Non-Termination Inference for Optimal Termination Conditions of Logic Programs
Frédéric Mesnard, Etienne Payet, Ulrich Neumerkel

To cite this version:
Frédéric Mesnard, Etienne Payet, Ulrich Neumerkel. Non-Termination Inference for Optimal Termination Conditions of Logic Programs. 11èmes Journées Francophones de Programmation en Logique et Programmation par Contraintes (JFPLC’02), Association Française pour la Programmation en Logique et la programmation par Contraintes (AFPLC), May 2002, Nice, France. pp.84-104. hal-01915407

HAL Id: hal-01915407
https://hal.univ-reunion.fr/hal-01915407
Submitted on 7 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Non-Termination Inference for Optimal Termination Conditions of Logic Programs

Fred Mesnard* — Etienne Payet* — Ulrich Neumerkel**

* Iremia - Université de La Réunion, France
{fred.epayet}@univ-reunion.fr
** Institut für Computersprachen - Technische Universität Wien, Austria
ulrich@mips.complang.tuwien.ac.at

ABSTRACT. In this paper, we present 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 then 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.

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.

KEYWORDS: logic programming, static analysis, non-termination, optimal termination condition.

RÉSUMÉ. Dans cet article, nous présentons une technique d’inférence de conditions de non-termination de programmes logiques. Notre travail repose sur une extension du “Lifting Theorem”, où des positions d’argument spécifiques peuvent être instanciées alors que les autres sont généralisées. Des requêtes atomiques qui bouclent à gauche sont alors générées de façon ascendante à partir de sous-ensembles des déploiements binaires du programme traité.

L’inférence de non-termination est alors utilisée pour tester l’optimalité de conditions de terminaison gauche générées par un outil d’inférence de terminaison. Pour chaque classe de requêtes atomiques non couverte par une condition de terminaison, nous tentons d’assurer l’existence d’une requête de cette classe qui mène à un arbre de recherche adjacent.

Quand les analyses de terminaison et de non-terminaison produisent des résultats complémentaires pour une procédure logique, on obtient une caractérisation du comportement opérationnel de la procédure par rapport à la règle de sélection gauche et au langage utilisé pour décrire les ensembles de requêtes atomiques.

MOTS-CLES : programmation logique, analyse statique, non-terminaison, condition optimale de terminaison.
1. Introduction

Since the work of N. Lindenstrauss on TermiLog [LIN 97, DER 01], several automatic tools for termination checking (e.g., TALP [ART 96]) or termination inference (e.g., cTI [MES 00, MES 01] or TerminWeb [GEN 01]) are now available to the logic programmer. One of them is even included in the Mercury compiler [SPE 97]. 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. Then 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 $L_{\text{term}}$.

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. [APT 82]), states that a SLD-derivation of $Q\emptyset$ can be lifted to a SLD-derivation $\xi$ 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 $\xi'$, where the length of $\xi$ and $\xi'$ are identical.

– A new application of binary unfoldings to left loop inference. [GAB 94] 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 [COD 99] 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 [MES 96], may detect optimal left termination conditions expressed in $L_{\text{term}}$ 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. Sections 4 and 5 propose an extension of the Lifting Theorem. We concentrate on non-termination inference in Section 6 and optimality proofs of termination conditions in Section 7.

2. Preliminaries

2.1. Logic Programming

We try to strictly adhere to the notations, definitions, and results presented in [APT 97]. Some of our results and our proofs are directly inspired by those written by Apt. When it is the case, we notify the reader. We recall here some basic facts.

$\mathbb{N}$ denotes the set of non-negative integers and for any $n \in \mathbb{N}$, $[1,n]$ denotes the set $\{1, \ldots, n\}$. If $n = 0$ then $[1,n] = \emptyset$. 
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 $\Pi$, and we assume that each relation symbol $p$ has an unique arity, denoted $\text{arity}(p)$. $TU_{\mathcal{L}}$ (resp. $TB_{\mathcal{L}}$) denotes the set of all (ground and non ground) terms of $\mathcal{L}$ (resp. atoms of $\mathcal{L}$). Let $A$ be an atom. Then $\text{rel}(A)$ denotes its relation symbol. A query $A$ is a finite sequence of atoms $A_1, \ldots, A_n$ (where $n \geq 0$). Let $t$ be a term. Then $\text{Var}(t)$ denotes the set of variables occurring in $t$. Let $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ be a substitution. We denote by $\text{Dom}(\theta)$ the set of variables $\{x_1, \ldots, x_n\}$, and by $\text{Ran}(\theta)$ the set of variables appearing in $t_1, \ldots, t_n$. We define $\text{Var}(\theta) = \text{Dom}(\theta) \cup \text{Ran}(\theta)$. Given a set of variables $V$, $\theta \mid V$ denotes the substitution obtained from $\theta$ by restricting its domain to $V$.

A logic program is a finite set of definite clauses. Clauses $H \leftarrow B_1, \ldots, B_n$ are written in program examples with the ISO-Prolog syntax $H := B_1, \ldots, B_n$. Let $P$ be a logic program. Then $\Pi_P$ denotes the set of relation symbols appearing in $P$. In this paper, we only focus on universal left termination. Consider a non-empty query (or goal) $A$, a and a clause $c$. Let $H \leftarrow B$ be a variant of $c$ variable disjoint with $A$, and assume that $A$ and $H$ unify. Let $\theta$ be an mgu of $A$ and $H$. Then $A, \frac{A \leftarrow B}{c} \theta$ is a left derivation step with $H \leftarrow B$ as its input clause. If the substitution $\theta$ or the clause $c$ is irrelevant, we drop a reference to it. We write $Q \xrightarrow{P}{\frac{P}{Q^1}}$ (resp. $Q \xrightarrow{P}{\frac{P}{Q^1}}$) to summarize a finite number ($> 0$) (resp. $\geq 0$) 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 $\theta$ and any $P' \subseteq P$. Similarly, left looping is generalization-closed: if there exists $\theta$ 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 [GAB 94] of a logic program, borrowed from [COD 99]. 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 $\text{true}$ 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 $G \xrightarrow{P} A, B$. We denote by $\text{calls}_P(G)$ the set of calls which occur in the left derivations of $\{G\} \cup P$. The specialization of the goal independent semantics for

1. More generally, a variant $\mathcal{c}'$ of $\mathcal{c}$ satisfying the standardization apart condition: $\mathcal{c}'$ has to be variable disjoint from the initial query, the substitutions and the clauses used so far in the computation.
call patterns for the left-to-right selection rule is given as the fixpoint of an operator $T_P^{\beta}$ over the domain of binary clauses, viewed modulo renaming. In the definition below, $\bar{d}l$ 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 $arity(p) = n$.

$$T_P^{\beta}(X) = \{ (H \leftarrow B)^\theta \mid c := H \leftarrow B_1, \ldots, B_m \in P, \ i \in [1, m],$$

$$\langle H_j \leftarrow true \rangle_{j=1}^{m+1} \in X \text{ renamed apart from } c,$$

$$H_i \leftarrow B \in X \cup \bar{d}l \text{ renamed apart from } c,$$

$$i < m \Rightarrow B \neq true$$

$$\theta = mgu\langle B_1, \ldots, B_i, \langle H_1, \ldots, H_i \rangle \rangle \}

We define its powers as usual. It can be shown that the least fixpoint of this monotonic operator always exists and we set $bin\_unf(P) := \theta P(T_P^{\beta})$. Then the calls that occur in the left derivations of $\{G\} \cup P$ can be characterized as follows: $calls_P(G) = \{ B\theta \mid H \leftarrow B \in bin\_unf(P), \theta = mgu(G, H) \}$. 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 [COD 99, GEN 01]:

**Theorem 1 (Codish and Taboada, 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 $bin\_unf(P)$.

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 $bin\_unf(P)$ (which can be an infinite set of binary clauses) hence $G$ left loops with respect to $P$.

**Example 1** Consider the following program $P$:

$$p(X, Z) :- p(Y, Z), q(X, Y). \quad p(X, X). \quad q(a, b).$$

The binary unfoldings of $P$ are:

$$T_P^{\beta} \uparrow 0 = \emptyset$$

$$T_P^{\beta} \uparrow 1 = \{ p(x, z) \leftarrow p(y, z), p(x, x) \leftarrow true, q(a, b) \leftarrow true \} \cup T_P^{\beta} \uparrow 0 \cup T_P^{\beta} \uparrow 1$$

$$T_P^{\beta} \uparrow 2 = \{ p(a, b) \leftarrow true, p(x, y) \leftarrow q(x, y), p(x, y) \leftarrow q(z, y) \} \cup T_P^{\beta} \uparrow 2$$

$$T_P^{\beta} \uparrow 3 = \{ p(x, b) \leftarrow q(x, a), p(x, b) \leftarrow q(y, b) \} \cup T_P^{\beta} \uparrow 3 \cup T_P^{\beta} \uparrow 4$$

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 $\text{arity}(p) = n$. First, we describe the language $L_{\text{term}}$ 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: $[m_p] = \{ p(t_1, \ldots, t_n) \in TB_c | \forall i \in m_p, \text{Var}(t_i) = \emptyset \}$ The set of all modes for $p$, i.e. $2^{[1,n]}$, is denoted $\text{modes}(p)$.

Note that if $m_p = \emptyset$ then $[m_p] = \{ p(t_1, \ldots, t_n) \in TB_c \}$. 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: $[M_p] = \bigcup_{m \in M_p} [m]$.

Note that if $M_p = \emptyset$, then $[M_p] = \emptyset$. 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 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:

$$\text{less_general}(M_p) = \{ m \in \text{modes}(p) | \exists m' \in M_p, m' \subseteq m \}$$
$$\text{more_general}(M_p) = \{ m \in \text{modes}(p) | \exists m' \in M_p, m \subseteq m' \}$$

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: $\text{modes}(p) = \text{less_general}(TC_p) \cup \text{more_general}(L_p)$.

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 $\text{APPEND}$:

\begin{verbatim}
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
\end{verbatim}
A well-known termination condition is \( TC_{\text{append}} = \{\{1\}, \{3\}\} \). Indeed, any query of the form \( \text{append}(t, Ys, Zs) \) or \( \text{append}(Xs, Ys, t) \), where \( t \) is a ground term (i.e. such that \( \text{Var}(t) = \emptyset \)), left terminates. We have:

\[
\text{less\_general}(TC_{\text{append}}) = \{\{1\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}
\]

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

We have already presented a tool for inferring termination conditions in [MES 01]. 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 \( \text{append} \) in Example 2 is such a candidate. Moreover, a very common programming technique called accumulator passing (see for instance e.g. [O’K 90], 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 \( \tau \):

\[
\Pi \longrightarrow 2^N
\]

\[
p \longmapsto I \subseteq [1, \text{arity}(p)]
\]

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

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

- The relation \( = \tau \):

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

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

- The relation “is a \( \tau \)-instance of”: \( Q \) is a \( \tau \)-instance of \( Q' \) iff there exists a substitution \( \eta \) such that \( Q = \tau Q' \eta \).

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

Finally we give a bunch of obvious definitions:

Definition 9 (Ordering sets of positions)
- $\tau \subseteq \tau'$ if for each relation symbol $p$ in $\Pi$, $\tau(p) \subseteq \tau'(p)$.
- $\tau \subset \tau'$ if $\tau \subseteq \tau'$ and $\tau \neq \tau'$.
- $\tau_{\min}$ is the set of positions verifying: for each $p$ in $\Pi$, $\tau_{\min}(p) = \emptyset$ and $\tau_{\max}$ is the set of positions verifying: for each $p$ in $\Pi$, $\tau_{\max}(p) = [1,\text{arity}(p)]$.

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 $\tau$-arguments do not play any rôle in the derivation process. The next subsection formalizes this intuition.

Definition 10 (Derivation Neutral) A set of positions $\tau$ is DN for a clause $p(s_1,\ldots,s_n) \leftarrow \text{Body}$: $$\forall i \in \tau(p), \begin{cases} \text{$s_i$ is a variable} \\ \text{$s_i$ occurs only once in $p(s_1,\ldots,s_n)$} \\ \text{for each $q(t_1,\ldots,t_m) \in \text{Body}$, if $s_i \in \text{Var}(t_j)$ then $j \in \tau(q)$.} \end{cases}$$

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 $\tau = \langle \text{append} \mapsto \{2\} \rangle$ is DN for the recursive clause defining \text{append}, but is not DN for the program APPEND since $\text{Ys}$ appears twice in the unit clause.

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 [APT 97], 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 $\tau := \langle p \mapsto \emptyset \rangle$ for any $p$). Proofs can be found in the long version of this paper, available at URL: www.univ-reunion.fr/~gcc/papers

Theorem 2 ($\tau$-Lifting) Let $\xi ::= Q \iff C_1 \iff Q_2 \iff \cdots$ be a left derivation and $Q'$ be a $\tau$-generalization of $Q$. Then there exists a left derivation $$\xi' : Q' \iff C_1 \iff Q'_2 \iff \cdots$$ where for each $Q_i \in \xi$, the corresponding $Q'_i$ is a $\tau$-generalization of $Q_i$. 
Example 6 Let \( P \) be a logic program. Let \( \tau \) 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 \( u \in T U_{\mathcal{L}} \), there exists a left derivation of \( \{p(s', u)\} \cup P \).

5. DN sets of positions for binary programs

We present in this section an algorithm for computing DN sets of positions. 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.

\[
\text{dna}(\text{BinProg}, \tau):
\]
\[
in: \text{BinProg}: \text{a finite set of binary clauses and } \tau: \text{a set of positions}
\]
\[
\text{out}: \text{a DN set of positions } \tau' \subseteq \tau
\]
\[
1: \quad \tau' \leftarrow \tau
\]
\[
2: \quad \text{while dna_one_step}(\text{BinProg}, \tau') \neq \tau' \text{ do}
\]
\[
3: \quad \tau' \leftarrow \text{dna_one_step}(\text{BinProg}, \tau')
\]
\[
4: \quad \text{return } \tau'
\]

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

Example 7 \( \text{dna}\{\text{append}([X|Xs], Ys, [X|Zs]) \leftarrow \text{append}(Xs, Ys, Zs)\}, \tau_{\text{max}}\) = \( \langle \text{append} \mapsto \{2\} \rangle \).

6. Inferring looping modes

Before we dive into algorithms and correctness proofs, let us try to give the intuitions. Assume we hold a binary program \( BP \subseteq \text{bin_unf}(P) \), \( \tau \) a DN set of positions for \( BP \), and an atom \( q(u_1, \ldots, u_{n'}) \) which loops with respect to \( BP \). We consider an \( n \)-ary relation symbol \( p \), a binary clause \( p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) \in \text{bin_unf}(P) \), and we would like to prove that the mode \( m_p \) is looping.

If \( q(t_1, \ldots, t_{n'}) \) is a \( \tau \)-generalization of \( q(u_1, \ldots, u_{n'}) \), then \( p(s_1, \ldots, s_n) \) loops. Now, to show that \( m_p \) is a looping mode, we can try to instantiate the variables of
Theorem 3

Figure 2. The correctness of our algorithms relies on:

6.3. The conditions (c) of Subsection 6.2: \( \text{Var}(\{s_i \mid i \in m_p\}) \cap \text{Var}(\{t_i \mid i \notin \tau(q)\}) = \emptyset \). We conclude by applying Theorem 1. Now assume that \( q = p \) with \( n' = n \). The reasoning above is of course valid.

Let \( P \) be a logic program, parametric for the subsections which follow.

6.1. Looping modes from one binary clause

\[
\text{unit_loop}(m_p, c):
\]

\[
\begin{align*}
\text{in:} & \quad m_p: \text{a mode of } P \text{ and } c: \text{a binary clause } \in \text{bin_unf}(P) \\
\text{out:} & \quad \text{a pair } (\tau, \{c\}) \text{, where } \tau \text{ is a DN set of positions for } \{c\} \text{, if } c \text{ allows to classify } m_p \text{ as a looping mode or the boolean } \text{false} \\
1: & \quad p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) : c \\
2: & \quad \tau := \text{DNA}(\{c\}, \tau_{\text{max}}) \\
3: & \quad \text{if } p(s_1, \ldots, s_n) \text{ is a } \tau\text{-instance of } q(t_1, \ldots, t_{n'}) /\# p=q, n=n' /*/ \\
& \quad \text{and } \text{Var}(\{s_i \mid i \in m_p\}) \cap \text{Var}(\{t_i \mid i \notin \tau(q)\}) = \emptyset \\
4: & \quad \text{then return } (\tau, \{c\}) \\
5: & \quad \text{else return false}
\end{align*}
\]

Termination of \( \text{unit_loop} \) relies on termination of \( \text{DNA} \). Partial correctness follows from partial correctness of \( \text{DNA} \) and the result below.

**Theorem 3** Let \( p \in \Pi_P, m_p \) be a mode of \( P \) and \( c \in \text{bin_unf}(P) \). If \( \text{unit_loop}(m_p, c) \neq \text{false} \), there exists \( A \in [m_p] \) such that \( A \) left loops w.r.t. \( P \).

6.2. Looping modes from a set of binary clauses

We now introduce a data structure which we call dictionary. It is a set of tuples \((\text{Atom}, \tau, \text{BinProg})\) where \( \text{Atom} \in TB_L, \text{BinProg} \) is a set of binary clauses and \( \tau \) a set of positions. Moreover:

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

Termination of \( \text{loop_with_dict} \) (see Figure 1) comes from finiteness of \( \text{Dict} \) and termination of \( \text{DNA} \). Partial correctness follows from partial correctness of \( \text{DNA} \) and:

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

6.3. Looping modes for a predicate

The function we use to infer looping modes for a predicate symbol is given in Figure 2. The correctness of our algorithms relies on:
Lemma 1: \( D \) always holds for \( \text{Dict}' \).

Concerning termination, note that calls to \texttt{modes.unit_loop}, \texttt{more_general} and \texttt{loop_with_dict} fulfill their specifications hence terminate. Since both \( M_p \) and \( \text{BinProg} \) are finite sets, termination is ensured. Partial correctness is a consequence of Lemma 1 and partial correctness of \texttt{unit_loop} and \texttt{loop_with_dict}. 

---

**Figure 1.** The function \texttt{loop_with_dict}.

**Figure 2.** Inference of looping modes for a predicate symbol.
6.4. 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 3. Notice that \( \Pi_P \) is finite and, for any non-negative integer \( \text{max} \), \( T^\beta_P \uparrow \text{max} \) is a finite set \( \subseteq \text{bin}_\text{unf}(P) \). Line 2, \( \text{Dict} \) is initialized to \( \emptyset \) which satisfies \( \mathbf{D} \). Hence all calls to \( \text{infer\_looping\_modes\_pred} \) fulfill their specification. This shows termination and partial correctness of the function \( \text{infer\_looping\_modes\_prog} \). 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, \( \text{Dict} \) 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.

Non-Termination Inference 11

\[
\text{infer\_looping\_modes\_prog}(P, \text{max}):\n\]

\[
in: \ P: \text{a logic program and } \text{max}: \text{an non-negative integer} \n\]

\[
out: \ \text{a set of pairs } (p, L_p) \text{ where, for each } p \in \Pi_P, L_p \text{ is a looping condition} \n\]

\[
1: \ BinProg := \text{the binary clauses of } T^\beta_P \uparrow \text{max} \n\]

\[
2: \ Dict := \emptyset \text{ and } Res := \emptyset \n\]

\[
3: \text{for each } p \in \Pi_P \text{ do} \n\]

\[
4: \ (L_p, Dict) := \text{infer\_looping\_modes\_pred}(BinProg, p, Dict) \n\]

\[
5: \ Res := Res \cup \{(p, L_p)\} \n\]

\[
6: \text{return } Res \n\]

Figure 3. The top-level function for inferring looping modes.

6.5. Running the algorithm

Example 8 We consider the program \textbf{APPEND3}:

\[
\text{append3}(X,Y,Z,T) :- \text{append}(X,Y,W), \text{append}(W,Z,T). \n\]

augmented by the \textbf{APPEND} program. Here are some elements of \( T^\beta_{\text{APPEND3}} \uparrow 2 \):

\[
\text{append}([\text{[A|E]}, C, [\text{[A|D]}]) :- \text{append}(B, C, D). \quad \% \ C1 \n\]

\[
\text{append3}(A, B, C) :- \text{append}(A, B, E). \quad \% \ C2 \n\]

\[
\text{append3}(\emptyset, A, B, C) :- \text{append}(A, B, C). \quad \% \ C3 \n\]

The dictionary \( \text{Dict} \), built from the binary clause \( \mathbf{C1} \) while processing \text{append}:

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

shows the looping mode \( \{2\} \), including the query: \text{append}([\text{[A|B]}, \text{void}, [\text{[A|C]}]) with all its \( \tau_1 \)-generalizations. For \text{append3}, from the binary clauses \( \mathbf{C2} \) and \( \mathbf{C3} \), the updated dictionary \( \text{Dict}' = \text{Dict} \cup \)}
\{(\text{append3}(x_1, x_2, x_3, x_4)), \\
\tau_2 = \langle \text{append3} \leftrightarrow \{2, 3, 4\}, \text{append} \leftrightarrow \{2\}\rangle, \\
\{\text{append3}(x_1, x_2, x_3, x_4) \leftarrow \text{append}(x_1, x_2, x_5), \\
\text{append}(x_1[x_2], x_3, x_4) \leftarrow \text{append}(x_2, x_3, x_4)\}\rangle, \\
(\text{append3}([], x_1, x_2, x_3), \\
\tau_3 = \langle \text{append3} \leftrightarrow \{3\}, \text{append} \leftrightarrow \{2\}\rangle, \\
\{\text{append3}([], x_1, x_2, x_3) \leftarrow \text{append}(x_1, x_2, x_3), \\
\text{append}(x_1[x_2], x_3, x_4) \leftarrow \text{append}(x_2, x_3, x_4)\}\rangle\}

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

- the query \text{append3}(A, \text{void}, \text{void}, \text{void}) with all its \tau_2-generalizations and
- the query \text{append3}(\text{[]}, A, \text{void}, B) with all its \tau_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 \text{APPEND3}.

7. Proving optimality of termination conditions

It turns out that a slight modification of \text{infer_looping_modes} enables to propose a function which may prove the optimality (see Definition 6) of termination conditions (as computed by a tool for termination inference, e.g. cTli [MES 01] or TerminWeb [GEN 01]). For each pair \((p, \emptyset)\) 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 \(L_{\text{term}}\). Termination and partial correctness rely on similar arguments than those in Subsections 6.3 and 6.4.

\begin{verbatim}
\text{optimal_tcl}(P, \text{max}, \{TC_p\}_{P \in \Pi_P}):
\text{in: \(P\): a logic program, \(\text{max}\): an non-negative integer and}\n\{TC_p\}_{P \in \Pi_P}: \text{a set of termination conditions}\n\text{out: a set of pair \((p, M_p)\) where, for each \(p \in \Pi_P, M_p\) is the multi-mode of \(p\) with no information with respect to its left behavior}\n\text{note: If for each \(p \in \Pi_P, M_p = \emptyset\), then \{TC_p\}_{P \in \Pi_P} is optimal}\n1: BinProg := T_P  \uparrow \text{max}, Dict := \emptyset \text{ and Res} := \emptyset \\
2: \text{for each } p \in \Pi_P \text{ do}\n3: \text{  \((L_p, Dict) := \text{infer_looping_modes}(\text{BinProg}, p, Dict)\)}
4: \text{  } M_p := \text{modes}(p) \setminus \text{less_general}(TC_p) \cup \text{more_general}(L_p))
5: \text{  Res} := \text{Res} \cup \{(p, M_p)\}
6: \text{return } Res
\end{verbatim}

**Example 9** We apply our algorithm to the program \text{APPEND3} of Subsection 6.5 (see also Example 2). We get, for \text{append}:

\[
L_{\text{append}} = \{2\}
\]

\[
\text{more_general}(L_{\text{append}}) = \{\emptyset, \{2\}\}
\]

\[
TC_{\text{append}} = \{1, 3\}
\]

\[
\text{less_general}(TC_{\text{append}}) = \{1, 3, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}
\]

\[
M_{\text{append}} = \{\}
\]
For append3, we have:

\[
\begin{align*}
L_{\text{append3}} &= \{\{1, 3\}, \{2, 3, 4\}\} \\
\text{more_general}(L_{\text{append3}}) &= \{\emptyset, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{2, 3, 4\}\} \\
TC_{\text{append3}} &= \{\{1, 2\}, \{1, 4\}\} \\
\text{less_general}(TC_{\text{append3}}) &= \{\{1, 2\}, \{1, 4\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{1, 2, 3, 4\}\} \\
M_{\text{append3}} &= \{\}
\end{align*}
\]

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

8. Conclusion

To our best knowledge, there is no other automated analysis dealing with optimality proofs of termination conditions for logic programs. But loop checking in logic programming is a subject related to non-termination, where Bol [BOL 91] sets up some solid foundations (see also [SKO 97]). 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 [KOM 82]. In this case, Bol emphasizes complete loop checkers, which were also studied in [BRU 92, SHE 01].

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 [DEV 93], 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 we have started to adapt the approach to constraint logic programming. For rational trees, [De 89] 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 worthwhile topic.
9. References


