TLP **13** (4–5): Online Supplement, July 2013. © 2013 [SENLIN LIANG and MICHAEL KIFER]
URL: http://dx.doi.org/10.1017/S1471068413000446

A practical analysis of non-termination in large logic programs

SENLIN LIANG and MICHAEL KIFER
Department of Computer Science, Stony Brook University, USA
(*e-mail:* {`sliang, kifer`}`@cs.stonybrook.edu`)

submitted 10 April 2013; revised 23 June 2013; accepted 5 July 2013

Abstract

A large body of work has been dedicated to termination analysis of logic programs but relatively little has been done to analyze *non*-termination. In our opinion, *explaining* non-termination is a much more important task because it can dramatically improve a user's ability to effectively debug large, complex logic programs without having to abide by punishing syntactic restrictions. Non-termination analysis examines program execution history when the program is suspected to not terminate and informs the programmer about the exact reasons for this behavior. In Liang and Kifer (2013), we studied the problem of non-termination in tabled logic engines with subgoal abstraction, such as XSB, and proposed a suite of algorithms for non-termination analysis, called `Terminyzer`. These algorithms analyze forest logging traces and output sequences of tabled subgoal calls that are the likely causes of non-terminating cycles. However, this feedback was hard to use in practice: the same subgoal could occur in multiple rule heads and in even more places in rule bodies, so `Terminyzer` left too much tedious, sometimes combinatorially large amount of work for the user to do manually.

Here we propose a new suite of algorithms, `Terminyzer+`, which closes this usability gap. `Terminyzer+` can detect not only sequences of subgoals that cause non-termination, but, importantly, the exact rules where they occur and the rule sequences that get fired in a cyclic manner, thus causing non-termination. This makes `Terminyzer+` suitable as a back-end for user-friendly graphical interfaces on top of `Terminyzer+`, which can greatly simplify the debugging process. `Terminyzer+` back-ends exist for the SILK system as well as for the open-source $\mathcal{F}$LORA-*2* system. A graphical interface has been developed for SILK and is currently underway for $\mathcal{F}$LORA-*2*. We also report experimental studies, which confirm the effectiveness of `Terminyzer+` on a host of large real-world knowledge bases. All tests used in this paper are available online.[1]

In addition, we make a step towards automatic remediation of non-terminating programs by proposing an algorithm that heuristically fixes some causes of misbehavior. Furthermore, unlike `Terminyzer`, `Terminyzer+` does not require the underlying logic engine to support subgoal abstraction, although it can make use of it.

KEYWORDS: terminyzer, termination and non-termination analysis, logic programming.

[1] http://rulebench.projects.semwebcentral.org/terminyzer+

1 Introduction

The problem of run-away computations in logic programs is much more serious than in procedural programming because of the declarative nature of the logic languages and the large gap between the declarative semantics and the actual evaluation strategy. This problem is even more vexing in high-level logic languages such as $\mathcal{F}$LORA-2 and SILK,[2] which position themselves as tools for developing complex knowledge bases by knowledge engineers who are not programmers. This type of users cannot be expected to debug the procedural aspects of the rule bases that they create and thus they require special support. Non-termination has been flagged as a key issue standing on the way of creation of complex biological knowledge base in the SILK project, where the use of function symbols is more common due to the higher-order features of HiLog (Chen *et al.* 1993) and F-logic (Kifer *et al.* 1995), and due to the proliferation of Skolemized head-existentials that are passed down to the engine by the knowledge acquisition system.

The first source of non-termination is the use of recursion, which plagues Prolog under the standard evaluation strategy. This can be illustrated by the following simple example:

```
p(X) :- p(X).
?- p(a).
```

The prevalent way to address this problem is to use *tabling*, which is also known under the more technical term of SLG-resolution. Tabling was pioneered by the XSB system (Swift and Warren 2012) and is now supported by a number of other systems, such as Yap (Costa *et al.* 2012), B-Prolog (Zhou 2012), and Ciao (Hermenegildo *et al.* 2012). In the above example, tabling a predicate, `p/1`, will cause the evaluation to terminate.

The second reason for non-termination, even under SLG, is the pattern of increasingly deep nested calls generated during the evaluation, as in the following example:

```
p(X) :- p(f(X)).
?- p(a).
```

Here query evaluation will successively call `p(a)`, `p(f(a))`, `p(f(f(a)))`, and so on. Since neither call subsumes the other, tabling will not help terminate the evaluation process. However, a surprisingly simple technique known as *subgoal abstraction*, also pioneered by XSB, takes care of this problem. The idea is to modify the calls by "abstracting" deeply nested subterms and replacing them with new variables. For instance, in the above example, we could abstract calls once the depth limit of 4 has been reached. As a result, `p(f(f(f(f(a)))))` and all the subsequent calls would be abstracted to `p(f(f(f(X)))), X=f(a)`. Tabling enhanced with subgoal abstraction is able to completely evaluate all queries that have finite number of answers.

[2] flora.sourceforge.net, silk.semwebcentral.org

The remaining major source of non-termination is when the number of answers to a query or its subqueries is infinite, such as for the query `?- p(X)` and the rules

```
p(a).
p(f(X)) :- p(X).
```

This query has an infinite number of answers: `p(a), p(f(a)), p(f(f(a))), ....` Clearly, such queries cannot be evaluated completely, but if the program is what the user intended, the user could ask the system to stop after getting the required number of answers. However, in our experience, the user usually does not intend to construct infinite predicates. Finding out how the infinite number of answers came about and fixing the problem is difficult even for an experienced programmer and even for programs that have just a few dozens of rules. For knowledge bases that have thousands of rules, like the ones we have been dealing with in the SILK project, diagnosing this problem is an onerous and frustrating job. In the absence of subgoal abstraction, this difficulty also exists for the aforesaid problem of detecting sequences of subgoals of increasing depth.

We remind that neither the problem of program termination nor that of whether the number of answers is finite is decidable (Schreye and Decorte 1994; Sipser 1996), so no algorithm can prove termination or non-termination in general. Sufficient conditions for termination of logic programs have been proposed in the literature (Bol *et al.* 1991; Schreye and Decorte 1994; Verbaeten *et al.* 2001; Lindenstrauss *et al.* 2004; Bruynooghe *et al.* 2007; Nguyen and De Schreye 2007; Nguyen *et al.* 2008; Schneider-kamp *et al.* 2010), but most deal with Prolog or Prolog-like evaluation strategies. Although many of these results are very deep, their practical impact is limited because they provide only sufficient conditions for termination. The precise classes of programs for which these algorithms work are typically inexpressive and usually not investigated at all (see Appendix B for a discussion). Moreover, neither tabling nor subgoal abstraction are taken into account by these works, so they have limited use for advanced logic engines like XSB and its derivatives, $\mathscr{F}$LORA-2 and SILK. In a recent work, (Liang and Kifer 2013) took a different approach and developed techniques to enable users to analyze the causes of non-termination. We proposed a suite of algorithms, called the *non-termination analyzer* or `Terminyzer`, which was able to detect sequences of tabled subgoal calls and functor applications that are the potential causes of non-termination. These algorithms analyze forest logging traces (see Section 2 for the details about forest logging) and output sequences of tabled subgoal calls that form non-terminating call-cycles. Unfortunately, in many cases this feedback was hard to use in practice, as it was fairly imprecise. The same subgoal could occur in multiple rule heads and in even more places in rule bodies, so `Terminyzer` left too much tedious, sometimes combinatorially large amount of manual work for the user to do.

Here we propose a new suite of algorithms, called `Terminyzer+`, which closes this usability gap. `Terminyzer+` can detect not only sequences of subgoals that cause non-termination, but, importantly, the exact rules where these calls occur and the rule sequences that are fired in a cyclic manner, which lead to non-termination. This makes `Terminyzer+` amenable to serving as a back-end for user-friendly graphical interfaces, which can greatly simplify the debugging process. Such an interface was constructed for the SILK project and is currently underway for the open-source $\mathcal{F}$LORA-*2* system.

The key idea that makes `Terminyzer+` possible is a program transformation that assigns a unique rule id to each rule *and* modifies the rules in such a way that the identifying information is preserved in the log forest trace. The transformation increases the size of each subgoal slightly, by adding an additional argument. The analysis that was originally performed by `Terminyzer` has been made much more precise so it can track the exact rule applications that cause non-termination. This is a major advance over the original system, as it closes the aforesaid usability gap. Furthermore, our new algorithms do not depend on subgoal abstraction, although they can take subgoal abstraction into account, if the underlying engine supports it. Finally, we make a step towards automatic remediation of non-terminating programs by proposing an algorithm that heuristically fixes some of the faulty programs.

The rest of this paper is organized as follows. Section 2 provides the necessary background. Section 3 presents the transformation that adds a unique id to each rule. Section 4 describes `Terminyzer+` for tabled logic engines with subgoal abstraction. Section 5 presents auto-repair techniques for certain non-terminating behaviors. Section 6 extends `Terminyzer+` to tabled logic engines that do not support subgoal abstraction, and Section 7 concludes the main part of the paper. In addition, the appendices supply further information on experimental studies, the related work, tabling and forest logging, unfinished call and answer flow analysis, and proofs of theorems.

2 Preliminaries

Tabling. The limitations of Prolog's standard SLD resolution-based evaluation strategy are well-known: it is incomplete and can go into an infinite loop even for simple Datalog rule sets. To address this problem, SLG resolution (also known as "tabling") was developed over 20 years ago and (Swift and Warren 2012) provides the most recent insight into this mechanism. In tabled evaluation, calls to tabled predicates are cached in a *table* for subsequent calls. It has been proven that tabled evaluation terminates for any program with the *bounded term depth property*, i.e., when all terms that are ever generated in the course of SLG resolution, including all subgoals and answers, have an upper bound on their depth (Swift and Warren 2012). To simplify the discussion, we assume all predicates are tabled in the rest of this paper.

Forest logging. Compared to Prolog systems, logic engines that support tabling are much more involved. They suspend and resume computation paths, delay negated subgoals that are involved in loops through negation, simplify these subgoals once their truth values become known, and manage the table accordingly. For debugging and performance optimization, programmers may need to inspect table operations during evaluation. To this end, XSB has recently provided a new facility, called *forest* *logging* (`logforest`), which makes the table events available to the programmer.[3] These events include:

- *Call to a tabled predicate*. If a subgoal *parent* calls another subgoal *child*, i.e., the evaluation of *parent* fires a rule that issues *child*, a Prolog fact of the form $tc(child, parent, status, timestamp)$ is logged. Here *timestamp* is the timestamp of the event representing its sequence order and *status* is the current status of *child*. It can take the following values:

  — *new* if *child* is a newly issued subgoal;
  — *cmp* if the evaluation of *child* has been completed; and
  — *incmp* if *child* is not a new subgoal, but is yet to be completely evaluated.

  If the subgoal is negative, a similar fact $nc(child, parent, status, timestamp)$ is logged. If *child* is the first tabled subgoal in an evaluation, *parent* is *root*.
- *Derivation of a new answer*. When a new answer, *ansr*, is derived for *sub* and added to the table, the fact $na(ansr, sub, timestamp)$ is added to the log. When a new conditional answer *ansr* `:-` *delayed_literals* is derived for *sub* and added to the table, a log record of the form $nda(ansr, sub, delayed\_literals, timestamp)$ is recorded. Here *ansr* is the answer substitution and *delayed_literals* are the delayed literals whose truth value is yet to be determined (this usually occurs due to recursive loops through negation).
- *Return of an answer to a consumer*. If an answer, *ansr*, is returned to a consumer, *child*, which was called by *parent*, the fact $ar(ansr, child, parent, timestamp)$ is added to the log. If the answer is conditional, $dar(ansr, child, parent, timestamp)$ is recorded.
- *Subgoal completion*. When all mutually recursive subgoals in a set, S, are completed, `logforest` records $cmp(sub, sccnum, timestamp)$ for each $sub \in S$, where *sccnum* is an ordinal that identifies S.
- *Other events*. `Logforest` also records delays of negative literals, table abolishes, and errors. These events are not needed for our purposes and are omitted.

More details and examples of tabling and forest logging are given in Appendix C.

3 Adding Ids to rules

The key enabling idea in `Terminyzer+` is a transformation that adds unique ids to rules in such a way that this information is preserved in the forest logging trace.

For our purposes, we want each subgoal call in the trace to "remember" the rule from which this call was issued. Although this information is not available in the original program, one can instrument any logic program so that each subgoal call would be stamped with the id of its *host rule*, i.e., rule from whose body the call was issued.

[3] Although currently XSB is the only system supporting forest logging, all tabling engins have the requisite information internally and could expose it to the user to take advantage o the advanced debugging facilities, such as `Terminyzer+`, that are enabled by this feature.

The transformation processes the original program rule by rule and assigns a new id to each newly encountered rule. In each such rule, each tabled predicate, p/N, is augmented with one more argument, so that p/N is replaced with $p/(N+1)$ as follows:

```
while unprocessed rules remain do
    Get the next program rule $R$: $h(t_1, ..., t_k)$ :- body.;
    Generate a new rule id, $id(R)$;
    if $h/k$ is tabled then Change the head literal to $h(t_1, ..., t_k, Newvar)$;
    else Leave the head literal unchanged;
    Replace each tabled subgoal, $p(s_1, ..., s_m)$, in body with $p(s_1, ..., s_m, id(R))$;
end
```

Algorithm 1: Program Transformation: Adding Rule Ids

Queries are modified as follows: if the query predicate is not tabled, the query is not changed. If that predicate is tabled, an additional (last) argument is added, which contains a new variable.

It is easy to see that the new program is equivalent to the original one in the sense that non-tabled queries to both programs have the same answers and the answers to the tabled predicates are the same if the last component in each answer tuple is chopped off.[4] However, now each subgoal call recorded in the log will be labeled with the id of the rule from which this call was issued. For instance, the query `?- r(a)` and the rule

```
r(X) :- p(X), s(X,Y), q(Y).
```

get transformed into the following, assuming that the assigned rule id is 123, that $r/1$, $p/1$, and $q/1$ are tabled, and that $s/2$ is not:

```
r(X,_) :- p(X,123), s(X,Y), q(Y,123).
?- r(a,_).
```

The above transformation has been implemented for $\mathcal{F}$LORA-2 and SILK, although the form of the last argument there is made more complex to provide additional support for truth maintenance.

4 `Terminyzer+` for tabled logic engines with subgoal abstraction

This section extends the call sequence analysis and answer flow analysis approaches in (Liang and Kifer 2013) for tabled logic engines that support subgoal abstraction. The analysis assumes that execution is stopped after a time limit set by the user or after the evaluation starts producing answers that exceed certain size limits (e.g., term depth), and then analyzing the logs. Our examples assume that the system stops when it generates query answers of depth greater than 10.

[4] We assume that the programs have no aggregate functions such as *count*, which are sensitive to duplicate answers.

We should stress that due to the undecidability results mentioned in the introduction, no algorithm can detect non-termination in all cases unless infinite logs are available. Pragmatically, this means that, in working with `Terminyzer+`, one must assume that the available logs are "long enough."

4.1 Call sequence analysis

Recall that, with subgoal abstraction, the only way for tabled query evaluation to not terminate is when the query or its subgoals have infinitely many answers. Call sequence analysis, in this case, finds the *exact* sequence of subgoal calls to tabled predicates and, for each subgoal, its host rule's id. Moreover, it identifies the potential sets of recursive predicates and rules that are causing non-termination.

As discussed in Section 2, when a tabled subgoal *sub* has been completely evaluated and all its answers have been recorded in the table, `logforest` records a log entry of the form $cmp(sub, sccnum, timestamp)$. We say that such calls are *finished*. Otherwise, the call is *unfinished* and can be found via the following rule:

```
unfinished(Child,Parent,Timestamp) :-
    (tc(Child,Parent,Stage,Timestamp); nc(Child,Parent,Stage,Timestamp)),
    (Stage == new ; Stage == incmp),
    not_exists(cmp(Child,SCCNum,Timestamp1)).
```

Here `not_exists` is the XSB well-founded negation operator, which, in this case, existentially quantifies `SCCNum` and `Timestamp1`.

The fact $unfinished(child, parent, timestamp)$ says that *unfinished* subgoal *child* is called by *parent* and the event timestamp is *timestamp*. Since *parent* is waiting for the answers from *child*, *parent* is a child of another unfinished subgoal. The initial subgoal, *root*, has no parent.

Theorem 1 (*Soundness of the call sequence analysis*)
Consider a query to a program all of whose predicates are tabled, and assume that the system supports subgoal abstraction. If there are unfinished calls in the complete infinite forest logging trace, then

i. the sequence of unfinished calls, sorted by their timestamps, is the exact sequence of unfinished calls that caused non-termination, and
ii. the ids of the rules that issued each of these unfinished calls appear in the last arguments of these calls. □

The proof of Theorem 1 is given in Appendix F. Clearly, however, one cannot obtain the complete infinite trace for a non-terminating evaluation. In practice, one would let the program execute long enough until it starts producing answers exceeding some size limits and then analyze the available portion of the log.

We now turn to developing a more precise machinery for this task.

The *unfinished-call child-parent graph* (CPG) for a forest logging trace is a directed graph $G_{uc} = (\mathscr{N}, \mathscr{E})$ whose nodes are $\mathscr{N} = \{child \mid unfinished(child, parent, timestamp)\} \cup \{root\}$. Subgoals that are variants of each other (i.e., identical up to the variable renaming) are treated as the same subgoal. Each $sub \in \mathscr{N}$ is labeled with the timestamp of the first call to *sub*; it is written as *sub.timestamp*.

The timestamp of the initial subgoal *root*, *root.timestamp*, is -1. A directed edge (sub_1, sub_2) is in $\mathscr{E}$ if and only if sub_1 is an unfinished parent-subgoal of sub_2, i.e., $unfinished(sub_2, sub_1, timestamp)$ is true. The edge that corresponds to the fact $unfinished(sub_2, sub_1, timestamp)$ is *labeled* with the timestamp of this fact and is denoted $(sub_1, sub_2).timestamp$. The timestamps of nodes and edges preserve the temporal order of their creation in the forest logging trace.

An *unfinished-call path* is a path with *no repeated edges* in G_{uc}; it is called an *unfinished-call loop* if it is a cycle. An unfinished-call path of the form $[sub, sub]$ means that there is an edge $(sub, sub) \in \mathscr{E}$ and it is also an unfinished-call loop. Loops that represent the same cycles in CPG are considered to be the same and we keep only one representative for each set of such loops. For instance, $[a, b, c, a]$ and $[b, c, a, b]$ are the same loop while $[a, b, c, a]$ and $[a, c, b, a]$ are not. Unfinished-call loops contain recursive subgoals that are potential causes of non-termination.

Example 1
Consider the non-terminating query `?- r(X)` and the rules, below, where `@!sym` indicates the id of the corresponding rule:

```
@!r1  p(a).                    @!r5  r(X) :- r(X).
@!r2  p(f(X)) :- q(X).         @!r6  r(X) :- p(X), s(X).
@!r3  q(b).                    @!r7  s(f(b)).
@!r4  q(g(X)) :- p(X).
```

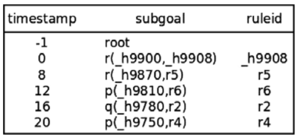
The evaluation produces logs containing these unfinished calls:

```
unfinished(r(_h9900,_h9908), root, 0)
unfinished(r(_h9870,r5), r(_h9870,_h9889), 8)
unfinished(r(_h9840,r5), r(_h9840,r5), 11)
unfinished(p(_h9810,r6), r(_h9810,r5), 12)
unfinished(q(_h9780,r2), p(_h9780,r6), 16)
unfinished(p(_h9750,r4), q(_h9750,r2), 20)
unfinished(q(_h9720,r2), p(_h9720,r4), 24)
```

This is the exact sequence of calls causing non-termination. There are 6 unfinished subgoals as shown in Figure 1(A), where each subgoal's timestamp and the host rule's id are also given. The unfinished-call CPG has 7 edges, shown in Figure 1(B), where timestamps are used to represent nodes instead of actual subgoals and each edge is labeled with its timestamp. There are 2 unfinished-call loops: [8, 8] and [16, 20, 16]. □

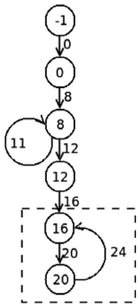
Theorem 2 (*Completeness of the call sequence analysis*)
Consider a query and a program all of whose predicates are tabled and assume that the system supports subgoal abstraction. If the evaluation does not terminate, then

i. there is at least one unfinished-call loop in the unfinished-call CPG constructed for the complete infinite forest logging trace, and the loop's subgoals are responsible for the generation of infinite number of answers, and
ii. the last arguments of these subgoals specify the rule ids from whose bodies these subgoals were called. □



timestamp	subgoal	ruleid
-1	root	
0	r(_h9900,_h9908)	_h9908
8	r(_h9870,r5)	r5
12	p(_h9810,r6)	r6
16	q(_h9780,r2)	r2
20	p(_h9750,r4)	r4

(A)



$$\begin{tikzcd}
-1 \arrow[d, "0"] \\
0 \arrow[d, "8"] \\
8 \arrow[loop left, "11"] \arrow[d, "12"] \\
12 \arrow[d, "16"] \\
16 \arrow[d, "20"] \\
20 \arrow[u, bend right, "24"']
\end{tikzcd}$$

(B)

Fig. 1. Unfinished-Call CPG of Example 1

Theorem 2 is proved in Appendix F, while the algorithms for constructing the unfinished-call CPG and for computing unfinished-call paths and loops are found in Appendix D. Since complete infinite traces for non-terminating computations cannot be had, in practice one would let the program execute long enough until it starts producing answers exceeding some size limits, and then analyze the available portion of the log. Clearly, this opens up the possibility for false negatives, i.e., for blaming sequences of calls that in actuality do terminate after a long time. However, even in this case, such sequences are possible computational bottlenecks and identifying them is useful in its own right.

Identification of the exact rules that cause infinite computations in Theorems 1 and 2 (and later in Theorems 3 and 4) is a major advance in `Terminyzer+` over the original `Terminyzer` (Liang and Kifer 2013), as the subgoals in the various loops found by our algorithms can come from multiple rules. Practically speaking, this means that `Terminyzer+` obviates potentially combinatorially large amount of work that the user would otherwise have to do manually. Rule ids can also be gainfully exploited by graphical tools, such as the one built for SILK.

4.2 Answer flow analysis

Call sequence analysis finds the exact sequences of subgoal calls and the corresponding host rules that are involved in a non-terminating computation. These subgoals are marked as incomplete in the trace because they are waiting for answers for themselves or their children. However, many of these subgoals do not actually produce an infinite number of answers and they are not true reasons for non-termination. A much more useful outcome of the call sequence analysis are the sets of recursive predicates that form the unfinished-call loops and cause generation of infinitely many answers. Unfortunately, the number of such loops in an unfinished-call CPG can be exponential and, moreover, not all of these loops may be the reason for non-termination. For instance, Figure 1 has two unfinished-call loops, but only [16, 20, 16] is at fault. This problem is dealt with using *answer flow analysis*, described below.

We say that an unfinished-call loop is a *culprit* if it is a cause for non-termination. Answer flow analysis looks for the log entries that specify the answers being returned to parents (the *ar*-facts and *dar*-facts) at the end of the `logforest` trace and produces child-parent relationships among unfinished subgoals. These child-parent relationships help to identify precisely which unfinished-call loops are culprits, so we could track how answers percolate through the unfinished subgoals.

When there are infinitely many answers, each new answer, *ansr*, to an unfinished subgoal, *sub*, is returned to the parents of *sub* and these parents use *ansr* to derive their own answers. The newly derived answers for the parents of *sub* are returned to the parents of the parents, and this gives rise to an endless process in which subgoals continue to receive, derive, and return answers. An *answer-flow child-parent sequence* is the sequence of child-parent pairs found in all the log entries for answers returned to parents; it captures the child-parent relationships in the above endless process. The pairs of an answer-flow child-parent sequence are sorted by their creation order (*timestamp*). A child might continue returning multiple answers to a certain parent before the parent starts deriving its own answers. In this case, only one child-parent pair is recorded for all such answer returns, since all these pairs are identical.

An answer-flow child-parent sequence, *cps*, contains a *child-parent pattern*, *cpp*, if *cpp* is a *finite* subsequence of *cps* such that $cps = prefix \bullet cpp^{\alpha}$, where $\bullet$ is the sequence concatenation operator, $\alpha > 1$ is a positive integer or ω (the first infinite ordinal), and cpp^{α} represents the concatenation of α *cpp*'s. We call cpp^{α} the *cpp*-suffix of *cps*. For instance, $[(c_2, p_2), (c_3, p_3)]$ is a child-parent pattern of length two in $[(c_1, p_1), (c_2, p_2), (c_3, p_3), (c_2, p_2), (c_3, p_3)]$, and its $[(c_2, p_2), (c_3, p_3)]$-suffix is $[(c_2, p_2), (c_3, p_3), (c_2, p_2), (c_3, p_3)]$. The *optimal child-parent pattern* in a child-parent sequence *cps* is the *shortest* child-parent pattern, *cpp*, such that the *cpp*-suffix is the longest in *cps* (longest by containment among all suffixes of child-parent patterns in *cps*). For an infinite trace, its child-parent sequence *cps* and the *cpp*-suffixes of any of its child-parent patterns are infinite, but all child-parent patterns have finite lengths. Since there can be only a finite number of unfinished subgoals due to subgoal abstraction, the answer-flow child-parent sequence of a non-terminating trace must have an optimal child-parent pattern (Theorem 3 below).

Let cpp_{opt} be the optimal child-parent pattern for the forest logging trace in question. We use $optimal\_cpp(child, parent)$ to denote the fact that $(child, parent)$ is in cpp_{opt}. As in the call sequence analysis, child-parent relationships in cpp_{opt} are modeled as a graph. An *answer-flow* CPG for a forest logging trace is a directed graph $G_{af} = (\mathcal{N}, \mathcal{E})$, defined as follows. $\mathcal{N}$ is the set of children and parent-subgoals in cpp_{opt}, i.e., $\mathcal{N} = \{sub \mid (sub, ...) \in cpp_{opt} \text{ or } (..., sub) \in cpp_{opt}\}$. Edges in G_{af} are the child-parent pairs in cpp_{opt}, i.e., $\mathcal{E} = \{(child, parent) \mid (child, parent) \in cpp_{opt}\}$. A path in G_{af} is called an *answer-flow path*; such a path is called an *answer-flow loop* if it is a cycle. Two answer-flow loops that consist of the same nodes and edges are considered to be the same and we will keep only one representative loop in such a case. We will see that non-termination implies the existence of an optimal child-parent pattern (Theorem 3).

Theorem 3 (*Completeness of the answer flow analysis*)

Consider a query to a program all of whose predicates are tabled and let the inference engine support subgoal abstraction. If the query evaluation does not terminate, then:

i. there is an optimal child-parent pattern in its complete infinite trace,
ii. $G_{af} = (\mathcal{N}, \mathcal{E})$ contains at least one answer-flow loop,
iii. every $sub \in \mathcal{N}$ appears in at least one answer-flow loop, and
iv. each edge $(sub_1, sub_2) \in \mathcal{E}$, where sub_1 is of the form *predicate*(..., *ruleid*), tells us that sub_2 calls sub_1 from the body of a rule with the id *ruleid*. □

Theorem 4 (*Soundness of the answer flow analysis*)
Consider a query to a program all of whose predicates are tabled. If the complete infinite trace of that query has an optimal child-parent pattern then the query evaluation does not terminate. □

The proof of Theorem 3 is found in Appendix F. Theorem 4 follows directly from the definitions, since the optimal child-parent pattern captures the information flow among unfinished subgoals in a non-terminating computation. These theorems tell us that the subgoals contained in the optimal child-parent pattern of a non-terminating trace, i.e., the nodes of the pattern's answer-flow CPG, are *exactly* the subgoals for which infinitely many answers keep being derived. We call these subgoals the *culprit* unfinished subgoals.

The algorithms for computing the artifacts involved in answer flow analysis, including the optimal child-parent patterns, are provided in Appendix E.

In call sequence analysis, an unfinished-call CPG is constructed and the suspected unfinished-call loops are flagged. Similarly, in answer-flow analysis, one builds answer-flow CPG and computes culprit loops, which shed light on how answers flow among culprit subgoals. The following Theorem 5 connects these two approaches. Its proof can be found in Appendix F.

Theorem 5 (*Relationship between unfinished-call CPGs and answer-flow CPGs*)
Let $G_{uc} = (\mathcal{N}_{uc}, \mathcal{E}_{uc})$ be the unfinished-call CPG and let $G_{af} = (\mathcal{N}_{af}, \mathcal{E}_{af})$ be the answer-flow CPG for a non-terminating forest logging trace. Then $\mathcal{N}_{af} \subset \mathcal{N}_{uc}$, and for every edge $(child, parent) \in \mathcal{E}_{af}$ there is an edge $(parent, child) \in \mathcal{E}_{uc}$. Furthermore, every answer-flow loop is a culprit unfinished-call loop. □

Theorem 6 (*No false-positives for finite traces*)
If the evaluation of a query, Q, terminates, then both the unfinished-call CPG and the answer-flow CPG for Q's forest logging trace are empty. □

Theorem 6, proved in Appendix F, assures that neither the unfinished call nor the answer flow analysis yield false-positive results for finite traces. Of course, for infinite traces, false-positives are possible, as one can inspect only a finite prefix in such cases.

5 Auto-repair of rules

Sometimes query evaluation does not terminate not because the query has infinitely many answers but because one of its subgoals does. In such cases, the query *may* terminate if a different evaluation order for its subgoals is used. This section describes a heuristic for fixing certain non-termination queries by delaying the evaluation of culprit subgoals.

Suppose that $G_{uc} = (\mathcal{N}_{uc}, \mathcal{E}_{uc})$ is the unfinished-call CPG of a non-terminating evaluation. For each $(parent, child) \in \mathcal{E}_{uc}$, we know that the call to *child* from *parent* has not been completed. Moreover, we know:

- the host rule for this call, and
- the common set of the unbound arguments of *parent* and *child*, which are also the arguments whose bindings are to be derived.

To reduce the possibility that *parent* gets an infinite number of bindings from *child* and thus diminish the possibility of non-termination caused by that call to *child*, we can delay the evaluation of *child* in the host rule until the aforesaid unbound arguments get bound. If later in the evaluation it is established that the arguments cannot be bound, the delay of *child* ceases and the subgoal is executed. Similar evaluation delays can be applied to all unfinished calls in $\mathcal{E}_{uc}$.

$\mathcal{F}$LORA-2 and SILK support delay quantifiers of the form `wish`(*cond*) and `must`(*cond*), where *cond* is an `and`/`or` combination of `ground`(*variables*) and `nonvar` (*variables*). This is similar to the `when/2` predicate found in many prologs with the difference being that the delayed subgoal is eventually tried even if the binding conditions are not met. A delayed literal is of the form *delay-quantifier*^*goal*. When such a literal is to be executed, the attached *delay-quantifier* is checked. If the quantifier's condition is satisfied, *goal* is executed immediately. Otherwise, the literal is delayed until such time that the condition is satisfied. If the condition is eventually satisfied, *goal* is called. If the engine determines that satisfying the quantifier's condition is impossible, *goal* is called anyway (in case of the `wish` quantifier) or an error is issued (in case of the `must` quantifier).

Example 2
Consider the program of Example 1. Our auto-repair heuristic will delay the unfinished subgoals by modifying the program as follows:

```
@!r1  p(a).                                @!r5  r(X) :- wish(ground(X))^r(X).
@!r2  p(f(X)) :- wish(ground(X))^q(X).     @!r6  r(X) :- wish(ground(X))^p(X), s(X).
@!r3  q(b).                                @!r7  s(f(b)).
@!r4  q(g(X)) :- wish(ground(X))^p(X).     ?- wish(ground(X))^r(X).
```

The modified program successfully terminates with an answer `X = f(b)`. □

It should be clear, however, that the above is only a heuristic and no automatic fool-proof auto-repair technique is possible, in general. Since `Terminyzer+` serves as a debugging tool, the user needs to tell the system whether it should attempt a repair upon detecting non-termination. A graphical interface can help to ease this process.

6 `Terminyzer+` for tabled logic engines without subgoal abstraction

We now turn to non-termination analysis that does *not* rely on subgoal abstraction. As discussed in Section 1, non-termination may then also be caused by generation of infinitely many subgoals. In this case, `Terminyzer+` analyzes the sequence of unfinished subgoals and reports the predicates and their respective rule ids that form increasingly deep nested subgoals. As before, we assume that users stop the execution after a time limit or when subgoals or answers become too large.

For an unfinished subgoal, its *simplified* version is constructed out of the subgoal's predicate and the rule id as *predicate*(*ruleid*). For instance, $p(f2(f1(a)), r3)$ is simplified to $p(r3)$. The *simplified unfinished subgoal sequence* is the sequence of simplified unfinished subgoals sorted by the order of their first appearance in the trace. When non-termination is due of an infinite number of subgoals, these subgoals must have increasingly deeply nested terms. Since a finite program has only a finite number of predicates and functors, there must be repetitions in the aforesaid sequence of simplified unfinished subgoals.

Similarly to the idea of optimal child-parent pattern in Section 4.2, the *optimal subgoal pattern* of a simplified unfinished subgoal sequence can be computed. This pattern will show which subgoals in which rules recursively call one another and create increasingly deeper and deeper terms.

Example 3
The evaluation of the query `?- r(a)` given the program

```
@!r1  p(a).                  @!r4  r(X) :- r(X).
@!r2  p(X) :- q(f1(X)).      @!r5  r(X) :- p(X), s(X).
@!r3  q(X) :- p(f2(X)).      @!r6  s(a).
```

produces a simplified unfinished subgoal sequence with the prefix `[root, r(_h46), r(r4), r(r4), p(r5), q(r2), p(r3), q(r2), p(r3)]`. The optimal subgoal pattern here is `[q(r2), p(r3)]`, which means that the predicates q in rule $r2$ and p in rule $r3$ are the ones causing the generation of increasingly deep subgoals. □

Theorem 7 (*Soundness and completeness*)
Consider a query to a tabled program and assume that the engine does *not* perform subgoal abstraction. The forest logging trace has an optimal subgoal pattern if and only if the computation is non-terminating due to infinitely many subgoals. □

The proof of Theorem 7 is in Appendix F. Once the optimal subgoal pattern is computed, the user can easily find the subgoals and the rules that are likely causes of non-termination. Note that without subgoal abstraction, the auto-repair technique presented in Section 5 does not apply since no subgoal reordering can cause the query to terminate.

7 Conclusion

`Terminyzer+` extends our previous non-termination analyzer, `Terminyzer` (Liang and Kifer 2013), in several ways to make it practically useful for debugging large programs. First, `Terminyzer+` gives more precise explanations by including the ids of the rules from which the unfinished subgoals were called. This is hugely important even for medium-size rule sets because the subgoals in the loops found by the `Terminyzer`'s algorithms can match multiple rules, which may result in a combinatorially large number of alternatives to be sifted through manually. `Terminyzer+` obviates that work. Rule ids can also be gainfully employed by graphical interfaces suitable for knowledge engineers who are not programmer. Second, a new analysis provided by `Terminyzer+` facilitated a heuristic for fixing non-terminating queries. Third, the use of rule ids made it possible to extend the analysis to systems that do not support subgoal abstraction. The usability of `Terminyzer+` was confirmed by multiple experiments, which include two very large real-life programs.

Acknowledgments

This work is part of the SILK (Semantic Inference on Large Knowledge) project sponsored by Vulcan, Inc. The authors were also partially supported by the NSF grant 0964196. We would like to thank Terry Swift for enabling this work by implementing subgoal abstraction and forest logging in XSB. We also thank Benjamin Grosof, David S. Warren, and the anonymous referees for a number of suggestions that helped improve this paper.

References

Bol, R. N., Apt, K. R. and Klop, J. W. 1991. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science 86,* 1, 35–79.

Bruynooghe, M., Codish, M., Gallagher, J. P., Genaim, S. and Vanhoof, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems 29*, 1–44.

Chen, W., Kifer, M. and Warren, D. S. 1993. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming 15,* 3, 187–230.

Costa, V. S., Damas, L. and Rocha, R. 2012. The YAP prolog system. *Theory and Practice of Logic Programming 12*, 5–34.

Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F. and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1-2, 219–252.

Kifer, M., Lausen, G. and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. *Journal of ACM 42*, 741–843.

Liang, S. and Kifer, M. 2013. Terminyzer: An automatic non-termination analyzer for large logic programs. In *Practical Aspects of Declarative Languages.* Springer-Verlag, Berlin, Heidelberg, New York.

Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. 2004. Proving termination for logic programs by the query-mapping pairs approach. In *Program Developments in Computational Logic*, Springer-Verlag LNCS, Berlin, Heidelberg, 453–498.

Nguyen, M. T. and De Schreye, D. 2007. Polytool: Proving termination automatically based on polynomial interpretations. In *Logic-Based Program Synthesis and Transformation*, Springer-Verlag, Berlin, Heidelberg, 210–218.

Nguyen, M. T., Giesl, J., Schneider-Kamp, P. and De Schreye, D. 2008. Termination analysis of logic programs based on dependency graphs. In *Logic-Based Program Synthesis and Transformation*, Springer-Verlag, Berlin, Heidelberg, 8–22.

Schneider-kamp, P., Giesl, J., StrÖder, T., Serebrenik, A. and Thiemann, R. 2010. Automated termination analysis for logic programs with cut. *Theory and Practice of Logic Programming 10,* 4-6 (July), 365–381.

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

Sipser, M. 1996. *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, Washington, DC.

Swift, T. and Warren, D. S. 2012. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming 12*, 157–187.

Verbaeten, S., De Schreye, D. and Sagonas, K. 2001. Termination proofs for logic programs with tabling. *ACM Transactions on Computational Logic 2,* 1 (January), 57–92.

Zhou, N.-F. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 189–218.