general than the previous one. It corresponds to filters whose associated term-conditions have all the form "is an instance of t" where t is a term that uniquely determines the term-condition. Notice that such term-conditions are variant independent, so it makes sense to consider such filters. Hence the following definition.

Definition 3.30. (Sets of Positions with Associated Terms) A *set of positions with associated terms*, denoted by τ^+, is a function from Π such that: for each $p \in \Pi$, $\tau^+(p)$ is a partial function from $[1, arity(p)]$ to $TU_{\mathcal{L}}$.

Example 3.31. Let p and q be two relation symbols whose arity is 2. Then

$$\tau^+ := \langle\, p \mapsto \langle 2 \mapsto X \rangle,\ q \mapsto \langle 2 \mapsto g(X) \rangle \,\rangle$$

is a set of positions with associated terms. □

The filter that is generated by a set of positions with associated terms is defined as follows.

Definition 3.32. (Associated Filter) Let τ^+ be a set of positions with associated terms. The *filter associated to* τ^+, denoted by $\Delta[\tau^+]$, is defined as: for each $p \in \Pi$, $\Delta[\tau^+](p)$ is the function

$$\begin{array}{lll} Dom(\tau^+(p)) & \rightarrowtail & \text{The set of term-conditions} \\ i & \mapsto & \begin{cases} TU_{\mathcal{L}} & \rightarrowtail \{\texttt{true}, \texttt{false}\} \\ t & \mapsto \texttt{true} \text{ iff } t \text{ is an instance of } \tau^+(p)(i) \end{cases} \end{array}$$

Example 3.33. (Example 3.31 continued) The filter associated to τ^+ is

$$\Delta[\tau^+] := \langle\ p \mapsto \langle 2 \mapsto f_1 \rangle,\ q \mapsto \langle 2 \mapsto f_2 \rangle\ \rangle$$

where

$$\begin{array}{rcl} f_1 : TU_{\mathcal{L}} & \rightarrowtail & \{\texttt{true}, \texttt{false}\} \\ t & \mapsto & \texttt{true} \text{ iff } t \text{ is an instance of } X \end{array}$$

$$\begin{array}{rcl} f_2 : TU_{\mathcal{L}} & \rightarrowtail & \{\texttt{true}, \texttt{false}\} \\ t & \mapsto & \texttt{true} \text{ iff } t \text{ is an instance of } g(X) \end{array}$$

□

As for sets of positions, we define a special kind of sets of positions with associated terms.

Definition 3.34. (DN Sets of Positions with Associated Terms) Let τ^+ be a set of positions with associated terms. We say that τ^+ is *DN for a binary clause* $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_m)$ if these conditions hold:

—**(DN1)** $\forall i \in Dom(\tau^+(p)), \forall j \in [1, n] \setminus \{i\}$: $Var(s_i) \cap Var(s_j) = \varnothing$,
—**(DN2)** $\forall \langle i \mapsto u_i \rangle \in \tau^+(p)$: s_i is more general than u_i,
—**(DN3)** $\forall \langle j \mapsto u_j \rangle \in \tau^+(q)$: t_j is an instance of u_j,
—**(DN4)** $\forall i \in Dom(\tau^+(p)), \forall j \notin Dom(\tau^+(q))$: $Var(s_i) \cap Var(t_j) = \varnothing$.

A set of positions with associated terms is *DN for a binary program* P if it is DN for each clause of P.

This definition says that any s_i where i is in the domain of $\tau^+(p)$ (*i.e.* position i is distinguished by τ^+): **(DN1)** does not share its variables with the other arguments of the head, **(DN2)** is more general than the term u_i that i is mapped to by $\tau^+(p)$, **(DN4)** distributes its variables to some t_j such that j is in the domain of $\tau^+(q)$ (*i.e.* position j is distinguished by τ^+). Moreover, **(DN3)** says that any t_j, where j is distinguished by τ^+, is such that t_j is an instance of the term u_j that j is mapped to by $\tau^+(q)$.

Example 3.35. (Example 3.31 continued) τ^+ is DN for the following program:

```
p(f(X),Y) :- q(X,g(X)).
q(a,g(X)) :- q(a,g(b)).
```

The preceding notion is closed under renaming:

PROPOSITION 3.36. *Let c be a binary clause and τ^+ be a set of positions with associated terms that is DN for c. Then τ^+ is DN for every variant of c.*

Notice that a set of positions is a particular set of positions with associated terms in the following sense.

PROPOSITION 3.37. *Let τ be a set of positions and X be a variable. Let τ^+ be the set of positions with associated terms defined as: for each $p \in \Pi$, $\tau^+(p) := (\ \tau(p) \rightarrowtail \{X\}\ )$. Then, the following holds.*

(*1*) *An atom A is $\Delta[\tau]$-more general than an atom B iff A is $\Delta[\tau^+]$-more general than B.*

(*2*) *For any binary clause c, τ is DN for c iff τ^+ is DN for c.*

PROOF. A proof follows from these remarks.

—Item 1 is a direct consequence of the definition of "Δ-more general" (see Definition 3.14) and the definition of the filter associated to a set of positions (see Definition 3.25) and to a set of positions with associated terms (see Definition 3.32).

—Item 2 is a direct consequence of the definition of DN sets of positions (see Definition 3.27) and DN sets of positions with associated terms (see Definition 3.34).

□

The sets of positions with associated terms of Definition 3.34 were named after "DN" because of the following result.

THEOREM 3.38. *Let P be a binary program and τ^+ be a set of positions with associated terms that is DN for P. Then $\Delta[\tau^+]$ is DN for P.*

PROOF. A proof of this result can be found in the long version of the paper (CoRR archive at `http://arxiv.org/archive/cs/intro.html` – Paper ID is `cs.PL/0406041`). □

As in the case of sets of positions, the set of DN sets of positions with associated terms of any binary program P is not empty because, by Definition 3.34, $\tau_0^+ := \langle p \mapsto \langle\rangle \mid p \in \Pi \rangle$ is DN for P. Moreover, an atom A is $\Delta[\tau_0^+]$-more general than an atom B iff A is more general than B. Finally, in Appendix A, we give an incremental algorithm (see Section 4.2) that computes a DN set of positions with associated terms. Its correctness proof can be found in the long version of this paper.

3.5 Examples

This section presents some examples where we use filters obtained from DN sets of positions and DN sets of positions with associated terms to infer looping queries. As the filters we use in each case are not "empty" (*i.e.* are not obtained from τ_0 or τ_0^+), we are able to compute more looping queries than using the classical subsumption test.

Example 3.39. Consider the program `APPEND` that we introduced in Example 3.6. Every infinite derivation w.r.t. `APPEND` starting from an atomic query only uses the non-unit clause `C2`. Therefore, as we aim at inferring looping atomic queries w.r.t. `APPEND`, we only focus on `C2` in the sequel of this example.

As in `C2` the body, which is $append(Xs, Ys, Zs)$, is more general than the head, which is $append([X|Xs], Ys, [X|Zs])$, by Corollary 3.2 we conclude that the query $append([X|Xs], Ys, [X|Zs])$ loops w.r.t. $\{$`C2`$\}$. Consequently, by the One Step Lifting Lemma 3.1, each query that is more general than $append([X|Xs], Ys, [X|Zs])$ also loops w.r.t. $\{$`C2`$\}$.

But we can be more precise than that. According to Definition 3.27, $\tau := \langle\ append \mapsto \{2\}\ \rangle$ is a DN set of positions for $\{$`C2`$\}$. The filter associated to τ

(see Definition 3.25) is $\Delta[\tau] := \langle\ append \mapsto \langle 2 \mapsto f_{true}\rangle\ \rangle$. By Theorem 3.29, $\Delta[\tau]$ is a DN filter for $\{\texttt{C2}\}$. Consequently, by Definition 3.17, each query that is $\Delta[\tau]$-more general than $append([X|Xs], Ys, [X|Zs])$ loops w.r.t. $\{\texttt{C2}\}$. This means that

$$\left\{ append(t_1, t_2, t_3) \in TB_{\mathcal{L}} \,\middle|\, \begin{array}{l} t_2 \text{ is any term and} \\ t_1, t_3 \text{ is more general than } [X|Xs], [X|Zs] \end{array} \right\}$$

is a set of atomic queries that loop w.r.t. $\{\texttt{C2}\}$, hence w.r.t. `APPEND`. This set includes the query $append(As, [\,], Bs)$. □

Example 3.40. Consider the program `REVERSE` that was introduced in Example 3.6. As in the example above, in order to infer looping atomic queries w.r.t. `REVERSE`, we only focus on the non-unit clauses `C1` and `C3` in the sequel of this example. More precisely, we process the relation symbols of the program in a bottom-up way, so we start the study with clause `C3` and end it with clause `C1`.

According to Definition 3.27, $\tau := \langle\ rev \mapsto \{2, 3\}\ \rangle$ is a DN set of positions for $\{\texttt{C3}\}$. The filter associated to τ (see Definition 3.25) is $\Delta[\tau] := \langle\ rev \mapsto \langle 2 \mapsto f_{true},\ 3 \mapsto f_{true}\rangle\ \rangle$. By Theorem 3.29, $\Delta[\tau]$ is DN for $\{\texttt{C3}\}$. As $rev(Xs, [X|R_0], R)$ (the body of `C3`) is $\Delta[\tau]$-more general than $rev([X|Xs], R_0, R)$ (the head of `C3`), by Proposition 3.19 we get that $rev([X|Xs], R_0, R)$ loops w.r.t. $\{\texttt{C3}\}$. Notice that, unlike the example above, here we do not get this result from Corollary 3.2 as $rev(Xs, [X|R_0], R)$ is not more general than $rev([X|Xs], R_0, R)$. Finally, as $\Delta[\tau]$ is DN for $\{\texttt{C3}\}$, by Definition 3.17 we get that each query that is $\Delta[\tau]$-more general than $rev([X|Xs], R_0, R)$ loops w.r.t. $\{\texttt{C3}\}$, hence w.r.t. `REVERSE`. This means that

$$\mathcal{Q} := \left\{ rev(t_1, t_2, t_3) \in TB_{\mathcal{L}} \,\middle|\, \begin{array}{l} t_2 \text{ and } t_3 \text{ are any terms and} \\ t_1 \text{ is more general than } [X|Xs] \end{array} \right\}$$

is a set of atomic queries that loop w.r.t. `REVERSE`. This set includes the query $rev(As, [\,], [\,])$.

Now, consider clause `C1`. As $rev(L, [], R)$ (its body) is an element of $\mathcal{Q}$, then $rev(L, [], R)$ loops w.r.t. $\{\texttt{C3}\}$, hence w.r.t. $\{\texttt{C1}, \texttt{C3}\}$. Consequently, by Corollary 3.3, $reverse(L, R)$ (the head of `C1`) loops w.r.t. $\{\texttt{C1}, \texttt{C3}\}$. Let $\tau' := \langle\ rev \mapsto \{2, 3\},\ reverse \mapsto \{2\}\ \rangle$. By Definition 3.27, τ' is DN for $\{\texttt{C1}, \texttt{C3}\}$, so $\Delta[\tau']$ is DN for $\{\texttt{C1}, \texttt{C3}\}$. Consequently, each query that is $\Delta[\tau']$-more general than $reverse(L, R)$ also loops w.r.t. $\{\texttt{C1}, \texttt{C3}\}$ hence w.r.t. `REVERSE`. This means that

$$\{ reverse(X, t) \in TB_{\mathcal{L}} \mid X \text{ is a variable and } t \text{ is any term} \}$$

is a set of atomic queries that loop w.r.t. `REVERSE`. This set includes the query $reverse(As, [\,])$. □

Example 3.41. Consider the recursive clauses of the following program `M`:

```
m([],Xs,Xs).
m(Xs,[],Xs).
m([X|Xs],[Y|Ys],[X|Zs]) :- m(Xs,[Y|Ys],Zs). % C3
m([X|Xs],[Y|Ys],[Y|Zs]) :- m([X|Xs],Ys,Zs). % C4
```

Every set of positions τ that is DN for $\{\texttt{C3}\}$ is such that $\tau(m) = \varnothing$ because each argument of the head of `C3` is not a variable (see Definition 3.27). Hence,

using Proposition 3.19 with a filter obtained from a DN set of positions leads to the same results as using Corollary 3.2: as $m(Xs, [Y|Ys], Zs)$ is more general than $m([X|Xs], [Y|Ys], [X|Zs])$, then $m([X|Xs], [Y|Ys], [X|Zs])$ loops w.r.t. $\{\texttt{C3}\}$. So, by the One Step Lifting Lemma 3.1, each query that is more general than $m([X|Xs], [Y|Ys], [X|Zs])$ also loops w.r.t. $\{\texttt{C3}\}$, hence w.r.t. `M`.

But we can be more precise than that. According to Definition 3.34, $\tau^+ := \langle m \mapsto \langle 2 \mapsto [Y|Ys] \rangle \rangle$ is a set of positions with associated terms that is DN for $\{\texttt{C3}\}$. Hence, by Theorem 3.38, the associated filter $\Delta[\tau^+]$ (see Definition 3.32) is DN for $\{\texttt{C3}\}$. So, by Definition 3.17, each query that is $\Delta[\tau^+]$-more general than $m([X|Xs], [Y|Ys], [X|Zs])$ loops w.r.t. $\{\texttt{C3}\}$. This means that

$$\left\{ m(t_1, t_2, t_3) \in TB_{\mathcal{L}} \,\middle|\, \begin{array}{l} t_2 \text{ is any instance of } [Y|Ys] \text{ and} \\ t_1, t_3 \text{ is more general than } [X|Xs], [X|Zs] \end{array} \right\}$$

is a set of atomic queries that loop w.r.t. `M`, which includes the query $m(As, [0], Bs)$. Finally, let us turn to clause `C4`. Reasoning exactly as above with the set of positions with associated terms $\langle m \mapsto \langle 1 \mapsto [X|Xs] \rangle \rangle$ which is DN for $\{\texttt{C4}\}$, we conclude that:

$$\left\{ m(t_1, t_2, t_3) \in TB_{\mathcal{L}} \,\middle|\, \begin{array}{l} t_1 \text{ is any instance of } [X|Xs] \text{ and} \\ t_2, t_3 \text{ is more general than } [Y|Ys], [Y|Zs] \end{array} \right\}$$

is a set of atomic queries that loop w.r.t. `M`. For instance, $m([0], As, Bs)$ is a query that belongs to this set. □

4. ALGORITHMS

We have designed a set of correct algorithms for full automation of non-termination analysis of logic programs. These algorithms are given in Appendix B and their correctness proofs can be found in the long version of the paper. In this section, we present the intuitions and conceptual definitions underlying our approach.

4.1 Loop Dictionaries

Our technique is based on a data structure called *dictionary* which is a set of pairs $(BinSeq, \tau^+)$ where $BinSeq$ is a finite ordered sequence of binary clauses and τ^+ is a set of positions with associated terms. In the sequel, we use the list notation of Prolog and a special kind of dictionaries that we define as follows.

Definition 4.1. (Looping Pair, Loop Dictionary) A pair $(BinSeq, \tau^+)$, where the list $BinSeq$ is a finite ordered sequence of binary clauses and τ^+ is a set of positions with associated terms, is a *looping pair* if τ^+ is DN for $BinSeq$ and:

—either $BinSeq = [H \leftarrow B]$ and B is $\Delta[\tau^+]$-more general than H,

—or $BinSeq = [H \leftarrow B, H_1 \leftarrow B_1 \mid BinSeq_1]$ and there exists a set of positions with associated terms τ_1^+ such that $([H_1 \leftarrow B_1 \mid BinSeq_1], \tau_1^+)$ is a looping pair and B is $\Delta[\tau_1^+]$-more general than H_1.

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

Example 4.2. The pair $(\mathit{BinSeq} := [H_1 \leftarrow B_1, H_2 \leftarrow B_2, H_3 \leftarrow B_3], \tau_1^+)$ where

$$H_1 \leftarrow B_1 := r(X) \leftarrow q(X, f(f(X)))$$
$$H_2 \leftarrow B_2 := q(X, f(Y)) \leftarrow p(f(X), a)$$
$$H_3 \leftarrow B_3 := p(f(g(X)), a) \leftarrow p(X, a)$$

and $\tau_1^+ := \langle p \mapsto \langle 2 \mapsto a\rangle, q \mapsto \langle 2 \mapsto f(X)\rangle\rangle$ is a looping pair:

—Let $\tau_3^+ := \langle p \mapsto \langle 2 \mapsto a\rangle\rangle$. Then τ_3^+ is a DN set of positions with associated terms for $[H_3 \leftarrow B_3]$. Moreover, B_3 is $\Delta(\tau_3^+)$-more general than H_3. Consequently, $([H_3 \leftarrow B_3], \tau_3^+)$ is a looping pair.

—Notice that B_2 is $\Delta(\tau_3^+)$-more general than H_3. Now, let $\tau_2^+ := \tau_1^+$. Then τ_2^+ is DN for $[H_2 \leftarrow B_2, H_3 \leftarrow B_3]$. So, $([H_2 \leftarrow B_2, H_3 \leftarrow B_3], \tau_2^+)$ is a looping pair.

—Finally, notice that B_1 is $\Delta(\tau_2^+)$-more general than H_2. As τ_1^+ is DN for BinSeq, we conclude that $(\mathit{BinSeq}, \tau_1^+)$ is a looping pair. □

A looping pair immediately provides an atomic looping query. It suffices to take the head of the first clause of the binary program of the pair:

Proposition 4.3. *Let* $([H \leftarrow B|\mathit{BinSeq}], \tau^+)$ *be a looping pair. Then* H *loops w.r.t.* $[H \leftarrow B|\mathit{BinSeq}]$.

Proof. By induction on the length of *BinSeq* using Proposition 3.19, Corollary 3.3 and Theorem 3.38. □

So, a looping pair denotes a proof outline for establishing that H left loops. Moreover, looping pairs can be built incrementally in a simple way as described below.

4.2 Computing a Loop Dictionary

Given a logic program P and a positive integer max, the function `infer_loop_dict` from Appendix B first computes $T_P^\beta \uparrow max$ (the first max iterations of the operator T_P^β), which is a finite subset of $\mathit{bin\_unf}(P)$. Then, using the clauses of $T_P^\beta \uparrow max$, it incrementally builds a loop dictionary *Dict* as follows.

At start, *Dict* is set to $\varnothing$. Then, for each clause $H \leftarrow B$ in $T_P^\beta \uparrow max$, the following actions are performed.

—`infer_loop_dict` tries to extract from $H \leftarrow B$ the most simple form of a looping pair: it computes a set of positions with associated terms τ^+ that is DN for $H \leftarrow B$, then it tests if B is $\Delta[\tau^+]$-more general than H. If so, the looping pair $([H \leftarrow B], \tau^+)$ is added to *Dict*.

—`infer_loop_dict` tries to combine $H \leftarrow B$ to some looping pairs that have already been added to *Dict* in order to build other looping pairs. For each $([H_1 \leftarrow B_1|\mathit{BinSeq}_1], \tau_1^+)$ in *Dict*, if B is $\Delta[\tau_1^+]$-more general than H_1, then a set of positions with associated terms τ^+ that is DN for $[H \leftarrow B, H_1 \leftarrow B_1|\mathit{BinSeq}_1]$ is computed and the looping pair $([H \leftarrow B, H_1 \leftarrow B_1|\mathit{BinSeq}_1], \tau^+)$ is added to *Dict*.

Notice that in the second step above, we compute τ^+ that is DN for $[H \leftarrow B, H_1 \leftarrow B_1|\mathit{BinSeq}_1]$. As we already hold τ_1^+ that is DN for $[H_1 \leftarrow B_1|\mathit{BinSeq}_1]$, it is more interesting, for efficiency reasons, to compute τ^+ from τ_1^+ instead of starting

from the ground. Indeed, starting from τ_1^+, one uses the information stored in τ_1^+ about the program $[H_1 \leftarrow B_1|BinSeq_1]$, which may speed up the computation substantially. This is why we have designed a function `dna` that takes two arguments as input, a binary program *BinProg* and a set of positions with associated terms τ^+. It computes a set of positions with associated terms that is DN for *BinProg* and that refines τ^+. On the other hand, the function `unit_loop` calls `dna` with τ^+_{max} which is the initial set of positions with associated terms defined as follows: $Dom(\tau^+_{max}(p)) = [1, arity(p)]$ for each $p \in \Pi$ and $\tau^+_{max}(p)(i)$ is a variable for each $i \in [1, arity(p)]$.

Example 4.4. Consider the program `APPEND3`

```
append3(Xs,Ys,Zs,Us) :- append(Xs,Ys,Vs), append(Vs,Zs,Us).
```

augmented with the `APPEND` program. The set $T^{\beta}_{\texttt{APPEND3}} \uparrow 2$ includes:

```
append([A|B],C,[A|D]) :- append(B,C,D).      % BC1
append3(A,B,C,D) :- append(A,B,E).           % BC2
append3([],A,B,C) :- append(A,B,C).          % BC3
```

From clause `BC1` we get the looping pair $(BinSeq_1, \tau_1^+)$ where

$$BinSeq_1 = \big[append([X_1|X_2], X_3, [X_1|X_4]) \leftarrow append(X_2, X_3, X_4)\big]$$

and $\tau_1^+(append) = \langle 2 \mapsto X_3 \rangle$. From this pair and the clause `BC2`, we get the looping pair $(BinSeq_2, \tau_2^+)$ where:

$$BinSeq_2 = \Big[\ \begin{array}{l} append3(X_1, X_2, X_3, X_4) \leftarrow append(X_1, X_2, X_5), \\ append([X_1|X_2], X_3, [X_1|X_4]) \leftarrow append(X_2, X_3, X_4)\ \Big] \end{array}$$

and $\tau_2^+(append) = \langle 2 \mapsto X_3 \rangle$ and $\tau_2^+(append3) = \langle 2 \mapsto X_2, 3 \mapsto X_3, 4 \mapsto X_4 \rangle$.

Finally, from $(BinSeq_1, \tau_1^+)$ and `BC3`, we get the looping pair $(BinSeq_3, \tau_3^+)$ where:

$$BinSeq_3 = \Big[\ \begin{array}{l} append3([], X_1, X_2, X_3) \leftarrow append(X_1, X_2, X_3), \\ append([X_1|X_2], X_3, [X_1|X_4]) \leftarrow append(X_2, X_3, X_4)\ \Big] \end{array}$$

and $\tau_3^+(append) = \langle 2 \mapsto X_3 \rangle$ and $\tau_3^+(append3) = \langle 3 \mapsto X_2 \rangle$. □

Example 4.5. Consider the program `PERMUTE`:

```
delete(X,[X|Xs],Xs).
delete(Y,[X|Xs],[X|Ys]) :- delete(Y,Xs,Ys).

permute([],[]).
permute([X|Xs],[Y|Ys]) :- delete(Y,[X|Xs],Zs), permute(Zs,Ys).
```

The set $T^{\beta}_{\texttt{PERMUTE}} \uparrow 1$ includes:

```
delete(B,[C|D],[C|E]) :- delete(B,D,E).      % BC1
permute([B|C],[D|E]) :- delete(D,[B|C],F).   % BC2
```

From clause BC1 we get the looping pair $(BinSeq_1, \tau_1^+)$ where

$$BinSeq_1 = \big[delete(X_1, [X_2|X_3], [X_2|X_4]) \leftarrow delete(X_1, X_3, X_4)\big]$$

and $\tau_1^+(delete) = \langle 1 \mapsto X_1 \rangle$. From this pair and the clause BC2, we get the looping pair $(BinSeq_2, \tau_2^+)$ where:

$$BinSeq_2 = \big[\ \begin{array}{l} permute([X_1|X_2], [X_3|X_4]) \leftarrow delete(X_3, [X_1|X_2], X_5), \\ delete(X_1, [X_2|X_3], [X_2|X_4]) \leftarrow delete(X_1, X_3, X_4)\ \big] \end{array}$$

and $\tau_2^+(delete) = \langle 1 \mapsto X_1 \rangle$ and $\tau_2^+(permute) = \langle 2 \mapsto [X_3|X_4] \rangle$. □

4.3 Looping Conditions

One of the main purposes of this article is the inference of classes of atomic queries that left loop w.r.t. a given logic program. Classes of atomic queries we consider are defined by pairs (A, τ^+) where A is an atom and τ^+ is a set of positions with associated terms. Such a pair denotes the set of queries $A_{\uparrow\tau^+}$, the definition of which is similar to that of the expansion of an atom, see Definition 3.20.

Definition 4.6. Let A be an atom and τ^+ be a set of positions with associated terms. Then $A_{\uparrow\tau^+}$ denotes the class of atomic queries defined as:

$$A_{\uparrow\tau^+} \stackrel{def}{=} \{A\} \cup \{B \in TB_{\mathcal{L}} \mid B \text{ is } \Delta[\tau^+]\text{-more general than A}\}\ .$$

Once each element of $A_{\uparrow\tau^+}$ left loops w.r.t. a logic program, we get what we call a *looping condition* for that program:

Definition 4.7. (Looping Condition) Let P be a logic program. *A looping condition for* P is a pair (A, τ^+) such that each element of $A_{\uparrow\tau^+}$ left loops w.r.t. P.

The function `infer_loop_cond` takes as arguments a logic program P and a non-negative integer max. Calling `infer_loop_dict`(P, max), it first computes a loop dictionary $Dict$. Then, it computes from $Dict$ looping conditions for P as follows. The function extracts the pair (H, τ^+) from each element $([H \leftarrow B|BinSeq], \tau^+)$ of $Dict$. By Proposition 4.3, H loops w.r.t. $[H \leftarrow B|BinSeq]$. As τ^+, hence $\Delta[\tau^+]$, is DN for $[H \leftarrow B|BinSeq]$, by Definition 3.17 each element of $H_{\uparrow\tau^+}$ loops w.r.t. $[H \leftarrow B|BinSeq]$. Finally, as $[H \leftarrow B|BinSeq] \subseteq T_P^\beta \uparrow max \subseteq bin\_unf(P)$, by Theorem 2.1, each element of $H_{\uparrow\tau^+}$ left loops w.r.t. P.

Example 4.8. (Example 4.4 continued) From each looping pair we have infered, we get the following information.

—$(append([X_1|X_2], X_3, [X_1|X_4]), \tau_1^+)$ is a looping condition. Therefore, each query $append(t_1, t_2, t_3)$, where $[X_1|X_2] = t_1\eta$ and $[X_1|X_4] = t_3\eta$ for a substitution η and t_2 is an instance of X_3 (because $\tau_1^+(append)(2) = X_3$), left loops w.r.t. APPEND3. In other words, each query $append(t_1, t_2, t_3)$, where $[X_1|X_2] = t_1\eta$ and $[X_1|X_4] = t_3\eta$ for a substitution η and t_2 is *any* term, left loops w.r.t. APPEND3.

—$(append3(X_1, X_2, X_3, X_4), \tau_2^+)$ is a looping condition. As $\tau_2^+(append3)(2) = X_2$, $\tau_2^+(append3)(3) = X_3$ and $\tau_2^+(append3)(4) = X_4$, this means that each query of form $append3(X_1, t_2, t_3, t_4)$, where t_2, t_3 and t_4 are *any* terms, left loops w.r.t. APPEND3.

—$(append3([], X_1, X_2, X_3), \tau_3^+)$ is a looping condition. So, as $\tau_3^+(append3)(3) = X_2$, this means that each query of form $append3([], X_1, t, X_3)$, where t is *any* term, left loops w.r.t. `APPEND3`. □

Example 4.9. (Example 4.5 continued) From each looping pair we have infered, we get the following information.

—$(delete(X_1, [X_2|X_3], [X_2|X_4]), \tau_1^+)$ is a looping condition. As $\tau_1^+(delete)(1) = X_1$, this means that each query of form $delete(t_1, t_2, t_3)$, where t_1 is *any term* and $[X_2|X_3] = t_2\eta$ and $[X_2|X_4] = t_3\eta$ for a substitution η, left loops w.r.t. `PERMUTE`.

—$(permute([X_1|X_2], [X_3|X_4]), \tau_2^+)$ is a looping condition. As $\tau_2^+(permute)(2) = [X_3|X_4]$, this means that each query of form $permute(t_1, t_2)$, where t_1 is more general than $[X_1|X_2]$ and t_2 is any instance of $[X_3|X_4]$, left loops w.r.t. `PERMUTE`. □

5. AN APPLICATION: PROVING OPTIMALITY OF TERMINATION CONDITIONS

[Mesnard and Neumerkel 2001] presents a tool for inferring termination conditions that are expressed as multi-modes, *i.e.* as disjunctions of conjunctions of propositions of form "the i-th argument is ground". In this section, we describe an algorithm that attempts proofs of optimality of such conditions using the algorithms for non-termination inference of the previous section.

5.1 Optimal Terminating Multi-modes

Let P be a logic program and $p \in \Pi_P$ be a relation symbol, with $arity(p) = n$. First, we describe the language we use for abstracting sets of atomic queries:

Definition 5.1. (Mode) A *mode* m_p for p is a subset of $[1, n]$, and denotes the following set of atomic goals: $[m_p] = \{p(t_1, \ldots, t_n) \in TB_{\mathcal{L}} \mid \forall i \in m_p \ Var(t_i) = \varnothing\}$. The set of all modes for p, *i.e.* $2^{[1,n]}$, is denoted $modes(p)$.

Note that if $m_p = \varnothing$ then $[m_p] = \{p(t_1, \ldots, t_n) \in TB_{\mathcal{L}}\}$. Since a logic procedure may have multiple uses, we generalize:

Definition 5.2. (Multi-mode) A *multi-mode* M_p for p is a finite set of modes for p and denotes the following set of atomic queries: $[M_p] = \cup_{m \in M_p}[m]$.

Note that if $M_p = \varnothing$, then $[M_p] = \varnothing$. Now we can define what we mean by terminating and looping multi-modes:

Definition 5.3. (Terminating mode, terminating multi-mode) A *terminating mode* m_p *for* p is a mode for p such that any query in $[m_p]$ left terminates w.r.t. P. A *terminating multi-mode* TM_p *for* p is a finite set of terminating modes for p.

Definition 5.4. (Looping mode, looping multi-mode) A *looping mode* m_p *for* p is a mode for p such that there exists a query in $[m_p]$ which left loops w.r.t. P. A *looping multi-mode* LM_p *for* p is a finite set of looping modes for p.

As left termination is instantiation-closed, any mode that is "below" (less general than) a terminating mode is also a terminating mode. Similarly, as left looping is generalization-closed, any mode that is "above" (more general than) a looping mode is also a looping mode. Let us be more precise:

Definition 5.5. (Less_general, more_general) Let M_p be a multi-mode for the relation symbol p. We set:

$$less\_general(M_p) = \{m \in modes(p) \mid \exists m' \in M_p\ [m] \subseteq [m']\}$$
$$more\_general(M_p) = \{m \in modes(p) \mid \exists m' \in M_p\ [m'] \subseteq [m]\}$$

We are now equipped to present a definition of optimality for terminating multi-modes:

Definition 5.6. (Optimal terminating multi-mode) A terminating multi-mode TM_p for p is *optimal* if there exists a looping multi-mode LM_p verifying:

$$modes(p) = less\_general(TM_p) \cup more\_general(LM_p)$$

Otherwise stated, given a terminating multi-mode TM_p, if each mode which is not less general than a mode of TM_p is a looping mode, then TM_p *characterizes* the operational behavior of p w.r.t. left termination and our language for defining sets of queries.

Example 5.7. Consider the program `APPEND`. A well-known terminating multi-mode is $TM_{append} = \{\{1\}, \{3\}\}$. Indeed, any query of the form `append(`t`,Ys,Zs)` or `append(Xs,Ys,`t`)`, where t is a ground term (*i.e.* such that $Var(t) = \varnothing$), left terminates. We have:

$$less\_general(TM_{append}) = \{\{1\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

On the other hand, `append(Xs,[],Zs)` left loops. Hence $LM_{append} = \{\{2\}\}$ is a looping condition and $more\_general(LM_{append}) = \{\varnothing, \{2\}\}$.

Since $modes(append) = less\_general(TM_{append}) \cup more\_general(LM_{append})$, we conclude that the terminating multi-mode TM_{append} is optimal. □

5.2 Algorithms

Suppose we hold a finite set L of looping conditions for P. Then, each element $(p(t_1, \ldots, t_n), \tau^+)$ of L provides an obvious looping mode for p: it suffices to take $\{i \in [1,n] \mid Var(t_i) = \varnothing\}$. But actually, we can extract more information from L. Let $p(t'_1, \ldots, t'_n)$ be an atom such that:

—for each $\langle i \mapsto u_i \rangle \in \tau^+(p)$, t'_i is a ground instance of u_i,

—for each i in $[1,n] \setminus Dom(\tau^+(p))$, $t'_i = t_i$.

Then, $p(t'_1, \ldots, t'_n)$ belongs to $p(t_1, \ldots, t_n)_{\uparrow \tau^+}$, hence it left loops w.r.t. P. Consequently, $Dom(\tau^+(p)) \cup \{i \in [1,n] \mid Var(t_i) = \varnothing\}$ is a looping mode for p. The function `looping_modes` of Fig. 1 is an application of these remarks.

Now we have the essential material for the design of a tool that attempts proofs of optimality of left terminating multi-modes computed by a termination inference tool as e.g. cTI [Mesnard and Neumerkel 2001] or TerminWeb [Genaim and Codish 2001]. For each pair $(p, \varnothing)$ in the set the function `optimal_tc` of Fig. 2 returns, we can conclude that the corresponding TM_p is *the* optimal terminating multi-mode which characterizes the operational behavior of p with respect to $\mathcal{L}_{term}$.

```
looping_modes(L, p):
  in:  L: a finite set of looping conditions
       p: a predicate symbol
  out: a looping multi-mode for p
   1:  LM_p := ∅
   2:  for each (p(t_1, ..., t_n), τ+) ∈ L do
   3:      m_p := Dom(τ+(p)) ∪ {i ∈ [1, n] | Var(t_i) = ∅}
   4:      LM_p := LM_p ∪ {m_p}
   5:  return LM_p
```

Fig. 1.

```
optimal_tc(P, max, {TM_p}_{p∈Π_P}):
  in:   P: a logic program
        max: a non-negative integer
        {TM_p}_{p∈Π_P}: a finite set of terminating multi-modes
  out:  a finite set of pairs (p, M_p) such that p ∈ Π_P and
        M_p is a multi-mode for p with no information w.r.t. its left behaviour
  note: if for each p ∈ Π_p, M_p = ∅, then {TM_p}_{p∈Π_P} is optimal
   1:  Res := ∅
   2:  L := infer_loop_cond(P, max)
   3:  for each p ∈ Π_P do
   4:      LM_p := looping_modes(L, p)
   5:      M_p := modes(p) \ (less_general(TM_p) ∪ more_general(LM_p))
   6:      Res := Res ∪ {(p, M_p)}
   7:  return Res
```

Fig. 2.

Example 5.8. (Example 4.8 continued) We apply our algorithm to the program `APPEND3` of Example 4.4. We get that

$$L := \{\ (\mathit{append}([X_1|X_2], X_3, [X_1|X_4]),\ \tau_1^+),\\ (\mathit{append3}(X_1, X_2, X_3, X_4),\ \tau_2^+),\\ (\mathit{append3}([], X_1, X_2, X_3),\ \tau_3^+)\ \}$$

is a finite set of looping conditions for `APPEND3` (see Example 4.8) with

$$\begin{aligned} \mathit{Dom}(\tau_1^+(\mathit{append})) &= \{2\}\\ \mathit{Dom}(\tau_2^+(\mathit{append3})) &= \{2,3,4\}\\ \mathit{Dom}(\tau_3^+(\mathit{append3})) &= \{3\} \end{aligned}$$

So, for *append* we have:

$$\begin{aligned} LM_{\mathit{append}} &:= \texttt{looping\_modes}(L, \mathit{append}) = \{\{2\}\}\\ \texttt{more\_general}(LM_{\mathit{append}}) &= \{\varnothing, \{2\}\}\\ TM_{\mathit{append}} &= \{\{1\}, \{3\}\}\\ \texttt{less\_general}(TM_{\mathit{append}}) &= \{\{1\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}\\ M_{\mathit{append}} &= \{\} \end{aligned}$$

For *append3*, we get:

—the looping mode $\{2,3,4\}$ from $(append3(X_1,X_2,X_3,X_4),\tau_2^+)$ and
—the looping mode $m_p := \{1,3\}$ from $(append3([],X_1,X_2,X_3),\tau_3^+)$ ($3 \in m_p$ because $Dom(\tau_3^+(append3)) = \{3\}$ and $1 \in m_p$ because of constant [] which is the first argument of $append3([],X_1,X_2,X_3)$.)

So, we have:

$$\begin{aligned}
LM_{append3} &:= \texttt{looping_modes}(L, append3) = \{\{2,3,4\},\{1,3\}\}\\
\texttt{more_general}(LM_{append3}) &= \{\varnothing,\{1\},\{2\},\{3\},\{4\},\{1,3\},\{2,3\},\{2,4\},\\
&\quad \{3,4\},\{2,3,4\}\}\\
TM_{append3} &= \{\{1,2\},\{1,4\}\}\\
\texttt{less_general}(TM_{append3}) &= \{\{1,2\},\{1,4\},\{1,2,3\},\{1,2,4\},\{1,3,4\},\\
&\quad \{1,2,3,4\}\}\\
M_{append3} &= \{\}
\end{aligned}$$

Hence in both cases, we have characterized the left behaviour of the predicates by using two complementary tools. □

5.3 An Experimental Evaluation

We have implemented[2] the algorithms presented in Sections 4 and 5.2. The binary unfoldings algorithm is derived from the one described in [Codish and Taboch 1999], where we added time stamps to precisely control what is computed at each iteration. Looping modes are computed starting from the leaves of the call graph then moving up to its roots. The cTI termination inference tool is detailed in [Mesnard and Neumerkel 2001; Mesnard and Bagnara 2004]. Here is the configuration we used for this experiment: Intel 686, 2.4GHz, 512Mb, Linux 2.4, SICStus Prolog 3.10.1, 24.8 MLips. Timings in seconds are average over 10 runs.

First we have applied them on some small programs from standard benchmarks of the termination analysis literature [Plümer 1990; Apt and Pedreschi 1994; De Schreye and Decorte 1994] (predefined predicates were erased). The column *opt?* of Table I indicates whether the result of cTI (see [Mesnard and Neumerkel 2001]) is proved optimal (✓) or not (*?*). The column *max* gives the least non-negative integer implying optimality or the least non-negative integer n where it seems we get the most precise information from non-termination inference (*i.e.* for n and $n+1$, the analyser delivers the same results). Then timings in seconds (t[s]) appear, followed by a pointer to a comment to the notes below.

Notes:

(1) The predicate `fold/3` is defined by:

```
fold(X,[],X).
fold(X,[Y|Ys],Z) :- op2(X,Y,V), fold(V,Ys,Z).
```

When the predicate `op2/3` is defined by the fact `op2(A,B,C)`, the result of cTI is optimal. When the predicate `op2/3` is defined by the fact `op2(a,b,c)`, no

[2]Available from `http://www.univ-reunion.fr/~gcc`

Table I. Some De Schreye's, Apt's, and Plümer's programs.

		cTI		Optimal			
program	top-level predicate	term-cond	t[s]	opt?	max	t[s]	cf.
`permute`	permute(X,Y)	X	0.01	✓	1	0.01	
`duplicate`	duplicate(X,Y)	$X \vee Y$	0.01	✓	1	0.01	
`sum`	sum(X,Y,Z)	$X \vee Y \vee Z$	0.01	✓	1	0.01	
`merge`	merge(X,Y,Z)	$(X \wedge Y) \vee Z$	0.02	✓	1	0.01	
`dis-con`	dis(X)	X	0.02	✓	2	0.01	
`reverse`	reverse(X,Y,Z)	X	0.02	✓	1	0.01	
`append`	append(X,Y,Z)	$X \vee Z$	0.01	✓	1	0.01	
`list`	list(X)	X	0.01	✓	1	0.01	
`fold`	fold(X,Y,Z)	Y	0.01	?	2	0.01	note 1
`lte`	goal	1	0.01	✓	1	0.01	
`map`	map(X,Y)	$X \vee Y$	0.01	✓	2	0.01	
`member`	member(X,Y)	Y	0.01	✓	1	0.01	
`mergesort`	mergesort(X,Y)	0	0.04	?	2	0.01	note 2
`mergesort_ap`	mergesort_ap(X,Y,Z)	Z	0.08	?	2	0.02	
`naive_rev`	naive_rev(X,Y)	X	0.02	✓	1	0.01	
`ordered`	ordered(X)	X	0.01	✓	1	0.01	
`overlap`	overlap(X,Y)	$X \wedge Y$	0.01	✓	2	0.01	
`permutation`	permutation(X,Y)	X	0.03	✓	1	0.01	
`quicksort`	quicksort(X,Y)	X	0.05	✓	1	0.01	
`select`	select(X,Y,Z)	$Y \vee Z$	0.01	✓	1	0.01	
`subset`	subset(X,Y)	$X \wedge Y$	0.01	✓	2	0.01	
`sum_peano`	sum(X,Y,Z)	$Y \vee Z$	0.01	✓	1	0.01	
`pl2.3.1`	p(X,Y)	0	0.01	?	1	0.01	note 3
`pl3.5.6`	p(X)	X	0.01	✓	2	0.01	
`pl4.4.6a`	perm(X,Y)	X	0.02	✓	1	0.01	
`pl4.5.2`	s(X,Y)	0	0.03	✓	1	0.01	
`pl4.5.3a`	p(X)	0	0.01	✓	1	0.01	
`pl5.2.2`	turing(X,Y,Z,T)	0	0.08	?	2	0.03	note 4
`pl7.2.9`	mult(X,Y,Z)	$X \wedge Y$	0.02	✓	4	0.03	note 5
`pl7.6.2a`	reach(X,Y,Z)	0	0.02	?	1	0.01	note 6
`pl7.6.2b`	reach(X,Y,Z,T)	0	0.02	?	1	0.01	
`pl7.6.2c`	reach(X,Y,Z,T)	$Z \wedge T$	0.02	?	2	0.02	
`pl8.3.1a`	minsort(X,Y)	X	0.03	✓	2	0.02	
`pl8.4.1`	even(X)	X	0.02	✓	2	0.01	
`pl8.4.2`	e(X,Y)	X	0.05	✓	3	0.04	

looping mode is found and the result of cTI is indeed sub-optimal as the query `fold(X,Y,Z)` terminates.

(2) Termination proofs for `mergesort` require the *list-size* norm, while cTI applies the *term-size* norm.

(3) The result of cTI is not optimal. The analyzed program:

```
p(A,B) :- q(A,C),p(C,B).
p(A,A).
q(a,b).
```

has finite binary unfoldings because there is no function symbol. Hence its termination is decidable (see [Codish and Taboch 1999]). This could be easily

detected at analyze time. We notice that no looping mode is found. But as any constant is mapped to 0 by the *term-size* norm, the modes $modes(p)$ remain undecided for cTI while they all terminate.

(4) The analyzed program (from [Plümer 1990], p. 64) simulates a Turing machine. The result of cTI is optimal.

(5) With respect to the program:

```
mult(s(A),B,C) :- mult(A,B,D),add(D,B,C).
mult(0,A,0).

add(s(A),B,s(C)) :- add(A,B,C).
add(0,A,A).
```

the query `mult(s(s(0)),A,B)` is automatically detected as looping, although `mult(0,A,B)` and `mult(s(0),A,B)` do terminate.

(6) These three programs propose various definitions of the reachability relation between two nodes in a list of edges. For the first and the third definition, cTI is indeed optimal. For the second one, cTI is not optimal.

Next, we have applied the couple of analyzers to some middle-sized Prolog programs, see Table II. Again, predefined predicates were all erased, while they are usually taken into account for cTI which of course improves the analysis. In other words, we only consider the logic programming skeleton of each program. The first two columns give the name of the program and its size (number of clauses). The fourth column indicates the running time (in seconds) of the termination analysis. Assuming that in well-written programs each predicate has at least one terminating mode, the third column is the ratio of predicates for which a non-false termination condition is computed over the total number of predicates defined in the program. For instance, cTI is able to show that there is at least one terminating mode for 48% of the predicates defined in the logic programming skeleton of the program `ann`. We ran the non-termination analyzer with $1 \leq max \leq 3$ iterations. For each value of max, we give the running time (in seconds) and the ratio of predicates for which looping modes complement terminating modes. For example, with respect to the program `ann`, for $max = 3$ we get the full *complete* mode termination behavior of 74% of all the defined predicates of the logic programming skeleton of the program. Consider now the second row of Table II, which describes the result of the combined analysis for the logic programming skeleton of the program `bid`. Here, cTI is able to find at least one non-false termination condition for each predicate. For $max = 3$, the non-termination analysis shows that 95% of the termination condition inferred by cTI are optimal. The remaining 5% indicates weakness of at least one analyzer. For instance, cTI does not generate termination proofs based on lexicographic ordering and some basic loop patterns are not caught by the non-termination component, see example 6.2. Finally, consider the result given for the logic programming skeleton of program `boyer`: cTI is able to find a non-false termination condition for 84% of the predicates. Then for $max = 3$ our non-termination analysis shows that each termination condition is optimal, so 16% of the predicates have no terminating mode.

Table II. Some middle-sized programs.

program		cTI		Optimal					
				max=1		max=2		max=3	
name	clauses	Q%	t[s]	Opt%	t[s]	Opt%	t[s]	Opt%	t[s]
`ann`	177	48	1.00	46	0.14	68	1.34	74	32.4
`bid`	50	100	0.14	55	0.02	90	0.08	95	0.50
`boyer`	136	84	0.30	80	0.03	96	0.22	100	3.66
`browse`	30	53	0.26	46	0.03	80	0.18	100	6.05
`credit`	57	100	0.11	91	0.02	95	0.11	100	4.46
`peephole`	134	88	1.08	23	0.06	70	3.62	70	406
`plan`	29	100	0.11	68	0.02	81	0.09	81	0.37
`qplan`	148	61	1.13	50	0.11	79	1.60	81	1911
`rdtok`	55	44	0.65	44	0.11	88	40.2	?	> 3600
`read`	88	52	1.72	39	0.04	47	0.80	47	10.9
`warplan`	101	32	0.49	37	0.07	83	0.99	91	21.5

We note that when we increase max, we obtain better results but the running times also increase, which is fairly obvious. For $max = 3$, we get good to optimal results but the binary unfoldings approach reveals its potentially explosive nature: we aborted the analysis of `rdtok` after one hour of computation.

In conclusion, from such a naive implementation, we were rather surprised by the quality of the combined analysis. Adopting some more clever implementation schemes, for instance computing the binary unfoldings in a demand driven fashion, could be investigated to improve the running times.

6. RELATED WORKS

Some extensions of the Lifting Theorem with respect to infinite derivations are presented in [Gori and Levi 1997], where the authors study numerous properties of finite failure. The non-ground finite failure set of a logic program is defined as the set of possibly non-ground atoms which admit a fair finitely failed SLD-tree w.r.t. the program. This denotation is shown correct in the following sense. If two programs have the same non-ground finite failure set, then any ground or non-ground goal which finitely fails w.r.t. one program also finitely fails w.r.t. the other. Such a property is false when we consider the standard ground finite failure set. The proof of correctness of the non-ground finite failure semantics relies on the following result. First, a derivation is called non-perpetual if it is a fair infinite derivation and there exists a finite depth from which unfolding does not instantiate the original goal any more. Then the authors define the definite answer goal of a non-perpetual derivation as the maximal instantiation of the original goal. A crucial lemma states that any instance of the definite answer goal admits a similar non-perpetual derivation. Compared to our work, note that we do not need fairness as an hypothesis for our results. On the other hand, investigating the relationships between non-ground arguments of the definite answer and neutral arguments is an interesting problem.

In [Shen et al. 2003], the authors present a dynamic approach to characterize (in the form of a necessary and sufficient condition) termination of general logic programs. Their technique employs some key dynamic features of an infinite gen-

eralized SLDNF-derivation, such as repetition of selected subgoals and recursive increase in term size.

Loop checking in logic programming is also a subject related to our work. In this area, [Bol et al. 1991] sets up some solid foundations. A loop check is a device to prune derivations when it seems appropriate. A loop checker is defined as *sound* if no solution is lost. It is *complete* if all infinite derivations are pruned. A complete loop check may also prune finite derivations. The authors show that even for function-free programs (also known as Datalog programs), sound and complete loop checks are out of reach. Completeness is shown only for some restricted classes of function-free programs.

We now review loop checking in more details. To our best knowledge, among all existing loop checking mechanisms only OS-check [Sahlin 1993], EVA-check [Shen 1997] and VAF-check [Shen et al. 2001] are suitable for logic programs with function symbols. They rely on a structural characteristic of infinite SLD-derivations, namely, the growth of the size of some generated subgoals. This is what the following theorem states.

THEOREM 6.1. *Consider an infinite SLD-derivation ξ where the leftmost selection rule is used. Then there are infinitely many queries Q_{i_1}, Q_{i_2}, ... (with $i_1 < i_2 < \ldots$) in ξ such that for any $j \geq 1$, the selected atom A_{i_j} of Q_{i_j} is an ancestor of the selected atom $A_{i_{j+1}}$ of $Q_{i_{j+1}}$ and $size(A_{i_{j+1}}) \geq size(A_{i_j})$.*

Here, *size* is a given function that maps an atom to its size which is defined in terms of the number of symbols appearing in the atom. As this theorem does not provide any sufficient condition to detect infinite SLD-derivations, the three loop checking mechanisms mentioned above may detect finite derivations as infinite. However, these mechanisms are *complete w.r.t. the leftmost selection rule i.e.* they detect all infinite loops when the leftmost selection rule is used.

OS-check (for OverSize loop check) was first introduced by Shalin [Sahlin 1990; 1993] and was then formalized by Bol [Bol 1993]. It is based on a function *size* that can have one of the three following definitions: for any atoms A and B, either $size(A) = size(B)$ or $size(A)$ (resp. $size(B)$) is the count of symbols appearing in A (resp. B) or $size(A) \leq size(B)$ if for each i, the count of symbols of the i-th argument of A is smaller than or equal to that of the i-th argument of B. OS-check says that an SLD-derivation may be infinite if it generates an atomic subgoal A that is *oversized*, *i.e.* that has ancestor subgoals which have the same predicate symbol as A and whose size is smaller than or equal to that of A.

EVA-check (for Extented Variant Atoms loop check) was introduced by Shen [Shen 1997]. It is based on the notion of *generalized variants* (if G_i and G_j $(i < j)$ are two goals in an SLD-derivation, an atom A in G_j is a generalized variant of an atom A' in G_i if A is a variant of A' except for some arguments whose size increases from A' to A via a set of recursive clauses.) EVA-check says that an SLD-derivation may be infinite if it generates an atomic subgoal A that is a generalized variant of some of its ancestor A'. Here the size function that is used applies to predicate arguments, *i.e.* to terms, and it is fixed: it is defined as the the count of symbols that appear in the terms. EVA-check is more reliable than OS-check because it is less likely to mis-identify infinite loops [Shen 1997]. This is mainly due to the

fact that, unlike OS-check, EVA-check refers to the informative internal structure of subgoals.

VAF-check (for Variant Atoms loop check for logic programs with Functions) was proposed by Shen *et al.* [Shen et al. 2001]. It is based on the notion of *expanded variants*. An atom A is an expanded variant of an atom A' if, after variable renaming, A becomes B that is the same as A' except that there may be some terms at certain positions in A', each $A'[i]\dots[k]$ of which grows in B into a function $B[i]\dots[k] = f(\dots, A'[i]\dots[k], \dots)$ (here, we use $A'[i]\dots[k]$ (resp. $B[i]\dots[k]$) to refer to the k-th argument of $\dots$ of the i-th argument of A' (resp. B)). VAF-check says that an SLD-derivation may be infinite if it generates an atomic subgoal A that is an expanded variant of some of its ancestor A'. VAF-check is as reliable as and more efficient than EVA-check [Shen et al. 2001].

Note that the loop checking mechanisms described above are "top-down" whereas our approach is "bottom-up". Another difference with our work is that we want to infer atomic queries which are guaranteed to be left looping. Hence, we consider *sufficient* conditions for looping, in contrast to the above mentioned methods which consider *necessary* conditions. Our technique returns a set of queries for which it has pinpointed *one* infinite derivation. Hence, we are not interested in soundness as we do not care for finite derivations, nor in completeness, as the existence of just one infinite derivation suffices. Of course, using the Δ-subsumption test as a loop checker leads to a device that is neither sound (since Δ-subsumption is a particular case of subsumption) nor complete (since the Δ-subsumption test provides a sufficient but not necessary condition). This is illustrated by the following example.

Example 6.2. Let $c := p(X, X) \leftarrow p(f(X), f(X))$. As the arguments of the head of c have one common variable X, every set of positions with associated terms τ^+ that is DN for $\{c\}$ is such that the domain of $\tau^+(p)$ is empty (see **(DN1)** in Definition 3.34). Notice that from the head $p(X, X)$ of c we get

$$p(X, X) \underset{c}{\Longrightarrow} p(f(X), f(X)) \underset{c}{\Longrightarrow} \cdots \underset{c}{\Longrightarrow} p(f^n(X), f^n(X)) \underset{c}{\Longrightarrow} \cdots$$

As the arguments of p grow from step to step, there cannot be any query in the derivation that is $\Delta[\tau^+]$-more general than one of its ancestors. Consequently, we cannot conclude that $p(X, X)$ left loops w.r.t $\{c\}$. □

On the other hand, using loop checking approaches to infer classes of atomic left looping queries is not satisfactory because, as we said above, non-looping queries may be mis-identified as looping.

Example 6.3. We cannot replace, in Corollary 3.2, the subsumption test by the expanded variant test used in VAF-check because, for instance, in the clause $c := p(a) \leftarrow p(f(a))$, we have: $p(f(a))$ is an expanded variant of $p(a)$, but $p(a)$ does not loop w.r.t. c.

Finally, [De Schreye et al. 1990] is also related to our study. In this paper, the authors describe an algorithm for detecting non-terminating queries to clauses of the type $p(\cdots) \leftarrow p(\cdots)$. The algorithm is able to check if such a given clause has no non-terminating queries or has a query which either loops or fails due to occur check. Moreover, given a *linear* atomic goal (*i.e.* a goal where all variable occurs at most once), the algorithm is able to check if the goal loops or not w.r.t. the clause. The

technique of the algorithm is based on directed weighted graphs [Devienne 1990] and on a necessary and sufficient condition for the existence of non-terminating queries to clauses of the type $p(\cdots) \leftarrow p(\cdots)$. This condition is proved in [De Schreye et al. 1989] and is expressed in terms of rational trees.

7. CONCLUSION

We have presented an extension of the relation "is more general than" which allows to disregard some arguments, termed neutral arguments, while checking for subsumption. We have proposed two syntactic criteria for statically identifying neutral arguments. From these results, in the second part of this report we have described algorithms for automating non-termination analysis of logic programs. Finally, we have applied these techniques to check the optimality of termination conditions for logic programs.

This paper leaves numerous questions open. For instance, it might be interesting to try to generalize this approach to constraint logic programming [Jaffar and Lassez 1987]. Can we obtain higher level proofs compared to those we give? Can we propose more abstract criteria for identifying neutral arguments? A first step in this direction is presented in [Payet and Mesnard 2004] where completeness of a weaker DN concept is studied. Also, our work aims at inferring classes of atomic left looping queries, using a bottom-up point of view. Experimental data show that it may sometimes lead to prohibitive time/space costs. How can we generate only the useful binary clauses without fully computing the iterations of this T_P-like operator? Or can we adapt our algorithms towards a more efficient correct top-down approach for checking non-termination?

ACKNOWLEDGMENTS

We thank Ulrich Neumerkel for numerous discussions on this topic, Roberto Bagnara and anonymous referees for interesting suggestions.

REFERENCES

Apt, K. R. 1997. *From Logic Programming to Prolog*. Prentice Hall.

Apt, K. R. and Pedreschi, D. 1994. Modular termination proofs for logic and pure Prolog programs. In *Advances in Logic Programming Theory*, G. Levi, Ed. Oxford University Press, 183–229.

Arts, T. and Zantema, H. 1996. Termination of logic programs using semantic unification. In *Logic Program Synthesis and Transformation*. Lecture Notes in Computer Science, vol. 1048. Springer-Verlag, Berlin. TALP can be used online at `http://bibiserv.techfak.uni.biekefeld.de/talp/`.

Bol, R. N. 1993. Loop checking in partial deduction. *Journal of Logic Programming 16*, 25–46.

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

Clark, K. L. 1979. Predicate logic as a computational formalism. Tech. Rep. Doc 79/59, Logic Programming Group, Imperial College, London.

Codish, M. and Taboch, C. 1999. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming 41,* 1, 103–123.

De Schreye, D., Bruynooghe, M., and Verschaetse, K. 1989. On the existence of nonterminating queries for a restricted class of Prolog-clauses. *Artificial Intelligence 41*, 237–248.

De Schreye, D. and Decorte, S. 1994. Termination of logic programs : the never-ending story. *Journal of Logic Programming 19-20*, 199–260.

De Schreye, D., Verschaetse, K., and Bruynooghe, M. 1990. A practical technique for detecting non-terminating queries for a restricted class of Horn clauses, using directed, weighted graphs. In *Proc. of ICLP'90*. The MIT Press, 649–663.

Deransart, P. and Ferrand, G. 1987. Programmation en logique avec négation: présentation formelle. Tech. Rep. 87/3, Laboratoire d'Informatique, Département de Mathématiques et d'Informatique, Université d'Orleans.

Dershowitz, N., Lindenstrauss, N., Sagiv, Y., and Serebrenik, A. 2001. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering,Communication and Computing 12,* 1/2, 117–156.

Devienne, P. 1988. Weighted graphs, a tool for expressing the behavious of recursive rules in logic programming. I. for New Generation Computer Technology (ICOT), Ed. OHMSHA Ltd. Tokyo and Springer-Verlag, 397–404. Proc. of the Inter. Conf. on Fifth Generation Computer Systems 88, Tokyo, Japan.

Devienne, P. 1990. Weighted graphs: A tool for studying the halting problem and time complexity in term rewriting systems and logic programming. *Theoretical Computer Science 75,* 2, 157–215.

Devienne, P., P.Lebègue, and Routier, J.-C. 1993. Halting problem of one binary Horn clause is undecidable. In *LNCS*. Vol. 665. Springer-Verlag, 48–57. Proc. of STACS'93.

Gabbrielli, M. and Giacobazzi, R. 1994. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the ACM Symposium on applied computing*. ACM Press, 394–399.

Genaim, S. and Codish, M. 2001. 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. TerminWeb can be used online from `http://www.cs.bgu.ac.il/~codish`.

Gori, R. and Levi, G. 1997. Finite failure is and-compositional. *Journal of Logic and Computation 7,* 6, 753–776.

Jaffar, J. and Lassez, J. L. 1987. Constraint logic programming. In *Proc. of the ACM Symposium on Principles of Programming Languages*. ACM Press, 111–119.

Lindenstrauss, N. 1997. TermiLog: a system for checking termination of queries to logic programs. `http://www.cs.huji.ac.il/~naomil`.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Springer-Verlag.

M. Falaschi, G. Levi, M. M. and Palamidessi, C. 1993. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation 102,* 1, 86–113.

Mesnard, F. 1996. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, M. J. Maher, Ed. MIT Press, 7–21.

Mesnard, F. and Bagnara, R. 2004. cTI: a constraint-based termination inference tool for ISO-Prolog. *Theory and Practice of Logic Programming*. To appear.

Mesnard, F. and Neumerkel, U. 2000. cTI: a tool for inferring termination conditions of ISO-Prolog. `http://www.univ-reunion.fr/~gcc`.

Mesnard, F. and Neumerkel, U. 2001. Applying static analysis techniques for inferring termination conditions of logic programs. In *Static Analysis Symposium*, P. Cousot, Ed. Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, Berlin, 93–110.

Mesnard, F., Payet, E., and Neumerkel, U. 2002. Detecting optimal termination conditions of logic programs. In *Proc. of the 9th International Symposium on Static Analysis*, M. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, Berlin, 509–525.

O'Keefe, R. 1990. *The Craft of Prolog*. MIT Press.

Payet, E. and Mesnard, F. 2004. Non-termination inference of logic programs. In *Proc. of the 11th International Symposium on Static Analysis*, R. Giacobazzi, Ed. Lecture Notes in Computer Science, vol. 3148. Springer-Verlag, Berlin.

Plümer, L. 1990. *Terminations proofs for logic programs*. Number 446 in LNAI. Springer-Verlag, Berlin.

Sahlin, D. 1990. The mixtus approach to automatic partial evaluation of full Prolog. In *Proc. of the 1990 North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, MA, 377–398.

Sahlin, D. 1993. Mixtus: an automatic partial evaluator for full Prolog. *New Generation Computing 12,* 1, 7–51.

Schmidt-Schauss, M. 1988. Implication of clauses is undecidable. *Theoretical Computer Science 59*, 287–296.

Shen, Y.-D. 1997. An extended variant of atoms loop check for positive logic programs. *New Generation Computing 15,* 2, 187–204.

Shen, Y.-D., You, J.-H., Yuan, L.-Y., Shen, S., and Yang, Q. 2003. A dynamic approach to characterizing termination of general logic programs. *ACM Transactions on Computational Logic 4,* 4, 417–434.

Shen, Y.-D., Yuan, L.-Y., and You, J.-H. 2001. Loops checks for logic programs with functions. *Theoretical Computer Science 266,* 1-2, 441–461.

A. AN ALGORITHM TO COMPUTE DN FILTERS

`dna`$\big(BinProg,\ \tau_1^+\big)$:

in: $BinProg$: a finite set of binary clauses
τ_1^+: a set of positions with associated terms

1: $\tau_2^+ := \tau_1^+$
2: $\tau_2^+ :=$ `satisfy_DN123`$(BinProg, \tau_2^+)$
3: **while** `satisfy_DN4`$(BinProg, \tau_2^+) \neq \tau_2^+$ **do**
4: $\quad\tau_2^+ :=$ `satisfy_DN4`$(BinProg, \tau_2^+)$
5: **return** τ_2^+

`satisfy_DN123`$\big(BinProg,\ \tau_1^+\big)$:

1: $\tau_2^+ := \tau_1^+$
2: **for each** $p(s_1, \ldots, s_n) \leftarrow B \in BinProg$ **do**
3: $\quad E := \{i \in [1, n] \mid Var(s_i) \cap Var(\{s_j \mid j \neq i\}) = \varnothing\}$
4: $\quad\tau_2^+(p) := \tau_2^+(p)|(Dom(\tau_2^+(p)) \cap E)$
5: **for each** $p(s_1, \ldots, s_n) \leftarrow B \in BinProg$ **do**
6: $\quad F := \varnothing$
7: $\quad$**for each** $i \in Dom(\tau_2^+(p))$ **do**
8: $\quad\quad u_i :=$ `less_general`$(s_i, \tau_2^+(p)(i))$
9: $\quad\quad$**if** $u_i = undefined$ **then** $F := F \cup \{i\}$
10: $\quad\quad$**else** $\tau_2^+(p)(i) := u_i$
11: $\quad\tau_2^+(p) := \tau_2^+(p)|(Dom(\tau_2^+(p)) \setminus F)$
12: **for each** $H \leftarrow q(t_1, \ldots, t_m) \in BinProg$ **do**
13: $\quad F := \varnothing$
14: $\quad$**for each** $i \in Dom(\tau_2^+(q))$ **do**
15: $\quad\quad$**if** $\tau_2^+(q)(i)$ is not more general than t_i **then** $F := F \cup \{i\}$
16: $\quad\tau_2^+(q) := \tau_2^+(q)|(Dom(\tau_2^+(q)) \setminus F)$
17: **return** τ_2^+

`satisfy_DN4`$\big(BinProg,\ \tau_1^+\big)$:

1: $\tau_2^+ := \tau_1^+$
2: **for each** $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_m) \in BinProg$ **do**
3: $\quad F := \varnothing$
4: $\quad$**for each** $i \in Dom(\tau_2^+(p))$ **do**
5: $\quad\quad$**for each** $j \in [1, m] \setminus Dom(\tau_2^+(q))$ **do**
6: $\quad\quad\quad$**if** $Var(s_i) \cap Var(t_j) \neq \varnothing$ **then** $F := F \cup \{i\}$
7: $\quad\tau_2^+(p) := \tau_2^+(p)|(Dom(\tau_2^+(p)) \setminus F)$
8: **return** τ_2^+

B. AN ALGORITHM TO COMPUTE LOOPING CONDITIONS

`unit_loop`($H \leftarrow B$, $Dict$):

in: $H \leftarrow B$: a binary clause
$Dict$: a loop dictionary
1: $Dict' := Dict$
2: $\tau^+ := \texttt{dna}([H \leftarrow B], \tau^+_{max})$
3: **if** B is $\Delta[\tau^+]$-more general than H **then**
4: $\quad Dict' := Dict' \cup \{([H \leftarrow B], \tau^+)\}$
5: **return** $Dict'$

`loops_from_dict`($H \leftarrow B$, $Dict$):

in: $H \leftarrow B$: a binary clause
$Dict$: a loop dictionary
1: $Dict' := Dict$
2: **for each** $\left([H_1 \leftarrow B_1 | BinSeq_1], \tau_1^+\right) \in Dict$ **do**
3: **if** B is $\Delta[\tau_1^+]$-more general than H_1 **then**
4: $\quad \tau^+ := \texttt{dna}([H \leftarrow B, H_1 \leftarrow B_1 | BinSeq_1], \tau_1^+)$
5: $\quad Dict' := Dict' \cup \{([H \leftarrow B, H_1 \leftarrow B_1 | BinSeq_1], \tau^+)\}$
6: **return** $Dict'$

`infer_loop_dict`(P, max):

in: P: a logic program
max: a non-negative integer
1: $Dict := \varnothing$
2: **for each** $H \leftarrow B \in T_P^\beta \uparrow max$ **do**
3: $\quad Dict := \texttt{unit\_loop}(H \leftarrow B, Dict)$
4: $\quad Dict := \texttt{loops\_from\_dict}(H \leftarrow B, Dict)$
5: **return** $Dict$

`infer_loop_cond`(P, max):

in: P: a logic program
max: a non-negative integer
1: $L := \varnothing$
2: $Dict := \texttt{infer\_loop\_dict}(P, max)$
3: **for each** $([H \leftarrow B | BinSeq], \tau^+) \in Dict$ **do**
4: $\quad L := L \cup \{(H, \tau^+)\}$
5: **return** L