Undefined 0 (2013) 1–0
IOS Press

Trace-Based Analysis of Large Rule-Based Computations

Terrance Swift
CENTRIA, Departamento de Informática, Faculdade de Ciencia e Tecnologia, Universidade Nova de Lisboa, Portugal. E-mail: terranceswift@gmail.com.

Abstract. Knowledge representation systems based on the well-founded semantics can offer the degree of scalability required for semantic web applications and make use of expressive semantic features such as Hilog, frame-based reasoning, and defeasibility theories. Such features can be compiled into Prolog tabling engines that have good support for indexing and memory management. However, due both to the power of the semantic features and to the declarative style typical of knowledge representation rules, the resources needed for query evaluation can be unpredictable. In such a situation, users need to understand the overall structure of a computation and examine problematic portions of it. This problem, of *profiling* a computation, differs from debugging and justification which address why a given answer was or wasn't derived, and so profiling requires different techniques. In this paper we present a trace-based analysis technique called *forest logging* which has been used to profile large, heavily tabled computations. In forest logging, critical aspects of a tabled computation are logged; afterwards the log is loaded and analyzed. As implemented in XSB, forest logging slows down execution of practical programs by a constant factor that is often small; and logs containing tens or hundreds of millions of facts can be loaded and analyzed in minutes.

Keywords: Scalable Reasoning, Tabled Resolution, Trace-Based Analysis

1. Introduction

Much of the literature on knowledge representation and reasoning (KRR) has concerned the use of expressive reasoning components such as ASP and $\mathcal{ALC}$-based description logics. However, there has also been interest in basing KRR systems on weaker deductive methods that more easily offer the type of scalability needed by semantic web applications. For description logics an example of such an approach is the $\mathcal{EL}$ family [2]. For rule-based systems, examples are Flora-2 [17] and its commercial extensions: the Silk and Fidji systems[1], all of which are based on logic programming under the well-founded semantics. The Fidji system, for instance, is currently used as a KRR tool to use web-based textual information to reason about financial regulations and medical informatics. Silk and Fidji support features that are not common for rule-based systems, including the object-oriented syntax of F-logic [9], higher-order syntax based on Hilog [3], rule descriptors, the intermixture of defeasibility theories [16], and the use of bounded rationality through a technique called *restraint* [6], along with various types of quantitative reasoning.

The use of these features can lead to concise representation of knowledge, but also to unpredictability in the time and space a computation requires. This unpredictability especially emerges when a knowledge base is produced by a team of knowledge engineers working in a loosely coordinated manner to create rules that may depend on one another. In such situations, the question arises whether the size of a resource intensive computation is due to the sophistication of the reasoning it requires; to redundant or unoptimized rules; or to rules that are simply incorrect. The following example illustrates a case that arose during a KRR effort for the SIlk project.

Example 1.1 *Over the course of several months, portions of the Cyc reasoner [2] and knowledge base were translated and compiled first into Flora-2 and then into XSB [15]. In addition, several hundred AP biology questions were then formulated and queried. The translated system was able to answer some of these*

[1] http://silk.semwebcentral.org, http://coherentknowledgesystems.com
[2] http://www.cyc.com.

questions quickly, often in less than 1 second of CPU time. Other questions took half a minute or more; while still others could not be answered because of timeouts, or because of aborts due to lack of memory. In general, a medium-sized query might take several minutes to execute.

Silk and Fidji are implemented using XSB [15], so that their operational semantics ultimately is based on tabled logic programming. In fact, because of the use of frames, defeasibility and Hilog, user predicates in Flora-2 and its extensions are tabled unless they are explicitly declared otherwise – a default that is the exact opposite of tabling in Prolog. To investigate the time and space required for queries like those of Example 1, a knowledge engineer who understood the operational semantics of Silk would use information about the tables to help determine why a computation was costly. For instance, she might want to examine which tabled subgoals were queried most often; how the answers were distributed among the tables; how the queries depended on one another; and how those dependencies affected the overall search. These questions indicate a need to model a tabled evaluation as a structure that can be examined in itself. Accordingly, we denote the problem of exploring large tabled computations as the *Profiling Problem*. Because profiling addresses the nature of a computation as a whole, rather than why given solutions are returned or omitted, it differs from previously reported approaches based on procedural or declarative debugging or on justification (e.g., [7,12]).

This paper presents *forest logging*, an approach to the profiling problem based on a trace-based analysis of SLG forests, an operational semantics for tabling. As its name implies, operational aspects of a computation are written to a log that is later loaded and analyzed. Specifically,

- We present the design of the logs, and formalize their properties; in particular we show how logs preserve dependency information, and specify the conditions under which the logs can construct a homomorphic image of an SLG forest.
- We present analysis predicates to display operational information about a tabled computation in an efficient manner, and describe how these routines can be customized in order to represent dependency and other information at different levels of abstraction.
- We show that the overhead of logging is a constant factor. We demonstrate the scalability of log analysis which can load and A analyze logs of hundreds of millions of facts.

Section 2 informally reviews SLG and presents the format of forest logs. Some basic properties are shown in Section 3, while Section 4 discusses the analysis routines and describes the implementation of forest logging along with performance results. Related work is covered in Section 6.

All forest logging features discussed in this paper are available in the latest release of XSB (version 3.4). In addition, these features form the basis of the forest logging library in the publically available version of Flora-2 (version 0.99.3), as well as in the commercial Silk and Fidji systems.

2. Representing an SLG Forest via a Log

In SLG resolution [4] as formulated in [14], an evaluation is a sequence of forests of SLG trees. Before discussing the logs themselves, we review those aspects of the forest of trees model for SLG that are necessary to understand forest logging and its applications. As SLG and its extensions have been presented in the literature our review is largely informal; for formal definitions see the references contained in [15]. All code examples are in Prolog syntax.

2.1. A Review of SLG by Examples

We begin our review with an example of SLG evaluation of a query to a definite program. For simplicity, in this paper we restrict our attention to finitely terminating evaluations (which correspond to finite forests), and always assume a left-to-right literal selection strategy[3].

Example 2.1 *Figure 1 shows a simple program along with an SLG forest for the query* reach(1,Y) *to the right-recursive tabled predicate* reach/2. *An SLG forest consists of an SLG tree associated with each tabled subgoal* S *(where variant subgoals are considered to be identical); each such tree has root* S:-|S. *Each SLG operation transforms a given forest* $\mathcal{F}_n$ *to a new forest* $\mathcal{F}_{n+1}$ *by adding a new tree, adding a new node, or by annotating a tree: so that an SLG tree represents*

[3] For presentation purposes we consider only tabling with call variance, and under the local scheduling strategy. However the forest logging features described here are also implemented for call subsumption and other scheduling strategies.

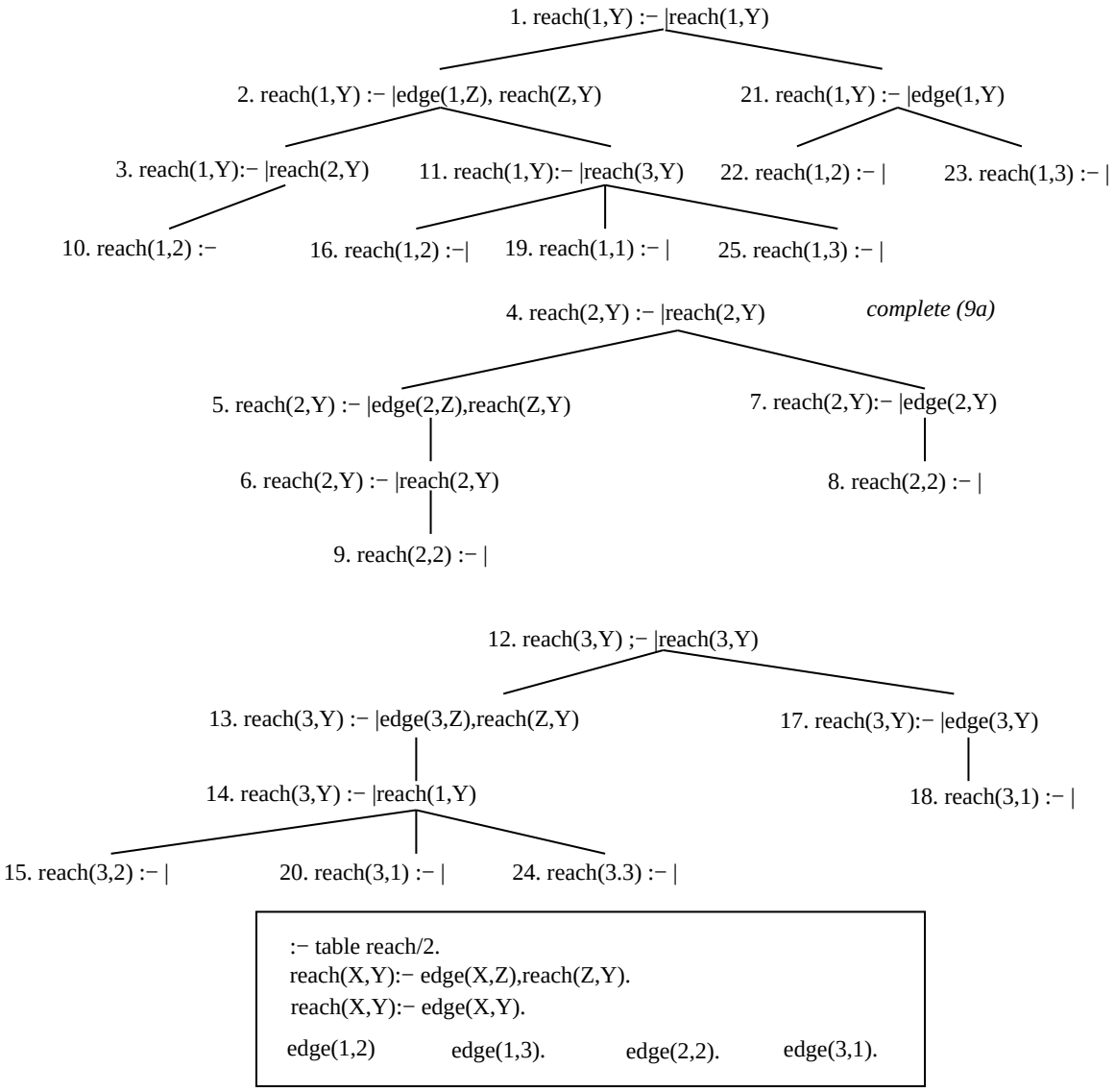

Fig. 1. A Definite Program and SLG Forest for Evaluation of the Query *reach(1,Y)*

the resolution steps that have been executed to derive answers for S.

Given an SLG tree $\mathcal{T}$ with root S :- |S, *$\mathcal{T}$ is sometimes referred to as* the tree for S. *In general, nodes of an SLG tree for* S *have the form* (S :- Delays|Goals)θ*; where* Goals *is the sequence of literals remaining to prove* $S\theta$*;* Delays *are used for negation and are explained below, as are the numbers associated with each node. Children of a root node are obtained through resolution against program clauses, modeled in SLG by the operation* PROGRAM CLAUSE RESOLUTION. *Children of non-root nodes are obtained through the SLG* ANSWER RESOLUTION *operation if the left most selected literal is tabled (e.g. children of the node* reach(1,Y):- | reach(2,Y))[4]*; or via* PROGRAM CLAUSE RESOLUTION *if the leftmost selected literal is not tabled (e.g. children of the node* reach(1,Y):- | edge(1,Z),reach(Z,Y)*). Nodes with empty* Goals *are termed* answers.

[4]We slightly abuse terminology since it is the predicate symbol of the atom within the literal that is tabled. We further abuse terminology by sometimes using selected literal to refer to the underlying atom on which the literal is based, when it is clear to do so.

The evaluation keeps track of each tabled subgoal S *that it encounters by creating a tree for* S *via the* NEW SUBGOAL *operation. Later if* S *is selected again, resolution will use answers from the tree for* S *rather than program clauses; if no answers are available, the computation will* suspend *and try to derive answers using some other computation path. Once additional answers have been derived, the evaluation will* resume *the suspended computation. Similarly, after a computation has resolved all answers available for* S *in a given state, the computation path will suspend, and resume after further answers are found. When it is determined that a (perhaps singleton) set $\mathcal{S}$ of subgoals can produce no more answers, the tree for every subgoal in $\mathcal{S}$ is marked as* complete *(cf. the tree for* reach(2,Y) *in Figure 1). In an implementation, stack space and other resources for a completed subgoal* S *can be reclaimed — apart from the table for* S *consisting of* S *and its answers.*

As seen from Example 2.1, a tabled evaluation evaluates mutually dependent sets of subgoals, marking them as complete when it is no longer possible to derive answers for these subgoals. In this way, a tabled evaluation can be viewed as a series of fixed point computations for sets of interdependent subgoals.

Much of the operational state of a SLG forest $\mathcal{F}$ can be captured by a *Subgoal Dependency Graph*.

Definition 2.1 (Subgoal Dependency Graph) *Let $\mathcal{F}$ be a forest, and let S_1:-|S_1 be the root of a non-completed tree in $\mathcal{F}$. The subgoal S_1* directly depends on *a subgoal S_2 iff S_2 is not completed in $\mathcal{F}$, and there is some node N in the tree for S_1 such that S_2 is the underlying subgoal of the selected literal of N.*

The Subgoal Dependency Graph of $\mathcal{F}$ SDG($\mathcal{F}$) = (V,E) *of $\mathcal{F}$ is a directed graph in which $(S_i, S_j) \in E$ iff subgoal S_i directly depends on subgoal S_j, and V is the underlying set of E. S_1* "depends on" *S_2 in $\mathcal{F}$ is there is a path from S_1 to S_2 in $SDG(\mathcal{F})$.*

Since $SDG(\mathcal{F})$ is a directed graph, sets of subgoals that are mutually recursive in $\mathcal{F}$ can be captured as *Strongly Connected Components (SCCs)* of $SDG(\mathcal{F})$. In Figure 1, there is a single SCC consisting of *reach(1,Y)* and *reach(3,Y)*, as *reach(2,Y)* is complete. While SCCs are critical for determining when subgoals can be completed, if an answer for a tabled subgoal *S* is derived that has the empty substitution, every ground atomic fact that unifies with *S* is true in the model of the program. Accordingly, *S* can be completed before the other subgoals in its SCC through *early completion*. Otherwise, a subgoal *S* can be completed when all possible resolution steps have been performed for *S* and the other subgoals in its SCC.

Understanding the changing dependencies of an evaluation is critical to a number of operational aspects. For instance, local scheduling restricts operations so that there is always a unique maximal independent SCC – that is, an SCC $\mathcal{S}$ whose subgoals depend on no other (non-completed) subgoals that are not in $\mathcal{S}$ itself. Local evaluation is efficient for many applications since it can be shown that it performs a "depth-first" search through SCCs. The number associated with each node in Figure 1 correspond to the node's creation under local evaluation.

2.1.1. Normal Programs

Arguably, the main difference between SLG resolution and other tabling methods is the use of DELAYING and SIMPLIFICATION to handle default negation.

Example 2.2 *Figure 2 shows a program with negation, P_{norm} and illustrates SLG resolution for the query* p(c) *to P_{norm}. The nodes in Figure 2 have been annotated with the order in which they were created under local scheduling; and as mentioned in Example 1, the symbol | in a node separates the unresolved goals to its right from the delayed goals to its left. In the evaluation state where nodes 1 through 10 have been created,* p(b) *has been completed, and* p(a) *and* p(c) *are in the same SCC. There are no more clauses or answers to resolve, but* p(a) *is involved in a loop through negation with itself in node 5, and nodes 2 and 10 involve* p(a) *and* p(c) *in a negative loop* [5].

In situations such as this, where all resolution has been performed for nodes in an SCC, an evaluation may have to apply a DELAYING *operation to a negative literal such as* not(p(a)), *in order to explore whether other literals to its right might fail. When multiple literals can be delayed (e.g., in nodes 2 and 10), an arbitrary literal is chosen to be delayed first. So the evaluation delays the selected literal of node 2 to generate node 12 producing a* conditional answer *– an answer with a non-empty* Delays *set. Next,* not p(a) *in node 5 is delayed, so that the new selected literal for its child, node 13, is* not p(b). *Since node 8 is an answer for* p(b) *with empty* Delays *(termed an* unconditional *answer), a* NEGATIVE RETURN *operation causes that computation path to fail (represented by*

[5] For expository purposes, we ignore the effects of early completion which would complete *p(b)* immediately upon creation of node 8, obviating the need to create node 9.

node 14, termed a failure node*). Afterwards* not p(c) *in node 10 is delayed to produce node 15, and a* NEGATIVE RETURN *operation fails the final computation path for* p(a). *At this stage the SCC* $\{p(a), p(c)\}$ *is* completely evaluated *meaning that there are no more operations applicable for goal literals (as opposed to delay literals). Since* p(a) *is completely evaluated with no answers, conditional or otherwise, the evaluation determines it to be* failed *and a* SIMPLIFICATION *operation can be applied to the conditional answer of node 12, removing* not p(a) *from its* Delays*. leading to the* unconditional *answer in node 17 and* success *of the literal* p(c)*.*

2.2. The Forest Log

Forest logging allows one to run a tabled query and produce a log from which a number of properties of the SLG forest can be inferred. The design of the log attempts to balance several goals: the log should be as informative as possible, but also easy to use and should not overly slow down computations. The log consists of Prolog-readable facts that may be loaded and analyzed, leading to the need to support quick load times and scalable analysis routines. The log facts described below correspond directly to SLG operations, except as noted. Each log fact has a counter $Cntr$, indicating the ordinal number of the fact within the log. Since logs can be very large, an effort is made to keep only the most critical information in the logs so that their memory footprint is kept to a minimum.

– *A call to a tabled subgoal* When a literal L is selected in a node N, where N is in the tree for S_{caller} and L is positive ($L = S_{called}$) then a fact

$$tc(S_{called}, S_{caller}, State, Cntr)$$

is logged. *State* is

- *new* if S_{called} is a new subgoal
- *cmp* if S_{called} is not a new subgoal and has been completed
- *incmp* if S_{called} is not a new subgoal but has *not* been completed

If $L = not(S_{called})$, a fact

$$nc(S_{called}, S_{caller}, State, Cntr)$$

is logged instead.

Note that if $state = new$, *tc/4* and *nc/4* correspond to the NEW SUBGOAL operation; otherwise they do not directly correspond to an SLG operation, but instead they directly log dependency information. If S_{called} is the first tabled subgoal called in an evaluation, then S_{called} is set to *null*.

– ANSWER RESOLUTION When an answer $S_{called}\theta$ is returned to a selected positive literal S_{called} in a tree for S_{caller}, a fact

$$ar(\theta, S_{called}, S_{caller}, Cntr)$$

is logged if A is unconditional and a fact

$$dar(\theta, S_{called}, S_{caller}, Cntr)$$

is logged if A is conditional. A log entry is made *only* if S_{called} is incomplete.

Although ANSWER RESOLUTION operations are logged, PROGRAM CLAUSE RESOLUTION are not; attempts to log these operations usually slowed down computations so much that logging became unusable for all but small computations. In XSB, resolving answers from completed tables is nearly identical to resolving program clauses, so for efficiency reasons these answers are not logged either. NEGATIVE RETURN operations are logged in a similar manner.

– NEGATIVE REGURN When a negative literal L with underlying subgoal S_{called} is resolved via NEGATIVE RETURN in a tree for S_{caller}, a fact

$$nr(S_{called}, S_{caller}, Cntr)$$

is logged. A log entry is made *only* if S_{called} is incomplete.

The logging of new answers does not correspond to an SLG operation but is useful for analysis.

– *New Answer* When a new answer $N = (S\text{:-}D|)\theta$ is derived for subgoal S (i.e. N is not already an answer for S) a fact

$$na(\theta, S, Cntr)$$

is logged if N is unconditional ($D = \emptyset$) and

$$na(\theta, S, D, Cntr)$$

is logged if N is conditional.

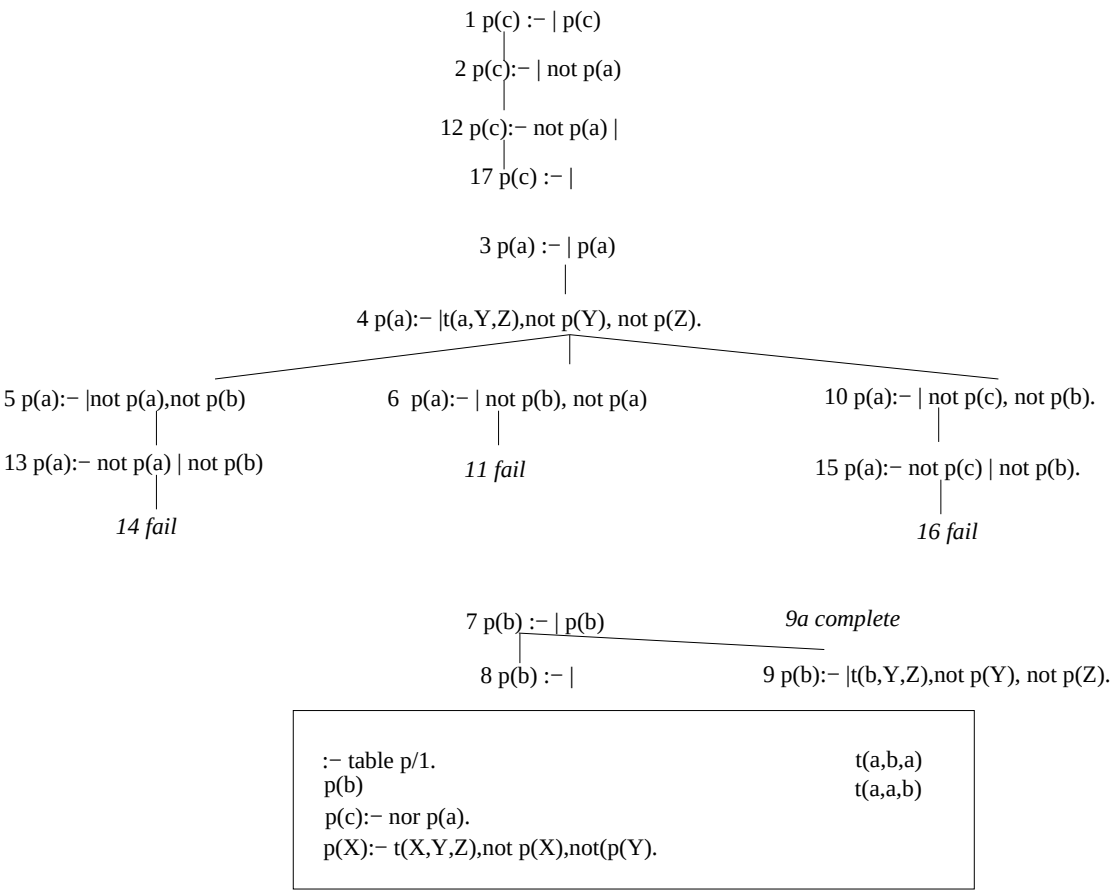

Fig. 2. A Normal Program P_{norm} and SLG Forest for Evaluation of the Query *p(c)*

Note that *na/3* can be seen as a specialization of *na/4* that reduces the memory footprint of the loaded log. A similar specialization is described below for simplification.

– COMPLETION When an SCC $\mathcal{S}$ is completed, a fact

$$cmp(S, SCC_{ind}, Cntr)$$

is logged for each $S \in \mathcal{S}$. Here SCC_{ind} is a index that groups subgoals into their mutually recursive components at the time they were completed. If S was early completed, a fact

$$cmp(S, ec, Cntr)$$

is logged at the time of early completion. When the original SCC for S is completed, another completion fact for S will be logged indicating its index as just described.

– DELAYING When the selected literal $not\ A$ is delayed in a node in a tree for S, a fact

$$dly(A, S, Cntr)$$

is logged.

– SIMPLIFICATION operations are logged as follows. Let $S_{caller}\theta$:-$D|$ be the answer to which SIMPLIFICATION is applied.

∗ If a literal $L \in D$ becomes failed, and $L = S_{called}\eta$ is positive, where S_{called} is a tabled subgoal, a fact

$$smpl\_fail(S_{caller}, \theta, S_{called}, \eta, Cntr)$$

is logged; if $L = not\ S_{called}$,

$$smpl\_fail(S_{caller}, \theta, S_{called}, Cntr)$$

is logged instead.

* If a literal $L \in D$ succeeds and if $L = S_{called}\eta$ is positive, where S_{called} is a tabled subgoal, a fact

$$smpl\_succ(S_{caller}, \theta, S_{called}, \eta, Cntr)$$

is logged; if $L = not\ S_{called}$,

$$smpl\_succ(S_{caller}, \theta, S_{called}, Cntr)$$

is logged instead.

– ANSWER COMPLETION If answer completion fails an answer $S\theta$ in a tree for S, a fact

$$ansc(\theta, S, Cntr)$$

is logged.

Example 2.3 *The forest for* reach(1,Y) *in the foregoing example has the log file as shown in Table 3. The actual log file facts are shown, along with the associated node they produced (if any) and an explanation* [6].

3. Properties of the Forest Log

Forest logs capture several important aspects of tabled computations. We begin by showing how they capture the subgoal dependency graph of a given forest (Definition 2.1), and then discuss the conditions under which a homomorphic image of an SLG forest can be constructed from a log.

3.1. Capturing Dependency Information

Definition 3.1 *Let $\mathcal{L}$ be a forest log with n facts, and let $0 \leq c \leq n$. Then the* log dependency graph induced by c *has an edge (S_1, S_2) for every fact $tc(S_2, S_1, state, c')$ or $nc(S_2, S_1, state, c')$ in $\mathcal{L}$ such that $c' \leq c$, $S_1 \neq null$ and*

$$\neg\exists \mathcal{S}_{scc}, c''.((cmp(S_1, \mathcal{S}_{scc}, c'') \vee cmp(S_2, \mathcal{S}_{scc}, c'')) \wedge c'' \leq c).$$

[6] As implemented in XSB, forest logging also records events that are not modeled by SLG or its extensions, including exceptions thrown during an evaluation, and table abolishes. However, the current version of forest logging does not log *ansc/3* facts, which are rarely needed.

Since the log dependency graph is parameterized by a log's counter, the log can be used to construct the SDG at various stages in the evaluation. This is formalized by Theorem 3.1 which states that the SDG for *any* forest of an evaluation can be reconstructed from the log dependency graph. This theorem directly underlies the analysis routines of Section 4; and because it holds for any forest, the theorem also underlies analysis of partial computations – e.g. computations that were interrupted because they were suspected to be non-terminating (cf. the discussion of the Terminyzer tool [11,10] in Section 6).

To be able to reconstruct the SDG of a given forest, there needs to be a guarantee of correspondence between when facts are logged and the state (i.e., forest) of an evaluation. A property termed *eager subgoal logging* is sufficient for this. Eager subgoal logging states that whenever a tabled literal L is selected in a tree S_{caller}, a $tc/4$ or $nc/4$ fact is logged, regardless of whether a NEW SUBGOAL operation is applicable. For instance, if the underlying atom of the positive literal L is S_{called}, then

$$tc(S_{called}, S_{caller}, \langle state \rangle, c_i + 1).$$

is logged, with the value of $state$ as *new*, *cmp* or *incmp*. There is thus a difference in the behavior of the logging mechanism from the formalism of SLG, as a NEW SUBGOAL operation is performed only if S_{called} is new to the evaluation. Eager subgoal logging is supported by XSB, and should be easy to guarantee for any tabling engine that implements forest logging[7].

Theorem 3.1 *Let $\mathcal{E} = \mathcal{F}_0, ..., \mathcal{F}_n$ be an SLG evaluation and $\mathcal{L}$ a log created using eager subgoal logging. Then for any $SDG(\mathcal{F}_i)$, $0 \leq i \leq n$, there is a* c *such that $SDG(\mathcal{F}_i)$ is isomorphic to the log dependency graph induced by* c. [8].

3.2. Constructing a Homomorphism of an SLG Forest

While dependency information among subgoals is critical to understanding an evaluation, other aspects are important as well. For example, applications in knowledge representation and business rule development may require analysis of dependencies or of answers that arise from application of a particular *rule* r for a predicate p/n, against a subgoal S. Such in-

[7] Within XSB this is done within the tabletry instruction (cf. [13]).
[8] Proofs are provided in the appendix of this paper.

Log File	Assoc. Node in Fig. 1	Explanation
tc(reach(1,_v0),null,new,0)	node 1	NEW SUBGOAL
tc(reach(2,_v0),reach(1,_v0),new,1)	node 4	NEW SUBGOAL
tc(reach(2,_v0),reach(2,_v0),incmp,2)	node 6	repeated subgoal registered
na([2],reach(2,_v0),3)	node 8	registered as answer
ar([2],reach(2,_v0),reach(2,_v0),4)	node 9	ANSWER RESOLUTION
cmp(reach(2,_v0),2,5)		reach(2,_v0) COMPLETION
na([2],reach(1,_v0),6)	node 10	registered as an answer
tc(reach(3,_v0),reach(1,_v0),new,7)	node 12	NEW SUBGOAL
tc(reach(1,_v0),reach(3,_v0),incmp,8)	node 14	repeated subgoal registered
ar([2],reach(1,_v0),reach(3,_v0),9)	node 15	ANSWER RESOLUTION
na([2],reach(3,_v0),10)	node 15	registered as an answer
na([1],reach(3,_v0),11)	node 17	registered as an answer
na([3],reach(1,_v0),12)	node 20	registered as an answer
ar([3],reach(1,_v0),reach(3,_v0),13)	node 21	ANSWER RESOLUTION
na([3],reach(3,_v0),14)	node 21	registered as an answer
ar([2],reach(3,_v0),reach(1,_v0),15)	node 22	ANSWER RESOLUTION
ar([1],reach(3,_v0),reach(1,_v0),16)	node 23	ANSWER RESOLUTION
na([1],reach(1,_v0),17)	node 23	registered as an answer
ar([3],reach(3,_v0),reach(1,_v0),18)	node 24	ANSWER RESOLUTION
ar([1],reach(1,_v0),reach(3,_v0),19)	node 25	ANSWER RESOLUTION
cmp(reach(1,_v0),1,20)		reach(1,_v0) COMPLETION
cmp(reach(3,_v0),1,21)		reach(3,_v0) COMPLETION

Fig. 3. A Log File Corresponding to the SLG Forest in Figure 1

formation can be easily obtained from the SLG tree $\mathcal{T}$ for S. The children of the root of $\mathcal{T}$ can be examined, the subtree corresponding to PROGRAM CLAUSE RESOLUTION by r determined, and dependency and answer information directly obtained. Further information about the behavior of r can be obtained by aggregating similar information from all subgoals of p/n in an evaluation.

A similar, but more precise problem is to identify a particular *position* of a literal in a given rule that has high computational cost. Such positions can be identified from an SLG forest via nodes with a large number of children, or nodes that have an underlying selected subgoal whose proof requires a large subforest not otherwise used in the evaluation (as determined by dependency information).

Both of these types of analysis problems require identifying the *parent-child* relations within an SLG tree. However, such relations are not always easy to construct from a forest log because PROGRAM CLAUSE RESOLUTION operations are not logged, due to the expense that their logging incurs. Of course, parent-child relations can be explicitly represented by rewriting a program. For instance, to obtain general information about the cost of rules, each rule *H :- Body* of interest, may be transformed by folding *Body* into a new tabled predicate, producing: *H :- tabledBody* and *tabledBody :- Body*. By logging an evaluation with such a transformed program, rule-based dependency information can be obtained, via Theorem 3.1. However, such rewriting leads to inefficiencies when there is a large overlap among the answers produced by different rules, so rewriting is most effective when potential rules or rule positions can be pre-identified.

In order to support rule-level or positional analysis without rewriting, sufficient conditions need to be determined under which the parent-child relations for a given tree can be constructed from the log. Since the log does not contain information about PROGRAM CLAUSE RESOLUTION or about ANSWER RESOLUTION from completed tables, we begin by characterizing a morphism that removes such information.

Definition 3.2 *Let $\mathcal{F}$ be an SLG forest. The graph morphism $\mathcal{H}(\mathcal{F})$ is defined as follows.*

- *For any node $n \in \mathcal{F}$, $\mathcal{H}(n)$ is defined as:*
  - *If the selected literal of n is tabled, then $\mathcal{H}(n) = n$;*
  - *otherwise, $\mathcal{H}(n)$ is closest parent-child ancestor of n whose selected literal is tabled.*

– *If there is an edge between nodes n_1 and n_2 in $\mathcal{F}$ there is an edge between $\mathcal{H}(n_1)$ and $\mathcal{H}(n_2)$.*

Note that since the root of any SLG tree has a selected tabled literal, any node whose selected literal is non-tabled has an ancestor that is tabled; because the ancestor relation is a tree, the closest such node is unique, so that $\mathcal{H}$ is well-defined. Given these considerations, it is evident that $\mathcal{H}$ defines a homomorphism of an SLG forest $\mathcal{F}$ where $\mathcal{F}$ is taken as a graph with labeled nodes.

In order to reconstruct an SLG tree in $\mathcal{F}$ from $\mathcal{H}(\mathcal{F})$, the parent of each logged fact f needs to be determined and the edges themselves constructed. When ANSWER RESOLUTION and other tabling operations are performed, their representation of the calling subgoal can be used for this purpose. However in the case of, e.g., program clauses, the program clauses must be sufficiently distinct so that the parent of each fact can be uniquely identified. These conditions are specified by Definition 3.3.

Definition 3.3 *Let $Body$ and $Body'$ be two sequences of literals. Then $Body$ and $Body'$ are* distinguishable *if*

– *$Body$ and $Body'$ are empty; or*
– *Both $Body$ and $Body'$ contain at least one tabled literal, $Body = L_1, ..., L_n$ and $Body' = L'_1, ..., L'_n$ and*
  1. *The leftmost literals L_1 and L'_1 are tabled and the sequences $L_2, ..., L_n$ and $L'_2, ..., L'_n$ are distinguishable.*
  2. *The leftmost tabled literal L_i of $Body$ does not unify with any literal in $Body'$, the leftmost tabled literal L'_j of $Body'$ does not unify with any literal in $Body$, and the sequences $L_{i+1}, ..., L_n$ and $L'_{j+1}, ..., L'_n$ are distinguishable.*

Note that if all predicates in a program are tabled, all rules will be distinguishable. When all rules for a predicate p/n are pairwise distinguishable, an SLG tree for a goal to p/n can be constructed by starting at the root node, and iteratively constructing the children of each node, using the information from the log and the rules themselves. This is formalized in the algorithm reconstruct_tree(), which can be found in the appendix of this paper.

Theorem 3.2 *Let P be a program, $\mathcal{E}$ a finitely terminating evaluation, $\mathcal{L}$ its log and $\mathcal{T}$ a completed tree with root $Subgoal$:-$|Subgoal$ in a forest of $\mathcal{E}$, and assume all rules in P whose head unifies with $Subgoal$ are distinguishable. Then* reconstruct_tree*(S) produces a graph, $EdgeSet$, that is isomorphic to $\mathcal{T}$.*

Assuming a fixed maximal size for terms in $\mathcal{T}$ and P, then the cost of reconstruct_tree*(S) is*

$$\mathcal{O}(size(\mathcal{T})log(size(\mathcal{T})) + size(P)).$$

As more predicates are tabled, the number of rules that are distinguishable increases. Thus, Proposition 3.2 implies that forest logging can often support rule level analysis for heavily tabled computations, such as those that occur in Flora-2.

4. Analyzing the Log; Seeing the Forest through the Trees

4.1. Using the Log to Analyze Dependencies

Continuing Example 1.1, we consider execution of a particular biology query that took more space and time than expected. This query took about 30 seconds of CPU time and created about 600,000 tables with about 300,000 answers total. Overall about 8.7 million tabled subgoals were called. The query required about 300 megabytes of table space, while XSB's combined trail and choice point stack region had allocated over 1 gigabyte of space[9]. The computation was rerun with forest logging. Forest logging has no impact on memory usage, although for this example the elapsed execution time increased form 30 to 52 seconds. The log file had a size of 3.6 gigabytes and contained 14.1 million facts.

After loading the log, the top-level analysis query, *forest_log_overview/0*, gave the results in Figure 4. The forest log overview first shows the total number of completed and non-completed subgoals and SCCs, along with a count of how many of the completed subgoals were early-completed (Section 2.1). Information about non-completed subgoals is useful for analyzing computations that do not terminate. The overview also distinguishes between positive and negative calls to tabled subgoals, and for each such class further distinguishes subgoals that were new, completed, or incom-

[9] All times reported in this paper were from a 64-bit machine with 3 Intel dual-core 3.47 GHz CPUs and 188 megabytes of RAM running under Fedora Linux. The default 64-bit, single-threaded SVN repository version of XSB was used for all tests.

```
There were 613448 subgoals in 463446 (completed) SCCs.
   93909 subgoals were early-completed.
   0 subgoals were not completed in the log.
   There were a total of 8638299 positive tabled subgoal calls:
      582754 were calls to new subgoals
      4460609 were calls to incomplete subgoals
      3594936 were calls to complete subgoals
   There were a total of 30694 negative tabled subgoal calls:
      30694 were calls to new subgoals
      0 were calls to incomplete subgoals
      0 were calls to complete subgoals
There were a total of 5 negative delays
There were a total of 6 simplifications
There were a total of 304447 unconditional answers derived:
There were a total of 6 conditional answers derived:

Number of SCCs with 1 subgoals is 463437
Number of SCCs with 4 subgoals is 1
Number of SCCs with 7 subgoals is 1
Number of SCCs with 52 subgoals is 1
Number of SCCs with 110 subgoals is 5
Number of SCCs with 149398 subgoals is 1
```

Fig. 4. Output of Forest Log Overview for the Program and Query in Example 1.1

plete. Recall that calls to completed tabled subgoals essentially treat the answers in the table as facts, so that such calls are efficient. Making a call to an incomplete subgoals on the other hand means that the calling and called subgoals are mutually recursive; [10] and execution of recursive sets of subgoals can be expensive, especially in terms of space. Aggregate counts of DELAYING and SIMPLIFICATION are also given along with counts of both conditional and unconditional answers. Negation does not appear to play a major role in this computation, and it appears likely that the program has a 2-valued well-founded model, although further exploration would be needed to determine this (cf. Section 4.3).

The overview also provides the distributions of tabled subgoals across SCCs. While most of the SCCs were small, one was very large with nearly 150,000 mutually dependent subgoals. Clearly the large SCC should be examined. The first step is to obtain the *index* of its SCC, which is simply a way to denote it. The query *get_scc_size(Index,Size), Size* $>$ *1000.* indicated that the index of the large SCC was 39. The query *analyze_an_scc(39)* then provided the information in Figure 5 [11]. It is evident from the count of edges in the first line of this report that the vast majority of the calls to incomplete tables during this computation occurred in the SCC under investigation. Since information on incomplete tables is kept in XSB's choice point stack (cf. [13]), the evaluation of SCC 39 is the likely culprit behind the large amount of stack space required. The subgoals in the SCC are first broken out by their predicate name and arity, then the edges within the SCC are broken out by the predicates of their caller and called subgoals. With this information, a programmer can review the various rules for *lookupSentence/3*, *forwardSentence/3* and other predicates to determine whether the recursion is intended and if so, whether it can be simplified. In the actual example, examination of these rules showed that the use of Hilog resulted in calling a number of unexpected predicates. Additional guards were placed on the Hilog call, greatly reducing the time and space needed for the computation.

4.2. Using abstraction in the analysis

Within the SCC analysis, information about a given tabled subgoal S is abstracted: only the functor and arity of S is presented. For SCC 39 in the running example, abstraction is necessary, as directly reporting 150,000 subgoals or 4,000,000+ edges would not provide a human with useful information. However, it could be the case that seeing the tabled subgoals themselves would be useful for a smaller SCC. Even for a large SCC, different levels of abstraction to provide

[10] This statement is true in the local scheduling strategy but not in batched scheduling.

[11] For purposes of space the lists of predicates and edges in the SCC have been abbreviated.

```
There are 149671 subgoals and 4461290 edges (average of 30.8073
      edges/subgoal) within the SCC

There are 2 subgoals in the SCC for backchainForbidden / 0
There are 2 subgoals in the SCC for
                  www.cyc.com/transformationPredicate / 0
:
There are 18770 subgoals in the SCC for forwardSentence / 3
There are 18771 subgoals in the SCC for lookupSentence / 3

Calls from assertedSentence/3 to lookupSentence/3:32
Calls from backchainForbidden/0 to www.cyc.com/transformationPredicate/0:2
:
Calls from transformationSentence/2 to sbhlSentence/3:5479
Calls from tvaSentence/3 to removalSentence/3:7695
```

Fig. 5. Output of SCC Analysis for the Program and Query in Example 1.1

```
There are 149671 subgoals and 4461290 edges (average of 30.8073
           edges per subgoal) within the SCC

There are 3 subgoals in the SCC for backchainRequired(g,g)
There are 2 subgoals in the SCC for backchainForbidden(g,g)
:
There are 29254 subgoals in the SCC for gpLookupSentence(g,g)
There are 29254 subgoals in the SCC for removalSentence(g,g)

Calls from assertedSentence(g,g) to lookupSentence(g,g):10
Calls from assertedSentence(m,g) to lookupSentence(m,g):22
:
Calls from transformationSentence(m,g) to sbhlSentence(m,g):741
Calls from tvaSentence(g,g) to removalSentence(g,g):7695
```

Fig. 6. Output of SCC Analysis for the Program and Query in Example 1.1

mode or type information can be useful. For this reason, forest log analysis predicates support calls such as *analyze_an_scc(39,abstract_modes(_,_))* which applies the predicate *abstract_modes/2* in the breakdowns of subgoals and edges. *abstract_modes(In,Out)* simply goes through each argument of the term *In* and unifies the corresponding argument of the term *Out* with a *v* if the argument is a variable, a *g* if the argument is ground, and *m* (for mixed) otherwise. The resulting output is shown in Figure 6. Examination of this output indicates that the SCC conists of a large number of fully ground calls to several predicates: rewriting code to make fewer but less instantiated calls to these predicates will often optimize a computation.

Of course, *abstract_modes/2* is simply an example: term abstraction predicates are easy to write, and any such predicate may be passed into the last argument of *analyze_an_scc/3* [12].

4.3. Analyzing Negation

Many programs that use negation are stratified in such a way that they do not require the use of DELAYING and SIMPLIFICATION operations. However if a program does not have a two-valued well-founded model, a user would often like to understand why, in addition to having the sort of dependency analysis described in the previous section. Even in a program that is two-valued, the heavy use of DELAYING and SIMPLIFICATION can indicate that some rules may need to be optimized by having their literals reordered.

[12] Because Flora-2 terms are represented in a particular way to support Hilog, abstraction was used to produce the output of Section 4.1, while a special version of `abstract_modes/2` was used here.

As indicated previously, the forest log overview includes a total count of DELAYING and SIMPLIFICATION operations, as well as a count of conditional answers. In addition, SCC analysis counts negative as well as positive edges within the SCC. Forest logging also provides an analysis routine to examine why answers have an undefined truth value. Recall from Example 2.2 that there are two types of causes of an undefined truth value: either 1) a negative literal explicitly undergoes a DELAYING operation; or 2) a conditional answer may be used to resolve a literal. It can be shown that in local scheduling, a conditional answer A will never be returned out of an SCC if A is successful or failed in the well-founded model of a program. This means that if an answer for S is undefined, then it would be caused operationally by a DELAYING operation within the SCC of S or within some other SCC on which S depends. So to understand why an atom is undefined it can be useful understand the "root causes" of the delay: to examine SCCs in which DELAYING operations were executed and conditional answers were derived, but where the answers could not be simplified.

Example 4.1 *As a use case, logging was made of execution of a Flora-2 program that tested out a new defeasibility theory. The forest log overview indicated that the top-level query was undefined:*

```
:
There were a total of 55 negative delays
There were a total of 0 simplifications
There were a total of 695 unconditional
                        answers derived
There were a total of 66 conditional
                        answers derived
```

The analysis predicate three_valued_scc(List) *produces a list of all SCC indices in which* DELAYING *caused the derivation of conditional answers. These SCCs were then analyzed as discussed in the previous sections.*

5. Implementation and Performance of Logging and Analysis Routines

A user of XSB may invoke forest logging so that the log is created as described in Section 2. Alternately, a user may invoke *partial logging*, which omits facts produced by the ANSWER RETURN and NEW ANSWER operations. Partial logging can save time and space and supports analysis of mutually recursive components as in Sections 4.1 and 4.2. However it does not support the negation analysis of Section 4.3.

Regardless of the level that is enabled, logging is performed by conditional code in large virtual machine instructions such as *tabletry* (NEW SUBGOAL), *answer_return*, *new_answer* and *check_completion* (COMPLETION) (cf. [13]). Subgoals and bindings are then written using registers, tables, answer templates, and lists of delayed literals. Calling subgoals (e.g., the second arguments of *tc/4* and *nc/4*) are obtained by the SLG-WAM's *root subgoal register*, which was originally introduced for tabled negation [13]. For efficiency, logging minimizes interaction with the operating system: information is written into a internal buffers; once the buffers contain all information for a log fact, they are written to the output stream using a single `printf()` statement. The subgoals and answers that are logged may be quite large, particularly when non-termination may be an issue: thus all buffers used are fully expandable.

All facts are written canonically[13] so that loading a log exploits XSB's efficient reading and asserting of canonical dynamic code. The *cmp/3* (COMPLETION) facts are trie-indexed (cf. [15]), while most other facts index on multiple arguments. For instance, *ar/4* (ANSWER RESOLUTION) facts are indexed on their second and third arguments (calling and called subgoals), so that indexing is used if either argument is bound. A type of indexing in XSB called star-indexing is used, which can index on up to the first four positions of a given argument [15].

Analysis routines are written in standard Prolog with one exception. Counting the number of (abstracted) edges in an SCC makes use of the code fragment

tc(T1,T2,incmp,_Ctr),
check_variant(cmp(T1,S,_),1),
check_variant(cmp(T2,S,_),1)

The predicate *check_variant(Goal,DontCareNum)* is implemented only for trie-asserted code (e.g., *cmp/3*). If *Goal* is an atom for predicate *p/n*, *check_variant/2* determines whether a variant of the first *N - DontCareNum* arguments of *Goal* is in the trie for *p/n*. *check_variant/2* is implemented at a low level, making direct use of the data structures used by XSB to represent tries. *check_variant/2* begins matching the leftmost element of a term t with the root of the trie, and proceeds to match each subsequent symbol with

[13] In Prolog, canonical syntax does not allow operator declarations so that all function symbols are prefixed and their arguments fully parenthesized; and restricts numbers to base 10.

a child node of the current trie position; if no match is found *check_variant/2* fails. As a result, only a single path from the root need be examined in order to determine whether a variant of t is in the trie. On the other hand, for large SCCs in which there are numerous subgoals that may unify with one another (but aren't variants), a Prolog search for variance may subject to a great deal of backtracking, and the time required may be proportional to the size of the trie, rather than to the size of t as with *check variant/2*. Not surprisingly, the use of *check variant/2* is critical to a good analysis time. For example, in the analysis of SCC 39 for the Cyc example presented above, the use of *check_variant/2* reduced the time for the forest log overview over 100-fold.

5.1. Performance

Figure 1 shows performance results for logging and analysis of various sets of examples:

- *Cyc Series. Cyc 1* is the working biology example used throughout this paper; *Cyc 3* is a similar, but larger, biology example, Both systems are based on the translation of the Cyc inference engine into Flora-2 and then into XSB.
- *Pref-kb Series. Pref-kb* contains a small set of tabled Prolog rules about personal preferences that demonstrate reasoning about existential information in a manner similar to description logics, and make use of default and explicit negation. Queries to these rules were run over sets of 3.7 million and 14.8 million base facts[14].
- *reach N Series.* This series tests logging of an open query to the right-recursive *reach/2* predicate in Figure 1 over fully connected graphs with 2000-12000 nodes. Since these queries measure reachability from all nodes in the graphs the cost of an open query scales quadratically with respect to the number of nodes in the graph. Although the tabling behavior of a simple transitive closure query such as *reach/2* is well understood, this series is included to test the scalability of logging and of its analysis.

5.1.1. Load Time

In part because of XSB's library predicates for loading canonical dynamic facts, XSB's load time is efficient for the various types of logs, loading approximately 100,000 facts per second for the *Cyc* series, over 150,000 facts per second for the *Pref-kb* series, and nearly 200,000 facts per second for the *reach N* series. After being loaded, the *Cyc* examples took about 500 bytes per fact, the *Pref-kb* examples about 300 bytes per fact, and the *reach N* facts about 200 bytes per fact. Much of this space is due to the heavy indexing of log facts. The reason that the *Cyc* logs take the longest to load and the most space to represent is because the subgoals and answers generated by Flora-2 compilation are larger for the *Cyc* series. For instance, the Hilog transformation used by Flora-2 transforms *n*-ary predicates and function symbols to *n+1* ary predicates and function symbols. As a slightly simplified instance, a term such as *p(a,f(b),1)* is converted to *flora_apply(p,a,flora_apply(f,b),1)*. In addition, Flora-2 represents module information as an argument of each atom, requiring further space.

5.1.2. Analysis Time

Once the log has been loaded, the indexing makes analysis fast enough to be interactive: for the *Cyc* biology example the top level analysis took around 10 seconds, and analyzing SCC 39 took about 20 seconds when the built-in predicate-arity abstraction was used, and about 60 seconds for the parameterizable version that used *abstract_modes/2*. Although computing the forest log overview requires several table scans in addition to indexed retrievals, timings for the both the *Pref-kb* and the *reach N* series show a sublinear growth of analysis time with respect to log size.

5.1.3. Logging Overhead

The overhead of query evaluation was also measured, i.e., the time it took to execute a query when forest logging was turned on compared to no logging. For the *Cyc* series, the overhead of logging increased the time for *Cyc 1* by 72% and for *Cyc 3* by 132% which was considered acceptable by KEs. Similarly, the *Pref-kb* series, which uses a heavily tabled Prolog program, has an average logging overhead of about 225%. On the other hand, for the *reach N* series the overhead of forest logging on query execution was naturally high (about 2 orders of magnitude), as *reach N* performed very little program clause resolution. This overhead may be considered as a worst-case for forest logging[15].

[14] Details of this series, including the code used to generate the datasets, are available at sites.unife.it/ai/termination.

[15] The *reach N* series was included to benchmark scalability, but partial logging as described in the next section can greatly reduce the logging overhead and log space of the *reach N* series, if needed.

Program	Number of facts	Load time (secs)	Load Space (bytes)	Forest Log Overview (secs)
Cyc 1	14,009,602	140.1	7,857,572,736	22.1
Cyc 3	66,256,186	612.2	36,950,074,144	92.2
Pref-kb 3,7	2,500,193	16.5	725,972,288	2.3
Pref-kb 14,8	8,000,140	52.5	2,336,039,512	7.3
reach 2000	12,006,002	78.4	2,496,927,880	8.4
reach 4000	48,012,002	280.1	9,985,835,352	13.2
reach 8000	192,024,003	1227.7	39,940,961,128	59.7
reach 12000	432,036,000	2332.9	89,864,542,056	132.8

Table 1
Timings for Loading and Analyzing Logs

5.1.4. Partial Logging

For some large examples, partial logging (mentioned at the beginning of this section) can reduce the the logging overhead, the time required to load a log, and the space the loaded log requires. An example of this is as follows.

Example 5.1 *In analysing the log for a query to* Pref-kb*, it became apparent that much of the resources the query required were due to large SCCs composed almost entirely of goals to* equals/2*, the predicate used for equality of non-identical terms. By examining the program, a rule for* equals/2 *was translated from a right-recursive form to a left-recursive form. Simplifying somewhat, this meant translating a rule of the form:*

equals(X,Z):- basePredicate(X,Y),equals(Y,Z)

to

equals(X,Z):- equals(Y,Z),basePredicate(X,Y)

The left-recursive form is usually faster for tabled Prolog, as Prolog's left-to-right literal selection strategy means that the right-recursive form will generate separate tabled queries for different instantiations of Y *while the left-recursive form will not.*

After performing the above translation, the query time for the transformed series, Pref-kb-lr *was reduced by 300-400%, and the maximum memory required for query evaluation was reduced by about 700-800%. However, while the translation optimized the query itself, when logging was turned on the left-recursive query slowed down substantially, even compared to the time required by the right-recursive form when using logging.*

Inspection of the log for the query to left-recursive *Pref-kb* showed that a large number of answers were produced for the top-level query and its tabled subqueries. Since partial logging removes most information about answer derivations it can substantially reduce the logging time and log size for queries with a large number of answers. Table 2 shows that partial logging reduces the size of the log for left-recursive *Pref-kb* by many orders of magnitude. On the other hand, evaluation of the query to right-recursive *Pref-kb* produces a large number of subgoals and relatively few answers, so that partial logging is not more efficient than full logging in this case[16].

6. Related Work

Trace-based analysis has been widely used to analyze the behavior of concurrent systems, security vulnerabilities, suitability for optimization strategies and other program properties. Within logic programming, it has been used to analyze how constraint evaluation affects program flow [5]; although perhaps the best known use of trace-based analysis is the Ciao preprocessor, which infers call and success conditions for a variety of domains based on execution of queries (see [8] for further details).

Based on XSB's forest logging, a system for analyzing non-termination of Flora-2, Silk and Fidji programs, called *Terminyzer* has been developed [11,10]. In addition to the logging mechanisms described so far, Terminyzer relies on special routines that translate compiled Flora-2 code back from a Prolog syntax to a more readable Flora-2 syntax. Displays for Terminyzer are shown in the IDEs of both Silk and Fidji and have been used for debugging by knowledge en-

[16] Although the left-recursive and the right-recursive forms of *Pref-kb* are semantically equivalent, the left-recursive form makes fewer queries than the right-recursive form but its queries not as instantiated. The left-recursive form thus has a larger search space than the right-recursive form, but it creates far fewer queries for its search and for that reason is more efficient under XSB's tabling implementation.

equals/2 form	EDB Size	Log Level	Log Overhead	Nbr of facts	Load time	Load Space	Forest Log Overview
Right-recursive	3.7 million	full	236%	2,500,254	16.5	725,972,288	2.3
Right-recursive	3.7 million	partial	236%	2,500,126	16.5	724,037,016	2.3
Left-recursive	3.7 million	full	2685%	11,983,203	89.3	3,904,201,328	1.1
Left-recursive	3.7 million	partial	$< 1\%$	115	< 0.1	80,202	< 0.1

Table 2
Comparing Full and Partial Logs for *Pref-kb*: Times are in Seconds and Space is in Bytes

gineers [1]. The analysis presented in Section 4 predates the termination analysis of [11,10], and is complementary to it. For instance, the analyses in Section 4.1 considered a program and query that terminated, but was inefficient due to unexpected dependencies among subgoals; while the negation analysis of Section 4.3 helped indicate why a 2-valued model was not obtained[17].

7. Discussion

The design of a forest log attempts to balance the amount of information logged against the time it takes to load and analyze a log. The propositions of Section 3 show that a forest log suffices to analyze dependency information and under certain conditions has the information available to construct a homomorphic image of an SLG forest. The analysis predicates of Section 4 show how the representation is used to provide meaningful information to users for tabled programs with and without negation. The benchmarks of Section 5 further demonstrate practicality of this approach and its scalability to logs with hundreds of millions of facts. As a result forest logging is now fully integrated into XSB and Flora-2, and underlies tools in the commercial Silk and Fidji IDEs.

More generally, trace-based analysis provides an alternative to static analysis for a number of program or query properties. Unlike static analysis, trace-based analysis requires realistic data along with a representative set of queries. On the other hand, for programs that include Hilog, defeasibility, equational reasoning and other features of Flora-2, Fidji and Silk, static analysis techniques may not exist, may not be implemented, or may not be powerful enough for practical use. As a result, trace-based analysis is a viable technique to determine properties of large tabled computations. Current work involves using forest logging to help suggest changes to tabling declarations and properties in order to optimize programs.

Acknowledgements This work was partially supported by Project Halo. The author thanks Fabrizio Riguzzi for making available the server on which the timings were run.

[17] Publication of the material in this paper was delayed while Vulcan Inc. which partially funded this work, considered whether to exercise its patent rights to forest logging and its analysis.

References

[1] C. Andersen, B. Benyo, M. Calejo, M. Dean, P. Fodor, B. Grosof, M. Kifer, S. Liang, and T. Swift. Understanding Rulelog computations in Silk. In *Workshop in Logic-based Methods in Programming Environments*, 2013.

[2] F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ envelope. In *International Joint Conference on Artificial Intelligence*, pages 364–369, 2005.

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

[4] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[5] M. Ducasse and L. Langevine. Automated analysis of clp(fd) program execution traces. In *International Conference on Logic Programming*, pages 470–471, 2002.

[6] B. Grosof and T. Swift. Radial restraint: A semantically clean approach to bounded rationality for logic programs. In *American Association for Artificial Intelligence Conference*, 2013.

[7] H. Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *International Conference on Logic Programming*, pages 150–165, 2001.

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

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

[10] S. Liang and M. Kifer. A practical analysis of non-termination in large logic programs. *Theory and Practice of Logic Programming*, 13(4-5):705–719, 2013.

[11] S. Liang and M. Kifer. Terminyzer: An automatic non-termination analyzer for large logic programs. In *Practical Applications of Declarative Languages*, 2013.

[12] E. Pontelli, T.C. Son, and O. Elkatib. Justificatinos for logic programs under the answer set semantics. *Theory and Practice of Logic Programming*, 9:1–56, 2009.

[13] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans-*

actions on Programming Languages and Systems, 20(3):586 – 635, May 1998.

[14] T. Swift. A new formulation of tabled resolution with delay. In *Progress in Art. Intel.*, pages 163–177, 1999.

[15] T. Swift and D.S. Warren. XSB: Extending the power of Prolog using tabling. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[16] H. Wan, B. Grosof, M. Kifer, P. Fodor, and S. Liang. Logic programming with defaults and argumentation theories. In *International Conference on Logic Programming*, pages 432–448, 2009.

[17] G. Yang, M. Kifer, and C. Zhao. FLORA-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. In *ODBASE-2003*, pages 671–688, 2003.

Appendix

A. Proofs of Theorems in Section 3

Theorem 1 *Let $\mathcal{E} = \mathcal{F}_0, ..., \mathcal{F}_n$ be an SLG evaluation and $\mathcal{L}$ a log created using eager subgoal logging. Then for any $SDG(\mathcal{F}_i)$, $0 \leq i \leq n$, there is a* c *such that $SDG(\mathcal{F}_i)$ is isomorphic to the log dependency graph induced by* c.

Proof: The proof is by induction on i such that $\mathcal{F}_i$ is a forest in $\mathcal{E}$.

For the base case, $SDG(\mathcal{F}_0)$ is empty, which corresponds to the log dependency graph induced by 0. To see this, note that the first *tc/4* or *nc/4* fact sets the calling subgoal to *null* and so by Definition 3.1 is not included in the log dependency graph induced by 0.

For the inductive case, assume the statement holds for $\mathcal{F}_i$ with log counter c_i and we consider the cases where $\mathcal{F}_{i+1}$ was produced by $\mathcal{F}_i$.

– NEW SUBGOAL. Suppose a tree with root

$$S_{called}\text{:-}|S_{called}$$

was created due to S being selected in a node $S_{caller}\theta$:-$Delays|Body$. In this case, by the eager subgoal logging property, a *tc/4* or *nc/4* fact with state *new* and counter $c_i + 1$ will be logged. E.g., if the dependency is positive, the log fact would be:

$$tc(S_{called}, S_{caller}, new, c_i + 1).$$

By Definition 3.1, setting c to $c_i + 1$ preserves the induction statement for $\mathcal{F}_{i+1}$, since neither subgoal will be completed.

– PROGRAM CLAUSE RESOLUTION. Note that this operation will affect the SDG only if the operation produces a child node with selected literal L whose underlying atom A is tabled, but has not been completed. In such a case, by the eager subgoal logging property, a *tc/4* or *nc/4* fact will be logged as the $c_i + 1$st fact. Setting c to $c_i + 1$ preserves the property for $\mathcal{F}_{i+1}$.

– ANSWER RESOLUTION. As with PROGRAM CLAUSE RESOLUTION, this operation will affect the SDG only if the operation produces a child node with selected literal L whose underlying atom A tabled, but has not been completed. By the eager subgoal logging property, after an *ar/3* or *dar/4* fact is logged as the $c + 1$st fact, a *tc/4* or *nc/4*

fact will be logged as the c_i + 2nd log fact. As in the previous case, the second argument of this facts will be S_{called}. By Definition 3.1, setting c to $c_i + 2$ preserves the property for $\mathcal{F}_{i+1}$.

- DELAYING, and NEGATION SUCCESS are argued in the same manner as ANSWER RESOLUTION[18].
- SIMPLIFICATION and ANSWER COMPLETION both affect only conditional answers. Since answers do not have a selected goal literal, they do not contribute to the SDG, so that the induction step holds trivially in these cases.
- NEGATION FAILURE adds a failure node, which does not affect the SDG, so that the induction step holds trivially in this case.
- COMPLETION. Completion of a subgoal S alters the SDG by removing all edges incident on S. In this case, the log contains a *cmp/3* fact for every early completion and every SCC completion. As a results, S will not be contained in the log dependency graph, and the induction statement holds if c is set to the counter of the last *cmp/3* fact for the SCC.

■

A.1. Proof of Theorem 2

Theorem 2, which states conditions for the existence of a homomorphism between a forest log and an SLG tree, is proved by showing the correctness of the algorithms reconstruct_tree (Figure 7) and create_children() (Figure 8). Both the proof and the algorithm create_children() use the definition of an SLG resolvent (originally from [4]), which differs from resolution in Horn rules in order to take into account delay literals in conditional answers.

Definition A.1 *Let N be a node A:-$D|L_1, ..., L_n$, where $n > 0$. Let $Ans = A'$:-$D'|$ be an answer whose variables are disjoint from N. N is* SLG resolvable *with Ans if $\exists i$, $1 \leq i \leq n$, such that L_i and A' are unifiable with a most general unifier (mgu) θ. The SLG resolvent of N and Ans on L_i has the form:*

$$(A\text{:-}D|L_1, ..., L_{i-1}, L_{i+1}, ..., L_n)\theta$$

if D' is empty; otherwise the resolvent has the form:

$$(A\text{:-}D, L_i|L_1, ..., L_{i-1}, L_{i+1}, ..., L_n)\theta$$

Note that SLG resolution delays L_i rather than propagating the answer's *Delays* D'. This is necessary, as shown in [4], to ensure polynomial data complexity.[19]

Theorem 2 *Let P be a program, $\mathcal{E}$ a finitely terminating evaluation, $\mathcal{L}$ its log and $\mathcal{T}$ a completed tree with root $Subgoal$:-$|Subgoal$ in a forest of $\mathcal{E}$, and assume all rules in P whose head unifies with $Subgoal$ are distinguishable. Then* reconstruct_tree*(S) produces a graph, $EdgeSet$, that is isomorphic to $\mathcal{H}(\mathcal{T})$.*

Assuming a fixed maximal size for terms in $\mathcal{T}$ and P, then the cost of reconstruct_tree*(S) is*

$$\mathcal{O}(size(\mathcal{T})log(size(\mathcal{T})) + size(P)).$$

Proof: We first show that $EdgeSet$ is isomorphic to $\mathcal{T}$, and then consider its cost.

reconstruct_tree($Subgoal$) (Figure 7) reconstructs the tree for $Subgoal$ in an iterative manner, starting with the root $Subgoal$:-$|Subgoal$, adding nodes to be expanded into $NodeSet$, and representing the resulting graph edges in $EdgeSet$. For the purposes of this proof, a *nearest tabled descendent* of a non-root node N is node N_{child} such that N_{child} is a descendent of N such and any other descendents of N that are ancestors of N_{child} (i.e., nodes between N and N_{child}) were formed by PROGRAM CLAUSE RESOLUTION. Note that these intermediate nodes have selected literals that are not tabled.

The proof of isomorphism is by induction on the number of iterations in the while loop in reconstruct_tree().

In the base case, reconstruct_tree() expands a root node. The children of a root node are created by resolution of program clauses whose heads unify with $Subgoal$: the rules themselves need not be distinguishable, as the children can be constructed immediately from P and $Subgoal$. Furthermore, it is immediate from Figure 7 that as all possible PROGRAM CLAUSE RESOLUTION operations are performed, all edges are added to $EdgeSet$ and all children are added to $NodeSet$.

In the inductive case, assume that $EdgeSet$ as created in the *n-1* iterations of the while loop in recon-

[18] The NEGATION SUCCESS operation is shorthand for a NEGATION RETURN operation where the selected literal succeeds and is resolved away.

[19] If the *Delays* sequence were propagated directly, then the *Delays* could effectively contain all derivations which could be exponentially many in the worst case.

reconstruct_tree($Subgoal$)
/* Assumes a program P and forest $\mathcal{F}$ */
$NodeSet := \{Subgoal\text{:-}|Subgoal\}$; $EdgeSet := \emptyset$; /* $Subgoal$:-|$Subgoal$ is a root node: */
For every clause H:-$Body$ whose head resolves with $Subgoal$ with mgu η
$NodeSet = NodeSet \cup (Subgoal\text{:-}|Body)\eta$
$EdgeSet = EdgeSet \cup (Node, (Subgoal\text{:-}|Body)\eta)$
While ($NodeSet \neq \emptyset$)
choose $Node$ from $NodeSet$; $NodeSet := NodeSet - Node$;
create_children($Node$,$NodeSet$,$Subgoal$);

Fig. 7. Top-level Algorithm to Perform $\mathcal{H}(\mathcal{F})$

struct_tree() is isomorphic to $\mathcal{H}(\mathcal{T})$ and that $Node = Subgoal\theta$:-$Delay|Body$ is chosen from $NodeSet$ in the n^{th} iteration.

We consider the cases for $Node$, and show how they are captured by create_children() (Figure 8).

– $Body$ is not empty. In this case, note that since $Node$ is not an answer, we do not have to consider either the effects of SIMPLIFICATION or ANSWER COMPLETION operations in producing the children of $Node$.

∗ The leftmost tabled literal in $Body$, L, exists and is positive (lines 4-13 of create_children() in Figure 8). Note that if there does not exist a leftmost tabled literal in $Body$, by Definition 3.3 of distinguishable rules, $Body$ must be empty, which falls under the case where $Node$ is an answer (lines 33-45).

∗ Consider first the case where L is the leftmost literal in $Body$ (lines 6-9 of Figure 8) – i.e., the leftmost literal is in fact tabled. In this case the fact that the rules for $Subgoal$ are distinguishable means that the ANSWER RESOLUTION operations that create children for $Node$ are identifiable by calling all facts of the form

$$ar(\eta, Subgoal_{called}, Subgoal_v, Ctr)$$

or

$$dar(\eta, Subgoal_{called}, Subgoal_v, Ctr)$$

such that $Subgoal_v$ is a variant of *Subgoal* and $Subgoal_{called}$ is a variant of L. In these cases create_children() properly creates children of the form

$$(Subgoal\theta\text{:-}Delay|Body')\eta$$

or

$$(Subgoal\theta\text{:-}Delay \cup L|Body')\eta$$

respectively.

∗ Next, consider the case where L is not the leftmost literal in $Body$ (lines 10-13 of Figure 8), so that create_children() creates the nearest tabled descendent of $Node$. In this case the fact that the rules for $Subgoal$ are distinguishable means that the nearest tabled descendent can be identified by calling all facts of the form

$$ar(\eta, Subgoal_{called}, Subgoal_v, Ctr)$$

or

$$dar(\eta, Subgoal_{called}, Subgoal_v, Ctr)$$

such that $Subgoal_v$ is a variant of *Subgoal* and $Subgoal_{called}$ unifies with L with mgu ξ. In these cases create_children() properly creates children of the form

$$(Subgoal\theta\text{:-}Delay|Body')\xi\eta$$

or

$$(Subgoal\theta\text{:-}Delay \cup L|Body')\xi\eta$$

∗ The leftmost tabled literal in $Body$, $L = not\ A$ exists and is negative (lines 14-32 of Figure 8).

create_children()($Node$,$NodeSet$,S);
If $Node = H$:-$Delay|Body$ is a non-root node where $Body$ is non-empty and $H = S\theta$
  If there is a leftmost tabled literal, L, in $Body$
    Let $Body = Body_{Left}, L, Body_{Right}$
    If L is positive
      If $Body_{Left}$ is empty /* L is the leftmost literal in $Body$, tabled or not */
        For each fact $ar(\eta, L, S, C)$ or $dar(\eta, L, S, C)$
          Let $Child$ be the SLG Resolvent of $Node$ and $L\eta$ on L
          $NodeSet := NodeSet \cup Child$; $EdgeSet := EdgeSet \cup (Node, Child)$;
      If L is not the leftmost literal in $Body$
        For each fact $ar(\eta, L', S, C)$ or $dar(\eta, L', S, C)$ such that L' unifies with L with mgu ξ
          Let $Child = Res\xi$, where Res is the SLG Resolvent of $Node$ and $L\eta$ on L
          $NodeSet := NodeSet \cup Child$; $EdgeSet := EdgeSet \cup (Node, Child)$;
    If L is negative, let $L = notA$
      If L is the leftmost literal in $Body$ /* tabled or not */
        For each fact $nr(L, S, C)$
          Let $Child = (H$:-$Delay|Body')$
          $NodeSet := NodeSet \cup Child$; $EdgeSet := EdgeSet \cup (Node, Child)$;
        For each fact $dly(L, S, C)$
          Let $Child = (H$:-$Delay \cup L|Body')$
          $NodeSet := NodeSet \cup Child$; $EdgeSet := EdgeSet \cup (Node, Child)$;
        If there are no facts of the form *nr(L,S,C)* or *dly(L,S,C)*
          $EdgeSet := EdgeSet \cup (Node, fail)$;
      If L is not the leftmost literal in $Body$
        For each fact $nr(L', S, C)$ such that L' unifies with L with mgu ξ
          Let $Child = (H$:-$Delay|Body')\xi$
          $NodeSet := NodeSet \cup Child$; $EdgeSet := EdgeSet \cup (Node, Child)$;
        For each fact $dly(L', S, C)$ such that L' unifies with L with mgu ξ
          Let $Child = (H$:-$Delay \cup L|Body')\xi$
          $NodeSet := NodeSet \cup Child$; $EdgeSet := EdgeSet \cup (Node, Child)$;
        If there are no facts of the form *nr(L',S,C)* or *dly(L',S,C)* such that L' unifies with A
          $EdgeSet := EdgeSet \cup (Node, fail)$;
  If $N = S\theta$:-$Delays|$ /* N is an answer */
    Let $\mathcal{S}$ be the set of facts $smpl\_fail(S_{called}, \eta, S, \theta, Cntr)$ or $smpl\_succ(S_{called}, \eta, S, \theta, Cntr)$
          such that $S_{called}\eta \in Delays$
    $\mathcal{S} := \mathcal{S} \cup smpl\_fail(S_{called}, S, \theta, Cntr)$ or $smpl\_succ(S_{called}, S, \theta, Cntr)$
          such that $not\ S_{called}\eta \in Delays$
    $\mathcal{S} := \mathcal{S} \cup ansc(\theta, S, Cntr)$
    while ($\mathcal{S} \neq \emptyset$)
      Let $f \in \mathcal{S}$ be such that the counter of f is the minimal counter for all facts in $\mathcal{S}$
      If $f = P(S_{called}, \eta, S, \theta, Cntr)$ where $P = smpl\_succ\_p$ or $smpl\_succ\_n$
        $Child = H$:-$Delay - Lit|$
      Else $Child = fail$
      $NodeSet := NodeSet \cup Child$; $EdgeSet := EdgeSet \cup (Node, Child)$;
      $\mathcal{S} := \mathcal{S} - f$

Fig. 8. Algorithm to create children of non-root nodes via the forest log

In this case, the non-failure children of $Node$ in $\mathcal{T}$ are produced by NEGATIVE RETURN or DELAYING.

* In the case where L is the leftmost literal in $Body$ (lines 15-23 of Figure 8), the fact that the rules for $Subgoal$ are distinguishable means that the NEGATION SUCCESS (i.e., NEGATION RETURN where the selected literal succeeds) and DELAYING operations that create children for $Node$ are identifiable by calling all facts of the form

  $$nr(Subgoal_{called}, Subgoal_v, Ctr)$$

  or

  $$dly(Subgoal_{called}, Subgoal_v, Ctr)$$

  such that $Subgoal_v$ is a variant of *Subgoal* and $Subgoal_{called}$ is a variant of A. For each operation create_children() properly creates children of the form

  $$(Subgoal\theta\text{:-}Delay|Body')$$

  or

  $$(Subgoal\theta\text{:-}Delay \cup L|Body').$$

  So far this case parallels the case where L is positive and leftmost. However, in the case that there are not such $nr/3$ or $dly/3$ facts in the log, create_children() adds a a child $fail$ corresponding to a NEGATION FAILURE operation on $Node$ in $\mathcal{T}$ (lines 22-23).
* The next case (lines 24-32 of Figure 8) L is not the leftmost literal in $Body$, so that create_children() creates the nearest tabled descendent of $Node$. In this case the fact that the rules for $Subgoal$ are distinguishable means that any nearest tabled descendent can be identified by calling all facts of the form

  $$nr(Subgoal_{called}, Subgoal_v, Ctr)$$

  or

  $$dly(Subgoal_{called}, Subgoal_v, Ctr)$$

  such that $Subgoal_v$ is a variant of *Subgoal* and $Subgoal_{called}$ unifies with A with mgu ξ. For each operation create_children() (lines 25-30) properly creates children of the form

  $$(Subgoal\theta\text{:-}Delay|Body')\xi$$

  or

  $$(Subgoal\theta\text{:-}Delay \cup L|Body')\xi.$$

  In the case that there are no such $nr/3$ or $dly/3$ facts in the log, create_children() (lines 31-32) adds a a child $fail$ corresponding to a NEGATION FAILURE operation on $Node$ in $\mathcal{T}$.

  – $Node = S\theta$:-$Delay|$. In other words, $Body$ is empty so that $Node$ is an answer (lines 33-45 of Figure 8). If $Delay$ is empty, $Node$ is an unconditional answer and will have no children. Otherwise if $Delay$ is non-empty its children will be produced by SIMPLIFICATION and ANSWER COMPLETION. Note that all of these operations are logged, and none of these operations changes the bindings of $S\theta$. Since all of the simplification log facts and *ansc/3* facts contain S, and $S\theta$, and the simplified literals as their arguments, the applicable operations can be identified (regardless of whether the rules are distinguishable). The only remaining issue in producing $\mathcal{H}(\mathcal{T})$ is to properly order the operations, which is done in a straightforward manner by create_children() (lines 39-45).

In each of the above cases, each log fact for $\mathcal{T}$ is accessed in constant time as the terms in $\mathcal{T}$ are assumed to have a fixed maximal size, while accessing all program clauses that unify with S can be performed with cost linear in the size of P as terms in P are also assumed to have a fixed maximal size. This set of facts are sorted, further accessed and compared to program clauses, and the sorting adds a log factor to the complexity of the operation. When edges are produced, they need to be compared to other edges with is constant time as a maximal term size is assumed.. As a result, the total cost of constructing the tree is $\mathcal{O}(size(\mathcal{T})\log(size(\mathcal{T})) + size(P))$. ∎