Non-Termination Inference for Optimal Ter-mination Conditions of Logic Programs

Frédéric Mesnard, Etienne Payet, Ulrich Neumerkel

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.*

RÉSUMÉ. *Dans cet article, nous présentons une technique d'inférence de conditions de non-terminaison 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 dépliages binaires du programme traité.*

L'inférence de non-terminaison 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 infini.

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 sélélection gauche et au langage utilisé pour décrire les ensembles de requêtes atomiques.*

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

MOTS-CLÉS : *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 $\mathcal{L}_{term}$.

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* 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 $Q\theta$ 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. [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 $\mathcal{L}_{term}$ 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. 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.

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 $\mathcal{L}$). Let A be an atom. Then $rel(A)$ denotes its relation symbol. A *query* $\mathbf{A}$ 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. 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.

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`$_1$`,...,B`$_n$. 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 (or goal) $A, \mathbf{A}$ and a clause c. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint[1] with $A, \mathbf{A}$ and assume that A and H unify. Let θ be an mgu of A and H. Then $A, \mathbf{A} \overset{\theta}{\underset{c}{\Longrightarrow}} (\mathbf{B}, \mathbf{A})\theta$ 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* [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 *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

---

1. 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.

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 $arity(p) = n$.

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

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 motivation 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 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 $bin\_unf(P)$.*

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).    p(X,X).    q(a,b).
```

The binary unfoldings of P are:

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

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 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:*

$$
\begin{aligned}
less\_general(M_p) &= \{m \in modes(p) \mid \exists m' \in M_p \; m' \subseteq m\} \\
more\_general(M_p) &= \{m \in modes(p) \mid \exists m' \in M_p \; m \subseteq m'\}
\end{aligned}
$$

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

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 [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 `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* τ:

$$\begin{array}{ccc} \Pi & \longrightarrow & 2^N \\ p & \longmapsto & I \subseteq [1, arity(p)] \end{array}$$

Example 3 (Example 2 continued) *If we want to spot 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* $=_\tau$:

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

$$A_1, \ldots, A_n =_\tau B_1, \ldots, B_m \quad \textit{iff} \quad \left\{ \begin{array}{l} n = m \\ \forall i \in [1, n],\ A_i =_\tau B_i \end{array} \right.$$

– *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 of append: $append([1], 2, [3, 4])$ 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:*

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

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 $\tau = \langle append \longmapsto \{2\}\rangle$ is DN for the recursive clause defining append, but is not DN for the program `APPEND` since `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 $\tau := \langle p \longmapsto \varnothing\rangle$ 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 (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 $\{p(s',u)\} \cup P$.*

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.

$\mathtt{dna}\big(BinProg, \tau\big)$:

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

1:   $\tau' := \tau$
2:   **while** $\mathtt{dna\_one\_step}(BinProg, \tau') \neq \tau'$ **do**
3:       $\tau' := \mathtt{dna\_one\_step}(BinProg, \tau')$
4:   **return** τ'

$\mathtt{dna\_one\_step}\big(BinProg, \tau\big)$:

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

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

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 $BP \subseteq bin\_unf(P)$, τ 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 $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) \in bin\_unf(P)$, and we would like to prove that the mode m_p is looping.

If $q(t_1, \ldots, t_{n'})$ is a τ-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

$\{s_i | i \in m_p\}$ by a grounding substitution σ. The looping behavior is preserved if the impact of σ only affects the arguments of $q(t_1, \ldots, t_{n'})$ which are neutral with respect to derivation. Hence the condition (**c**) of Subsection 6.2: $Var(\{s_i \mid i \in m_p\}) \cap Var(\{t_i \mid i \notin \tau(q)\}) = \varnothing$. 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*

$\texttt{unit\_loop}(m_p, c)$:

  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 the boolean **false**

  1: $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) := c$
  2: $\tau := \texttt{dna}(\{c\}, \tau_{max})$
  3: **if** $p(s_1, \ldots, s_n)$ is a τ-instance of $q(t_1, \ldots, t_{n'})$ /* p=q, n=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**

Termination of $\texttt{unit\_loop}$ relies on termination of $\texttt{dna}$. Partial correctness follows from partial correctness of $\texttt{dna}$ and the result below.

Theorem 3 *Let $p \in \Pi_P$, m_p be a mode of p, and $c \in bin\_unf(P)$.*
If $\texttt{unit\_loop}(m_p, c) \neq \textbf{false}$, 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 $(Atom, \tau, BinProg)$ 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 $\textbf{D}$ if $Dict$ is a finite set such that for any $(Atom, \tau, BinProg) \in Dict$ we have: $BinProg$ is a finite subset of $bin\_unf(P), \tau$ is DN for $BinProg$, and $Atom$ loops w.r.t. $BinProg$.*

Termination of $\texttt{loop\_with\_dict}$ (see Figure 1) comes from finiteness of $Dict$ and termination of $\texttt{dna}$. Partial correctness follows from partial correctness of $\texttt{dna}$ and:

Theorem 4 *Let $p \in \Pi_P$, m_p be a mode of p, and $c \in bin\_unf(P)$.*
If $Dict$ satisfies $\textbf{D}$ and $\texttt{loop\_with\_dict}(m_p, c, Dict) \neq \textbf{false}$ 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:

`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 the
boolean **false**

1:   $p(s_1, \ldots, s_n) \leftarrow q(t_1, \ldots, t_{n'}) := c$

2:   **for each** $\big(q(u_1, \ldots, u_{n'}), \tau, BinProg\big) \in Dict$ **do**

3:      **if** $\begin{cases} q(u_1, \ldots, u_{n'}) \text{ is a } \tau\text{-instance of } q(t_1, \ldots, t_{n'}) \text{ and} \\ \textbf{(c)} Var(\{s_i \mid i \in m_p\}) \cap Var(\{t_i \mid i \notin \tau(q)\}) = \varnothing \end{cases}$

4:      **then return** $(\mathtt{dna}(BinProg \cup \{c\}, \tau), BinProg \cup \{c\})$

5:   **return false**

Figure 1. *The function* `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:   $M_p := \mathtt{modes}(p)$, $L_p := \varnothing$ and $Dict' := Dict$

2:   **for each** $c := p(s_1, \ldots, s_n) \leftarrow B \in BinProg$ **with** $B \neq true$ **do**

3:      **for each** $m_p \in M_p$ **do** /*NB: M_p is modified lines 6 and 11 */

4:         **if** $\mathtt{unit\_loop}(m_p, c) \neq$ **false then**

5:            $(\tau, BP) := \mathtt{unit\_loop}(m_p, c)$

6:            $M_p := M_p \backslash \mathtt{more\_general}(m_p)$ /* cf. Definition 5 */

7:            $L_p := L_p \cup \{m_p\}$

8:            $Dict' := Dict' \cup \big\{(p(s_1, \ldots, s_n), \tau, BP)\big\}$

9:         **elsif** $\mathtt{loop\_with\_dict}(m_p, c, Dict') \neq$ **false then**

10:            $(\tau, BP) := \mathtt{loop\_with\_dict}(m_p, c, Dict')$

11:            $M_p := M_p \backslash \mathtt{more\_general}(m_p)$

12:            $L_p := L_p \cup \{m_p\}$

13:            $Dict' := Dict' \cup \big\{(p(s_1, \ldots, s_n), \tau, BP)\big\}$

14:   **return** $(L_p, Dict')$

Figure 2. *Inference of looping modes for a predicate symbol.*

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 `unit_loop` and `loop_with_dict`.

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 Π_P is finite and, for any

---

$\texttt{infer\_looping\_modes\_prog}(P, max)$:

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:      $(L_p, Dict) := \texttt{infer\_looping\_modes\_pred}(BinProg, p, Dict)$
  5:      $Res := Res \cup \{(p, L_p)\}$
  6:   **return** Res

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

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 $\texttt{infer\_looping\_modes\_pred}$ fulfill their specification. This shows termination and partial correctness of the function $\texttt{infer\_looping\_modes\_prog}$. 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.

6.5. *Running the algorithm*

Example 8 *We consider the program* APPEND3:

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

augmented by the APPEND *program. Here are some elements of* $T_{APPEND3}^\beta \uparrow 2$:

```
append([A|B],C,[A|D]):-append(B,C,D).       % C1
append3(A,B,C,D):-append(A,B,E).            % C2
append3([],A,B,C):-append(A,B,C).           % C3
```

The dictionary $Dict$*, built from the binary clause* C1 *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 the query:* append([A|B],void,[A|C]) *with all its* τ_1*-generalizations. For* append3*, from the binary clauses* C2 *and* C3*, 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 query* `append3(A,void,void,void)` *with all its* τ_2*-generalizations and*

– *the query* `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`.

7. Proving optimality of termination conditions

It turns out that a slight modification of `infer_looping_modes_prog` 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. cTI [MES 01] or TerminWeb [GEN 01]). 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 6.3 and 6.4.

`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 the 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:       $(L_p, Dict) :=$ `infer_looping_modes_pred(`$BinProg, p, Dict$`)`
   4:       $M_p :=$ `modes(`p`)`$\backslash($`less_general(`TC_p`)` $\cup$ `more_general(`L_p`)`$)$
   5:       $Res := Res \cup \{(p, M_p)\}$
   6:   **return** Res

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

$$L_{append} \quad = \quad \{\{2\}\}$$
$$\texttt{more\_general}(L_{append}) \quad = \quad \{\varnothing, \{2\}\}$$
$$TC_{append} \quad = \quad \{\{1\}, \{3\}\}$$
$$\texttt{less\_general}(TC_{append}) \quad = \quad \{\{1\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\},$$
$$M_{append} \quad = \quad \{\}$$

For `append3`, *we have:*

$$
\begin{aligned}
L_{append3} &= \{\{1,3\},\{2,3,4\}\} \\
\texttt{more\_general}(L_{append3}) &= \{\varnothing,\{1\},\{2\},\{3\},\{4\},\{1,3\},\{2,3\},\{2,4\}, \\
&\qquad \{3,4\},\{2,3,4\}\} \\
TC_{append3} &= \{\{1,2\},\{1,4\}\} \\
\texttt{less\_general}(TC_{append3}) &= \{\{1,2\},\{1,4\},\{1,2,3\},\{1,2,4\},\{1,3,4\}, \\
&\qquad \{1,2,3,4\}\} \\
M_{append3} &= \{\}
\end{aligned}
$$

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

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 worth-while topic.

9. References

[APT 82] APT K. R., VAN EMDEN M. H., "Contributions to the theory of Logic Programming", *Journal of the ACM*, vol. 29, num. 3, 1982, p. 841–862.

[APT 97] APT K. R., *From Logic Programming to Prolog*, Prentice Hall, 1997.

[ART 96] ARTS T., ZANTEMA H., "Termination of logic programs using semantic unification", *Logic Program Synthesis and Transformation*, vol. 1048 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1996, TALP can be used online at `http://bibiserv.techfak.uni.biekefeld.de/talp/`.

[BOL 91] BOL R., "Loop Checking in Logic Programming", PhD thesis, CWI, Amsterdam, 1991.

[BRU 92] BRUYNOOGHE M., DE SCHREYE D., MARTENS B., "A General Criterion for Avoiding Infinite Unfolding during Partial Deduction", *New Generation Computing*, vol. 11, num. 1, 1992, p. 47–79.

[COD 99] CODISH M., TABOCH C., "A semantic basis for the termination analysis of logic programs", *Journal of Logic Programming*, vol. 41, num. 1, 1999, p. 103-123.

[De 89] DE SCHREYE D., BRUYNOOGHE M., VERSCHAETSE K., "On the existence of nonterminating queries for a restricted class of Prolog-clauses", *Artificial Intelligence*, vol. 41, 1989, p. 237–248.

[DER 01] DERSHOWITZ N., LINDENSTRAUSS N., SAGIV Y., SEREBRENIK A., "A General Framework for Automatic Termination Analysis of Logic Programs", *Applicable Algebra in Engineering, Communication and Computing*, vol. 12, num. 1/2, 2001, p. 117-156.

[DEV 93] DEVIENNE P., P.LEBÈGUE, ROUTIER J.-C., "Halting problem of one binary Horn clause is undecidable", *LNCS*, vol. 665, Springer-Verlag, 1993, p. 48–57, Proc. of STACS'93.

[GAB 94] GABBRIELLI M., GIACOBAZZI R., "Goal independency and call patterns in the analysis of logic programs", *Proceedings of the ACM Symposium on applied computing*, ACM Press, 1994, p. 394–399.

[GEN 01] GENAIM S., CODISH M., "Inferring Termination Condition for Logic Programs using Backwards Analysis", *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`.

[KOM 82] KOMOROWSKI H. J., "Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language : a Theory and Implementation in the Case of Prolog", *Proc. of the 9th POPL*, 1982, p. 255–267.

[LIN 97] LINDENSTRAUSS N., "TermiLog: a system for checking termination of queries to Logic Programs", 1997, `http://www.cs.huji.ac.il/~naomil`.

[MES 96] MESNARD F., "Inferring left-terminating classes of queries for constraint logic programs by means of approximations", MAHER M. J., Ed., *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, MIT Press, 1996, p. 7–21.

[MES 00] MESNARD F., NEUMERKEL U., "cTI: a tool for inferring termination conditions of ISO-Prolog", april 2000, `http://www.complang.tuwien.ac.at/cti`.

[MES 01] MESNARD F., NEUMERKEL U., "Applying static analysis techniques for inferring termination conditions of logic programs", COUSOT P., Ed., *Static Analysis Symposium*, vol. 2126 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 2001, p. 93–110.

[O'K 90] O'KEEFE R., *The Craft Of Prolog*, MIT Press, 1990.

[SHE 01] SHEN Y.-D., YUAN L.-Y., YOU J.-H., "Loops Checks for Logic Programs with Functions", *Theoretical Computer Science*, vol. 266, num. 1-2, 2001, p. 441–461.

[SKO 97] SKORDEV D., "An Abstract Approach to Some Loop Detection Problems", *Fundamenta Informaticae*, vol. 31, 1997, p. 195–212.

[SPE 97] SPEIRS C., SOMOGYI Z., SØNDERGAARD H., "Termination Analysis for Mercury", VAN HENTENRYCK P., Ed., *Proc. of the International Static Analysis Symposium*, vol. 1302 of *LNCS*, Springer-Verlag, 1997, p. 160–171.