Online appendix for the paper

A Practical Analysis of Non-Termination in Large Logic Programs

published in Theory and Practice of Logic Programming

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 May 2013; accepted 23 June 2013

Appendix A Experiments

`Terminyzer+` has been implemented for the $\mathcal{F}$LORA-*2* and SILK systems, and we report our experiments below. All tests were performed on a dual core 2.4GHz Lenovo X200 with 3 gigabytes of main memory running Ubuntu 11.04 with Linux kernel 2.6.38. The sources of the test programs as well the reports produced by `Terminyzer+` are available online at `http://rulebench.projects.semwebcentral.org/terminyzer+`.

Test programs. Here we include four test cases: T_1, T_2, T_3, and T_4, and none of them terminates. The first three tests are performed with subgoal abstraction enabled, while T_4 was tested without subgoal abstraction. T_1 is the query and the rule set of Example 1. T_2 and T_3 are very large programs which were derived from $\mathcal{F}$LORA-*2* programs used in the SILK project. T_2 has 844 rules and facts, and its corresponding XSB program (after $\mathcal{F}$LORA-*2*-to-XSB translation) is estimated to have 2,000 rules and facts. T_3 consists of 4,774 rules and 919 facts, and its XSB program has over 1,000 facts and over 5,500 rules.[1] T_4 is the program of Example 3.

For T_1 and T_2, we set XSB to abort after the answer depth reached 30. For T_3, we let the evaluation continue until all available memory was consumed. The reason is that T_3 is a really complex program, and in order to get a usable prefix of its infinite trace, we have to let it run "long enough." The execution of T_2 produces a log trace of 3 megabytes with around 26,000 log entries, and the trace for T_3 is in excess of 2 gigabytes with more than 14 million log entries.

Test results. `Terminyzer+` produced expected results in all the test cases. For T_1, `Terminyzer+` constructed the unfinished-call graph shown in Figure 1 and identified its culprit loop. The auto-repair technique presented in Section 5 successfully fixed the non-termination problem as demonstrated in Example 2.

For T_2, `Terminyzer+` determined that the predicate `entailed(X)` of the following rule was generating infinitely many answers:

```
entailed(conjunction(Antecedent1,Antecedent2)) :-
```

[1] We also tested other, fairly large real programs from the SILK project with similarly positive results.

```
entailed(Antecedent1), entailed(Antecedent2).
```

The heuristic auto-repair method of Section 5 fails to fix this non-terminating query since it is the query itself, not its subqueries, that has infinitely many answers.

For T_3, the unfinished-call CPG has 14 nodes and 34 edges, and its answer-flow CPG has 9 nodes and 28 edges. Our auto-repair method successfully removes the cause of non-termination and the remedied program terminates with one answer. We should mention that an experienced knowledge engineer spent hours debugging T_3 — all in vein.

For T_4, `Terminyzer+` successfully identified the optimal subgoal pattern, as described in Example 3.

Computation times. For T_1, T_2, and T_4, `Terminyzer+` took less than 1 second for each program. For the much more complex T_3, it took 170 seconds. Compared to the fruitless hours spent by our knowledge engineer, `Terminyzer+` appears to be a much more inviting alternative.

Appendix B Related Work

There have been many works on termination analysis for logic programs (Bol et al. 1991; Sahlin 1993; Schreye and Decorte 1994; Decorte et al. 1998; Shen 1997; Ohlebusch et al. 2000; Shen et al. 2001; 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; Shen et al. 2010) while *non*-termination analysis received much less attention (Neumerkel and Mesnard 1999; Payet 2007; Payet and Mesnard 2006; Shen et al. 2001; Voets and De Schreye 2009; Shen et al. 2010; Voets and De Schreye 2011). Most of these studies are either *norm*-based or *transformation*-based. In norm-based approaches (Bol et al. 1991; Sahlin 1993; Schreye and Decorte 1994; Decorte et al. 1998; Shen 1997; Shen et al. 2001; Verbaeten et al. 2001; Lindenstrauss et al. 2004; Bruynooghe et al. 2007; Shen et al. 2010; Voets and De Schreye 2011), termination analysis is performed by proving certain well-founded sufficient conditions for termination, which involve norms, i.e., abstractions of the size of a term (e.g., the number of symbols, depth, etc.). Transformation-based algorithms (Neumerkel and Mesnard 1999; Ohlebusch et al. 2000; Payet 2007; Payet and Mesnard 2006; Nguyen et al. 2008; Schneider-kamp et al. 2010) rewrite logic programs so that the termination property of the rewritten program could be used to prove termination of the original program.

There are three main points that differentiate `Terminyzer+`. First, a log-based approach to *debugging* expounded by `Terminyzer+` is fundamentally different from the works on *proving* termination. We do not aim to prove termination because if a query terminates then there is nothing for `Terminyzer+` to do. Second, the problems discussed in most previous work of the subject—except (Decorte et al. 1998; Verbaeten et al. 2001)—are non-issues in our framework, since they stem from the severe incompleteness of the Prolog inference mechanism and, therefore, do not apply to the inference engines under consideration. Third, `Terminyzer+` aims at helping the programmer to debug programs *without syntactic restrictions*. All other approaches perform static or dynamic analysis in order to *prove* termination or non-termination for *restricted* classes of logic programs, such as function-free programs, positive programs, etc. These restrictions, if at all stated, are typically very strong; stated or not, they always exist because both of

the above problems are undecidable. This also applies to (Decorte et al. 1998; Verbaeten et al. 2001), which are the only works that study the termination problem for tabling engines.

Among all these previous studies, only the loop checker approach in (Shen et al. 2001) resembles our analysis of non-termination in the *absence of subgoal abstraction*. This work aims at detecting repetitions of subgoals and clauses, which are akin to `Terminyzer+`'s optimal subgoal patterns. However, there are two major differences between `Terminyzer+`'s analysis in Section 6 and the loop checkers in (Shen et al. 2001). First, Shen et al. work with Prolog without tabling. For instance the following query:

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

terminates without answers in our framework (with subsumptive tabling or subgoal abstraction) and thus is a non-issue at all, while their loop checker will report non-termination because it detects an infinite SLD-derivation. Second, they perform static analysis of the original program clauses and try to detect possible loops, while `Terminyzer+` analyzes logs for actual execution. For instance, this query

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

will be reported as terminating in their framework since there is a successful SLD-derivation. However, this is a drawback because this analysis considers only *some* derivations, while Prolog may explore more. For instance, in the above example, if the user asks for *another answer* by typing a ";" then Prolog will go into an infinite loop. So, in that sense, this analysis is overly optimistic and not completely adequate. In contrast, `Terminyzer+` would consider the actual executions. For tabled engines *without* subgoal abstraction that, like Prolog, return one answer at a time (e.g., the batched engines of XSB and YAP), `Terminyzer+` will report the first successful derivation of `p(a)` as terminating and the subsequent ones as non-terminating. Furthermore, *with* subgoal abstraction, the computation terminates and `Terminyzer+` will report this properly.

Appendix C Tabling and Forest Logging in XSB

In tabled (SLG) evaluation, calls to tabled predicates are cached in a *table* $\mathcal{T}$ for subsequent calls. $\mathcal{T}$ can be viewed as a set of pairs of the form $(sub, ansrs)$ where $ansrs$ are proven instances of sub. When a tabled subgoal, sub, is issued, SLG examines whether there is a pair $(sub', ansrs') \in \mathcal{T}$ such that sub is *similar* (to be explained shortly) to sub'. If so, then answer clause resolution is performed instead of program clause resolution, i.e., $ansrs'$ are used to satisfy sub. In this case, sub is referred to as the *consumer* of sub' while sub' is the *producer* of sub. If no tabled answers can be used above, a new table entry of the form $(sub, ansrs)$ is added to $\mathcal{T}$, where initially $ansrs = \emptyset$. Then sub is resolved against program clauses, as usual in Prolog. In such a case, all newly derived answers for sub are added to $ansrs$, sub becomes a producer of these answers, and all subsequent subgoals that are similar to sub become consumers of sub's answers.

There are two main ways to define subgoal-similarity mentioned above. Depending on which notion is chosen, the tabling strategy is called *variant* or *subsumptive*. In variant tabling, sub is similar to sub' if sub is a *variant* of sub', i.e., they are identical up to

variable renaming. In subsumptive tabling, *sub* is similar to *sub′* if *sub* is *subsumed* by *sub′*, i.e., there is a variable substitution σ such that $\sigma(sub') = sub$. Note that in this case the notion of similarity is asymmetric. Since only unique answers are added to the table and returned to consumers, tabled evaluation terminates *if* there is only a finite number of tabled subgoals and each tabled subgoal has finitely many answers. For instance, this is the case in Datalog, i.e., when function symbols are not present.

The workings of SLG resolution can be captured by an *SLG forest*, which has an *SLG tree* for every new (dissimilar) subgoal to a tabled predicate. The SLG tree for *sub* has root of the form *sub* :- *sub*, and each non-root node is of the form $\theta(sub)$:- $\theta(left\_subs)$, where θ is the substitution obtained from resolving *sub* against the knowledge base and $\theta(left\_subs)$ are the remaining subgoals needed to prove *sub*. If $\theta(left\_subs)$ is an empty clause, $\theta(sub)$ is an answer to *sub*. Children of a root node are obtained through resolution of a tabled subgoal against program clauses. Children of non-root nodes are obtained through answer clause resolution if the leftmost selected literal is tabled or through program clause resolution if the leftmost selected literal is not tabled. Each edge in the tree corresponds to a derivation step of program or answer clause resolution.

Example 1
The SLG forest for the following XSB program is shown in Figure C 1, where each node is labeled with an ordinal denoting the creation order (a timestamp) of the node during evaluation.

```
:- table path/2.
edge(1,2).    edge(1,3).    edge(2,1).
path(X,Y) :- edge(X,Y).     path(X,Y) :- edge(X,Z), path(Z,Y).
?- path(1,Y).
```

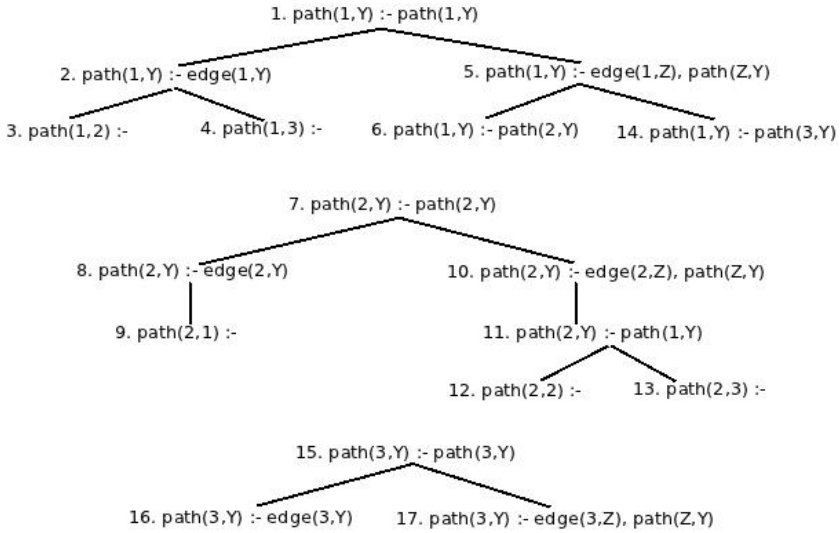

Fig. C 1. The SLG Forest for Example 1

Children of node 11 are obtained through answer clause resolution since its leftmost selected literal `path(1,Y)` is tabled, while children of nodes 2 and 5 are obtained through

answer clause resolution because their leftmost selected literals not tabled. This is a simplified version of an example in (Swift et al. 2013). □

As introduced in Section 2, XSB provides a new facility, called `logforest`, which makes the table events available to the programmer and thus helps program debugging and optimization.

Example 2
For the SLG forest of Example 1, the `logforest` trace is given in the first column of Table C 1. The second column in the table is the label of the node in the trees of Figure C 1

Log	Node	Explanation
`tc(path( 1,_v0),root,new,0)`	1	initial call
	2	program clause resolution
`na([ 2],path( 1,_v0),1)`	3	program clause resolution, new answer
`na([ 3],path( 1,_v0),2)`	4	program clause resolution, new answer
	5	program clause resolution
	6	program clause resolution
`tc(path( 2,_v0),path( 1,_v0),new,3)`	7	new call made by node 6
	8	program clause resolution
`na([ 1],path( 2,_v0),4)`	9	program clause resolution, new answer
	10	program clause resolution
`tc(path( 1,_v0),path( 2,_v0),incmp,5)`	11	repeated unfinished call
`ar([ 2],path( 1,_v0),path( 2,_v0),6)`	12	answer clause resolution
		answer to consumer
`na([ 2],path( 2,_v0),7)`	12	new answer
`ar([ 3],path( 1,_v0),path( 2,_v0),8)`	13	answer clause resolution
		answer to consumer
`na([ 3],path( 2,_v0),9)`	13	new answer
	14	program clause resolution
`tc(path( 3,_v0),path( 1,_v0),new,10)`	15	new call made by node 14
	16	program clause resolution
	17	program clause resolution
`cmp(path( 3,_v0),3,11)`	15	evaluation completed
`ar([ 1],path( 2,_v0),path( 1,_v0),12)`	9	return to consumer
`na([ 1],path( 1,_v0),13)`	6	new answer
`ar([ 2],path( 2,_v0),path( 1,_v0),14)`	12	return to consumer
`ar([ 3],path( 2,_v0),path( 1,_v0),15)`	13	return to consumer
`ar([ 1],path( 1,_v0),path( 2,_v0),16)`	1	return to consumer
`cmp(path( 1,_v0),1,17)`	1	evaluation completed
`cmp(path( 2,_v0),1,18)`	7	evaluation completed

Table C 1. *Forest Log for the Evaluation of Example 1*

where a corresponding event happens. The third column is an explanation. An answer for a subgoal is represented as a substitution for the list of variables in the subgoal. For instance, in the second log entry $na([2], path(1, \_v0), 1)$, the answer is represented as $[2]$ and the list of variables in the subgoal $path(1, \_v0)$ are $[\_v0]$. It means that the substitution $\_v0 = 2$ is an answer. □

Appendix D Unfinished-Call CPG, Path, and Loop Computation

Algorithm 1, below, constructs the unfinished-call CPG $G_{uc} = (\mathcal{N}, \mathcal{E})$ from the set of unfinished calls of a forest logging trace. Construction starts by adding the root call to the CPG. Then, for each log record of the form *unfinished*(*child*, *parent*, *timestamp*) such that *parent* is already in the CPG, the node *child* and the edge (*parent*, *child*) are added, if *child* has not been added before. All unfinished calls are processed in the order of their timestamps, i.e., their addition to the log, which is also the order in which these unfinished calls are made during evaluation. Thus, when the record *unfinished*(*child*, *parent*, *timestamp*) is encountered, we know that *parent* must have been added to the graph as an child-subgoal of its parent, i.e., *unfinished*(*parent*, p', *timestamp*$'$) must be true for some p' and *timestamp*$' <$ *timestamp*. We have two cases:

1. *child* $\in \mathcal{N}$. The evaluation calls a previously issued subgoal.
2. *child* $\notin \mathcal{N}$. A new subgoal is called and a new node is added to the graph.

In the first case, an unfinished-call loop exists, so the current evaluation path of *parent* is suspended and alternative derivations is explored. This implies an important property of unfinished-call CPGs: an unfinished-call loop is created out of an (acyclic) path always by adding a final edge of the form (sub_1, sub_2), where sub_1.*timestamp* $\geq$ sub_2.*timestamp*. We call such an edge a *critical loop edge*—see the edges labeled with 11 and 24 in Figure 1 of Example 1.

Let UC be the set of all unfinished calls $\mathcal{E} = \emptyset$; $\mathcal{N} = \{root\}$; $root.timestamp = -1$;
while $UC \neq \emptyset$ **do**
  Remove *unfinished*(*child*, *parent*, *timestamp*) from UC, where *timestamp* is the smallest among UC;
  if *child* $\notin \mathcal{N}$ **then** $\{\mathcal{N} = \mathcal{N} \cup \{child\};\ child.timestamp = timestamp\}$;
  $\mathcal{E} = \mathcal{E} \cup \{(parent, child)\};\ (parent, child).timestamp = timestamp$;
end
return $G_{uc} = (\mathcal{N}, \mathcal{E})$

Algorithm 1: Unfinished-Call CPG Construction

If critical loop edges are taken out, any unfinished-call CPG becomes a connected directed acyclic graph (i.e., a tree) in which every edge goes from a node with a smaller timestamp to a node with a larger timestamp. A path, connecting the root node to a subgoal, can be represented as the predicate `uc_path(Sub,Path)` defined by the following rules.

```
:- table reversed_uc_path/2.
uc_path(C,P) :- reversed_uc_path(C,RevP), reverse(RevP,P).
reversed_uc_path(C,[C,root]) :- unfinished(C,root,_).
reversed_uc_path(C,[C|P]) :- unfinished(C,Parent,_),
    Parent.timestamp<C.timestamp, reversed_uc_path(Parent,P).
```

After computing all unfinished-call paths, without critical loop edges, from *root* to other nodes, all distinct unfinished-call loops can be computed by checking whether there exists a critical loop edge from the last vertex of a path to any other node in the same path. Consider an unfinished-call path $P = [root, sub_1, \ldots, sub_n]$. If there is a critical loop edge (sub_n, sub_i), $1 \leq i \leq n$, then the part of P from sub_i to sub_n, $[sub_i, \ldots, sub_n]$, is an unfinished-call loop.

Appendix E Answer-Flow CPG Construction

Given a child-parent sequence (defined in Section 4.2), let *pat* be the subsequence containing the last n elements in the sequence. The predicate `pattern`(*cps*, *len*, *pat*, *times*) specifies the number of times a child-parent pattern *pat* of length *len* repeats at the end of *cps*. Patterns of different lengths can be computed by posing `?- pattern`(*cps*, *len*, *Pat*, *Times*) to the following rules, where the *len* parameter successively assumes the values 1, 2, and so on. In this way, we will either find an optimal child-parent pattern or determine that there is no pattern.

```
pattern(CPS,Len,Pat,Times) :-
    length(Pat,Len),
    %% This binds Pat to the suffix of CPS of length Len
    append(CPSPrefix,Pat,CPS),
    aux_pattern(CPSPrefix,Pat,Times).
aux_pattern(CPS,Pat,Times) :-
    append(CPSPrefix,Pat,CPS), !,
    pattern(CPSPrefix,Pat,TimesPrefix),
    Times is TimesPrefix+1.
aux_pattern(CPS,Pattern,1).
```

Example 3
The child-parent sequence of the forest logging trace for Example 1 is the *cps* below:

```
[(q(_h599,r2),p(_h599,r4)), (p(_h599,r4),q(_h599,r2)), (q(_h619,r2),p(_h619,r4)),
 (p(_h639,r4),q(_h639,r2)), (q(_h659,r2),p(_h659,r4)), (p(_h679,r4),q(_h679,r2)),
 (q(_h699,r2),p(_h699,r4)), (p(_h719,r4),q(_h719,r2)), (q(_h739,r2),p(_h739,r4)),
 (p(_h759,r4),q(_h759,r2)), (q(_h779,r2),p(_h779,r4))].
```

This *cps* has two child-parent patterns. The first is $cpp_1 = [(\mathtt{p}(\mathtt{\_h759}, \mathtt{r4}), \mathtt{q}(\mathtt{\_h759}, \mathtt{r2})), (\mathtt{q}(\mathtt{\_h779}, \mathtt{r2}), \mathtt{p}(\mathtt{\_h779}, \mathtt{r4}))]$ of length two and it repeats five times. The second one is $cpp_2 = cpp_1^2$ of length four, which repeats twice. The optimal child-parent pattern is cpp_1, as it covers $2 \times 5 = 10$ entries in *cps* compared to cpp_2, which covers only $4 \times 2 = 8$ entries. □

Let G_{af} be the answer-flow CPG for the forest logging trace in question. Its answer-flow paths and loops can be computed in a way similar to the computation of unfinished-call paths and loops. All answer-flow paths from node *child* to node *parent* can be computed using the predicate `af_path`(*child*, *parent*, *path*); all answer-flow loops starting from *child* can be computed using the predicate `af_loop`(*child*, *loop*), defined below.

```
:- table af_path/3.
af_path(Child,Parent,[Child]) :- optimal_cpp(Child,Parent).
af_path(Child,Parent,[Child|P]) :- optimal_cpp(Child,Sub),
    af_path(Sub,Parent,P), \+ member(Child,P).
af_loop(Sub,Loop) :- af_path(Sub,Sub,Loop).
```

Example 4
Consider cpp_1, the optimal child-parent pattern of Example 3. Its answer-flow graph is the subgraph shown inside the rectangle in Figure 1. The only answer-flow loop is [16, 20, 16], which tells us that subgoal *p* called from rule *r4* and subgoal *q* called from rule *r2* return answers to each other in an infinite answer derivation loop. □

Appendix F Proofs of Theorems

Proof of Theorem 1
(i) Clearly, non-termination can be caused only by unfinished calls. As described in Section 2, either $tc(child, parent, \_, timestamp)$ or $nc(child, parent, \_, timestamp)$ must be logged whenever a tabled subgoal *child* is called by *parent*, and only when *child* is completely evaluated, $cmp(child, \_, timestamp)$ is recorded. The timestamps of these log entries preserve the sequential order of the corresponding events. Therefore, the sequence of unfinished calls defined in Section 4.1, sorted by their timestamps, records exactly those unfinished calls that cause that specific non-termination.

(ii) The program transformation described in Algorithm 1 generates a new rule id for each rule and embeds it in each of the rule's tabled body subgoals as the last argument, and these rule ids appear in each call to these subgoals. Therefore, each log entry for each unfinished subgoal includes the id of the rule calling that subgoal. □

Proof of Theorem 2
(i) There has to be at least one loop. Suppose there is no unfinished-call loop in the corresponding unfinished-call CPG $G_{uc} = (\mathcal{N}, \mathcal{E})$. Subgoal abstraction ensures that only a finite number of calls to tabled predicates can exist, so G_{uc} is a finite graph. Since there is no unfinished-call loop, there must be terminal nodes that have no outgoing edges. Let $\mathcal{S} \subseteq \mathcal{N}$ denote this set of nodes. It means that their SLG-children are not in $\mathcal{N}$, i.e., they are not unfinished subgoals. Therefore the SLG-children of $\mathcal{S}$ are either completely evaluated tabled subgoals or base facts. But then, after long enough time, all subgoals in $\mathcal{S}$ should have been completely evaluated and completed. This contradicts the assumption that $\mathcal{S} \subseteq \mathcal{N}$, i.e., the subgoals in $\mathcal{S}$ are unfinished.

At least one of the loops must be responsible for the generation of an infinite number answers; otherwise all answers would be derived and the evaluation would terminate.

(ii) This is proved by the same argument as in Theorem 1 (ii). □

Proof of Theorem 3
(i) There can be only a finite number of unfinished subgoals due to subgoal abstraction, and thus there must be at least one child-parent pattern. Otherwise the evaluation would have terminated. Therefore, an optimal child-parent pattern must exist in the forest logging trace.

(ii) Suppose there is no answer-flow loop in G_{af}. There must be a set $\mathcal{S} \subseteq \mathcal{N}$ of terminal nodes and, since these nodes are terminal, the graph has no edges going out of $\mathcal{S}$. The SLG-children of these terminal nodes are therefore not in $\mathcal{N}$ and answers for these SLG-children are *not* being repeatedly derived. Recall that, due to subgoal abstraction, G_{af} can have only a finite number of nodes and, if we let the engine run long enough, all possible edges in G_{af} will be generated and further computation will not change that graph. Therefore, the nodes for which answers are not derived repeatedly cannot stay unfinished (in the sense of unfinished SLG subgoals) infinitely long. So, after a while, all SLG-children of $\mathcal{S}$ must either become completely evaluated tabled subgoals or they must have been base facts all along. This implies that, given enough time, all subgoals in $\mathcal{S}$ would be completed, contrary to the assumption that $\mathcal{S} \subseteq \mathcal{N}$. Therefore, there must be an answer-flow loop.

(iii) If $sub \in \mathcal{N}$ and *sub* is not contained in any answer-flow loop, then *sub*'s evaluation would have been completed and it cannot be part of any child-parent pattern, a contradiction.

(*iv*) Consider an edge $(sub_1, sub_2) \in \mathcal{E}$, where sub_1 is of the form $predicate(..., ruleid)$. We know sub_2 calls sub_1 and sub_1 keeps returning answers to sub_2, by the definition of the edges in G_{af}. It follows from the argument made in (ii) of Theorem 1 that this call of sub_1 must have been made from the rule with the id *ruleid*. □

Proof of Theorem 5
If $sub \in \mathcal{N}_{af}$ then it must be an unfinished subgoal, since answers to *sub* continue to be derived. That is, the evaluation of *sub* has not been completed and $\mathcal{N}_{af} \subseteq \mathcal{N}_{uc}$. In fact, we even have that $\mathcal{N}_{af} \subset \mathcal{N}_{uc}$, since $root \in \mathcal{N}_{uc} \setminus \mathcal{N}_{af}$. For any edge $(child, parent) \in \mathcal{E}_{af}$, we know that *child* returns answers to *parent*, i.e., it is issued in a SLG tree for *parent*. Therefore $(parent, child) \in \mathcal{E}_{uc}$. This implies that any answer-flow loop is also an unfinished-call loop. □

Proof of Theorem 6
We know that the set of nodes of the unfinished-call CPG for a trace is its set of unfinished subgoals. In case of a terminating evaluation, all subgoals are completed and thus there are no unfinished subgoals, i.e., its unfinished-call CPG must be empty, as it has no nodes. It follows from Theorem 5 that the corresponding answer-flow CPG is likewise empty. □

Proof of Theorem 7
Soundness: If an optimal subgoal pattern exists then the evaluation does not terminate. Indeed, if the evaluation terminates, there would be no unfinished subgoals and thus no optimal subgoal pattern. Suppose the evaluation produces only a finite number of subgoals. Since there are only two causes for non-termination in a tabled logic engine *without* subgoal abstraction—infinite number of answers or infinite number of subgoals—non-termination must be due to an infinite number of answers. As described in Section 4.2, this means that a *finite* subset of these subgoals, contained in the trace's optimal child-parent pattern, keeps receiving, deriving, and returning answers. Since there is an optimal subgoal pattern, this requires certain predicates from certain rules to recursively and *repeatedly* call each other, supplying deeper and deeper terms as arguments. These calls would then be causing new subgoals of bigger and bigger sizes to appear in the child-parent pattern, contrary to the assumption that the evaluation produces only a finite set of subgoals.

Completeness: As discussed above, when non-termination happens because of an infinite number of subgoals, these subgoals must have increasingly deep function terms as arguments, and these subgoals' predicates must be recursive. Otherwise there would be only a finite number of terms in a finite program. Therefore, there must be repetitions in the simplified unfinished subgoal sequence of the trace, which implies that there must be an optimal subgoal pattern. □

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.

Decorte, S., De Schreye, D., Leuschel, M., Martens, B., and Sagonas, K. F. 1998. Termination analysis for tabled logic programming. In *Logic-based program synthesis and transformation*. Springer-Verlag, London, UK, UK, 111–127.

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.

Neumerkel, U. and Mesnard, F. 1999. Localizing and explaining reasons for non-terminating logic programs with failure-slices. In *International Conference on Principles and Practice of Declarative Programming*. Springer-Verlag, London, UK, 328–342.

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.

Ohlebusch, E., Claves, C., and March, C. 2000. TALP: A tool for the termination analysis of logic programs. In *Rewriting Techniques and Applications*. Springer-Verlag, LNCS, Berlin, Heidelberg, New York, 270–273.

Payet, E. 2007. Detecting non-termination of term rewriting systems using an unfolding operator. In *Logic-based program synthesis and transformation*. Springer-Verlag, Berlin, Heidelberg, 194–209.

Payet, E. and Mesnard, F. 2006. Nontermination inference of logic programs. *ACM Transactions on Programming Languages and Systems 28*, 256–289.

Sahlin, D. 1993. Mixtus: An automatic partial evaluator for full prolog. *New Generation Computing 12,* 1 (March), 7–51.

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.

Shen, Y.-D. 1997. An extended variant of atoms loop check for positive logic programs. *New Generation Computing 15,* 2 (May), 187–203.

Shen, Y.-d., De Schreye, D., and Voets, D. 2010. Termination prediction for general logic programs. *Theory and Practice of Logic Programming 9,* 6 (Jan.), 751–780.

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

Swift, T., Warren, D. S., Sagonas, K., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Marques, R. F., Saha, D., Dawson, S., and Kifer, M. 2013. *The XSB System, Version 3.3.x. Volume 1: Programmer's Manual.* XSB documentation.

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

Voets, D. and De Schreye, D. 2009. A new approach to non-termination analysis of logic programs. In *Int'l Conference on Logic Programming*. Springer-Verlag, Berlin, Heidelberg, 220–234.

Voets, D. and De Schreye, D. 2011. Non-termination analysis of logic programs using types. In *Logic-based program synthesis and transformation*. Springer-Verlag, Berlin, Heidelberg, 133–148.