Detecting Optimal Termination Conditions of Logic Programs

Fred Mesnard[1], Etienne Payet[1], and Ulrich Neumerkel[2]

[1] Iremia - Université de La Réunion, France
{fred,epayet}@univ-reunion.fr
[2] Institut für Computersprachen - T. U. Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract. In this paper, we begin with an approach to non-termination inference of logic programs. Our framework relies on an extension of the Lifting Theorem, where some specific argument positions can be instantiated while others are generalized. Atomic left looping queries are generated bottom-up from selected subsets of the binary unfoldings of the program of interest. Then non-termination inference is tailored to attempt proofs of optimality of left termination conditions computed by a termination inference tool. For each class of atomic queries not covered by a termination condition, the aim is to ensure the existence of one query from this class which leads to an infinite search tree. An experimental evaluation is reported. When termination and non-termination analysis produce complementary results for a logic procedure, they induce a *characterization* of the operational behavior of the logic procedure with respect to the left most selection rule and the language used to describe sets of atomic queries.

1 Introduction

Since the work of N. Lindenstrauss on TermiLog [16, 10], several automatic tools for termination checking (e.g. TALP [4]) or termination inference (e.g. cTI [18, 19] or TerminWeb [13]) are now available to the logic programmer. One of them is even included in the Mercury compiler [24]. 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, then 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. The aim is to ensure the existence, for each class of atomic queries not covered by a termination condition, of *one* query from this class which leads to an infinite search tree. The main contributions of this work are:

- A generalization of the Lifting Theorem from the Logic Programming theory. The Lifting Theorem, at the heart of the completeness proof of SLD-resolution (see e.g. [3]), states that a SLD-derivation of $Q\theta$ can be lifted to a SLD-derivation ξ of Q. We prove that some specific arguments of Q, called "derivation neutral", can be instantiated as well, while retaining the existence of a lifted derivation ξ', where the length of ξ and ξ' are identical.
- A new application of binary unfoldings to left loop inference. [12] 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 [7] while abstracting the standard operational semantics associated to SLD-resolution. We present an algorithm to infer left looping classes of atomic goals, where such classes are computed bottom-up from selected subsets of the binary unfoldings of the analyzed program.
- An algorithm which, when combined with termination inference [17], may detect optimal left termination conditions expressed in $\mathcal{L}_{term}$ for logic programs.

We organize the paper as follows: Section 2 presents the notations. Then we define in Section 3 what we call an optimal termination condition. In Section 4 we propose an extension of the Lifting Theorem. We concentrate on non-termination inference in Section 5 and optimality proofs of termination conditions in Section 6.

2 Preliminaries

2.1 Logic Programming

We try to strictly adhere to the notations, definitions, and results presented in [1].
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 $[1, n] = \varnothing$.
Let $\mathcal{L}$ be a language of programs. We assume that $\mathcal{L}$ contains an infinite number of constant symbols including *void*. The set of relation symbols of $\mathcal{L}$ is Π, and we assume that each relation symbol p has an *unique* arity, denoted $arity(p)$. $TU_{\mathcal{L}}$ (resp. $TB_{\mathcal{L}}$) denotes the set of all (ground and non ground) terms of $\mathcal{L}$ (resp. atoms of $\mathcal{L}$). A *query* (or *goal*) $\mathbf{A}$ is a finite sequence of atoms $A_1, \ldots, A_n$ (where $n \geq 0$). Let t be a term. Then $Var(t)$ denotes the set of variables occurring in t.
A *logic program* is a finite set of definite 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 universal left termination. Consider a non-empty query $A_1, A_2, \ldots, A_n$ and a clause c. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint[3] with $A_1, A_2, \ldots, A_n$ and assume that A_1 and H unify. Let θ be an mgu of A_1

[3] More generally, a variant c' of c satisfying the *standardization apart* condition: c' has to be variable disjoint from the initial query, the substitutions and the clauses used so far in the computation.

and H. Then $A_1, A_2, \ldots, A_n \overset{\theta}{\underset{c}{\Longrightarrow}} (\mathbf{B}, A_2, \ldots, A_n)\theta$ 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. We write $Q \overset{+}{\underset{P}{\Longrightarrow}} Q'$ (resp. $Q \overset{*}{\underset{P}{\Longrightarrow}} Q'$) to summarize a finite number (> 0) (resp. ≥ 0) of left derivation steps from Q to Q', where each input clause is a variant of a clause of P.
Let Q be a query. A *left derivation* of $\{Q\} \cup P$ is a *maximal* sequence of left derivation steps starting from the query Q, where each input clause is a variant of a clause of P. 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 *left terminates* (resp. *left loops*) with respect to P if every left derivation of $\{Q\} \cup P$ is finite (resp. there exists an infinite left derivation of $\{Q\} \cup P$). We recall that for logic programs, left termination is instantiation-closed: if Q left terminates with respect to P, then $Q\theta$ left terminates with respect to P' for any substitution θ and any $P' \subseteq P$. Similarly, left looping is generalization-closed: if there exists θ such that $Q\theta$ left loops with respect to P', then Q left loops with respect to any $P \supseteq P'$.

2.2 The binary unfoldings of a logic program

Let us present the main ideas about the *binary unfoldings* [12] of a logic program, borrowed from [7]. This technique transforms a logic program P (without any query of interest) 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) necessary leads to a call to B (or its corresponding instance). More precisely, let G be an atomic query. Then A is a *call* in a left derivation of $\{G\} \cup P$ if $G \overset{+}{\underset{P}{\Longrightarrow}} A, \mathbf{B}$. We denote by $calls_P(G)$ the set of calls which occur in the left derivations of $\{G\} \cup P$. 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 $p(x_1, \ldots, x_n) \leftarrow p(x_1, \ldots, x_n)$ for any $p \in \Pi_P$, where $arity(p) = n$.

$$
\begin{aligned}
T_P^\beta(X) = \{(H \leftarrow B)\theta \mid\ & c := H \leftarrow B_1, \ldots, B_m \in P,\ i \in [1, m], \\
& \langle H_j \leftarrow true \rangle_{j=1}^{i-1} \in X \text{ renamed apart from } c, \\
& H_i \leftarrow B \in X \cup id \text{ renamed apart from } c, \\
& i < m \Rightarrow B \neq true \\
& \theta = mgu(\langle B_1, \ldots, B_i \rangle, \langle H_1, \ldots, H_i \rangle)\}
\end{aligned}
$$

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 $\{G\} \cup P$ can be characterized as follows: $calls_P(G) = \{B\theta | H \leftarrow B \in bin\_unf(P), \theta = mgu(G, H)\}$. 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 goal G with respect to P is identical to universal termination of G 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 [7]:

Theorem 1 (Codish and Taboch, 99). *Let P be a program and G an atomic goal. Then G left loops with respect to P iff G 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 sections 5 and 6, we compute only the first max iterations of T_P^β where max is a parameter of the analysis. As an immediate consequence of Theorem 1 frequently used in our proofs, assume that we detect that G loops with respect to a subset of the binary clauses of $T_P^\beta \uparrow i$, with $i \in N$. Then G loops with respect to $bin\_unf(P)$ hence G left loops with respect to P.

Example 1. Consider the following program P:

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

The binary unfoldings of P are:

$T_{P'}^\beta \uparrow 0 = \varnothing$
$T_{P'}^\beta \uparrow 1 = \{p(x,z) \leftarrow p(y,z), p(x,x) \leftarrow true, q(a,b) \leftarrow true\} \cup T_{P'}^\beta \uparrow 0$
$T_{P'}^\beta \uparrow 2 = \{p(a,b) \leftarrow true, p(x,y) \leftarrow q(x,y), p(x,y) \leftarrow q(z,y)\} \cup T_{P'}^\beta \uparrow 1$
$T_{P'}^\beta \uparrow 3 = \{p(x,b) \leftarrow q(x,a), p(x,b) \leftarrow q(y,a)\} \cup T_{P'}^\beta \uparrow 2$
$T_{P'}^\beta \uparrow 4 = T_{P'}^\beta \uparrow 3 = bin\_unf(P')$

The mere existence of the clause $p(x,z) \leftarrow p(y,z) \in T_P^\beta \uparrow 1$ implies that $\{p(x,b)\} \cup \{p(x,z) \leftarrow p(y,z)\}$ loops. Hence $\{p(x,b)\} \cup P$ left loops.

3 Optimal termination conditions

Let P be a logic program and p be a relation symbol $\in \Pi_P$, with $arity(p) = n$. First, we describe the language $\mathcal{L}_{term}$ presented in Section 1 for abstracting sets of atomic queries:

Definition 1 (Mode). *A* mode m_p *for p is a subset of $[1,n]$, and denotes the following set of atomic goals: $[m_p] = \{p(t_1,\ldots,t_n) \in TB_\mathcal{L} |\ \forall i \in m_p\ Var(t_i) = \varnothing\}$. 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 2 (Multi-mode). *A* multi-mode *for p is a 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 termination and looping condition:

Definition 3 (Terminating mode, termination condition). *A* terminating mode m_p *for* p *is a mode for* p *such that any query* $\in [m_p]$ *left terminates with respect to* P. *A* termination condition TC_p *for* p *is a set of terminating modes for* p.

Definition 4 (Looping mode, looping condition). *A* looping mode m_p *for* p *is a mode for* p *such that there exists a query* $\in [m_p]$ *which left loops with respect to* P. *A* looping condition L_p *for* p *is a 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 for p. Similarly, as left looping is generalization-closed, any mode that is "above" (more general than) a looping mode is also a looping mode for p. Let us be more precise:

Definition 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 termination conditions:

Definition 6 (Optimal termination condition). *An* optimal *termination condition* TC_p *for* p *is a termination condition for* p *such that there exists a looping condition* L_p *verifying:*

$$modes(p) = less\_general(TC_p) \cup more\_general(L_p)$$

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

Example 2. Consider the program `APPEND`:

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

A well-known termination condition is $TC_{append} = \{\{1\}, \{3\}\}$. 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:

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

On the other hand, `append(Xs,[],Zs)` left loops. Hence $L_{append} = \{\{2\}\}$ is a looping condition and $more\_general(L_{append}) = \{\varnothing, \{2\}\}$. Since $modes(append) = less\_general(TC_{append}) \cup more\_general(L_{append})$, we conclude that the termination condition TC_{append} is optimal.

We have already presented a tool for inferring termination conditions in [19]. We now describe the concepts underlying the inference of looping modes.

4 Neutral arguments for left derivation

A basic idea in the work we present lies in identifying arguments in clauses which we can disregard when unfolding a query. For instance, the second argument of the non-unit clause of `append` in Example 2 is such a candidate. Moreover, a very common programming technique called *accumulator passing* (see for instance e.g. [20], p. 21–25), always produces such patterns.
We first give a technical tool to describe specific arguments inside a program and present a generalization of the relation "is an instance of". In Subsection 4.2, we formalize the concept of derivation neutrality. Subsection 4.3 gives the main result, in the form of a generalized Lifting Theorem, with an application to loop checking.

4.1 Sets of positions

Definition 7 (Set of positions). *A* set of positions *is a mapping τ that maps each predicate symbol $p \in \Pi$ to a subset of $[1, arity(p)]$.*

Example 3 (Example 2 continued). If we want to disregard the second argument of the relation symbol *append*, we set $\tau := \langle append \longmapsto \{2\}\rangle$.

Definition 8 (τ-instance and τ-generalization). *Let τ be a set of positions. We make use of the following relations:*

- *The relation $=_\tau$:*

$$A =_\tau B \quad iff \quad \begin{cases} A = p(s_1, \ldots, s_n) \\ B = p(t_1, \ldots, t_n) \\ \forall i \in [1, n] \setminus \tau(p),\ t_i = s_i \end{cases}$$

$$A_1, \ldots, A_n =_\tau B_1, \ldots, B_m \quad iff \quad \begin{cases} n = m \\ \forall i \in [1, n],\ A_i =_\tau B_i \end{cases}$$

- *The relation "is a τ-instance of": Q is a τ-instance of Q' iff there exists a substitution η such that $Q =_\tau Q'\eta$.*
- *The relation "is a τ-generalization": Q is a τ-generalization of Q' iff Q' is a τ-instance of Q.*

Example 4 (Example 3 continued). Since $\tau = \langle append \longmapsto \{2\}\rangle$, we do not care of what happens to the second argument: $append([1], 2, [3, 4])$ is a τ-instance of $append([1|x], f(x), [3|z])$, with $\eta = \{x/[], z/[4]\}$. Otherwise stated, $append([1|x], f(x), [3|z])$ is a τ-generalization of the atom $append([1], 2, [3, 4])$.

Finally we give a bunch of obvious definitions:

Definition 9 (Ordering sets of positions).

- *$\tau \subseteq \tau'$ if for each relation symbol p in Π, $\tau(p) \subseteq \tau'(p)$.*
- *$\tau \subset \tau'$ if $\tau \subseteq \tau'$ and $\tau \neq \tau'$.*
- *τ_{min} is the set of positions verifying: for each p in Π, $\tau_{min}(p) = \varnothing$ and τ_{max} is the set of positions verifying: for each p in Π, $\tau_{max}(p) = [1, arity(p)]$.*

4.2 DN sets of positions

We give here a precise definition of the kind of arguments we are interested in. The name "derivation neutral" stems from the fact that τ-arguments do not play any rôle in the derivation process. The next subsection formalizes this intuition.

Definition 10 (Derivation Neutral). *A set of positions τ is* DN *for a clause $p(s_1, \ldots, s_n) \leftarrow Body$ if:*

$$\forall i \in \tau(p), \begin{cases} s_i \text{ is a variable} \\ s_i \text{ occurs only once in } p(s_1, \ldots, s_n) \\ \text{for each } q(t_1, \ldots, t_m) \in Body,\ [s_i \in Var(t_j) \Rightarrow j \in \tau(q)] \end{cases}.$$

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

Example 5 (Example 4 continued). The set of positions $\tau = \langle append \longmapsto \{2\}\rangle$ is DN for `C2`, the recursive clause defining *append*, but is *not* DN for the program `APPEND` since `Ys` appears twice in the unit clause `C1`.

The preceding notion is closed under renaming:

Proposition 1. *Let τ be a set of positions, c be a clause, and c' be a variant of c. If τ is DN for c then τ is DN for c'.*

4.3 Left derivation and DN sets of positions

Our goal here is to generalize the Lifting Theorem of Logic Programming (see Sections 3.4 and 3.5 of [1], p. 56–60) in the following sense: while lifting a left derivation, we may safely ignore derivation neutral arguments which can be instantiated to any terms. As a consequence, loop detection with DN sets of positions generalizes loop detection with the subsumption test (take $\tau := \langle p \longmapsto \varnothing\rangle$ for any p). Proofs can be found in the long version of this paper available at `www.univ-reunion.fr/~gcc/papers`.

Theorem 2 (τ-Lifting). *Let $\xi := Q \underset{c_1}{\Longrightarrow} Q_1 \underset{c_2}{\Longrightarrow} Q_2 \underset{c_3}{\Longrightarrow} \cdots$ be a left derivation and Q' be a τ-generalization of Q. Then there exists a left derivation*

$$\xi' : Q' \underset{c_1}{\Longrightarrow} Q'_1 \underset{c_2}{\Longrightarrow} Q'_2 \underset{c_3}{\Longrightarrow} \cdots$$

where for each $Q_i \in \xi$, the corresponding Q'_i is a τ-generalization of Q_i.

Example 6. Let P be a logic program. Let τ be a DN set of positions for P, with $\tau(p) = \{2\}$. Assume that there exists a successful left derivation of $\{p(s,t)\} \cup P$. Then we hold a similar left derivation when generalizing s, whatever the second argument is: for any term s' (including s) which generalizes s, for *any* term $u \in TU_{\mathcal{L}}$, there exists a left derivation of $\{p(s',u)\} \cup P$.

As a consequence of Theorem 2, we get the following result that we use in the correctness proofs of the algorithms of Section 5.

Corollary 1. *If* $(A \overset{*}{\underset{P}{\Longrightarrow}} B_1, \mathbf{B}_1 \ ; \ B_1 \overset{+}{\underset{P}{\Longrightarrow}} B_2, \mathbf{B}_2)$ *with* B_1 *a* τ*-instance of* B_2*, then* $P \cup \{A\}$ *left loops.*

Example 7. Let $c := p(f(X), Y) \leftarrow p(X, g(Y))$ be a binary clause. Then $\tau := \langle p \mapsto \{2\}\rangle$ is a DN set of positions for c. Notice that we have $p(f(X), Y) \overset{\theta}{\underset{c}{\Longrightarrow}} p(X', g(Y'))$ where $p(f(X'), Y') \leftarrow p(X', g(Y'))$ is the input clause and $\theta = \{X/X', Y/Y'\}$. Applying Corollary 1, as $p(f(X), Y)$ is a τ-instance of $p(X', g(Y'))$, we get that $p(f(X), Y)$ left loops w.r.t. $\{c\}$. We point out that we do not get this result from the classical Lifting Theorem as $p(f(X), Y)$ is not an instance of $p(X', g(Y'))$.

4.4 DN sets of positions for binary programs

We present below an algorithm for computing DN sets of positions. Its correctness is discussed in the long version of this paper. We shall show in the next section that we can incrementally build selected sets of binary clauses, together with their corresponding DN sets of positions. So, although the algorithm below can be generalized to arbitrary logic programs, we only consider binary programs, *i.e.* finite sets of binary clauses. Moreover, our interest lies in defining an *incremental* algorithm for computing DN sets of positions.

`dna`$(BinProg, \tau)$:

in: $BinProg$: a finite set of binary clauses
τ: a set of positions
out: a DN set of positions $\tau' \subseteq \tau$

1: $\tau' := \tau$
2: **while** `dna_one_step`$(BinProg, \tau') \neq \tau'$ **do**
3: $\quad \tau' :=$`dna_one_step`$(BinProg, \tau')$
4: **return** τ'

`dna_one_step`$(BinProg, \tau)$:

1: $\tau' := \tau$
2: **for each** $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) \in BinProg$ **do**
3: $\quad E := \{i \in [1, n] \mid s_i$ is a variable that occurs only once in $p(s_1, \ldots, s_n)\}$
4: $\quad F := \varnothing$
5: $\quad$ **for each** $i \in \tau'(p) \cap E$ **do**
6: $\quad\quad$ **for each** $j \in [1, n'] \setminus \tau'(q)$ **do**
7: $\quad\quad\quad$ **if** $s_i \in Var(t_j)$ **then** $F := F \cup \{i\}$
8: $\quad \tau'(p) := (\tau'(p) \cap E) \setminus F$
9: **return** τ'

Example 8 (Example 2 continued). `dna`$(\{$`C2`$\}, \tau_{max}) = \langle append \mapsto \{2\}\rangle$.

5 Inferring looping modes

In this section, we give a set of algorithms that allow to infer looping modes for the predicate symbols defined in a given logic program.
Let P be a logic program, parametric for the subsections which follow.

5.1 Looping modes from one binary clause

Let p be an n-ary relation symbol and m_p be a mode for p. Suppose that we want to prove that m_p is looping. Assume that we hold a binary clause $c := p(s_1, \ldots, s_n) \leftarrow p(t_1, \ldots, t_n) \in bin\_unf(P)$ and τ a DN set of positions for c. Very intuitively, we have $p(s_1, \ldots, s_n) \underset{c}{\Longrightarrow} p(t_1, \ldots, t_n)$. If $p(s_1, \ldots, s_n)$ is a τ-instance of $p(t_1, \ldots, t_n)$, then, by the τ-Lifting Theorem 2, there exists a query Q_1 such that $p(t_1, \ldots, t_n) \underset{c}{\Longrightarrow} Q_1$ and $p(t_1, \ldots, t_n)$ is a τ-instance of Q_1. So, by Corollary 1, $p(t_1, \ldots, t_n)$ loops w.r.t. $\{c\}$. Now, to show that m_p is a looping mode, we can try to instantiate the variables of $\{s_i | i \in m_p\}$ by a grounding substitution σ. If σ only affects the arguments of $p(t_1, \ldots, t_n)$ that are neutral with respect to derivation, then, by the τ-Lifting Theorem 2, $p(t_1, \ldots, t_n)\sigma$ also loops w.r.t. $\{c\}$. But, very intuitively, we have $p(s_1, \ldots, s_n)\sigma \underset{c}{\Longrightarrow} p(t_1, \ldots, t_n)\sigma$. Therefore, $p(s_1, \ldots, s_n)\sigma$ loops w.r.t. $\{c\}$. Hence, as $c \in bin\_unf(P)$, $p(s_1, \ldots, s_n)\sigma$ loops w.r.t. $bin\_unf(P)$, so, by Theorem 1, $p(s_1, \ldots, s_n)\sigma$ left loops w.r.t. P. Hence m_p is a looping mode.
The function `unit_loop` below relies on the above remarks. Its termination relies on that of `dna` and its partial correctness follows from that of `dna` and the result:

Theorem 3. *Let $p \in \Pi_P$, m_p be a mode of p, and $c \in bin\_unf(P)$.*
If $\mathtt{unit\_loop}(m_p, c) \neq \textbf{\textit{false}}$, there exists $A \in [m_p]$ such that A left loops w.r.t. P.

`unit_loop`(m_p, c):

in: m_p: a mode of p and c: a binary clause $\in bin\_unf(P)$
out: a pair $(\tau, \{c\})$, where τ is a DN set of positions for $\{c\}$ if c allows to classify m_p as a looping mode or **false**

1: $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) := c$
2: $\tau :=$ `dna`$(\{c\}, \tau_{max})$
3: **if** $p(s_1, \ldots, s_n)$ is a τ-instance of $q(t_1, \ldots, t_{n'})$ **and** $Var(\{s_i \mid i \in m_p\}) \cap Var(\{t_i \mid i \notin \tau(p)\}) = \varnothing$
4: **then return** $(\tau, \{c\})$
5: **else return false**

5.2 Looping modes from a set of binary clauses

We now introduce a data structure which we call *dictionary* and that we use both in the algorithms that follow and in their correctness proofs. A dictionary is a set of tuples $(Atom, \tau, BinProg)$ where $Atom \in TB_{\mathcal{L}}$, $BinProg$ is a set of binary clauses and τ a set of positions. Moreover:

Definition 11 (D). *A dictionary Dict enjoys the property* ***D*** *if Dict is a finite set such that for any* $(Atom, \tau, BinProg) \in Dict$ *we have:* $BinProg$ *is a finite subset of* $bin\_unf(P)$, τ *is DN for* $BinProg$, *and* $Atom$ *loops w.r.t.* $BinProg$.

Assume we hold: a tuple $(q(u_1, \ldots, u_{n'}), \tau, BP)$ of a dictionary satisfying **D**; a binary clause $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) \in bin\_unf(P)$; and we would like to prove that a given mode m_p is looping. If $q(t_1, \ldots, t_{n'})$ is a τ-generalization of $q(u_1, \ldots, u_{n'})$, then, by the τ-Lifting Theorem 2, $q(t_1, \ldots, t_{n'})$ loops w.r.t. BP. Then, to show that m_p is a looping mode, we can reason as in Subsection 5.1.
The function `loop_with_dict` relies on the above remarks. Its termination relies on that of `dna` and finiteness of *Dict* and its partial correctness follows from that of `dna` and the result below.

`loop_with_dict`(m_p, c, $Dict$):

in: m_p: a mode of p, c: a binary clause $\in bin\_unf(P)$ and $Dict$: a dictionary satisfying **D**

out: a pair $(\tau, BinProg)$, where τ is a DN set of positions for $Binprog \subseteq bin\_unf(P)$, which allows to classify m_p as a looping mode or **false**

1: $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) := c$
2: **for each** $(q(u_1, \ldots, u_{n'}), \tau, BinProg) \in Dict$ **do**
3: **if** $\begin{cases} q(u_1, \ldots, u_{n'}) \text{ is a } \tau\text{-instance of } q(t_1, \ldots, t_{n'}) \text{ and} \\ Var(\{s_i \mid i \in m_p\}) \cap Var(\{t_i \mid i \notin \tau(q)\}) = \varnothing \end{cases}$
4: **then return** (`dna`$(BinProg \cup \{c\}, \tau), BinProg \cup \{c\})$
5: **return false**

Theorem 4. *Let* $p \in \Pi_P$, m_p *be a mode of* p, *and* $c \in bin\_unf(P)$. *If Dict satisfies* ***D*** *and* `loop_with_dict`$(m_p, c, Dict) \neq$ ***false*** *then there exists* $A \in [m_p]$ *such that* A *left loops w.r.t.* P.

5.3 Looping modes for a predicate

The function we use to infer looping modes for a predicate symbol is given in Figure 1. Our algorithm maintains the following invariant:

Lemma 1. ***D*** *always holds for* $Dict'$.

Concerning termination, note that calls to `modes`, `unit_loop`, `more_general` and `loop_with_dict` fulfill their specifications hence terminate. Since both M_p and $BinProg$ are finite sets, termination is ensured. Partial correctness is a consequence of Lemma 1 and partial correctness of the functions `unit_loop` and `loop_with_dict`.

`infer_looping_modes_pred` ($BinProg$, p, $Dict$):

in: $BinProg$: a finite set of binary clauses $\subseteq bin\_unf(P)$,
p: a relation symbol $\in \Pi_P$ and $Dict$: a dictionary satisfying **D**
out: a pair $(L_p, Dict')$ where L_p is a looping condition for p and $Dict'$ is a dictionary satisfying **D**

```
1:  M_p := modes(p), L_p := ∅ and Dict' := Dict
2:  for each c := p(s_1,...,s_n) ← B ∈ BinProg with B ≠ true do
3:      for each m_p ∈ M_p do
        /* NB: M_p is modified line 6 and line 11 */
4:          if unit_loop(m_p, c) ≠ false then
5:              (τ, BP) := unit_loop(m_p, c)
6:              M_p := M_p\more_general(m_p) /* cf. Definition 5 */
7:              L_p := L_p ∪ {m_p}
8:              Dict' := Dict' ∪ {(p(s_1,...,s_n), τ, BP)}
9:          elsif loop_with_dict(m_p, c, Dict') ≠ false then
10:             (τ, BP) := loop_with_dict(m_p, c, Dict')
11:             M_p := M_p\more_general(m_p)
12:             L_p := L_p ∪ {m_p}
13:             Dict' := Dict' ∪ {(p(s_1,...,s_n), τ, BP)}
14: return (L_p, Dict')
```

Fig. 1. Inference of looping modes for a predicate symbol.

5.4 Looping modes for a logic program

The top-level function we use to infer looping modes for each predicate symbol of any logic program P is given in Figure 2. Notice that Π_P is finite and, for any non-negative integer max, $T_P^\beta \uparrow max$ is a finite set $\subseteq bin\_unf(P)$. Line 2, $Dict$ is initialized to $\varnothing$ which satisfies **D**. Hence all calls to `infer_looping_modes_pred` fulfill their specification. This shows termination and partial correctness of the function `infer_looping_modes_prog`. We point out that correctness is independent of whether the relation symbols are analyzed according a topological sort of the strongly connected components of the call graph of P. However, $Dict$ is always increasing and, due to the definition of binary unfoldings, inference of looping modes is much more efficient if relation symbols are processed bottom-up.

5.5 Running the algorithm

Example 9. We consider the program `APPEND3`:

```
append3(X,Y,Z,T):-append(X,Y,W),append(W,Z,T).
```

augmented by the `APPEND` program. $T^\beta_{\texttt{APPEND3}} \uparrow 2$ includes:

`infer_looping_modes_prog`(P, max):

in: P: a logic program and max: an non-negative integer
out: a set of pairs (p, L_p) where, for each $p \in \Pi_P$, L_p is a looping condition for p

1: $BinProg :=$ the binary clauses of $T_P^\beta \uparrow max$
2: $Dict := \varnothing$ and $Res := \varnothing$
3: **for each** $p \in \Pi_P$ **do**
4: $\quad (L_p, Dict) :=$ `infer_looping_modes_pred`($BinProg$, p, $Dict$)
5: $\quad Res := Res \cup \{(p, L_p)\}$
6: **return** Res

Fig. 2. The top-level function for inferring looping modes.

```
append([A|B],C,[A|D]):-append(B,C,D).      % C3
append3(A,B,C,D):-append(A,B,E).           % C4
append3([],A,B,C):-append(A,B,C).          % C5
```

The dictionary $Dict$, built from `C3` while processing `append`:

$$\{(append([x_1|x_2], x_3, [x_1|x_4]),\ \tau_1 = \langle append \longmapsto \{2\}\rangle,\ \{append([x_1|x_2], x_3, [x_1|x_4]) \leftarrow append(x_2, x_3, x_4)\})\}$$

shows the looping mode $\{2\}$, including, with all its τ_1-generalizations, the query: `append([A|B],void,[A|C])`. For `append3`, from `C4` and `C5`, the updated dictionary $Dict' = Dict \cup$

$$\{(append3(x_1, x_2, x_3, x_4),\ \tau_2 = \langle append3 \longmapsto \{2,3,4\}, append \longmapsto \{2\}\rangle,\ \{append3(x_1, x_2, x_3, x_4) \leftarrow append(x_1, x_2, x_5),\ append([x_1|x_2], x_3, [x_1|x_4]) \leftarrow append(x_2, x_3, x_4)\}),$$
$$(append3([], x_1, x_2, x_3),\ \tau_3 = \langle append3 \longmapsto \{3\}, append \longmapsto \{2\}\rangle,\ \{append3([], x_1, x_2, x_3) \leftarrow append(x_1, x_2, x_3),\ append([x_1|x_2], x_3, [x_1|x_4]) \leftarrow append(x_2, x_3, x_4)\})\}$$

allows the elimination of the looping modes $\{2,3,4\}$ and $\{1,3\}$ including the queries:

- `append3(A,void,void,void)` with all its τ_2-generalizations and
- `append3([],A,void,B)` with all its τ_3-generalizations.

Note that we do not have to guess the constant [] for the last query as it appears naturally in the binary unfoldings of `APPEND3`.

6 Proving optimality of termination conditions

It turns out that a slight modification of `infer_looping_modes_prog` gives a function which may prove the optimality (see Definition 6) of

termination conditions (as computed by a tool for termination inference, e.g. cTI [19] or TerminWeb [13]). For each pair $(p, \varnothing)$ in the set the function returns, we can conclude that the corresponding TC_p is *the* optimal termination condition which characterizes the operational behavior of p with respect to $\mathcal{L}_{term}$. Termination and partial correctness rely on similar arguments than those in Subsections 5.3 and 5.4.

`optimal_tc`(P, max, $\{TC_p\}_{p\in\Pi_P}$):

in: P: a logic program, max: an non-negative integer and $\{TC_p\}_{p\in\Pi_P}$: a set of termination conditions
out: a set of pair (p, M_p) where, for each $p \in \Pi_P$, M_p is a multi-mode of p with no information with respect to its left behavior
note: If for each $p \in \Pi_P$, $M_p = \varnothing$, then $\{TC_p\}_{p\in\Pi_P}$ is optimal

1: $BinProg := T_P^\beta \uparrow max$, $Dict := \varnothing$ and $Res := \varnothing$
2: **for each** $p \in \Pi_P$ **do**
3: $\quad(L_p, Dict) :=$ `infer_looping_modes_pred`($BinProg$, p, $Dict$)
4: $\quad M_p :=$ `modes`$(p)\backslash($`less_general`$(TC_p) \cup$ `more_general`$(L_p))$
5: $\quad Res := Res \cup \{(p, M_p)\}$
6: **return** Res

Example 10. We apply our algorithm to the program `APPEND3` of Subsection 5.5 (see also Example 2). We get, for `append`:

$$\begin{aligned}
L_{append} &= \{\{2\}\} \\
\texttt{more_general}(L_{append}) &= \{\varnothing, \{2\}\} \\
TC_{append} &= \{\{1\}, \{3\}\} \\
\texttt{less_general}(TC_{append}) &= \{\{1\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}, \\
M_{append} &= \{\}
\end{aligned}$$

For `append3`, we have:

$$\begin{aligned}
L_{append3} &= \{\{1,3\}, \{2,3,4\}\} \\
\texttt{more_general}(L_{append3}) &= \{\varnothing, \{1\}, \{2\}, \{3\}, \{4\}, \{1,3\}, \{2,3\}, \{2,4\}, \\
&\quad \{3,4\}, \{2,3,4\}\} \\
TC_{append3} &= \{\{1,2\}, \{1,4\}\} \\
\texttt{less_general}(TC_{append3}) &= \{\{1,2\}, \{1,4\}, \{1,2,3\}, \{1,2,4\}, \{1,3,4\}, \\
&\quad \{1,2,3,4\}\} \\
M_{append3} &= \{\}
\end{aligned}$$

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

7 An experimental evaluation

We have implemented[4] the algorithms presented in Sections 4, 5 and 6. Then we have applied them on some small programs from standard benchmarks of the termination analysis literature [21, 2, 9] (predefined predicates were erased). Here is the configuration we used for this experiment (see Table 1): PowerPC, 333MHz, 192Mb, Linux 2.2, SICStus Prolog 3.8.5, 3.8 MLips. Timings are average over 10 runs. The column *opt?* indicates whether the result of cTI is proved optimal (y) or not ($?$). The column *max* gives the least non-negative integer implying optimality (or the least non-negative integer where it seems we get the most precise information from non-termination inference). Then timings appear, followed by a pointer to a comment to the notes below.

Notes:

1. The result of cTI is optimal.
2. For this program (and the following), cTI does not compute an optimal termination condition for `split`. Nevertheless, the termination condition for `mergesort` is shown optimal.
3. The result of cTI is not optimal.
4. This is an example where the binary unfoldings reveal its explosive nature. The analyzed program P (from [21], p. 64), simulates in Prolog a Turing machine. The first iteration $T_P^\beta \uparrow 1$ generates more than one hundred non-unit binary clauses. The second iteration generates more than two thousands new non-unit binary clauses.
5. `mult(s(s(0)),A,B)` and `mult(s(s(s(0))),A,void)` are detected as left looping, although the queries `mult(0,A,B)` and `mult(s(0),A,B)` do *not* left loop.

8 Conclusion

To our best knowledge, there is no other automated analysis dealing with optimality proofs of termination conditions for logic programs.

Some extensions of the Lifting Theorem with respect to infinite derivations are presented in [14], where the authors study properties of finite failure.

Loop checking in logic programming is a subject related to non-termination, where Bol [5] sets up some solid foundations (see also [23]). 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. Bol shows that even for function-free programs (also known as Datalog programs), sound and complete loop checks are out of reach. If such a mechanism is to be included into a logic programming system, then Bol advocates and studies sound loop checkers. Completeness is shown only for some restricted classes of function-free programs. Loop checking is also important for partial deduction [15]. In this case, Bol emphasizes complete loop checkers, which were also studied in [6, 22].

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

Table 1. Some De Schreye's, Apt's, and Plümer's programs.

		cTI		Optimal			
program	top-level predicate	term-cond	time[s]	opt?	max	time[s]	cf.
`permute`	permute(x,y)	x	0.24	**y**	1	0.01	
`duplicate`	duplicate(x,y)	$x \vee y$	0.09	**y**	1	0.01	
`sum`	sum(x,y,z)	$x \vee y \vee z$	0.26	**y**	1	0.01	
`merge`	merge(x,y,z)	$(x \wedge y) \vee z$	0.39	?	1	0.01	note 1
`dis-con`	dis(x)	x	0.35	**y**	1	0.01	
`reverse`	reverse(x,y,z)	x	0.14	**y**	1	0.01	
`append`	append(x,y,z)	$x \vee z$	0.14	**y**	1	0.01	
`list`	list(x)	x	0.05	**y**	1	0.01	
`fold`	fold(x,y,z)	y	0.15	?	1	0.01	note 1
`lte`	goal	1	0.19	**y**	1	0.01	
`map`	map(x,y)	$x \vee y$	0.13	**y**	2	0.01	
`member`	member(x,y)	y	0.06	**y**	1	0.01	
`mergesort`	mergesort(x,y)	x	0.75	**y**	2	0.04	note 2
`mergesort_ap`	mergesort_ap(x,y,z)	$x \vee z$	1.20	**y**	2	0.10	
`naive_rev`	naive_rev(x,y)	x	0.19	**y**	1	0.01	
`ordered`	ordered(x)	x	0.07	**y**	1	0.01	
`overlap`	overlap(x,y)	$x \wedge y$	0.08	**y**	1	0.01	
`permutation`	permutation(x,y)	x	0.25	**y**	1	0.01	
`quicksort`	quicksort(x,y)	x	0.59	**y**	1	0.03	
`select`	select(x,y,z)	$y \vee z$	0.12	**y**	1	0.01	
`subset`	subset(x,y)	$x \wedge y$	0.14	?	1	0.01	note 1
`sum_peano`	sum(x,y,z)	$y \vee z$	0.19	**y**	1	0.01	
`pl2.3.1`	p(x,y)	0	0.03	?	1	0.01	note 3
`pl3.5.6`	p(x)	x	0.09	**y**	2	0.01	
`pl4.4.6a`	perm(x,y)	x	0.19	**y**	1	0.01	
`pl4.5.2`	s(x,y)	0	0.25	**y**	1	0.04	
`pl4.5.3a`	p(x)	0	0.02	**y**	1	0.01	
`pl5.2.2`	turing(x,y,z,t)	0	3.41	?	1	0.50	note 4
`pl7.2.9`	mult(x,y,z)	$x \wedge y$	0.33	**y**	3	0.03	note 5
`pl7.6.2a`	reach(x,y,z)	0	0.22	?	1	0.01	note 1
`pl7.6.2b`	reach(x,y,z,t)	0	0.35	?	1	0.01	note 3
`pl7.6.2c`	reach(x,y,z,t)	$z \wedge t$	0.46	**y**	2	3.00	
`pl8.3.1a`	minsort(x,y)	x	0.36	**y**	2	0.04	
`pl8.4.1`	even(x)	x	0.19	**y**	1	0.01	
`pl8.4.2`	e(x,y)	x	0.78	**y**	1	0.02	

The main difference with our work is that we want to pinpoint *some* infinite derivations that we build bottom-up. We are not interested in completeness nor in soundness. Moreover, in [11], the undecidability of the halting problem for programs with one binary clause and one atomic query is shown. This clearly puts an upper bound on what one can expect to do.
Nonetheless, we point out that the combination of termination inference and non-termination inference may give a strong result for the program being analyzed. Although the two methods are both incomplete, when their results are complementary, it implies that each analysis is optimal. Altogether they can sometimes *characterize* the operational behavior of logic programs with respect to the left most selection rule and the language used to describe classes of atomic queries.
More work is needed to refine the implementation into an efficient analyzer. In particular, the binary unfoldings need to be either computed with care or abstracted, due to the potential exponential number of binary clauses it may generate. How to take the predefined predicates into account is another problem to solve. Finally, for rational trees, we note that [8] provides an undecidable necessary and sufficient condition for the existence of a query which loops with respect to a binary clause. Moving to other constraint structures seems a worth-while topic.

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
2. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
3. K. R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.
4. T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *Logic Program Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996. TALP can be used online at `http://bibiserv.techfak.uni.biekefeld.de/talp/`.
5. R. Bol. *Loop Checking in Logic Programming*. PhD thesis, CWI, Amsterdam, 1991.
6. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
7. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
8. D. De Schreye, M. Bruynooghe, and K. Verschaetse. On the existence of nonterminating queries for a restricted class of Prolog-clauses. *Artificial Intelligence*, 41:237–248, 1989.
9. D. De Schreye and S. Decorte. Termination of logic programs : the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.

10. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering,Communication and Computing*, 12(1/2):117–156, 2001.
11. P. Devienne, P.Lebègue, and J-C. Routier. Halting problem of one binary Horn clause is undecidable. In *LNCS*, volume 665, pages 48–57. Springer-Verlag, 1993. Proc. of STACS'93.
12. M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the ACM Symposium on applied computing*, pages 394–399. ACM Press, 1994.
13. S. Genaim and M. Codish. Inferring termination condition for logic programs using backwards analysis. In *Proceedings of Logic for Programming, Artificial intelligence and Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2001. TerminWeb can be used online from `http://www.cs.bgu.ac.il/~codish`.
14. R. Gori and G. Levi. Finite failure is and-compositional. *Journal of Logic and Computation*, 7(6):753–776, 1997.
15. H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language : a theory and implementation in the case of Prolog. In *Proc. of the 9th POPL*, pages 255–267, 1982.
16. N. Lindenstrauss. TermiLog: a system for checking termination of queries to logic programs, 1997. `http://www.cs.huji.ac.il/~naomil`.
17. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In M. J. Maher, editor, *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, pages 7–21. MIT Press, 1996.
18. F. Mesnard and U. Neumerkel. cTI: a tool for inferring termination conditions of ISO-Prolog, april 2000. `http://www.complang.tuwien.ac.at/cti`.
19. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In P. Cousot, editor, *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 93–110. Springer-Verlag, Berlin, 2001.
20. R. O'Keefe. *The Craft Of Prolog*. MIT Press, 1990.
21. L. Plümer. *Terminations proofs for logic programs*. Number 446 in LNAI. Springer-Verlag, Berlin, 1990.
22. Y-D. Shen, L-Y. Yuan, and J-H. You. Loops checks for logic programs with functions. *Theoretical Computer Science*, 266(1-2):441–461, 2001.
23. D. Skordev. An abstract approach to some loop detection problems. *Fundamenta Informaticae*, 31:195–212, 1997.
24. C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. Van Hentenryck, editor, *Proc. of the International Static Analysis Symposium*, volume 1302 of *LNCS*, pages 160–171. Springer-Verlag, 1997.