A

Terminating Evaluation of Logic Programs with Finite Three-Valued Models

FABRIZIO RIGUZZI, University of Ferrara
TERRANCE SWIFT, Coherent Knowledge Systems, Inc. and NOVALincs, Universidade Nova de Lisboa, Portugal

As evaluation methods for logic programs have become more sophisticated, the classes of programs for which termination can be guaranteed have expanded. From the perspective of answer set programs that include function symbols, recent work has identified classes for which grounding routines can terminate either on the entire program [Calimeri et al. 2008] or on suitable queries [Baselice et al. 2009]. From the perspective of tabling, it has long been known that a tabling technique called *subgoal abstraction* provides good termination properties for definite programs [Tamaki and Sato 1986], and this result was recently extended to stratified programs via the class of bounded term-size programs [Riguzzi and Swift 2013]. In this paper we provide a formal definition of tabling with subgoal abstraction resulting in the SLG_{SA} algorithm. Moreover, we discuss a declarative characterization of the queries and programs for which SLG_{SA} terminates. We call this class *strongly bounded term-size* programs and show its equivalence to programs with finite well-founded models. For normal programs strongly bounded term-size programs strictly includes the finitely ground programs of [Calimeri et al. 2008]. SLG_{SA} has an asymptotic complexity on strongly bounded term-size programs equal to the best known and produces a residual program that can be sent to an answer set programming system. Finally, we describe the implementation of subgoal abstraction within the SLG-WAM of XSB and provide performance results.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: Tabled Logic Programming, Termination

1. INTRODUCTION

The study of termination has proven a fruitful topic in logic programming. The majority of work has focussed on analyzing termination of definite programs under SLD resolution and its extensions, such as arithmetic (e.g., [Decorte et al. 1999; Serebrenik and De Schreye 2004; Codish et al. 2005; Serebrenik and De Schreye 2005; Nguyen and De Schreye 2005; Bruynooghe et al. 2007; Nguyen et al. 2007; Schneider-Kamp et al. 2010; Voets and De Schreye 2011]). Another recent branch of work has focused on defining classes of disjunctive programs for which a model-preserving ground instantiation can be obtained in finite time, along with algorithms to produce these instantiations [Syrjanen 2001; Gebser et al. 2007; Calimeri et al. 2008; Baselice et al. 2009; Lierler and Lifshitz 2009; Baselice and Bonatti 2010; Calimeri et al. 2011; Greco et al. 2012; Greco et al. 2013]. A third branch of work has explored the termination

Author's addresses: F. Riguzzi, fabrizio.riguzzi@unife.it Dipartimento di Matematica e Informatica – University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy; T. Swift, tswift@cs.suysb.edu CENTRIA – Universidade Nova de Lisboa.

DOI:http://dx.doi.org/10.1145/0000000.0000000

properties of query evaluation for definite or normal programs under tabling [Verbaeten et al. 2001; Riguzzi and Swift 2013]. The study of termination for tabling is of particular importance as tabling has come to underly several research and commercial knowledge representation systems [Alferes et al. 2013; Yang et al. 2013; Grosof et al. 2012].

Rather than starting with SLD resolution, with grounding techniques or with tabling, one could ask what evaluation methods terminate for programs with models that are finitely representable in some manner. Of course, a model can be represented using sets of different elements — from one perspective a program itself is a set of rules representing its model — but for this paper we use the standard approach of representing models as sets of ground atoms from a program's Herbrand base[1]. The next question is what portion of a model needs to be represented. Let P be a normal program with an infinite Herbrand base. A two-valued interpretation, $\mathcal{I}_P$, of P is arguably best represented by its set of true atoms ($true(\mathcal{I}_P)$), as reasoning can still be done in a complete manner on the false atoms ($false(\mathcal{I}_P)$) when the closed-world assumption is used. However, both tabling systems and grounders work with programs whose well-founded model may be three-valued, and if $\mathcal{I}_P$ is three-valued, at least two of its three truth assignments must be represented via finite sets. In this paper, we focus on programs that have three-valued models where both $true(\mathcal{I}^P)$ and atoms whose truth assignment is undefined (*undef*$(\mathcal{I}^P)$) can be represented as finite sets of ground atoms. We term such finite models *canonical*.

Example 1.1. Consider the normal program P_{inf}:

$$p(s(X)) \leftarrow p(X).$$
$$p(0).$$
$$q(0).$$

P_{inf} does not have a finite well-founded model (denoted $WFM^{P_{inf}}$) as both $true(WFM^{P_{inf}})$ and $false(WFM^{P_{inf}})$ are infinite. However, the superficially similar program, P_{fin}:

$$p(X) \leftarrow p(f(X)).$$
$$p(0).$$
$$q(0).$$

does have a (canonical) finite model, as $true(WFM^{P_{fin}})$ and $undef(WFM^{P_{fin}})$ are both finite. Finally, the program P_{inf_undef}:

$$p(X) \leftarrow p(f(X)).$$
$$p(0).$$
$$r(f(X)) \leftarrow not\, r(X), not\, r(f(X)).$$
$$q(0).$$

does not have a canonical finite model, as $undef(WFM^{P_{inf\_undef}})$ is no longer finite.

[1]Although we do not consider them in this paper, more general definitions are sometimes useful, such as allowing non-ground universally quantified atoms, or allowing non-ground atoms whose variables are subject to constraints over some domain.

This paper explores how programs that have canonical finite models relate to previous termination classes, and how such programs can be evaluated in a top-down manner. Specifically, the results of this paper are as follows.

— We extend the fixed-point definition of bounded term-size programs [Riguzzi and Swift 2013] to *strongly bounded term-size* programs, and show that this new notion coincides with the class of programs that have a canonical finite well-founded model. We then show that for programs that are both normal and safe, bounded term-size programs strictly include finitely ground programs [Calimeri et al. 2008].
— We show that tabled SLG resolution, extended with subgoal abstraction, [Tamaki and Sato 1986; Riguzzi and Swift 2013] finitely terminates and correctly computes queries to safe, strongly bounded term-size programs. In addition, when depth-based abstraction functions are used, the abstract complexity of query evaluation equals the best complexity that is known[2]. As usual with SLG, the derived answers can be seen as a partially transformed program that preserves the stable model semantics, and so can be used by a grounder.
— We describe a publically available, engine-level implementation of subgoal abstraction that is sound and complete for safe, strongly bounded term-size programs, and provide performance results concerning this engine.
— We discuss how these results have recently been used for applications (e.g., [Liang and Kifer 2013], and discuss further areas to which these results are relevant (e.g., [Jansen et al. 2013; Grosof and Swift 2013]).

2. BACKGROUND

We recall those concepts of logic programming used in this paper. For a general treatment see e.g., [Lloyd 1987].

We assume a language $\mathcal{L}$ containing a finite set $\mathcal{F}$ of predicate and function symbols, and a countable set of program variables from the set $\mathcal{V}$. A term is either a variable (e.g., Y), a function symbol of arity 0 (e.g., c) or a function symbol of arity n applied to a tuple of n terms (e.g., $f(t_1, \ldots, t_n)$). Symbols within a term may be represented through *positions* which are members of the set Π. A position in a term is either the empty string Λ that reaches the root of the term, or the string $\pi.i$ that reaches the ith child of the term reached by π, where π is a position and i an integer. For a term t we denote the symbol at position π in t by $t|_\pi$. For example, $p(a, f(X))|_{2.1} = X$. We suppose that $\mathcal{L}$ also contains a countable set of variables $\hat{\mathcal{V}}$, disjoint from $\mathcal{V}$, called *position variables* of the form X_π, where π is a position. A position variable is used in order to associate a given variable with a position of interest in a term.

An atom A for a predicate symbol p of arity n is p applied to a tuple of n terms: $p(t_1, \ldots, t_n)$; $pred(A)$ indicates the predicate of the atom A. A literal is either an atom A or the negation of an atom $not\, A$. A term, atom or literal is ground if it does not contain variables. A substitution θ is a set of pairs V/s where V is a variable and s is a term. A substitution applied to a term/atom/literal t, indicated with $t\theta$, replaces each variable V in t that appears in a pair V/t in θ with t. An atom A subsumes an atom B if there is a substitution θ such that $A\theta = B$[3].

We assume that a program P is defined over a language $\mathcal{L}$. The set of ground terms of a language $\mathcal{L}$ is called the Herbrand universe of $\mathcal{L}$ and is denoted by $\mathcal{H}_\mathcal{L}$, or as $\mathcal{H}_P$

[2]The complexity results of Section 4.6 are significantly more precise than previous results for SLG, which showed that an evaluation required a number of operations that was polynomial in the size of a ground program.

[3]We assume that the non-position variables of A and B are standardized apart and that unification is performed with an occur check. Note that since each position variable is tied to a specific position in a term, position variables do not need to be standardized apart.

if $\mathcal{L}$ consists of the predicate and function symbols in P. The set of ground atoms of a language $\mathcal{L}$ is called the Herbrand base and is denoted as $\mathcal{B}_\mathcal{L}$ or as $\mathcal{B}_P$. Two atoms are considered identical if they are variants of each other (informally, if the atoms are the same up to variable renaming).

Throughout this paper we restrict our attention to normal programs, and to queries that are simply atoms. A normal program is a set of normal rules. We also assume a fixed strategy for selecting literals in a clause: without loss of generality we assume the selection strategy is left-to-right. In accordance with this strategy, a normal rule has the form

$$r = H \leftarrow A_1, \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n \tag{1}$$

where $A_1, \ldots, A_n$ are atoms. We say that a predicate symbol p occurs positively (negatively) in r if p is the predicate symbol of an atom that occurs positively (negatively) in r. As notation, $literals(r)$ denotes the set of literals in the body of r and $head(r)$ denotes the head H. A rule r is *safe* if each variable in r occurs in a positive literal in the body of r, and a program is safe if all its rules are safe. For example, the rule $p(X, Y, Z) \leftarrow q(Y), not\, r(Z).$ is not safe, because X does not appear in the body and Z appears only in a negative literal.

Given a program P, $Ground(P)$ denotes the grounding of P; $Facts(P)$ denotes the set of rules with an empty body in P and $Heads(P)$ is the set of atoms in the head of some rule in P.

A *two-valued interpretation* $\mathcal{I}_T$ is a subset of $\mathcal{B}_P$. $\mathcal{I}_T$ is the set of true atoms. A *three-valued interpretation* $\mathcal{I}$ is a pair $\langle \mathcal{I}_T; \mathcal{I}_F \rangle$ where $\mathcal{I}_T$ and $\mathcal{I}_F$ are subsets of $\mathcal{B}_P$ and represent respectively the set of true and false atoms. Alternatively, a three-valued interpretation can be represented with a set of literals[4]. The union of two three-valued interpretations $\langle \mathcal{I}_T, \mathcal{I}_F \rangle$ and $\langle \mathcal{J}_T, \mathcal{J}_F \rangle$ is defined as $\langle \mathcal{I}_T, \mathcal{I}_F \rangle \cup \langle \mathcal{J}_T, \mathcal{J}_F \rangle = \langle \mathcal{I}_T \cup \mathcal{J}_T, \mathcal{I}_F \cup \mathcal{J}_F \rangle$. A three-valued interpretation $\mathcal{I}$ is a subset of a three-valued interpretation $\mathcal{J}$ iff $\mathcal{I} \subseteq \mathcal{J}$ where $\mathcal{I}$ and $\mathcal{J}$ are represented as sets of literals.

To give a semantics to normal logic programs, we need to identify one or more interpretations as the "intended models" of the program, i.e., as the interpretations giving its meaning. Many semantics have been proposed for normal programs. Among these, the well-founded semantics [van Gelder et al. 1991] and the stable model semantics [Gelfond and Lifschitz 1988] are the most prominent.

2.1. Well-Founded Semantics

The well-founded semantics (WFS) assigns a three-valued model to a program, i.e., it identifies a three-valued interpretation as the meaning of the program. The WFS was given in [van Gelder et al. 1991] in terms of the least fixed point of an operator that is composed by two sub-operators, one computing consequences and the other computing unfounded sets. We give here the alternative definition of the WFS of [Przymusinski 1989] that is based on a different iterated fixed point.

Definition 2.1. For a normal program P, sets Tr and Fa of ground atoms, and a 3-valued interpretation $\mathcal{I}$ we define

$OpTrue^P_\mathcal{I}(Tr) =$. $\{A | A$ is not true in $\mathcal{I}$; and there is a clause $B \leftarrow L_1, ..., L_n$ in P, a grounding substitution θ such that $A = B\theta$ and for every $1 \leq i \leq n$ either $L_i\theta$ is true in $\mathcal{I}$, or $L_i\theta \in Tr\}$;

[4]Note that this definition, which uses sets of true and false atoms, differs from the canonical finite models of Section 1 which uses sets of true and undefined atoms. The definition using true and false atoms will be used as a basis for the definition of the well-founded semantics in Section 2.1. Alternately, the definition of canonical finite models will be used for to characterize termination results in Sections 2.4 and 4.4.

$OpFalse^P_{\mathcal{I}}(Fa) =$. $\{A | A$ is not false in $\mathcal{I}$; and for every clause $B \leftarrow L_1, ..., L_n$ in P and grounding substitution θ such that $A = B\theta$ there is some i $(1 \leq i \leq n)$ such that $L_i\theta$ is false in $\mathcal{I}$ or $L_i\theta \in Fa\}$.

Przymusinski [1989] shows that $OpTrue^P_{\mathcal{I}}$ and $OpFalse^P_{\mathcal{I}}$ are both monotonic, and defines $\mathcal{T}^P_{\mathcal{I}}$ as the least fixed point of $OpTrue^P_{\mathcal{I}}(\emptyset)$ and $\mathcal{F}^P_{\mathcal{I}}$ as the greatest fixed point of $OpFalse^P_{\mathcal{I}}(\mathcal{B}_P)$[5]. In words, the operator $\mathcal{T}_{\mathcal{I}}$ extends the interpretation $\mathcal{I}$ to add the new atomic facts that can be derived from P knowing $\mathcal{I}$; while $\mathcal{F}_{\mathcal{I}}$ adds the new negations of atomic facts that can be shown false in P by knowing $\mathcal{I}$ (via the uncovering of unfounded sets). An iterated fixed point operator builds up *dynamic strata* by constructing successive partial interpretations as follows.

Definition 2.2 (*Iterated Fixed Point and Dynamic Strata*). For a normal program P let

$$\begin{aligned} WFM_0 &= \langle \emptyset; \emptyset \rangle; \\ WFM_{\alpha+1} &= WFM_\alpha \cup \langle \mathcal{T}_{WFM_\alpha}; \mathcal{F}_{WFM_\alpha} \rangle; \\ WFM_\alpha &= \bigcup_{\beta<\alpha} WFM_\beta, \text{ for limit ordinal } \alpha. \end{aligned}$$

Let WFM^P denote the fixed point interpretation WFM_δ, where δ is the smallest (countable) ordinal such that both sets $\mathcal{T}_{WFM_\delta}$ and $\mathcal{F}_{WFM_\delta}$ are empty. We refer to δ as the *depth* of P. The *stratum* of atom A is the least ordinal β such that $A \in WFM_\beta$ (where A may be either in the true or false component of WFM_β).

Przymusinski [1989] shows that the iterated fixed point WFM^P is in fact the well-founded model, and that undefined atoms of the well-founded model do not belong to any stratum – i.e. they are not added to WFM_δ for any ordinal δ. He called a program *dynamically stratified* if every atom belongs to a stratum. He also showed that a program has a two-valued well-founded model iff it is dynamically stratified, so that it is the weakest notion of stratification that is consistent with the well-founded semantics.

Example 2.3. Let us consider the program P_1

$$\begin{array}{ll} a(1). & \\ a(2) & \leftarrow not\, p(1,2). \\ t(f(X)) & \leftarrow a(X), not\, q(X). \\ q(g(1)). & \\ q(X) & \leftarrow t(f(X)), p(Y,X), not\, a(3). \\ p(X,Y) & \leftarrow q(g(X)), t(f(Y)), a(X). \\ p(2,3) & \leftarrow not\, p(2,1). \end{array}$$

inspired by Example 1 of [Calimeri et al. 2008]. Its iterated fix point is

$$\begin{aligned} WFM_0 &= \langle \emptyset; \emptyset \rangle; \\ WFM_1 &= \langle \{a(1), q(g(1))\}; \mathcal{B}_{P_1} \setminus \{a(1), a(2), t(f(1)), t(f(2)), q(g(1)), p(1,1), p(1,2), \\ &\quad q(1), q(2), p(2,3)\} \rangle; \\ WFM_2 &= \langle \{a(1), q(g(1)), p(2,3)\}; \mathcal{B}_{P_1} \setminus \{a(1), a(2), t(f(1)), t(f(2)), q(g(1)), p(1,1), \\ &\quad p(1,2), q(1), q(2), p(2,3)\} \rangle; \\ WFM_3 &= WFM_2 \end{aligned}$$

Thus the depth of P_1 is 3 and, for example, the stratum of $p(2,3)$ is 2. The well-founded model of P_1 can be represented as WFM_2 or as

$$true(WFM^{P_1}) = \{a(1), q(g(1)), p(2,3)\}$$

[5]Below, we will sometimes omit the program P in these operators when the context is clear.

$$undef(WFM^{P_1}) = \{a(2), t(f(1)), t(f(2)), p(1,1), p(1,2), q(1), q(2)\}$$

So WFM^{P_1} is three-valued and P_1 is not dynamically stratified.

Given a normal program P, the atom dependency graph of P is used to bound the search space of a derivation of a query Q under the WFS.

Definition 2.4 (*Atom Dependency Graph*). Let P be a normal program. Then the *atom dependency graph* of P is a graph (V, E) such that $V = \mathcal{B}_P$ and an edge $(v_1, v_2) \in E$ iff there is a grounding r of a clause in P such that $v_1 = head(r)$ and v_2 or $not\, v_2 \in literals(r)$

2.2. Stable Model Semantics

The stable model semantics [Gelfond and Lifschitz 1988] is the main alternative to the WFS. The stable models semantics associates zero, one or more two-valued models to a normal program.

Definition 2.5 (*Reduction*). Given a normal program P and an interpretation $\mathcal{I}$, the *reduction* $\frac{P}{\mathcal{I}}$ of P relative to $\mathcal{I}$ is obtained from $ground(P)$ by deleting

(1) each rule that has a negative literal $not\, A$ such that $A \in \mathcal{I}$
(2) all negative literals in the body of the remaining rules.

Thus if $\mathcal{I}$ is a full two-valued interpretation, then $\frac{P}{\mathcal{I}}$ is a program without negation as failure and has a unique least Herbrand model $lhm(\frac{P}{\mathcal{I}})$.

Definition 2.6 (*Stable Model*). A two-valued interpretation $\mathcal{I}$ is a *stable model* or an *answer set* of a program P if $\mathcal{I} = lhm(\frac{P}{\mathcal{I}})$.

The relationships between the WFS and the stable models semantics is given by the following two theorems [van Gelder et al. 1991].

THEOREM 2.7. *If P has a well-founded total model, then that model is the unique stable model.*

THEOREM 2.8. *The well-founded model of P is a subset of every stable model of P seen as a three-valued interpretation.*

The approach of solving problems by computing the answer sets of a logic program is called Answer Set Programming (ASP).

2.3. Bounded term-size Programs

Given the definitions of dynamic stratification, we are now in a position to define bounded term-size programs.

2.3.1. Norms of Terms and Atoms. Our definition of bounded term-size programs extends that of [Riguzzi and Swift 2013] to use arbitrary norms [6]. The same concept of a norm will later be used in the definition of SLG_{SA} and in the proofs of its properties.

Definition 2.9. A *norm* $N(\cdot)$ is a (total) function from terms to non-negative integers such that

(1) $N(t) = 0$ iff t is a variable.
(2) t subsumes t' implies $N(t) \leq N(t')$

[6]The use of the term *bounded* in bounded term-size is independent of other usages in the literature of termination of logic programs: [Pedreschi et al. 2002; Greco et al. 2013].

A norm is *finitary* iff for any finite non-negative integer k, the cardinality of the set $\{t|t \in \mathcal{H}_\mathcal{L} \wedge N(t) < k\}$ is finite. A norm may be defined on an atom by considering that atom as a term.

Definition 2.9 is quite general and differs from many definitions of norms that are used in the literature of termination analysis: in particular it relies on the use of subsumption. This use of subsumption is motivated by the use of norms to define term abstractions in Section 4 [7].

Example 2.10. The trivial norm, which sets $N(t) = 0$ for any term t, satisfies Definition 2.9 but is not finitary. A finitary norm has the additional property that it can be used to define finite subsets of $\mathcal{H}_\mathcal{L}$ (or $\mathcal{B}_\mathcal{L}$). For instance, $depth(\cdot)$, a norm based on the maximal depth of any subterm of a given term t (defined formally in Definition 4.10) is finitary, since for any non-negative integer n, the set of terms in $\mathcal{H}_\mathcal{L}$ for which $depth(\cdot) \leq n$ is finite. Similarly, the size norm, $size(\cdot)$, based on the total number of constant and function symbols in a term, is also finitary. More complex finitary norms may also be defined, for instance weighing list functors differently than other functors.

Definition 2.11 (*Bounded Term-size Programs*). Let P be a normal program, $norm(\cdot)$ a finitary norm, $\mathcal{I}$ a 3-valued interpretation and $Tr \subseteq \mathcal{B}_P$. Then an application of $OpTrue^P_\mathcal{I}(Tr)$ (Definition 2.1) has the *bounded term-size* property if there is an integer N such that $norm(A)$ ($= norm(B\theta)$ of Definition 2.1) is less than N for all A in $OpTrue^P_\mathcal{I}(Tr)$. P itself has the *bounded term-size* property if there is some N for which every application of $OpTrue^P_\mathcal{I}$ used to construct $WFM(P)$ has the bounded term-size property.

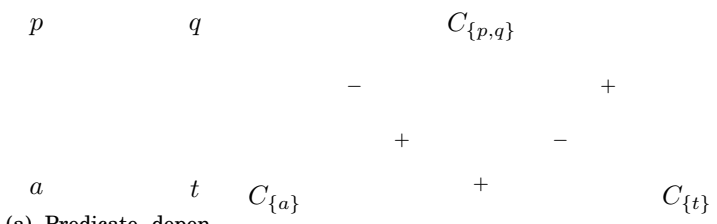
Example 2.12. P_{inf} from Example 1.1 does not have the bounded term-size property. To see this, note that having the bounded term-size property would mean that there is a bound N and a finitary norm, $norm_{fin}(A)$ such that for every atom A in the range of every application of $OpTrue^{P_{\text{inf}}}_\mathcal{I}$, $norm_{fin}(A) \leq N$. As there are an infinite number of atoms in $true(WFP(P_{\text{inf}}))$ this is impossible. On the other hand P_{fin} and P_1 from Example 2.3 do have the bounded term-size prpoperty.

While determining whether a program P is bounded term-size is clearly undecidable in general, T_{fin} shows that $ground(P)$ need not be finite if P is bounded term-size.

2.3.2. Bounded Term-size Queries. Although bounded term-size programs have appealing properties, there are many interesting programs that are not bounded term-size. For instance, a program containing the Prolog predicate *member/2* would not be bounded term-size, although as any Prolog programmer knows, a query to *member/2* will terminate whenever the second argument of the query is ground. We capture this intuition with *bounded term-size queries*.

Definition 2.13 (*Bounded Term-size Queries*). Let P be a normal program, and Q an atomic query to P (not necessarily ground). Then the *atomic search space* of Q consists of the union of all ground instantiations of Q in $\mathcal{B}_P$ together with all atoms reachable in the atom dependency graph of P from any ground instantiation of Q. Let $P_Q = \{r|r$ is a grounding of a rule of P and $head(r)$ is in the atomic search space of $Q\}$. The query Q is *bounded term-size* if P_Q is a bounded term-size program.

[7] Term abstractions are used to parameterize the NEW SUBGOAL operation of SLG_{SA} to capture both a new form of SLG that is complete for canonical finite well-founded models, and also previous versions of SLG described in the literature.



(a) Predicate dependency graph of P_1.

(b) Component graph of P_1.

Fig. 1. Graphs for P_1.

2.4. Finitely Ground Programs

Finitely ground programs were introduced in [Calimeri et al. 2008] as a class of logic programs with function symbols for which the set of ground instances of those rules that influence the computation of answer sets is finite.

The definition of finitely ground programs relies on the notion of *intelligent instantiation*, which is a method to obtain a ground program from a program with variables so that no grounding of rules that matters for the computation of answer sets is excluded.

Intelligent instantiations and finitely ground programs were defined in [Calimeri et al. 2008] with respect to disjunctive normal programs. Here we restrict these definitions to the case of non-disjunctive normal programs. Moreover, we do not distinguish between extensional and intensional predicates.

We first restate definitions from [Calimeri et al. 2008] that define dependency graphs for predicates and their components.

Definition 2.14. The *predicate dependency graph* $\mathcal{G}(P)$ of a program P is a directed graph whose nodes are the predicates of P. There is an edge (p_2,p_1) in $\mathcal{G}(P)$ if a rule for p_1 contains a positive literal for p_2 in its body[8].

Example 2.15. Program P_1 of Example 2.3 has the predicate dependency graph shown in Figure 1(a).

Definition 2.16. Given a program P and its predicate dependency graph $\mathcal{G}(P)$, the *component graph* of P, $\mathcal{G}^C(P)$, is a directed labeled graph having a node for each maximal strongly connected component (SCC) of $\mathcal{G}(P)$. $\mathcal{G}^C(P)$ is obtained by collapsing the predicate dependency graph on its SCC, i.e., $\mathcal{G}^C(P)$ has an edge (C_2, C_1) iff $C_1 \neq C_2$ and there is a rule for some $p_1 \in C_1$ such that p_2 occurs in its body. If p_2 occurs positively, the edge is labeled "+" and if p_2 occurs negatively, the edge is labeled '-' unless (C_2, C_1) can be labeled as $+$.

An ordering can be defined over the component graph.

Definition 2.17. A path in $\mathcal{G}^C(P)$ is *strong* if all its edges are labeled +, and is *weak* otherwise. A *component ordering* $\mathcal{C} = (C_0, \ldots, C_n)$ for P is a total ordering of the nodes in $\mathcal{G}^C(P)$ such that for any C_j, C_i with $i < j$ then 1) there are no strong paths from C_j to C_i and 2) if there is a weak path from C_j to C_i, then there is a weak path from C_i to C_j.

Example 2.18. Program P_1 of Example 2.3 has three SCCs: $C_{\{a\}}$, containing only predicate a, $C_{\{t\}}$, containing only predicate t, and $C_{\{p,q\}}$, containing predicates p and q.

[8]Note that this definition, unlike that of Definition 2.4, only creates edges for positive dependencies.

The component graph for P_1 is shown in Figure 1(b). There are strong paths between $C_{\{a\}}$ and $C_{\{t\}}$, $C_{\{a\}}$ and $C_{\{p,q\}}$, $C_{\{t\}}$ and $C_{\{p,q\}}$ and weak paths between any couple of components. Thus the only component ordering is $\mathcal{C} = (C_{\{a\}}, C_{\{t\}}, C_{\{p,q\}})$, so $C_0 = C_{\{a\}}$, $C_1 = C_{\{t\}}$, $C_2 = C_{\{p,q\}}$.

The set of components can be seen as a partition on the predicates of P. Moreover, each component C_i corresponds to a *module* $P(C_i)$, a subprogram of P containing all the rules with a predicate of C_i in the head.

We now turn to definitions regarding program and rule instantiations. Supposing $\mathcal{T}$ is a set of atoms that are potentially true, we define a $\mathcal{T}$-restricted instance of a rule as one that is *supported* by $\mathcal{T}$.

Definition 2.19. Let r be a rule and $\mathcal{T}$ a set of ground atoms. A *$\mathcal{T}$-restricted instance* r' of r is a ground instance of r such that if an atom a occurs positively in the body of r then $a \in \mathcal{T}$. The set of all $\mathcal{T}$-restricted instances of a program P is denoted as $Inst_P(\mathcal{T})$.

Example 2.20. Given the program module $P_1(C_{\{t\}})$ of program P_1 of Example 2.3, then

$$
\begin{aligned}
&Inst_{P_1(C_{\{t\}})}(\{a(1), a(2)\}) = \{t(f(1)) \leftarrow a(1), not\, q(1).,\ t(f(2)) \leftarrow a(2), not\, q(2).\}\\
&Inst_{P_1(C_{\{p,q\}})}(\{a(1), a(2), t(f(1)), t(f(2)), q(g(1))\}) = \{q(g(1)).,\\
&\qquad p(1,1) \leftarrow q(g(1)), t(f(1)), a(1).,\ p(1,2) \leftarrow q(g(1)), t(f(2)), a(2).,\\
&\qquad p(2,3) \leftarrow not\, p(2,1).\}\\
&Inst_{P_1(C_{\{p,q\}})}(\{a(1), a(2), t(f(1)), t(f(2)), q(g(1)), p(1,1), p(1,2)\}) = \{q(g(1)).,\\
&\qquad p(1,1) \leftarrow q(g(1)), t(f(1)), a(1).,\ p(1,2) \leftarrow q(g(1)), t(f(2)), a(2).,\\
&\qquad q(1) \leftarrow t(f(1)), p(1,1), not\, a(3).,\ q(2) \leftarrow t(f(2)), p(1,2), not\, a(3).,\\
&\qquad p(2,3) \leftarrow not\, p(2,1).\}
\end{aligned}
$$

Assuming the program is evaluated from the bottom up using a component ordering, we can identify rule groundings that do not matter for the computation of answer sets and we can simplify the bodies of some others.

Definition 2.21. Given a program P and a component ordering $(C_0, \ldots, C_n)$ for P, a set $\mathcal{S}_j$ of ground rules for component C_j and a set of ground rules $\mathcal{R}$ for the components preceding C_j, the *simplification of $\mathcal{S}_j$ with respect to $\mathcal{R}$*, $Simpl(\mathcal{S}_j, \mathcal{R})$, is obtained from $\mathcal{S}_j$ by

(1) deleting each rule whose body contains some negative literal $not\, a$ such that $a \in Facts(\mathcal{R})$.
(2) eliminating from the remaining rules in $\mathcal{S}_j$ each literal l such that
  (a) l is positive and $l \in Facts(\mathcal{R})$; or
  (b) $l = not\, a$, $pred(a) \in C_i$, $i < j$, and $a \notin Heads(\mathcal{R})$.

Example 2.22. Given program P_1 of Example 2.3, then

$$
\begin{aligned}
&Simpl(\{a(1).,\ a(2) \leftarrow not\, p(1,2).\}, \emptyset) =\\
&\qquad \{a(1)., a(2) \leftarrow not\, p(1,2).\}\\
&Simpl(\{t(f(1)) \leftarrow a(1), not\, q(1).,\ t(f(2)) \leftarrow a(2), not\, q(2).\},\\
&\qquad \{a(1)., a(2) \leftarrow not\, p(1,2).\}) =\\
&\qquad \{t(f(1)) \leftarrow not\, q(1).,\ t(f(2)) \leftarrow a(2), not\, q(2).\}\\
&Simpl(\{p(1,1) \leftarrow q(g(1)), t(f(1)), a(1).,\ p(1,2) \leftarrow q(g(1)), t(f(2)), a(2).,\\
&\qquad p(2,3) \leftarrow p(2,1).\},\\
&\qquad \{a(1).,\ a(2) \leftarrow not\, p(1,2).,\ t(f(1)) \leftarrow not\, q(1).,\ t(f(2)) \leftarrow a(2), not\, q(2).\} =\\
&\qquad \{p(1,1) \leftarrow q(g(1)), t(f(1)).,\ p(1,2) \leftarrow q(g(1)), t(f(2)), a(2).,\\
&\qquad p(2,3) \leftarrow p(2,1).\}
\end{aligned}
$$

$$Simpl(\{q(1) \leftarrow t(f(1)), p(1,1), not\, a(3).,\ q(2) \leftarrow t(f(2)), p(1,2), not\, a(3).\},$$
$$\{t(f(1)) \leftarrow not\, q(1).,\ t(f(2)) \leftarrow a(2), not\, q(2).,$$
$$p(1,1) \leftarrow q(g(1)), t(f(1)).,\ p(1,2) \leftarrow q(g(1)), t(f(2)), a(2).\} =$$
$$\{q(1) \leftarrow t(f(1)), p(1,1).,\ q(2) \leftarrow t(f(2)), p(1,2).\}$$

The operator ϕ defined below is used to select and simplify ground rules from a module $P(C_j)$ on the basis of a set of ground rules for preceding modules.

Definition 2.23. Let P be a program with component ordering $\mathcal{C} = (C_0, \ldots, C_n)$, a component C_j, a set $\mathcal{X}_j$ of ground rules of $P(C_j)$, and a set $\mathcal{R}$ of ground rules of modules of components C_i with $i < j$, let

$$\phi_{C_j,\mathcal{R}}(\mathcal{X}_j) = Simpl(Inst_{P(C_j)}(Heads(\mathcal{R} \cup \mathcal{X}_j)), \mathcal{R})$$

Since $Simpl(\mathcal{S}_j, \mathcal{R})$ is monotonic in its first argument, $\phi_{C_j,\mathcal{R}_j}$ is monotonic as well and has a least fixed point $lfp(\phi_{C_j,\mathcal{S}_{j-1}}(\emptyset))$. We can consider $lfp(\phi_{C_j,\mathcal{S}_{j-1}}(\emptyset))$ as an operator to be applied to components in order to drop many rules that do not influence answer set computation.

Definition 2.24. Let P be a program and $\mathcal{C} = (C_0, \ldots, C_n)$ a component ordering for P. The intelligent instantiation $P^{\mathcal{C}}$ of P for $\mathcal{C}$ is the last element $\mathcal{S}_n$ of the sequence

$$\mathcal{S}_0 = lfp(\phi_{C_0,\emptyset}(\emptyset)); \mathcal{S}_j = \mathcal{S}_{j-1} \cup lfp(\phi_{C_j,\mathcal{S}_{j-1}}(\emptyset))$$

Example 2.25. Given program P_1 of Example 2.3, then

$$\phi^1_{C_0,\emptyset}(\emptyset) = \{a(1).,\ a(2) \leftarrow not\, p(1,2).\}$$
$$\phi^2_{C_0,\emptyset}(\emptyset) = \phi^1_{C_0,\emptyset}(\emptyset))$$
$$\mathcal{S}_0 = lfp(\phi_{C_0,\emptyset}(\emptyset)) = \{a(1).,\ a(2) \leftarrow not\, p(1,2).\}$$

$$\phi^1_{C_1,\mathcal{S}_0}(\emptyset) = \{t(f(1)) \leftarrow not\, q(1).,\ t(f(2)) \leftarrow a(2), not\, q(2).\}$$
$$\phi^2_{C_1,\mathcal{S}_0}(\emptyset) = \phi^1_{C_1,\mathcal{S}_0}(\emptyset))$$
$$\mathcal{S}_1 = \mathcal{S}_0 \cup lfp(\phi_{C_1,\mathcal{S}_0}(\emptyset)) = \{a(1).,\ a(2) \leftarrow not\, p(1,2).,$$
$$t(f(1)) \leftarrow not\, q(1).,\ t(f(2)) \leftarrow a(2), not\, q(2).\}$$

$$\phi^1_{C_2,\mathcal{S}_1}(\emptyset) = \{q(g(1)).,p(2,3) \leftarrow p(2,1).\}$$
$$\phi^2_{C_2,\mathcal{S}_1}(\emptyset) = \{q(g(1)).,p(1,1) \leftarrow t(f(1)).,\ p(1,2) \leftarrow t(f(2)), a(2).,p(2,3) \leftarrow p(2,1).\}$$
$$\phi^3_{C_2,\mathcal{S}_1}(\emptyset) = \{q(g(1)).,\ p(1,1) \leftarrow t(f(1)).,\ p(1,2) \leftarrow t(f(2)), a(2).,$$
$$q(1) \leftarrow t(f(1)), p(1,1).,\ q(2) \leftarrow t(f(1)), p(1,2).,p(2,3) \leftarrow p(2,1).\}$$
$$\phi^4_{C_2,\mathcal{S}_1}(\emptyset) = \phi^3_{C_2,\mathcal{S}_1}(\emptyset))$$
$$\mathcal{S}_2 = \mathcal{S}_1 \cup lfp(\phi_{C_2,\mathcal{S}_1}(\emptyset)) = \{a(1).,\ a(2) \leftarrow not\, p(1,2).,$$
$$t(f(1)) \leftarrow not\, q(1).,\ t(f(2)) \leftarrow a(2), not\, q(2).,$$
$$q(g(1)).,\ p(1,1) \leftarrow t(f(1)).,\ p(1,2) \leftarrow t(f(2)), a(2).,$$
$$q(1) \leftarrow t(f(1)), p(1,1).,\ q(2) \leftarrow t(f(1)), p(1,2).,p(2,3) \leftarrow p(2,1).\}$$

We are now ready to define *finitely ground* programs.

Definition 2.26. A program P is *finitely ground* if its intelligent instantiation $P^{\mathcal{C}}$ is finite for all component orderings $\mathcal{C}$.

Example 2.27. Program P_1 of Example 2.3 is finitely ground as its intelligent instantiation is finite for the only component ordering.

Finitely ground programs enjoy the following three properties [Calimeri et al. 2008].

THEOREM 2.28. *Let P be a finitely ground program and let $P^{\mathcal{C}}$ be its intelligent instantiation for a component ordering $\mathcal{C}$. Then P and $P^{\mathcal{C}}$ have the same answer sets.*

COROLLARY 2.29. *A finitely ground program has finitely many answer sets, and each of them is finite.*

THEOREM 2.30. *Recognizing whether a program is finitely ground is semi-decidable.*

The well-founded model enjoys the following property.

PROPOSITION 2.31. *If P is a finitely ground program then $true(WFM^P)$ and $undef(WFM^P)$ are finite.*

PROOF. It is clear that the size of the set of $true(WFM^P) \cup undef(WFM^P)$ has as an upper bound the number of clauses in the intelligent instantiation, $P^{\mathcal{C}}$, of P, which is finite. □

3. STRONGLY BOUNDED TERM-SIZE PROGRAMS AND QUERIES

A program that is bounded term-size may have an infinite number of undefined atoms. We define here strongly bounded term-size programs and queries.

Definition 3.1. A normal program P is *strongly bounded term-size* iff it is bounded term-size, and in addition, $undef(WFM^P)$ is finite.

In [Riguzzi and Swift 2013] it was shown that for a normal program P, P is bounded term-size iff WFM^P has a finite number of true atoms. The following statement holds as a simple extension:

COROLLARY 3.2. *Let P be a safe normal program. Then WFM^P is a canonical finite well-founded model iff P is strongly bounded term-size.*

We now consider the relationship between strongly bounded term-size and finitely ground programs.

Example 3.3. The following program is strongly bounded term-size (in fact, bounded term-size) but is not finitely ground.

$$
\begin{array}{ll}
p(0) & \leftarrow not\, q. \\
p(f(X)) & \leftarrow p(X). \\
q. & \\
q & \leftarrow not\, p(1). \\
q & \leftarrow p(1).
\end{array}
$$

Its well-founded models is $\langle \{q\}, \mathcal{B} \setminus \{q\} \rangle$. Its components are $C_0 = \{p\}$ and $C_1 = \{q\}$, it has a strong path from $\{p\}$ to $\{q\}$ and weak paths from $\{p\}$ to $\{q\}$ and vice-versa. Accordingly, its only component ordering is (C_0, C_1) and its intelligent instantiation is

$$
\begin{array}{ll}
p(0) & \leftarrow not\, q. \\
p(f(0)) & \leftarrow p(0). \\
p(f(f(0))) & \leftarrow p(f(0)). \\
 & \ldots \\
q. & \\
q & \leftarrow not\, p(1). \\
q & \leftarrow p(1).
\end{array}
$$

If a normal program P is finitely ground then it is strongly bounded term-size as an immediate consequence of Proposition 2.31 and Corollary 3.2. Programs like that of Example 3.3 show that the inclusion is proper.

COROLLARY 3.4. *The class of programs that are strongly bounded term-size strictly includes the class of normal programs that are finitely ground.*

Strongly bounded term-size programs are undecidable just as are finitely ground programs.

THEOREM 3.5. *Recognizing whether a program is strongly bounded term-size is semi-decidable.*

PROOF. Since strongly bounded term-size includes the class of finitely ground programs, then recognizing strongly bounded term-size program is undecidable by Theorem 2.30. Semi-decidability follows by asking the query $P_{grounding}$ to the program built as in Section 4.5. If SLG_{SA} evaluation (Section 4.3) is used and answer "yes" is returned if the evaluation terminates in finite time, then the program is strongly bounded term-size. □

Strongly bounded term-size queries are defined analogously to bounded term-size queries.

Definition 3.6 (*Strongly Bounded Term-size Queries*). Let P be a normal program, and Q an atomic query to P (not necessarily ground). Then Q is *strongly bounded term-size* if P_Q is a strongly bounded term-size program.

Despite the undecidability of strongly bounded term-size programs, the class is important because it declaratively characterizes not only canonical finite models, but also the programs for which tabled SLG resolution with subgoal abstraction terminates correctly: a topic to which we now turn.

4. TABLED EVALUATION OF STRONGLY BOUNDED TERM-SIZE PROGRAMS

In this section we present a tabled evaluation method that correctly evaluates strongly bounded term-size programs. Our approach is based on SLG evaluation [Chen and Warren 1996] which models well-founded computation for logic programs at an operational level, ensuring goal-directedness, termination and optimal complexity for a large class of programs. In this section we first present the main aspects of SLG informally through an example, and then briefly recall the definitions of SLG. Afterwards, we present our extension, SLG_{SA}, along with its properties.

4.1. An Informal Review of SLG

In the forest-of-trees model of SLG [Swift 1999], an evaluation is a possibly transfinite sequence of forests (sets) of trees corresponding to subgoals that have been encountered in an evaluation. The nodes in each tree contains sets of literals divided into those literals that have not been examined, and others that have been examined, but their resolution delayed (cf. Definition 4.2). The need to delay some literals arises for the following reason. Modern Prolog engines rely on a fixed order for selecting literals in a rule, e.g., always left-to-right. However, well-founded computations cannot be performed using a fixed-order literal selection function. Hence in SLG, the DELAY operation may postpone evaluation of some literals that may be later resolved through an operation called SIMPLIFICATION. In addition to supporting the operational behavior of Prolog, the use of delay and simplification supports the termination and complexity results discussed later in this section.

Example 4.1. Consider the following program

$r_1:\ p(b).$
$r_2:\ p(c) \leftarrow not\, p(a).$
$r_3:\ p(X) \leftarrow t(X,Y,Z), not\, p(Y), not\, p(Z).$
$r_4:\ p(a) \leftarrow p(b), p(a).$
$r_5:\ t(a,a,b).$
$r_6:\ t(a,b,a).$

and query $p(c)$ (clauses are annotated with names for the purposes of explanation). The SLG forest at the end of this evaluation is shown in Figure 2 where each node is labeled with a number indicating the order in which it was created.

Nodes consist either of the symbol *fail*; or of a head representing the bindings made to an atomic subgoal along with a body containing a set of delayed literals *Delays*, followed by the | symbol, followed by a sequence of literals *Goals* that are still to be examined. The evaluation begins by creating a tree for the initial query with root $p(c) \leftarrow |p(c)$ in node 1. Children of root nodes are created via the operation PROGRAM CLAUSE RESOLUTION just as in the SLD resolution of Prolog. Accordingly, the evaluation uses rule r_2 to create node 2. The (only possible) literal $not\, p(a)$ in node 2 is

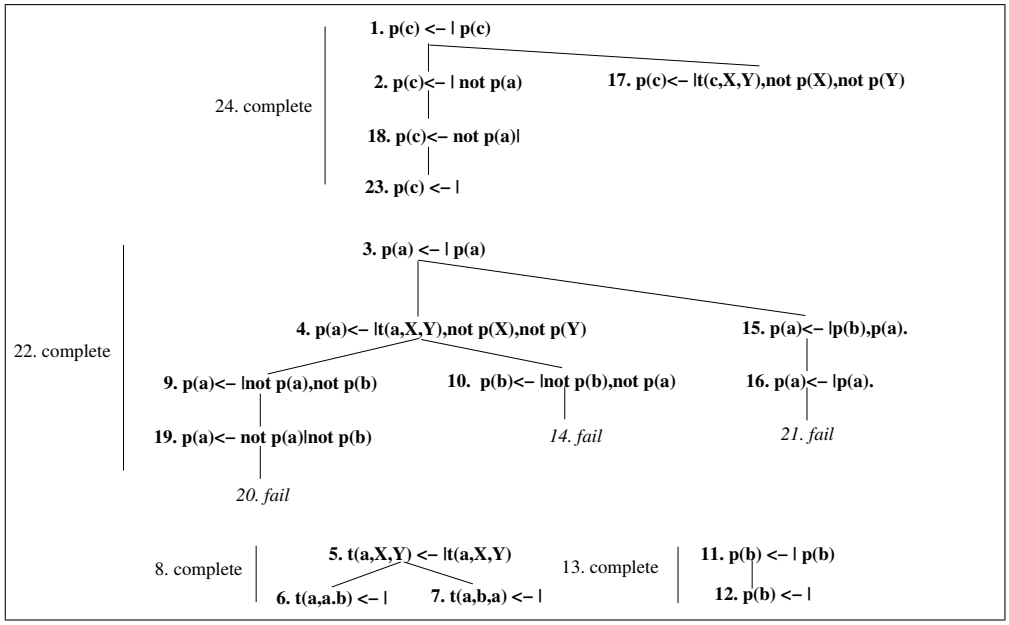


Fig. 2. Final forest for the query $p(c)$ to P_1.

selected. This literal has an underlying subgoal $p(a)$ that does not correspond to the root of any tree in the forest so far. Thus, the SLG operation NEW SUBGOAL creates a new tree for $p(a)$ (node 3), whose child, node 4, is created by PROGRAM CLAUSE RESOLUTION using rule r_3. The NEW SUBGOAL operation is again used to create a new tree for the selected literal $t(a, X, Y)$ (node 5), and children nodes 6 and 7 are created by PROGRAM CLAUSE RESOLUTION from rules r_5 and r_6. These latter nodes have empty *Goals* and are termed *answers*; moreover, since they also have empty *Delays*, they are

unconditional answers.[9] Any atom in the ground instantiation of an unconditional answer is true in the well-founded model of the program, cf. Theorem 4.15. In a case like that of the tree for $t(a, X, Y)$, when it can be determined that no more answers can be produced for the subgoal, the tree is marked as *complete* (step 8). The SLG operation POSITIVE RETURN is used to resolve the first of answer to $t(a, X, Y)$ against the selected literal of node 4, producing node 9. The selected literal of this latter node has $p(a)$ as its underlying subgoal, but there is already a tree for $p(a)$ in the forest and there are no answers for $p(a)$ to return. Since there is another unconditional answer for $t(a, X, Y)$ (node 7), POSITIVE RETURN can be used to produce node 10. The underlying subgoal $p(b)$ is selected, the tree for $p(b)$ is created by NEW SUBGOAL (node 11), and it is eventually determined that the subgoal $p(b)$ has an unconditional answer (node 12); accordingly, using the NEGATION FAILURE operation, the *failure node*, node 14, is created. Then the computation, via PROGRAM CLAUSE RESOLUTION and rule r_4, produces another child for $p(a)$, node 15, and resolves away $p(b)$ creating node 16. An application of PROGRAM CLAUSE RESOLUTION to node 1 using clause r_3 produces node 17, which cannot be further resolved. At this stage (up to node 17) the subgoal $p(a)$ is neither true, as no unconditional answers have been derived for it, nor false as one of its possible derivations, node 9, effectively has a loop through negation. However, it is possible to apply the DELAYING operation to a node such as node 2 that has $not\ p(a)$ as its selected negative literal. This operation moves $not\ p(a)$ from the *Goals* to the right of the | symbol into the *Delays* to the left of the | symbol, producing node 18, which is termed a *conditional* answer as it has empty *Goals* but non-empty *Delays*[10]. DELAYING also produces node 19 whose new selected literal $not\, p(b)$ now fails (given the unconditional answer in node 12), producing the failure node 20. At this stage, all possible operations for non-answer nodes in $p(a)$ and the trees it depends on have been performed so that the tree for $p(a)$ may be marked as complete (step 22). The completed subgoal $p(a)$ has no answers, and so is termed *failed* and is false in the well-founded model. This failed literal can be removed from the *Delays* of node 18 through the SIMPLIFICATION operation producing the unconditional answer node 23.

4.2. SLG Evaluation

SLG does not especially differ from other Prolog-like tabling formalisms in the case of programs that do not use default negation. However, as indicated in Example 4.1, for negation it introduces the concept of delaying literals in order to be able to find witnesses of failure anywhere in a rule, along with the concept of simplifying these delayed literals whenever their truth value becomes known.

An SLG evaluation proceeds by constructing a sequence of forests according to the set of SLG operations. Such forests, along with the trees and nodes it contains, are defined as follows:

Definition 4.2. A *node* has the form

$$AnswerTemplate \leftarrow Delays|Goals \qquad \text{or} \qquad \textit{fail}.$$

In the first form, $AnswerTemplate$ is an atom, while *Delays* and *Goals* are sequences of literals. The second form is called a *failure node*. An SLG *tree* T has a root of the form $S \leftarrow |S$ for some atom S: we call S *the root node for* T and T *the tree for* S. An *SLG forest* $\mathcal{F}$ is a set of SLG trees. A node N is an *answer* when it is a leaf node for

[9] In a practical program, a predicate defined by simple facts would not be evaluated using tabling, but rather would use SLD resolution as in Prolog.

[10] Choosing DELAYING in this order is not optimal and is made for purposes of illustrating the operations of SLG. This does not affect the result of the query itself since SLG is confluent [Chen and Warren 1996].

which *Goals* is empty. If the *Delays* sequence of an answer is empty, it is termed an *unconditional answer*; otherwise, it is a *conditional answer*. A tree T may be marked with the symbol *complete*.

The *underlying subgoal* of a literal L is L if L is a positive literal; otherwise it is S if $L = not\, S$.

An SLG evaluation $\mathcal{E}$ of an atomic query Q to a program P is a sequence of forests. $\mathcal{E}$ starts with an initial forest containing the single node $Q \leftarrow |Q$ and creates the n^{th} forest in the sequence by applying an SLG operation if n is a successor ordinal, or by taking the union of forests in previous sequences if n is a limit ordinal. If no further operation is applicable, then the *final forest* for the evaluation of Q has been reached. If there are selected non-ground negative literals in $\mathcal{F}$ then the evaluation is termed *floundered*. We introduce SLG operations incrementally, in Definitions 4.4, 4.6, and 4.9. Before we present the first set of operations, we introduce the definition of answer resolution, which differs from resolution in SLD in order to take account of *Delays* in conditional answers.

Definition 4.3 (*SLG Resolvent*). Let N be a node $A \leftarrow D|L_1, ..., L_n$, where $n > 0$. Let $Ans = A' \leftarrow D'|$ 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 θ. The SLG resolvent of N and Ans on L_i has the form:

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

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

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

SLG resolution delays L_i rather than propagating the answer's *Delays*, D', which means that L_i in the *Delays* of the resolvent is only resolved once all of the delay literals of D' have become true or false. This is necessary, as shown in [Chen and Warren 1996], to ensure polynomial data complexity.[11]

Definition 4.4 (*SLG Operations: 1*). Let P be a program and assume that a leftmost selection function is used to select a literal from the *Goals* in a node. Given a forest $\mathcal{F}_n$ of an SLG evaluation of P, $\mathcal{F}_{n+1}$ may be produced by one of the following operations.

(1) NEW SUBGOAL: Let $\mathcal{F}_n$ contain a tree with non-root node
$$N = Ans \leftarrow Delays|G, Goals$$
where S is the underlying subgoal of G, while $\mathcal{F}_n$ contains no tree with root S. Then add the tree $S \leftarrow |S$ to $\mathcal{F}_n$.

(2) PROGRAM CLAUSE RESOLUTION: Let $\mathcal{F}_n$ contain a tree with root node $N = S \leftarrow |S, C$ be a rule $Head \leftarrow Body$ such that $Head$ unifies with S with mgu θ, and assume N does not have a child $N_{child} = (S \leftarrow |Body)\theta$. Then add N_{child} as a child of N.

(3) POSITIVE RETURN: Let $\mathcal{F}_n$ contain a tree with non-root node
$$N = Ans \leftarrow Delays|S, Goals$$
whose selected literal S is positive. Let Ans be an answer for S in $\mathcal{F}_n$ and N_{child} be the SLG resolvent of N and Ans on S. Assume that in $\mathcal{F}_n$, N does not have a child N_{child}. Then add N_{child} as a child of N.

As illustrated in Example 4.1, NEW SUBGOAL creates a new tree in the forest $\mathcal{F}$ for a selected literal in the *Goals* of some (non-root) node in a tree in $\mathcal{F}$. Once a root node N

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

is created, the PROGRAM CLAUSE RESOLUTION operation can create children for N, given the rules in the program. POSITIVE RETURN resolves positive literals in nodes with answers already in the forest, using SLG resolution according to Definition 4.3.

If a sequence of SLG operations yields a (possibly intermediate) forest containing an unconditional answer, then this answer is considered to be true. Likewise, if no more operations are applicable to a set of trees (i.e., the set of subgoals associated to these trees is completely evaluated, Definition 4.7), and if in addition none of the trees contains an unconditional answer, then we can interpret all these subgoals as false. Extending this correspondence, we associate an SLG forest with a partial interpretation. For a final forest, this interpretation is proved to correspond to the well-founded model. (cf. Theorem 4.15 below).

Definition 4.5. Let $\mathcal{F}$ be an SLG forest. Then the *interpretation induced by* $\mathcal{F}$, $\mathcal{I}_\mathcal{F}$, is the smallest set of literals such that:

— An atom $A \in \mathcal{I}_\mathcal{F}$ iff A is in the ground instantiation of some unconditional answer $Ans \leftarrow |$ in $\mathcal{F}$.
— A literal $not\, A \in \mathcal{I}_\mathcal{F}$ iff A is in the ground instantiation of an atom whose tree in $\mathcal{F}$ is marked as *complete*, and A is not in the ground instantiation of any answer in a tree in $\mathcal{F}$.

An atom S is *successful* (resp. *failed*) in $\mathcal{F}$ if S' (resp. $not\, S'$) is in $\mathcal{I}_\mathcal{F}$ for every S' in the ground instantiation of S. A literal $not\, S$ is *successful* (resp. *failed*) in $\mathcal{F}$ if $not\, S'$ (resp. S') is in $\mathcal{I}_\mathcal{F}$ for every S' in the ground instantiation of S.

Given a three-valued interpretation $\mathcal{J}$ and forest $\mathcal{F}$, the *restriction of* $\mathcal{J}$ *to* $\mathcal{F}$, $\mathcal{J}|_\mathcal{F}$, is the interpretation such that $true(\mathcal{J}|_\mathcal{F})$ ($false(\mathcal{J}|_\mathcal{F})$) consists of those atoms in $true(\mathcal{J})$ ($false(\mathcal{J})$) that are in the ground instantiation of some subgoal whose tree is in $\mathcal{F}$.

Whenever an atom A is successful, we can fail its default negation $not\, A$. If an atom A is failed, then we can simplify away $not\, A$. Ground default negated literals that are neither failed nor successful may be delayed and later simplified. More precisely:

Definition 4.6 (*SLG Operations: 2*). Let P be program and assume a selection function as in Definition 4.4. Given a forest $\mathcal{F}_n$ of an SLG evaluation of P, $\mathcal{F}_{n+1}$ may be produced by one of the following operations.

(4) NEGATIVE RETURN: Let $\mathcal{F}_n$ contain a tree with a leaf node whose selected literal $not\, S$ is ground

$$N = Ans \leftarrow Delays | not\, S, Goals.$$

   (a) NEGATION SUCCESS: If S is failed in $\mathcal{F}_n$, then create a child for N of the form $Ans \leftarrow Delays|Goals$.
   (b) NEGATION FAILURE: If S succeeds in $\mathcal{F}_n$, then create a child for N of the form *fail*.

(5) DELAYING: Let $\mathcal{F}_n$ contain a tree with leaf node of the form

$$N = Ans \leftarrow Delays | not\, S, Goals$$

   whose selected literal $not\, S$ is ground, but S is neither successful nor failed in $\mathcal{F}_n$. Then create a child for N of the form $Ans \leftarrow Delays, not\, S|Goals$.

(6) SIMPLIFICATION: Let $\mathcal{F}_n$ contain a tree with leaf node of the form

$$N = Ans \leftarrow Delays|$$

   and let $L \in Delays$
   (a) If L is failed in $\mathcal{F}$, then create a child of the form *fail* for N.
   (b) If L is successful in $\mathcal{F}$, then create a child of the form $Ans \leftarrow Delays'|$ for N, where $Delays' = Delays \setminus \{L\}$.

SLG also includes an operation that marks a set of trees as *complete* if the corresponding set of subgoals is completely evaluated.

Definition 4.7. A set $\mathcal{S}$ of subgoals in a forest $\mathcal{F}$ is *completely evaluated* if at least one of the following conditions holds for each $S \in \mathcal{S}$:

(1) The tree for S contains an answer $S \leftarrow |$; or
(2) For each node N in the tree for S:
   (a) The underlying subgoal of the selected literal of N is marked as *complete*; or
   (b) The underlying subgoal of the selected literal of N is in $\mathcal{S}$ and there are no applicable NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN (Definition 4.4), NEGATIVE RETURN or DELAYING (Definition 4.6) operations for N.

Intuitively, a set of subgoals is completely evaluated if no further SLG operations will add information about these subgoals. If condition 1 holds, the COMPLETION operation can be applied, as any atom in the ground instantiation of S has already been determined to be true. Otherwise, condition 2(b) of the above definition prevents the COMPLETION operation from being applied to a tree from a set if certain other operations are applicable to the trees in the set. This notion of completion is incremental in the sense that once a set $\mathcal{S}$ of mutually dependent subgoals is fully evaluated, the derivation need not be concerned with the trees for $\mathcal{S}$ apart from any answers they contain. In an actual implementation, resources for such trees are reclaimed.

In certain cases the propagation of delayed literals through SLG resolution (Definition 4.3) can lead to a set of *unsupported answers* – conditional answers that are false in the well-founded model[12]. Intuitively, these answers, which have positive mutual dependencies through delay literals, correspond to an unfounded set, but their technical definition is based on the form of conditional answers.

Definition 4.8. Let $\mathcal{F}$ be an SLG forest and A be an atom that occurs in the head of some answer in a tree with root S. Then A is supported in $\mathcal{F}$ if and only if:

(1) S is not completely evaluated; or
(2) there exists an answer node $A' \leftarrow Delays|$ in S such that A' subsumes A and for every positive literal $L \in Delays$, L is supported in $\mathcal{F}$.

We are now able to characterize the last two SLG operations: one allows the completion of trees, and the other removes unsupported answers.

Definition 4.9 (*SLG Operations: 3*). Let P be a program. Given a forest $\mathcal{F}_n$ of an SLG evaluation of P, $\mathcal{F}_{n+1}$ may also be produced by one of the following operations.

(8) COMPLETION: Given a completely evaluated set $\mathcal{S}$ of subgoals (Definition 4.7), mark the trees for all subgoals in $\mathcal{S}$ as *complete*.
(9) ANSWER COMPLETION: Given a set of unsupported atoms $\mathcal{UA}$, create a failure node as a child for each answer whose head is in $\mathcal{UA}$.

Each of the operations (1)–(9), in Definitions 4.4, 4.6 and 4.9, can be seen as a function that associates a forest with a new forest by adding a new tree, adding a new node to an existing tree, or marking a set of trees as *complete*.

[12]As an aside, we note that unsupported answers appear to be uncommon in evaluation strategies that minimize the use of delay, such as those used by XSB [Swift and Warren 2012].

4.3. Extending SLG with Subgoal Abstractions

An *abstraction* of a term t, denoted $abs(t)$, may replace subterms of t by position variables: formally, $abs(t)$ is a term such that if $abs(t)|_\pi \in (\mathcal{F} \cup \mathcal{V})$, then $abs(t)|_\pi = t|_\pi$. For instance $p(f(g(X_{1.1.1}), X_{1.2}), X_2)$ is an abstraction of $p(f(g(a), X), X)$. It is easy to see that $abs(t)$ subsumes t. An abstraction $abs(\cdot)$ is finitary if the cardinality of $\{abs(t)|t \in \mathcal{H}_\mathcal{L}\}$ is finite[13]. As with norms, abstractions may be applied to atoms by taking those atoms as terms.

Definition 4.10. A *depth norm*, denoted $depth(\cdot)$, is a norm that maps a term t to the maximal *depth* of any position in t, where

— $depth(\Lambda)$ is 0 if $t|\Lambda$ is a position variable, and is 1 otherwise;
— $depth(\pi.i)$ is $depth(\pi)+1$ if $t|_{\pi.i}$ is a not a position variable, and is $depth(\pi)$ otherwise.

For a non-negative integer k, a *depth-k abstraction* of t maps $t|\pi$ to itself if $depth(\pi) \leq k$, and maps $t|\pi$ to X_π if $depth(\pi) = k + 1$.

Example 4.11. Within the atom $A = p(a, f(b, g(c)))$ the depth of c is 4. The depth 3 abstraction of A is $p(a, f(b, g(X_{2.2.1})))$, and the depth 2 abstraction of A is $p(a, f(X_{2.1}, X_{2.2}))$. Both the depth norm and the family of depth-k abstractions are finitary for any non-negative integer k. As a convention, we consider the identity function as a $depth$-ω abstraction.

The following simple lemma is immediate from the definition and will be used for the proof of the complexity of SLG_{SA}.

LEMMA 4.12. *If k is a non-negative integer, the depth-k abstraction of t is unique.*

PROOF. Immediate given that any term rooted in a position with depth $k + 1$ is replaced by a position variable that is determined by the position. □

Depth-k abstractions are simple to understand and to implement. However, the number of terms whose depth is less than k may grow exponentially in many languages. Thus, other abstractions can be practically useful: such as those based on the size of a term, or those that weigh the occurrence of certain types of function symbols over others (e.g., weighing list symbols less than other function symbols). Finally, note that the identity function on terms is an abstraction function, but is not finitary (as it maps infinite Herbrand bases to infinite sets).

The single extension to basic SLG needed to ensure finite evaluations for strongly bounded term-size programs is the addition of the use of an abstraction function to the NEW SUBGOAL operation.

Definition 4.13. NEW SUBGOAL: Let $\mathcal{F}_n$ contain a tree with non-root node

$$N = Ans \leftarrow Delays|G, Goals$$

where S is the underlying subgoal of G, while $\mathcal{F}_n$ contains no tree with root $abs(S)$. Then add the tree $abs(S) \leftarrow |abs(S)$ to $\mathcal{F}_n$.

We denote this extended version as SLG_{SA} to distinguish it from previous versions in the literature. Note that since the identity function is an abstraction function, SLG_{SA} includes SLG as a subcase.

Example 4.14. Consider the goal $p(1)$ to the program P_{fin} from Example 1.1. If this goal is evaluated with basic SLG, an infinite number of goals will be created: $p(1)$,

[13]Since abstractions replace terms by position variables, if $abs(\cdot)$ is finitary, the cardinality of $\{abs(t)|t \in \mathcal{H}_\mathcal{L}\}$ will be finite without needing to consider equivalences of terms, such as variance.

$p(f(1))$, $p(f(f(1)))$, and so on. However, if evaluated with a depth-3 abstraction function, only the first two of these goals together with $p(f(f(X_{1.1.1})))$ would be created, neither of which would have any answers. Note that the technique of "call subsumption", which is used by some tabling methods, would not help in the basic case where subgoal abstraction is not used, as none of the goals $p(1)$, $p(f(1))$, $p(f(f(1)))$, ... subsume one another.

4.4. Results

The following theorem shows how SLG_{SA} preserves the correctness of SLG, regardless of whether it is evaluating a query to a program that is strongly bounded term-size or not. The theorem is restricted to safe programs in order to exclude possibilities where abstraction leads to nonground calls to negative literals. The therorem holds for transfinite evaluations, (which have not been presented in this paper)[14].

THEOREM 4.15. *Let $\mathcal{E}$ be an SLG$_{\text{SA}}$ evaluation of a query Q to a safe program P with final forest $\mathcal{F}_{fin}$. Then $\mathcal{I}_{\mathcal{F}_{fin}} = WFM(P)|_{\mathcal{F}_{fin}}$.*

As stated below, SLG_{SA} terminates on any strongly bounded term-size program. However if subgoal abstraction is used for a goal to a rule that has non-safe negation, abstraction may introduce non-ground negative goals and hence floundered evaluations. For instance, given the rule *p(X) ← not q(X)* the goal *p(f(f(f(a))))* will not lead to unsafe negation, but if the goal were abstracted to *p(f(f(*$X_{1.1.1}$*)))*, the goal *not q(f(f(*$X_{1.1.1}$*)))* would be unsafe.

THEOREM 4.16. *Let Q be a query to a strongly bounded term-size program P. Then any SLG$_{\text{SA}}$ evaluation $\mathcal{E}$ of Q that uses a finitary abstraction operation reaches a final forest $\mathcal{F}_{fin}$ after a finite number of steps. If P is safe, then $\mathcal{F}_{fin}$ will not be floundered.*

4.5. SLG$_{\text{SA}}$ and Intelligent Instantiation

SLG_{SA} and intelligent instantiation both terminate on finitely ground programs, so it is natural to compare the two approaches.

Example 4.17. Consider the program:

$$p(X,Y) \leftarrow t(X,Y,Z), not\, p(Y,Z).$$
$$t(a,b,c).$$

The intelligent instantiation of this program is the set of clauses

$$\{t(a,b,c),\ p(a,b) \leftarrow not\, p(b,c)\}.$$

Note that because $p(b,c)$ and $p(a,b)$ are in the same component, $not\, p(b,c)$ cannot be removed from the body of the second clause. However, because SLG evaluates negation based on dynamic dependencies, the interpretation of the final forest for the top-level goal $p(X,Y)$ will assign $p(a,b)$ as true, and all other instantiations of $p(X,Y)$ as false.

In order to compare SLG_{SA} to intelligent instantiation more precisely, we need a framework to compare their results. First, since intelligent instantiation grounds an entire program while SLG_{SA} is query-oriented, we introduce the notion of a *grounding predicate* to ensure that an entire program is grounded. Let P be a safe normal program for which every predicate is tabled. Then the grounding predicate $P_{grounding}$ is

[14]The proof for this theorem, as well as other longer proofs, is given in the appendix.

defined by the set of rules:

$$P_{grounding} \leftarrow pred_1(\vec{X_1}).$$
$$P_{grounding} \leftarrow pred_n(\vec{X_n}).$$

where for each predicate $pred_i$ occurring in P, $pred_i(\vec{X_i})$ is an atom of $pred_i$ whose arguments consist solely of position variables.

Next, we specify a way to compare two ground instantiations of the same program. Let $r_1 = Head \leftarrow Body_1$ and $r_2 = Head \leftarrow Body_2$ be ground clauses: r_1 is *at least as reduced* as r_2, denoted $r_1 \geq_{red} r_2$, iff $literals(Body_1) \subseteq literals(Body_2)$. Similarly, for ground programs P_1, P_2, $P_1 \geq_{red} P_2$ iff for every rule r_1 in P_1 there is a rule r_2 in P_2 such that $r_1 \geq_{red} r_2$. Finally as notation, if $\mathcal{F}$ is an SLG$_{\mathsf{SA}}$ forest, then $answers(\mathcal{F})$ is the set of answers in $\mathcal{F}$ taken as program clauses.

The following theorem indicates that, for finitely ground programs, SLG$_{\mathsf{SA}}$ is at least as effective a grounder as intelligent instantiation. Its proof essentially follows from the correctness of SLG (and SLG$_{\mathsf{SA}}$) with respect to the stable model semantics, combined with Theorem 2.29.

THEOREM 4.18. *Let P be a safe, finitely ground program P. Let $\mathcal{E}$ be an SLG$_{\mathsf{SA}}$ evaluation of a grounding predicate of P whose final forest is $\mathcal{F}_{fin}$, and $P_{tabled} = answers(\mathcal{F}_{fin})$.*

(1) $ground(P_{tabled})$ is equal to $\frac{ground(P)}{WFM(P)}$.
(2) Let P_{ii} be the intelligent instantiation of P. Then $ground(P_{tabled}) \geq_{red} P_{ii}$.

4.6. Complexity of SLG$_{\mathsf{SA}}$

While the abstract complexity of query evaluation has been studied for SLG and its extensions (e.g., [Chen and Warren 1996; Alferes et al. 2013]), the results obtained are typically that evaluation of a ground query has polynomial complexity in the size of a given function-free program. Since SLG$_{\mathsf{SA}}$ differs only from SLG in its NEW SUBGOAL operation, a similar result can be shown for SLG$_{\mathsf{SA}}$, assuming proper conditions for $abs(\cdot)$. However, such a result does not provide any insight into the behavior of SLG$_{\mathsf{SA}}$ on strongly bounded term-size programs that contain function symbols: the very type of programs it is designed to address.

In previous approaches to complexity (e.g., [van Gelder 1989]), P is a (finite) ground program without function symbols. Define $size(r)$ for a rule r as one plus the number of body literals in r; $size(P)$ for a program P is the sum of the size of each rule. Next, let $atoms(P)$ indicate the set of atoms appearing in P. Then the best currently known bound on worst-case complexity for computing the well-founded semantics of an unrestricted ground normal program P is $size(P) \times |atoms(P)|$ [van Gelder 1989], and is shown by induction on the alternating fixed point computation of P.

In order to determine the complexity of SLG$_{\mathsf{SA}}$ on strongly bounded term-size programs that contain function symbols, a new cost model $\mathcal{C}_{function}$ is needed, as neither P nor P_Q (Definition 3.6) need be finite. Accordingly, let P be a ground strongly bounded term-size program with function symbols, and Q a ground query. In the cost model $\mathcal{C}_{function}$, the size of a rule r is defined as above: that is, one plus the number of body literals in r. Therefore $size(\cdot)$ does not consider the number of symbols or the depth of terms within an atom or literal.

Next, assume that P is a ground strongly bounded term-size program and Q a ground query. Since P may not be finite, we need to devise a finite parameter for $\mathcal{C}_{function}$ to stand in for P. By Theorem 4.16, an SLG$_{\mathsf{SA}}$ evaluation $\mathcal{E}$ of Q against P that uses a finitary abstraction function will produce a final forest $\mathcal{F}_{fin}$ after a finite number of steps, and $\mathcal{F}_{fin}$ will itself be finite. Since P is ground, then given $\mathcal{E}$, we can

construct the set of ground rules that were used in some PROGRAM CLAUSE RESOLUTION operation and denote this set as the program $P_Q(\mathcal{E})$. Since $\mathcal{E}$ is finite, $P_Q(\mathcal{E})$ must always be finite. Further, as P_Q is constructed from the atom dependency graph rather than from a dynamic computation, it is evident that $P_Q(\mathcal{E}) \subseteq P_Q$. Next, define $atoms(\mathcal{F}_{fin})$ as the set of atoms A such that that A occurs as the head of some node in $\mathcal{F}_{fin}$: in other words, $atoms(\mathcal{F}_{fin})$ captures the set of atoms that were evaluated by $\mathcal{E}$. It is evident that $atoms(\mathcal{F}_{fin})$ is finite, although $atoms(P_Q)$ may not be[15]. Nonetheless in the worst case, if $atoms(P_Q)$ is finite then $atoms(\mathcal{F}_{fin})$ may contain every atom in P_Q plus those roots of trees that have been abstracted via a non-trivial application of the depth-k abstraction function. Thus $|atoms(\mathcal{F}_{fin})|$ is bounded by $2 \times |atoms(P_Q)|$ if $atoms(P_Q)$ is finite. We state these observations formally.

LEMMA 4.19. *Let P be a ground strongly bounded term-size program and Q a ground query. Let $\mathcal{E}$ be an SLG$_{\mathsf{SA}}$ evaluation of Q against P_Q that uses a finitary abstraction function, and let the final forest of $\mathcal{E}$ be $\mathcal{F}_{fin}$. Then $P_Q(\mathcal{E})$ is finite and $P_Q(\mathcal{E}) \subseteq P_Q$. In addition, $atoms(\mathcal{F}_{fin})$ is finite and if $atoms(P_Q)$ is also finite then $|atoms(\mathcal{F}_{fin})| \leq 2 \times |(atoms(P_Q)|$.*

Since $size(P_Q(\mathcal{E})) \times |atoms(\mathcal{F}_{fin})|$ is finite, and at most $2 \times size(P) \times |atoms(P)|$, showing that an SLG$_{\mathsf{SA}}$ evaluation has complexity of at most $size(P_Q(\mathcal{E})) \times |atoms(\mathcal{F}_{fin})|$ will ensure that it has the optimal known complexity for evaluating a query according to WFS. Towards this end, the following lemma bounds the number of nodes in the final forest of an evaluation. Its proof depends on showing that a ground program clause is used to produce a node in exactly one tree in a computation. This property holds under $\mathcal{C}_{function}$ for depth-k abstractions, although it remains open for arbitrary abstractions.

LEMMA 4.20. *Let P be a ground program, Q a ground query, and $\mathcal{E}$ a terminating SLG$_{\mathsf{SA}}$ evaluation of Q against P that uses depth-k abstraction. Then the number of nodes in the final forest $\mathcal{F}_{fin}$ is at most $\mathcal{O}(size(P_Q(\mathcal{E})))$.*

As a next step in defining $\mathcal{C}_{function}$, we consider the cost of each SLG$_{\mathsf{SA}}$ operation. First, since the scope of an abstraction function is an atom, the cost of applying an abstraction function is constant in $\mathcal{C}_{function}$[16]. Note that the NEW SUBGOAL operation creates a root node for a given atomic subgoal, and thus may be considered a constant-time operation. Similarly the POSITIVE RETURN, NEGATIVE RETURN, DELAYING, and SIMPLIFICATION operations each affect one goal or delay literal and may also be considered constant-time. The PROGRAM CLAUSE RESOLUTION, however, has a cost proportional to the size of the rule it applies. The COMPLETION operation applies to a set of subgoals $\mathcal{S}$ in a forest $\mathcal{F}$ so that its cost is proportional to the cardinality of $\mathcal{S}$: in the worst case this is $|atoms(\mathcal{F})|$. Similarly, the ANSWER COMPLETION operation must determine an unsupported set of answers and its worst-case cost is $size(P_Q(\mathcal{E}))$.

The cost model $\mathcal{C}_{function}$ thus consists of

— The definition of the size of a program that contains function symbols based on those parts of a program dynamically traversed by an SLG$_{\mathsf{SA}}$ evaluation: $P_Q(\mathcal{E})$;
— The number of its atoms in the evaluation to which operations might be applied: $atoms(\mathcal{F}_{fin})$; and

[15]Note, that there is a maximum number n_{sym} of symbols for any atom in a given set $atoms(\mathcal{F}_{fin})$ since $atoms(\mathcal{F}_{fin})$ is finite. This consideration helps motivate the choice that $size(\cdot)$ does not consider the number of symbols within particular atoms, as $size(\cdot)$ can be parameterized by n_{sym}.

[16]In a practical implementation of SLG$_{\mathsf{SA}}$, atoms need to be traversed to determine whether operations are applicable. Although any abstraction function is constant in $\mathcal{C}_{function}$, a practical abstraction function should have a low cost as a function of the actual size of an atom to which it is applied (i.e., as a function of the total number of positions in an atom).

— The costs for each individual SLG_{SA} operation.

Under this cost model, it is shown in the appendix that the following theorem holds. The proof makes use of the way in which the definitions of SLG_{SA} operations prevent their redundant application, and effectively "amortizes" the cost of the non-constant-time operations.

THEOREM 4.21. *Let P be a ground program, Q a ground query, and $\mathcal{E}$ a terminating SLG_{SA} evaluation of Q against P that uses depth-k abstraction, and with final forest $\mathcal{F}_{fin}$. Then under the cost model $\mathcal{C}_{function}$, the cost of $\mathcal{E}$ is $\mathcal{O}(|atoms(\mathcal{F}_{fin})| \times size(P_Q(\mathcal{E})))$.*

5. IMPLEMENTATION OF SUBGOAL ABSTRACTION

Depth-k subgoal abstraction is built into the XSB engine and supported in Versions 3.3.8 and higher of XSB (xsb.sourceforge.net). Subgoal abstraction can be invoked in various ways, the most flexible of which is to set it on a per-predicate basis by the directive:

:- table <predspec> as subgoal_depth(<n>),...

For the predicate(s) in *predspec*, this directive ensures depth-k abstraction, setting *subgoal_depth(<n>)* as a table property that can be combined with other properties such as *incremental*, *thread-private*, *thread-shared*, and so on[17]. In order to avoid the creation of floundering subgoals, XSB only abstracts positive literals; however, this limitation does not affect termination for strongly bounded term-size programs that are safe, as the binding of each variable in a negative goal must have been produced as part of an answer to a positive subgoal. Within XSB, abstraction is permitted on subgoals with attributed variables (which support constraint-based reasoning) as described below.

At a high level, the implementation of subgoal abstraction can be seen as a dynamically performed rewriting of a subgoal:

$$\ldots, G\theta, \ldots \Rightarrow \ldots, abs(G\theta), abs(G\theta) = G\theta, \ldots \tag{2}$$

i.e., the goal $G\theta$ is replaced by the depth-k abstraction $abs(G\theta)$ and $abs(G\theta)$ is called; any answers returned for $abs(G\theta)$ are unified with the original goal $G\theta$ – a step we term *post-unification*[18].

Example 5.1. As a concrete example, suppose the goal $p(1)$ was made to the program P_{sbts}

$$\begin{array}{ll} p(X) & \leftarrow p(f(X)). \\ p(f(f(X))) & \leftarrow q(X). \\ q(0). & \\ q(1). & \end{array}$$

in an evaluation where depth-3 abstraction is used. The tabled subgoal $p(1)$ produces the subgoal $p(f(1))$ by PROGRAM CLAUSE RESOLUTION against the clause $p(X) \leftarrow p(f(X))$, and then the subgoal $p(f(f(1)))$ is produced by resolution against the same clause. Setting $G\theta = p(f(f(1)))$, then its depth-3 abstraction, $abs(G\theta)$, is $p(f(f(X_{1.1.1})))$ and by Formula (2), the subgoal $p(f(f(X_{1.1.1})))$ would be called. This

[17] See the XSB manual for the current list of properties with which depth-k abstraction is compatible.

[18] As an aside, we note that the proof of correctness of SLG_{SA} (Theorem 4.15) is based on just such a transformation.

subgoal is completely evaluated producing the answers $p(f(f(0)))$ and $p(f(f(1)))$. Afterwards, both solutions to $p(f(f(X_{1.1.1})))$ would be post-unified with $p(f(f(1)))$ but would succeed only for $p(f(f(1)))$, which allows $X_{1.1.1}$ to unify with 1.

5.1. Implementation within the SLG-WAM

Our description of engine-level details of subgoal abstraction assumes some knowledge of the SLG-WAM engine, as presented in [Sagonas and Swift 1998; Ramakrishnan et al. 1999] and other papers. Let $G\theta$ be a tabled subgoal and assume that a maximum depth, k, has been set for the underlying predicate, G_P, of $G\theta$ and that the action is set to *abstract*. The abstraction is performed during the tabletry instruction (the SLG-WAM instruction corresponding to the NEW SUBGOAL operation). Within this instruction, a single-pass *check-insert traversal* of the subgoal $G\theta$ checks whether a variant of $G\theta$ has already been encountered during an evaluation, and creates a table for the subgoal if not. During this traversal, a depth counter is initialized by checking a cell in the table information frame for the predicate G_P; if the depth k is reached at position π_i, a pointer to the subterm rooted at π_i ($G\theta|_{\pi_i}$) is added to an *abstraction stack* together with the (heap) address of π_i; then a free (position) variable X_{π_i} is created at position π_i, and trailed with a *pre-assignment cell* in its trail frame, as used for mutable variables in XSB and other Prolog systems. Such a cell contains information about the value of a variable *before* a binding, and so supports backtrackable "destructive" assignment within a Prolog engine. After the abstraction and trailing, X_{π_i} is copied into the table in the normal manner (as part of the abstracted goal $abs(G\theta)$). If $G\theta$ is part of a set of mutually dependent subgoals, the SLG-WAM may need to repeatedly suspend and resume computation of $abs(G\theta)$ as answers for other subgoals are derived and used for resolution. In general, the trail for the SLG-WAM supports suspending and resuming environments with a *value cell*: that is, it trails the value of the binding to a variable along with the variable itself. However, abstractions may also need to be undone and re-applied during environment switching; because pre-assignment trailing is used, the abstracted variables are reset to their prior (non-abstracted) terms when backtracking above the call to $G\theta$, then reset to their abstracted value via the value cell. To summarize, trail frames for abstracted variables require 1) the pre-assignment cell (in this case, pointing to $G\theta|_{\pi_i}$); 2) the value cell (in this case, pointing to X_{π_i}); and 3) the variable address cell, just as in the WAM (in this case, the address of X_{π_i}). Trailing for non-abstracted variables does not require the pre-assignment cell[19].

During the same check-insert traversal of the subgoal $G\theta$, the SLG-WAM creates a *substitution factor*: a vector that corresponds to the set of variables in $G\theta$. Substitution factors are maintained in the heap, and are not part of permanent table storage. The use of substitution factors allows the SLG-WAM to represent answers in a table as substitutions to the variables in a subgoal, which is more compact than representing answers as atoms that are instantiations of the subgoal [Ramakrishnan et al. 1999]. Using the substitution factor, when an answer is derived for a generator of $G\theta$ or returned to a consumer of $G\theta$, the engine need only copy bindings into or out of the table by traversing the substitution factor, rather than having to re-traverse the entire subgoal and answer. When subgoal abstraction is used for $G\theta$, the abstraction code ensures that the frames of the abstraction stack are also added to the substitution factor. The abstraction frames are then used for post-unification: the subterm in each abstracted position of the answer ($abs(G\theta)\eta|_{\pi_i}$) is unified with the original subterm at that position ($G\theta|_{\pi_i}$). Only if all such unifications succeed is the answer return successful.

[19] As noted in [Sagonas and Swift 1998] since the SLG-WAM trail represents a tree rather than a stack, frames also contain a cell that points to their previous trail frame. Such a cell is required regardless of whether a trailed variable has been abstracted.

Example 5.2. Continuing from Example 5.1, for the abstracted subgoal $p(f(f(X_{1.1.1})))$, the substitution factor would consist of the variable $X_{1.1.1}$. When evaluating $p(f(f(X_{1.1.1})))$ the use of the substitution factor allows the engine to traverse only the bindings (0 and 1) to $X_{1.1.1}$ when copying answers into the table, and to store only these bindings: as indicated in [Ramakrishnan et al. 1999], no retraversal or storage of ancestor positions is necessary. As mentioned above, support for SLG_{SA} requires augmenting the substitution factor with a series of abstraction frames, here a single frame containing $X_{1.1.1}$ and 1. When copying answers out of the table the variable $X_{1.1.1}$ of the substitution factor is bound, and once this is accomplished, the post-unifications of the abstraction stack are performed: here unifying the bound value of $X_{1.1.1}$ with 1, so that only the binding $X_{1.1.1} = 1$ succeeds.

If a tabled subgoal contains attributed variables, the attributed variables are handled as follows. XSB tables subgoals with attributed variables by copying variable attributes into the table as specially designated terms. Suppose subgoal abstraction replaces a term t rooted in position π with a free variable X_π. If t contains an attributed variable as a subterm, then the post-unification of the attributed variable may call a unification hook, just as any unification would, so that the abstraction code need not treat such abstracted variables in a special manner. However, if the depth bound is exceeded while traversing a variable attribute, abstraction is disabled until the attribute has been traversed. The reason for this is that abstracting midway through a variable attribute would break the unification hooks for many classes of attributes.

To summarize, the engine-level implementation of subgoal abstraction essentially involves the ability to dynamically implement the rewriting of Formula (2). First of all, rewriting $G\theta$ into $abs(G\theta)$ requires the ability to calculate the norm of a goal, and to apply an abstraction function. In the case of depth-k abstraction, calculation of the norm and abstraction application can be done without an additional term traversal beyond that needed for tabling. Once the abstraction is performed, the abstraction vector is used to perform the post-unification: $abs(G) = G\theta$. Because tabling requires not just backtracking, but suspension and resumption of subgoals, the trail must be extended to undo the abstraction on backtracking caused by suspension, as well as to redo the abstraction when a tabled subgoal is resumed. The implementation of each of these operations requires care, as they interact with some of the lowest-level functionality of a tabling engine. At the same time, these operations affect only a few parts of an engine in very specific ways. As a result, the implementation of the depth check and abstraction, of abstraction vectors, and of the post-unification of abstracted answers with the original goal required a total of about 300-400 lines of code in XSB.

5.2. Tests for Performance Overhead of Subgoal Abstraction

Subgoal abstraction obviously improves performance by ensuring termination when the atom dependency graph of a program contains an infinite path (as in P_{fin} of Example 1.1). Additionally, if an abstraction allows different subgoals to share the same table that otherwise would not, it can benefit performance in a manner similar to call subsumption. Because of these obvious benefits, its is natural to ask if there are cases when subgoal abstraction should *not* be used, a question we address here by measuring the performance overhead of subgoal abstraction.

To investigate the overhead of subgoal abstraction, a series of tests were executed on a Macintosh laptop with a 2.43 GHz Intel Core i5 processor and 4 GBytes of memory, running OS X 10.6.8. Timings on this platform show a variance of up to 6% for the same executable and program.

As a preliminary step, various cases of linear recursion were tested, comparing a version of XSB with subgoal abstraction implemented but not turned on, against the

$$p\_1(X,F) \leftarrow q\_1(X). \qquad p\_1(X,F) \leftarrow succ1mil(X,Z), p\_1(Z,F).$$

$$p\_2(X,F,Y) \leftarrow q\_2(X,Y). \quad p\_2(X,F,Y) \leftarrow succ1mil(X,Z), p\_2(Z,F,Y).$$

Fig. 3. P_{over}: A program for benchmarking the overhead of subgoal abstraction.

previous version without subgoal abstraction. These timings (not shown) indicated no significant difference in times. This is not surprising: if subgoal abstraction is not invoked, the only overhead is the maintenance of the depth counter during the check-insert step for tabled subgoals (cf. Section 5.1).

The next two series of timings test the overhead of subgoal abstraction when it is turned on, but does not provide an advantage in sharing tables for different subgoals. These tests use the programs $p\_1/2$ and $p\_2/3$ in Figure 3 with results displayed in Table I. In these series, each benchmark makes use of the predicate $succ1mil/2$, the successor function for integers less than 1 million.

The first series of benchmarks was constructed as follows.

— $p_1/2$*: no answer:* In these tests, the base predicate $q\_1/1$ was set so that it never succeeded. Under this setting, the goal $p\_1(0,F)$ creates 1 million variant subgoals, but no answers for any of these subgoals.
— $p_1/2$*: unique answer:* In these tests, the fact $q\_1(1000000)$ was added so that each of the million goals for $p\_1/2$ contained a single answer.
— $p_1/2$*: redundant answers:* In these tests, 4 or 16 facts of the form $p\_1(\_,\_)$ were also added. In these tests each subgoal had one answer that was redundantly rederived 4 or 16 times.

The second series of benchmarks was similar, but here the base predicate, $q\_2/2$, was adjusted so that 1, 4, or 8 answers were derived for each of 1 million subgoals. Specifically, facts of the form $q(1000000, \langle I \rangle)$ were added with $\langle I \rangle$ ranging from 1 to 1, 4 or 8. In this second series, no redundant answers were derived.

Both benchmark series had two additional parameters. For each set of benchmarks, the depth limit for subgoal abstraction was either turned off or set to 6. In addition, the top-level goals were $p\_1(0,F)$ or $p\_2(0,F,\_)$ – where in each case F was bound to terms of the form $f^n(1)$ for n equal to 0, 2, 4, 8, 16 and 32 (i.e., $f^0(1) = 1$, $f^1(1) = f(1)$, $f^2(1) = f(f(1))$, etc.). Note that the different values of F do not affect the number of answers derived for any of these benchmark programs, but when F was set to $f^n(1)$ for $n = 8$, 16, and 32 the subgoals are (non-trivially) abstracted if the depth-limit is set.

We consider the first series, based on $p\_1/2$. When no answers are derived, the timings for $p\_1/2$ (Table I) show that subgoal abstraction reduces runtime up to 98% in the case of $f^{32}(1)$ compared to the runtime when subgoal abstraction is not used (cf. the first two lines of Table I). In this case, if subgoal abstraction is not used, each goal needs to be fully traversed and copied into the table, but when subgoal abstraction is used, subterms with depth greater than k do not need to be traversed: instead a pointer to the subterm is simply added to the abstraction stack, leading to efficiency for subgoal abstraction. As a contravening factor, if abstraction is not used, the F argument is ground, while if abstraction is used the abstracted variable is added to the substitution factor, and must be traversed when copying an answer into or out of the table. Accordingly, when $p\_1/2$ derives a single answer per goal, subgoal abstraction only shows improvements above the noise level for $f^{16}(1)$ and $f^{32}(1)$. Indeed, as further redundant answers are derived, the cost of traversing the binding to the redundant answer for the abstracted goal outweighs the savings made for the abstracted call, by up to 217% when 16 redundant answers are derived per goal. Although adding redundant

Table I. Benchmark results for tests of subgoal abstraction overhead (times in seconds)

Program	1	f(1)	$f^2(1)$	$f^4(1)$	$f^8(1)$	$f^{16}(1)$	$f^{32}(1)$
First series							
p_1/2 (no answers) no abstr	0.424	0.471	0.517	0.693	0.96	1.214	1.702
p_1/2 (no answers) abstr	0.431	0.489	0.533	0.72	0.864	0.862	0.864
p_1/2 (1 answer) no abstr	0.524	0.576	0.62	0.808	1.071	1.324	1.841
p_1/2 (1 answer) abstr	0.529	0.579	0.621	0.809	1.009	1.008	1.01
p_1/2 (1 answer + 4 redund) no abstr	0.534	0.583	0.623	0.809	1.072	1.332	1.839
p_1/2 (1 answer + 4 redund) abstr	0.524	0.578	0.623	0.81	1.375	1.376	1.377
p_1/2 (1 answer + 16 redund) no abstr	0.529	0.582	0.623	0.809	1.069	1.32	1.831
p_1/2 (1 answer + 16 redund) abstr	0.525	0.58	0.625	0.809	2.336	2.341	2.352
Second series							
p_2/3 (1 answer) no abstr	0.633	0.676	0.765	0.928	1.147	1.398	1.917
p_2/3 (1 answer) abstr	0.625	0.664	0.769	0.938	1.135	1.14	1.136
p_2/3 (4 answers) no abstr	1.042	1.079	1.187	1.351	1.553	1.812	2.316
p_2/3 (4 answers) abstr	1.024	1.067	1.168	1.334	1.651	1.658	1.649
p_2/3 (8 answers) no abstr	1.485	1.535	1.625	1.795	2.016	2.256	2.768
p_2/3 (8 answers) abstr	1.489	1.529	1.623	1.797	2.199	2.209	2.202

answers leads to a performance cost for subgoal abstraction due to the added cost for answer checks, the post-unification step was used only once: the first answer added to the table.

On the other hand for the second series, no answers for $p\_2/2$ are redundant, so that in addition to the cost of answer check/insert, the cost of post-unification of each answer is also measured. Timings for $p\_2/3$ overall show less savings than those for $p\_1/2$ for abstraction compared to non-abstraction; however the second series still shows savings for $f^{16}(1)$ and $f^{32}(1)$.

These timings show that subgoal abstraction can be implemented so that its overhead is negligible if it is not used. If the number of answers per subgoal is relatively low and the subgoals are large, subgoal abstraction provides performance improvement by traversing the subgoal in almost a "lazy" manner. However, the cost for this is that the bindings for answers to abstracted goals will be larger than for non-abstracted goals, and traversing these answers to check for redundancy or to perform post-unification can lead to performance degradation, particularly when there are numerous answers per subgoal.

5.3. A Benchmark in the Style of Knowledge Representation

To test out the use of subgoal abstraction in a somewhat more realistic situation, a simple program, P_{krr}, was created that abstractly represents personal preferences (Figure 4). Despite its simplicity, P_{krr} captures certain aspects of reasoning over knowledge bases such as the use of default and explicit negation. More to our purposes, P_{krr} uses logical functions to represent existential information in a manner similar to some description logics. For instance, the parent of a given human being, p_1, can be denoted as $parent\_of(p_1)$, which might or might not correspond to a given atomic constant in P_{krr}. Equality among terms is represented by the predicate $equals/2$. In Figure 4, the first rule for this predicate states that $equals/2$ is symmetric, and the second and third that it is reflexive. In addition, $parent\_of(X)$ is equal to a term Y such that the relation $parent\_of(X, Y)$ holds. Thus, $parent\_of(p_1)$ may be shown equal to, say p_2, if $parent\_of(p_2, p_1)$ holds; if equality cannot be proved, it is treated as an existential fact.

Parenthood is used to prove that a given object is a human: this can be shown either by an explicit statement that the object is a person or by default, if it cannot be shown that an object is not non-human. An object is non-human if it is known to be a dog, or if the object's parent is non-human. The $loves/2$ relation makes use of existential knowl-

$$
\begin{array}{l}
:-table\ human/1, neg\_human/1\ as\ subgoal\_abstract(3).\\
human(X) \leftarrow person(X).\\
human(X) \leftarrow human(parent\_of(X)), not\ neg\_human(parent\_of(X)).\\
neg\_human(X) \leftarrow dog(X).\\
neg\_human(X) \leftarrow neg\_human(parent\_of(X)).\\
\\
:-table\ equals/2\ as\ subsumptive.\\
equals(X,Y) \leftarrow equals(Y,X).\\
equals(X,X) \leftarrow human(X).\\
equals(X,X) \leftarrow neg\_human(X).\\
equals(parent\_of(X),Y) \leftarrow parent\_of(X,Y).\\
equals(parent\_of(parent\_of(X)), parent\_of(Z)) \leftarrow parent\_of(X,Z).\\
equals(parent\_of(parent\_of(X)),Y) \leftarrow equals(parent\_of(X),Z), parent\_of(Z,Y).\\
\\
:-table\ loves/2, neg\_loves/2.\\
loves(X,Y) \leftarrow loves(Y,X).\\
loves(X,Y) \leftarrow friend(X,Y), human(X), human(Y), not\ neg\_loves(X,Y).\\
loves(X,Y) \leftarrow has\_pet(X,Y), human(X), neg\_human(Y).\\
loves(X,Y) \leftarrow equals(parent\_of(X),Y).\\
loves(X,Y) \leftarrow grandparent\_of(X,Y).\\
\\
neg\_loves(X,Y) \leftarrow works\_for(X,Y), human(X), human(Y), not\ loves(X,Y).\\
\\
grandparent\_of(X,Y) \leftarrow equals(parent\_of(prent\_of(X)),Y).\\
\\
/* \text{Base predicates} : friend/2, has\_pet/2, works\_for/2, parent\_of/2, dog/1, person/1\ */
\end{array}
$$

Fig. 4. P_{krr}: A program abstracting aspects of personal preferences.

edge and equality, default and explicit negation, along with various base predicates: $friend/2, has\_pet/2, works\_for/2, parent\_of/2, dog/1, person/1$.

To test out performance and scalability, extensions were randomly generated for the base predicates, creating extensional databases (EDBs) of from 3.7 million to 14.8 million facts. These facts were loaded into XSB along with the program in Figure 4, and the query *loves(Person,Object)* was queried for 10 and 20 randomly chosen instantiations of *Person*. Results for these queries are shown in Table II[20]. Table II reports query time as measured by XSB and total time as measured by the GNU time program; it also reports the maximum amount of memory as returned by GNU time, the amount of memory used by XSB for table space and the number of tabled subgoals.

As the EDB size increases, load time increases linearly, and the query time increases slightly more than linearly; both maximum memory and the total size for table space also increase linearly. Within a given EDB, increasing the number of queries increases the time and table space less than linearly, indicating a reuse of tables. Although XSB scales well on these benchmarks, it is worthwhile noting that the resources used by XSB proved highly sensitive to the way the *equals/2* predicate was written[21]. Query evaluation time was slowed down 4-5 times when the the last rule for *equals/2* was translated into the semantically equivalent right-recursive form:

$$equals(parent\_of(parent\_of(X)),Y) \leftarrow parent\_of(X,Z), equals(parent\_of(Z),Y).$$

[20]The tests were performed on a Linux server with two Intel Xeon X5690 CPU at 3.47GHz and 188 GB RAM. Details of the tests, including the code used to generate the datasets, are available at sites.unife.it/ai/termination.

[21]*equals/2* was implemented with call subsumption rather than subgoal abstraction as the subgoals to *equals/2* tended to vary in their instantiations, but — because of the use of subgoal abstraction in *human/1* — these instantiations already had a bounded size.

Table II. XSB benchmark results for queries to P_{krr}. Times are in seconds, Maximum Memory and Table Space are in GB.

EDB/Queries	Query Time	Total Time	Maximum Memory	Table Space	Tabled Subgoals
3.7M / 10	10.78	17.37	3.67	0.45	47
3.7M / 20	13.36	19.98	3.67	0.45	87
7.4M / 10	22.38	36.53	7.28	0.91	47
7.4M / 20	29.58	43.52	7.28	0.91	87
14.8M / 10	50.03	78.79	14.55	1.82	47
14.8M / 20	66.12	94.53	14.55	1.82	87

Although P_{krr} is a finitely ground program, it is not stratified for some EDBs as it contains a loop through negation between *loves/2* and *neg_loves/2*; however the queries used for the benchmark did not have conditional answers (i.e., they were not undefined in the well-founded model of the program). Despite its semantic simplicity, P_{krr} realistically represents certain issues that arise in evaluating queries in languages such as Flora-2 [Yang et al. 2013] and its commercial extensions Silk (silk.semwebcentral.org) and Fidji (coherentknowledge.com). The results from these experiments indicate that subgoal abstraction can be an important tool to implement scalable knowledge representation systems.

5.4. Comparisons with DLV

While XSB is an ISO-Prolog that supports tabling, DLV [Leone et al. 2002; Calimeri et al. 2008; Alviano et al. 2010] is an ASP system that has been extended to support function symbols. Despite their differences, the functionality of the two systems overlaps when computing queries to stratified programs, or grounding non-stratified programs. Repeated independent comparisons of DLV, XSB and other systems have been made in 2009, 2010 and 2011 by [OpenRuleBench 2011] and in 2011 in the Third Open Answer Set Competition [Calimeri et al. 2012]. While the comparisons as a whole illustrate general performance differences between the two systems, here we discuss the benchmarks of Figure 3 and Figure 4.

We first tested the performance overhead tests of Section 5.2, converting P_{pver} (Figure 3) into an equivalent safe program that could be handled by DLV. We measured the execution time and the maximum memory using GNU time on the Linux server used in Section 5.3. For the first series of overhead tests, on $p\_1/2$ with no answers, we could not get a result after several hours with the default DLV options. By disabling the magic set optimization, DLV could answer each query in a few seconds. On $p\_1/2$ with 1 answer, we could not get an answer after several hours even with magic sets disabled. By recursing *p_1/2* 100000 times rather than 1000000 times, DLV terminated in 20-30 minutes for each query [22].

While XSB performs better on the overhead tests, this is not surprising as the main purpose of these benchmarks was to test tradeoffs within the XSB engine. The results on P_{krr} are potentially more significant, and were repeated for DLV on the same machine and with the same methodology as for XSB, using both the left- and the right-recursive forms of *equals/2*. For the default DLV options, the best total time and maximum memory ranged from over 100 seconds and 12 GB for 3.7M / 10 queries to over 600 seconds and 52 GB for 14.8M / 20 queries. DLV performs better than on the $p\_1/2$ experiments, possibly because the bottom-up method of intelligent instantiation works better for queries where there are relatively few answers as in P_{krr}, as opposed to those where there are many answers as with P_{over}. In general while XSB performed well in our experiments, the results should not be regarded as conclusive, as the DLV benchmarks were not conducted by members of the DLV team. However the experiments

[22] While the XSB timings of Table I do not include the loading time, this is usually of the order of 1-2 seconds.

do show that both systems are robust and scalable enough to handle complex queries involving functions over large EDBs.

6. DISCUSSION

In this paper, we have examined the class of programs with canonical finite models and shown that it coincides with the class of strongly bounded term-size programs (Corollary 3.2), whose definition is adapted from a well-known iterated fixed point definition of WFS [Przymusinski 1989]. Strongly bounded term-size programs, in their turn, strictly include normal finitely ground programs (Theorem 3.4), a class motivated by termination properties of grounders. Queries to strongly bounded term-size programs terminate correctly under SLG_{SA} using a finitary abstraction operation (Theorems 4.15 and 4.16). Further, SLG_{SA} has optimal complexity when using depth-k abstractions (Theorem 4.21), and may produce a smaller program than other grounders (Theorem 4.18). Finally, subgoal abstraction has been implemented at the engine level of XSB with good performance results in terms of overhead, query optimization, scalability, and comparison with other systems. Because the code changes are local within a tabling engine, subgoal abstraction should be implementable without undue effort by other tabled Prologs, at least for definite programs.

After presenting a brief overview of termination analysis for logic programming and related disciplines, we discuss the applicability of these results in terms of recent and current work.

Related Work in Termination Analysis and its Applicability. Although a full survey of related work on termination analysis is beyond the scope of this paper, we briefly discuss recent work in termination analysis of Prolog programs, sets of database constraints evaluated using chase algorithms, and logic programs under the stable model semantics.

There has been a large body of work analyzing termination conditions in logic programs that use SLD resolution. [De Schreye and Decorte 1994] presents a survery of early work, while more recent work includes [Decorte et al. 1999; Serebrenik and De Schreye 2004; Codish et al. 2005; Serebrenik and De Schreye 2005; Nguyen and De Schreye 2005; Bruynooghe et al. 2007; Nguyen et al. 2007; Nishida and Vidal 2010; Schneider-Kamp et al. 2010; Voets and De Schreye 2011]. To the extent that they study pure logic programs, the approaches developed in these works are directly applicable to showing termination of queries under SLG_{SA} resolution. However the properties of SLD resolution, which does not terminate even for function-free definite programs, means that these techniques are sometimes more restrictive than needed for tabled resolution. More closely related is [Verbaeten et al. 2001] which considers termination for evaluations that mix SLD resolution with tabling, a different focus than in this paper. Overall, work in termination for Prolog and its extensions provides valuable tools for developing static analyses of when normal logic programs have canonical well-founded models.

Another related topic is that of termination for the chase algorithm as used in databases [Fagin et al. 2005; Deutsch et al. 2008; Marnette 2009; Meier et al. 2009; Greco and Spezzano 2010; Greco et al. 2011]. While relevant to the evaluation of canonical well-founded models, there are important differences. First, chase algorithms consider sets of database constraints that may have non-Herbrand models in which null values may be equated to constants; in that sense their semantics is more general than that of normal logic programs. However, these database constraints do not admit the general use of function symbols that is possible in normal logic programs. In addition, the papers cited above do not consider non-stratified negation. As with the termination analysis of Prolog programs, work on chase termiation may prove relevant to decidable

methods for determining when normal logic programs have canonical well-founded models.

The literature on whether a given program has a set of finite set of stable models is the most directly applicable to the results in this paper, and in fact proved the starting point for several of the results presented here. While strongly bounded term-size programs are semi-decidable (as are finitely ground programs), the various decidable subclasses of finitely ground programs that have been identified in the literature can be used for static analysis of termination of strongly bounded term-size programs as well (e.g., [Syrjanen 2001; Gebser et al. 2007; Lierler and Lifshitz 2009; Alviano et al. 2010; Greco et al. 2012; Greco et al. 2013]). The results shown in this paper indicate that these classes, restricted when necessary from disjunctive to normal programs, can be used for analysis of tabling systems with subgoal abstraction, in addition to analyzing ASP systems. Accordingly, the query-orientation of SLG_{SA} makes it comparable to evaluation techniques such as presented in [Baselice and Bonatti 2010; Calimeri et al. 2011] on finitely recursive programs for which given queries may be strongly bounded term-size, but that may not have models representable as finite sets of ground atoms[23]. Indeed, syntactic restrictions of finitely recursive programs, such as [Eiter and Šimkus 2009] will ensure ground query termination under SLG_{SA} for WFS.

To summarize, extending the termination analysis literature mentioned above is an important topic for tabling. As discussed in Section 2.3.2, adding a predicate such as *member/2* prevents a program from having a canonical finite model, but if a subgoal to *member/2* has the second argument ground, it will terminate even if the first argument is non-ground. Analysis methods that extend finitely ground programs and their static subclasses with (not necessarily ground) queries would be able to prove termination for many practical tabling programs.

Dynamic Detection of Non-Termination. In addition to analysis of termination, some very recent work addresses detection and prevention of non-termination, based on SLG_{SA}. Liang and Kifer [2013] discuss *Terminyzer*, a tool for detection of non-termination in Flora-2 programs [Yang et al. 2013]. These Flora-2 programs are implemented using XSB, and the algorithms of Terminyzer were written to exploit SLG_{SA}. Specifically, Terminyzer analyzes a program trace, available in XSB, of an interrupted computation that is suspected to be non-terminating. By performing call-sequence analysis, Terminyzer can pinpoint the sequence either of subgoals or of Flora-2 rules leading to non-terminating behavior. By performing answer flow analysis, Terminyzer can pinpoint those subgoals within a mutually recursive component that are causing non-termination[24]. Terminyzer has been used commercially in the knowledge representation and reasoning systems Silk and Fidji (cf. Section 5.3).

An alternate approach to detecting termination under SLG_{SA} queries is that of *radial restraint* [Grosof and Swift 2013]. Radial restraint performs abstraction of answers when their norm is greater than a given bound, specified on a per-predicate basis. When answers are abstracted, their truth value is set to *unknown* so that computations using SLG_{SA} always terminate soundly, but in doing so may sacrifice completeness. Once an evaluation has terminated, users may examine which answers were subject to restraint (abstraction), and investigate the dependencies of the subgoals giving rise to the restrained answers.

[23]In [Riguzzi and Swift 2013] it was shown that finitely recursive and bounded term-size programs are incompatible, but finitely recursive programs are a proper subclass of those programs for which all ground atomic queries are bounded term-size.

[24]Trace-based analysis was chosen for Terminyzer in part because Flora-2 programs are substantially different than e.g,, normal logic programs, and support features such as Hilog, frame-style inheritance and defeasibility theories that substantially complicate static analysis.

Implementation of Grounders. As mentioned above, the results of this paper indicate that tabling with subgoal abstraction may be useful for grounding ASP or other systems. On-going work on the IDP3 system uses tabling in XSB as part of a grounding pass, after which a program is sent to a model generator [Jansen et al. 2013]. This recent work does not make use of subgoal abstraction, although it can directly benefit from it. Even so, results indicate that the time for grounding using tabling in XSB is comparable to that used in Clingo or DLV, although at present slightly slower.

Current Implementational Work. Current work is focussed on enhancing the utility of subgoal abstraction within the Fidji system, mentioned above. Note that there are cases where subgoal abstraction should *not* be used, as it can undo bindings needed for a bounded term-size query to terminate (e.g., making the second argument of *member/2* non-ground). Current experiments using XSB involve an interrupt mechanism for long-running computations in which an interrupt handler inspects table structures or a log to determine whether or not subgoal abstraction, restraint, or other tabling features should be used on a given predicate. Since many tabling attributes are adjustable dynamically, this approach allows termination behavior for tabled computation to be dynamically adaptive based on self-inspection.

Acknowledgements

This work was partially supported by FCT Project ERRO PTDC/ EIACCO/121823/2010. The authors would like to thank the anonymous reviewers and Thomas Ströder for their comments, and Francesco Calimeri for his advice in optimizing DLV performance.

REFERENCES

J.J. Alferes, M. Knorr, and T. Swift. 2013. Query-driven Procedures for Hybrid MKNF Knowledge Bases. *ACM Transactions on Computational Logic* 14, 2 (2013).

M. Alviano, W. Faber, and N. Leone. 2010. Disjunctive ASP with Functions: Decidable Queries and Effective Computation. *Theory and Practicce of Logic Programming* 10, 4-6 (2010), 497–512.

S. Baselice and P. Bonatti. 2010. A decidable subclass of finitary programs. *Theory and Practicce of Logic Programming* 10, 4-6 (2010), 481–496.

S. Baselice, P. Bonatti, and G. Criscuolo. 2009. On finitely recursive programs. *Theory and Practice of Logic Programming* 9, 2 (2009), 213–238.

M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof. 2007. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *ACM Transactions on Programming Languages and Systems* 29, 2 (2007).

F. Calimeri, S. Cozza, G. Ianni, and N. Leone. 2008. Computable Functions in ASP: Theory and Implementation. In *International Conference on Logic Programming (LNCS)*, Vol. 5366. Springer, 407–424.

Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. 2011. Finitely recursive programs: Decidability and bottom-up computation. *AI Commun.* 24, 4 (2011), 311–334.

Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. 2012. The third open Answer Set Programming competition. *CoRR* abs/1206.3111 (2012).

W. Chen and D. S. Warren. 1996. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the Association for Computing Machinery* 43, 1 (1996), 20–74.

M. Codish, V. Lagoon, and P. Stuckey. 2005. Testing for Termination with Monotonicity Constraints. In *International Conference on Logic Programming*. 326–340.

D. De Schreye and S. Decorte. 1994. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming* 19 (1994), 199–260.

S. Decorte, D. De Schreye, and H. Vandecasteele. 1999. Constraint-based Termination Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems* 21 (1999), 1137–1195.

A. Deutsch, A. Nash, and J. Remmel. 2008. The Chase Revisited. In *ACM Conference on Principles of Database Systems*.

T. Eiter and M. Šimkus. 2009. FDNC: Decidable Nonmontonic Disjunctive Logic Progrmas with Function Symbols. *ACM Transactions on Computational Logic* 9, 9 (2009), 1–45.

R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* 336, 1 (2005), 89–124.

M. Gebser, T. Schaub, and S. Thiele. 2007. GrinGo: A New Grounder for Answer Set Programs. In *Logic Programming and Non-Monotonic Reasoning*. 267–280.

M. Gelfond and V. Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *International Conference and Symposium on Logic Programming*. 1070–1080.

S. Greco, C. Molinaro, and I. Trubitsyna. 2013. Bounded Programs: A New Decidable Class of Logic Programs with Function Symbols. In *Internation Joint Conference on Artificial Intelligence*. 926–932.

S. Greco and F. Spezzano. 2010. Chase Termination: A Constraints Rewriting Approach. In *Conference on Very Large Databases*. 93–104.

S. Greco, F. Spezzano, and I. Trubitsyna. 2011. Stratification Criteria and Rewriting Techniques for Checking Chase Termination. In *Conference on Very Large Databases*. 1158–1168.

S. Greco, F. Spezzano, and I. Trubitsyna. 2012. On the termination of logic programs with function symbols. In *International Conference on Logic Programming*. 323–333. Technical Communications.

B. Grosof, M. Dean, and M. Kifer. 2012. Semantic Web Rules: Fundamentals, Applications, and Standards. (2012). Tutorial, 11th International Semantic Web Conference.

B. Grosof and T. Swift. 2013. Radial Restraint: A Semantically Clean Approach to Bounded Rationality for Logic Programs. In *Conference of the American Association of Artificial Intelligence*.

J. Jansen, A. Jorissen, and G. Janssens. 2013. Compiling Input* FO(·) Inductive Definitions into Tabled Prolog Rules for IDP3. *Theory and Practicce of Logic Programming* 13, 4-5 (2013), 601–704.

N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, K. Koch, S. Perri, and A. Polleres. 2002. The DLV system. In *JELIA*. 537–540.

S. Liang and M. Kifer. 2013. A Practical Analysis of Non-Termination in Large Logic Programs. *Theory and Practicce of Logic Programming* 13, 4-5 (2013), 705–719.

Y. Lierler and V. Lifshitz. 2009. One more decidable class of finitely ground progrmas. In *International Conference on Logic Programming*.

J. W. Lloyd. 1987. *Foundations of Logic Programming* (2nd extended ed.). Springer-Verlag.

B. Marnette. 2009. Generalized schema-mappings: from termination to tractability. In *ACM Conference on Principles of Database Systems*. 13–22.

M. Meier, M. Schmidt, and G. Lausen. 2009. On Chase Termination Beyond Stratification. In *Conference on Very Large Databases*. 970–981.

M. Nguyen and D De Schreye. 2005. Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. In *International Conference on Logic Programming*. 311–325.

M. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. 2007. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR*. 8–22.

N. Nishida and G. Vidal. 2010. Termination of narrowing via termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing* 21, 3 (2010), 177–225.

OpenRuleBench. 2009-2011. OpenRuleBench: Benchmarks for Semantic Web Rule Engines. (2009-2011). \sfrulebench.projects.semwebcentral.org

D. Pedreschi, S. Ruggieri, and J.G. Smaus. 2002. Classes of Terminating Logic Programs. *Theory and Practicce of Logic Programming* 2, 3 (2002), 369–418.

T. Przymusinski. 1989. Every Logic Program has a Natural Stratification and an Iterated Least Fixed Point Model. In *ACM Symposium on Principles of Database Systems*. 11–21.

I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. 1999. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming* 38, 1 (1999), 31–55.

F. Riguzzi and T. Swift. 2013. Well-Definedness and Efficient Inference for Probabilistic Logic Programming under the Distribution Semantics. *Theory and Practicce of Logic Programming* 13, 2 (2013), 279–302.

K. Sagonas and T. Swift. 1998. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems* 20, 3 (May 1998), 586 – 635.

P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. 2010. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic* (2010).

A. Serebrenik and D. De Schreye. 2004. Inference of Termination Conditions for Numerical Loops in Prolog. *Theory and Practicce of Logic Programming* 4, 5-6 (2004), 719–751.

A. Serebrenik and D. De Schreye. 2005. Termination of Floating Point Computations. *Journal of Automated Reasoning* 34, 2 (2005), 141–177.

T. Swift. 1999. A New Formulation of Tabled Resolution with Delay. In *Recent Advances in Artifiial Intelligence (LNAI)*, Vol. 1695. Springer, 163–177.

T. Swift and D.S. Warren. 2012. XSB: Extending the Power of Prolog using Tabling. *Theory and Practicce of Logic Programming* 12, 1-2 (2012), 157–187.

T. Syrjanen. 2001. Omega-Restricted Logic Programs. In *Logic Programming and Non-Monotonic Reasoning*. 267–280.

H. Tamaki and T. Sato. 1986. OLDT Resolution with Tabulation. In *International Conference on Logic Programming (LNCS)*, Vol. 225. Springer, 84–98.

A. van Gelder. 1989. The Alternating Fixpoint of Logic Programs with Negation. In *ACM Symposium on Principles of Database Systems*. 1–10.

A. van Gelder, K. A. Ross, and J. S. Schlipf. 1991. The Well-founded Semantics for General Logic Programs. *Journal of the Association for Computing Machinery* 38, 3 (1991), 620–650.

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

D. Voets and D. De Schreye. 2011. Non-termination analysis of logic programs with integer arithmetics. *Theory and Practicce of Logic Programming* 11, 4-5 (2011), 521–536.

G. Yang, M. Kifer, H. Wan, and C. Zhao. 2013. *FLORA-2: User's Manual Version 0.99.3*. http://flora.sourceforge.net.

A. LONGER PROOFS

A.1. Proofs for Results of Section 4.4 (Correctness and Termination of SLG$_{SA}$)

THEOREM 4.15 *Let $\mathcal{E}$ be an SLG$_{SA}$ evaluation of a query Q to a safe program P with final forest $\mathcal{F}_{fin}$. Then $\mathcal{I}_{\mathcal{F}_{fin}} = WFM^P|_{\mathcal{F}_{fin}}$.*

PROOF. Note that the only difference between SLG$_{SA}$ and the version of SLG from Section 4.2 is that in SLG$_{SA}$ the root subgoals of some trees in a forest $\mathcal{F} \in \mathcal{E}$ may have been abstracted. We consider the two cases in which an abstraction operation may be used and show that the action of the abstraction function is the same as a simple rewriting of a rule that preserves logical equivalence, but can be evaluated by SLG without an abstraction function. The theorem then holds by the correctness of SLG (cf. [Chen and Warren 1996], and [Swift 1999] for the forest of trees model).

— *Positive selected literals*. Let $N = (G \leftarrow Delays|Goals)\theta$ be a node with selected positive literal S. Note that N must have an ancestor in the tree for G that was created by a program clause

$$r = H \leftarrow A_1, \ldots, A, \ldots, L_n$$

and assume that S corresponds to the selected literal A so that $S = A\theta$. Suppose that a NEW SUBGOAL operation creates a tree with root subgoal $A' = abs(A\theta)$. We have that $A\theta = A'\eta$ for some η, as A' subsumes $A\theta$.
Let

$$r' = (H \leftarrow A_1, \ldots, abs(A, A'), A', A' = A, \ldots, L_n)$$

The predicate $abs/2$ simply sets $A' = abs(A)$ (its implementation is outside of the semantics of the SLG evaluation $\mathcal{E}$). A' is then called, and if it succeeds, A' is unified with A. Note that r' is logically equivalent to r: A' subsumes A so that every solution to A will also be a solution to A', and the unification $A' = A$ in r' ensures that only those solutions that also unify with A will succeed.

— *Negative selected literals*. The argument is essentially the same, but the transformation is instead:

$$r' = (H \leftarrow A_1, \ldots, abs(A, A'), not\, exists\_ans(A', A), \ldots, L_n)$$

where

$$exists\_ans(A', A) \leftarrow A', A = A'.$$

By assumption, P is safe, so that A' is ground when the literal $A' = A$ is called. Note that $not\, exists\_ans(A', A)$ succeeds iff there is no (ground) answer to the goal A' that unifies with the ground A. As above, the program r' is logically equivalent to r.

The program P' is constructed from P by the transformations of the various rules (sometimes replacing a rule by a series of transformations), and adding $abs/2$ and *exists_ans/2*. Note that the transformation is completely local to a rule, so that replacing all rules in P by forms equivalent to r' makes P' a conservative extension of P. □

THEOREM 4.16 *Let Q be a query to a strongly bounded term-size program P. Then any SLG$_{SA}$ evaluation $\mathcal{E}$ of Q that uses a finitary abstraction operation reaches a final forest $\mathcal{F}_{fin}$ after a finite number of steps. If P is safe, then $\mathcal{F}_{fin}$ will not be floundered.*

PROOF. We first prove the statement that any SLG evaluation $\mathcal{E}$ of Q that uses a finitary abstraction operation reaches a final forest $\mathcal{F}_{fin}$ after a finite number of steps. The proof is given by induction on the maximal dynamic stratum of any answer in $\mathcal{E}$.

— For the base case, assume the maximal stratum is 1 (Definition 2.2). Because Q is in stratum 1, the only applicable SLG operations in $\mathcal{E}$ are NEW SUBGOAL, PROGRAM

CLAUSE RESOLUTION, POSITIVE RETURN, and COMPLETION. Note that each of these operations is applicable only once to a given node or set of subgoals.

— NEW SUBGOAL: The use of a finitary abstraction operation means that there may only be a finite number of NEW SUBGOAL operations in $\mathcal{E}$, and hence a finite number of trees in any forest of $\mathcal{E}$.
— PROGRAM CLAUSE RESOLUTION: Since there are a finite number of trees, and a finite number of program clauses resolvable against the root subgoal of any tree, $\mathcal{E}$ contains only a finite number of PROGRAM CLAUSE RESOLUTION operations, and the root of any tree has only a finite number of immediate children.
— POSITIVE RETURN: Next, since P is strongly bounded term-size, $true(WFM^P)$ is finite, and since the maximal stratum of $\mathcal{E}$ is 1, any answer returned will be unconditional. Accordingly there are a finite number of answers that can be resolved against any selected subgoal. Because interior nodes of SLG trees can only be extended by POSITIVE RETURN operations, any non-root node in any tree may have only a finite number of children. In addition, the depth of any tree in $\mathcal{E}$ is bounded by the maximal number of body literals in any rule in P, which is finite. Thus the subtrees of any tree in $\mathcal{E}$ have a finite depth and a finite branching factor, and so are finite. There can thus be only a finite number of POSITIVE RETURN operations.
— COMPLETION: Finally, since there are a finite number of trees in any forest, there can be only a finite number of COMPLETION operations.

Since the number of occurrences of each type of operation in $\mathcal{E}$ is finite, $\mathcal{E}$ itself must be finite.

— For the inductive case, assume that the statement is true for all atoms whose (finite) stratum is less than n in order to prove it true for those atoms whose stratum is n.
— NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION and COMPLETION are argued in the same manner as for the base case.
— POSITIVE RETURN: Because P is strongly bounded term-size there are only a finite number of undefined atoms. Since each literal in the $Delays$ of a node must come from delaying or SLG resolution of a literal in the body of a rule, and the maximum number of literals in the body of a rule is finite, there are only a finite number of *conditional* answers. The statement that there are only a finite number of POSITIVE RETURN operations follows as for the base case.
— NEGATIVE RETURN: First, consider that a NEGATIVE RETURN operation can be applied at most once to any node N. As a result of this operation, any node N to which a NEGATIVE RETURN operation is applied can have only a single child: either a failure node in the case of NEGATION FAILURE, or a single child with the selected literal removed from the $Goals$ of N in the case of NEGATION SUCCESS. In the case of NEGATION FAILURE this is enough to show that the finiteness of $\mathcal{E}$ is not affected, as a failure node cannot be further expanded. In the case of NEGATION SUCCESS, the fact that the selected literal is removed from $Goals$, means that the child of N will have a smaller $Goals$ sequence. Since $Goals$ is finite, any path from N may have only a finite number of NEGATIVE RETURN operations.
— DELAYING: Considerations analogous to the NEGATION SUCCESS case show that DELAYING will not affect the finiteness of $\mathcal{E}$.
— ANSWER COMPLETION: Next, note that ANSWER COMPLETION will produce a single failure node as a child of each answer node to which it is applied, and a failure node cannot be further expanded. So ANSWER COMPLETION does not affect the finiteness of any forest, no matter how many times it is applied.
— In the case of SIMPLIFICATION, an application of the SIMPLIFICATION operation that produces a failure node does not affect the finiteness of any forest (Defini-

tion 4.6, 6a) as a failure node cannot be further expanded.. On the other hand, if an application of a SIMPLIFICATION operation to a node N produces a non-failure child (Definition 4.6, 6b), note that similar to the case of NEGATION SUCCESS, the child of N will have a smaller $Delays$. Since $Delays$ is finite, any path from N to its descendents may have only a finite number of SIMPLIFICATION operations.

Since each operation can be applied only a finite number of times, $\mathcal{E}$ must be finite.

Note that the proof of the previous statement shows that $\mathcal{E}$ is finite, but $\mathcal{F}_{fin}$ may be floundered: i.e., a node in $\mathcal{F}_{fin}$ may have a selected non-ground negative literal. We next show that if P is safe, then $\mathcal{F}_{fin}$ will not be floundered. It is straightforward to show that if P is safe, any answer in any forest in $\mathcal{E}$ will be ground. Then, let r be a rule in a *safe* program P, and L_j a given negative literal in r. Because of the safety of P, the action of PROGRAM CLAUSE RESOLUTION and POSITIVE RETURN operations on previously selected subgoals to the left of L_j in r will ensure that L_j is ground by the time it becomes a selected subgoal. □

A.2. Proofs for Results of Section 4.5 (Comparison with Intelligent Instantiation)

THEOREM 4.18

Let P be a safe, finitely ground program P. Let $\mathcal{E}$ be a SLG_{SA} evaluation of a grounding predicate of P whose final forest is $\mathcal{F}_{fin}$, and $P_{tabled} = answers(\mathcal{F}_{fin})$.

(1) P_{tabled} is equal to $\frac{ground(P)}{WFM^P}$.
(2) Let P_{ii} be the intelligent instantiation of P. Then $P_{tabled} \geq_{red} P_{ii}$.

PROOF.

(1) (Sketch) We consider first the case of SLG (i.e., where no abstraction operation is used). By the correctness of SLG, the use of the grounding predicate ensures that $\mathcal{I}_{\mathcal{F}_{fin}} = WFM^P$. The safety of P ensures that all answers are ground, regardless of whether they are conditional. In such a case, the correctness of SLG with respect to the stable model semantics (cf. [Chen and Warren 1996]), ensures that if there is rule r with a non-empty body in $\frac{ground(P)}{WFM^P}$, then r must be a conditional answer in $\mathcal{F}_{fin}$.

    For SLG_{SA}, Theorem 4.15 ensures that P_{tabled} is also equal to $\frac{ground(P)}{WFM^P}$. Note that the final forest of an SLG and an SLG_{SA} evaluation will both have the same trees for subgoals immediatedly called by the grounding predicate (which we call here the grounding subgoals), and the answers in each of these trees will be the same. For other trees, suppose that the SLG_{SA} $\mathcal{E}_{\text{SA}}$ evaluation differed from the SLG evaluation $\mathcal{E}$ in that $\mathcal{E}_{\text{SA}}$ created a tree for an abstracted subgoal, while $\mathcal{E}$ did not. Both of these subgoals are more specific than the grounding subgoals, and the (ground) answers of both are contained in the set of ground answers for the grounding subgoals.
(2) By the definition of intelligent instantiation, the facts in P_{ii} are a subset of $true(WFM^P)$, while $\mathcal{B}_P$ minus the set of all head atoms in P_{ii} is a subset of $false(WFM^P)$. The statement then holds by part 1 of this theorem.

□

A.3. Proofs for Complexity Results of Section 4.6

LEMMA 4.20. *Let P be a ground program, Q a ground query, and $\mathcal{E}$ a terminating SLG_{SA} evaluation of Q against P that uses depth-k abstraction. Then the number of nodes in the final forest $\mathcal{F}_{fin}$ is at most $\mathcal{O}(size(P_Q(\mathcal{E})))$.*

PROOF. First note that since no step of an SLG_{SA} evaluation ever removes a node or tree from a forest, showing the upper bound for $\mathcal{F}_{fin}$ carries over to all forests in $\mathcal{E}$. We consider first the special case in which Q (finitely) terminates using the identity function as an abstraction function (i.e., depth-ω abstraction), before considering the case of general abstraction functions.

(1) $abs(\cdot)$ *is the identity function*. Within this case, we consider first the subcase where P is definite, and then consider the case where P is any normal program.
   — *P is definite*. First, since P is ground, no two trees in any forest of $\mathcal{E}$ may have root subgoals that unify. Because of this fact, each rule in P_Q may be used for resolution in at most one tree in $\mathcal{F}_{fin}$. Continuing, consider a rule r of P_Q that creates a node n_r in some tree in $\mathcal{F}_{fin}$. The number of descendants of n_r is at most the number of literals in the body of r. To see this, first note that the selected literal of any node is ground and thus may have at most once descendant. Next, since each POSITIVE RETURN answer operation removes a body literal from the $Goals$ of a node when creating a new node, the number of descendants of n is limited to the number of body literals in n_r. Thus there are at most $size(P_Q)$ non-root nodes in $\mathcal{F}_{fin}$ and $\mathcal{O}(size(P_Q(\mathcal{E})))$ nodes overall.
   — *P is a normal program*. If P is not definite, then a node n_r as above may have at most $2 \times size(r)$ descendents, as each body literal first may be delayed and then either simplified or failed via an ANSWER COMPLETION operation. So in this case there are still $\mathcal{O}(size(P_Q(\mathcal{E})))$ nodes overall.

(2) $abs(\cdot)$ *is a depth-k abstraction function for positive integer k*. To show this result,we first show a subsidiary statement.
   Consider two subgoals $abs(S_1)$ and $abs(S_2)$ at the root of two distinct trees in $\mathcal{F}_{fin}$ generated from the application of NEW SUBGOAL to S_1 and S_2. We show that $abs(S_1)$ cannot unify with $abs(S_2)$. Recall that $abs(S_1)$ and $abs(S_2)$ cannot be identical (up to variance), since they would not correspond to distinct trees in that case. There are three cases to consider.
   (a) $abs(S_1) = S_1$ and $abs(S_2) = S_2$. Since P is ground, this means that S_1 and S_2 are ground, so $abs(S_1)$ does not unify with $abs(S_2)$.
   (b) $abs(S_1) = S_1$ but $abs(S_2) \neq S_2$. In this case, there must be a position π in S_2 in which a constant or function symbol was replaced by a position variable. However, S_1 is ground since P is ground, but S_1 does not have a position of depth k as $abs(S_1) = S_1$. Accordingly there must be some position π' that is a constant in $abs(S_1)$, but is a non-constant function symbol in $abs(S_2)$, so that $abs(S_1)$ and $abs(S_2)$ do not unify.
   (c) $abs(S_1) \neq S_1$ and $abs(S_2) \neq S_2$. In this case, one of the following situations arises:
      i. There is some position π in one of the subgoals, say $abs(S_1)$, that is a constant or function symbol, while $abs(S_2)$ has a position variable at π. For this subcase the argument is identical to case 2(b) above.
      ii. Otherwise assume that $abs(S_1)$ and $abs(S_2)$ contain position variables in exactly the same positions. Recall that $abs(S_1)$ and $abs(S_2)$ cannot be variants of each other by the definition of the NEW SUBGOAL operation. So there must be some position π in which $abs(S_1)$ differs from $abs(S_2)$ and neither $abs(S_1)|_\pi$ nor $abs(S_2)|_\pi$ is a position variable. Since P is ground, $abs(S_1)|_\pi$ cannot unify with $abs(S_2)|_\pi$

   Since no two subgoals for trees in any forest $\mathcal{F}$ of $\mathcal{E}$ may unify and since P is ground, each program clause of P_Q can appear in at most one tree in $\mathcal{F}$. Given this fact the argument of case 1 can be applied when depth-k abstraction is performed for some integer $k > 0$.

□

THEOREM 4.21. *Let P be a ground program, Q a ground query, and $\mathcal{E}$ a terminating SLG$_{\mathsf{SA}}$ evaluation of Q against P that uses depth-k abstraction, and with final forest $\mathcal{F}_{fin}$. Then under the cost model $\mathcal{C}_{function}$, the cost of $\mathcal{E}$ is $\mathcal{O}(|atoms(\mathcal{F}_{fin})| \times size(P_Q(\mathcal{E})))$.*

PROOF. From Lemma 4.20 there are $\mathcal{O}(size(P_Q(\mathcal{E})))$ nodes in $\mathcal{F}_{fin}$. First note that each SLG$_{\mathsf{SA}}$ operation either produces a distinct node or nodes in $\mathcal{F}_{fin}$ or marks a set of trees in $\mathcal{F}_{fin}$ as complete. Thus, the cost of $\mathcal{E}$ can be broken down by analyzing the costs of creating distinct set of nodes.

— NEW SUBGOAL, POSITIVE RETURN, NEGATIVE RETURN, DELAYING, SIMPLIFICATION. Since each of these operations is constant-time in $\mathcal{C}_{function}$, they produce a node in $\mathcal{F}_{fin}$ and there can be at most $\mathcal{O}(size(P_Q(\mathcal{E})))$ such operations, the combined cost of these operations is $\mathcal{O}(size(P_Q(\mathcal{E})))$.
— PROGRAM CLAUSE RESOLUTION. The cost of the PROGRAM CLAUSE RESOLUTION operation is non-constant, but is proportional to the size of the rule that it applies to a subgoal. However, each rule in $P_Q(\mathcal{E})$ can only be applied once. This is because no two trees in $\mathcal{F}_{fin}$ have roots that unify with each other, due to the fact that P is ground and a depth-k abstraction function is used (cf. the proof of Lemma 4.20). Because each rule can only be applied once, the combined cost of the PROGRAM CLAUSE RESOLUTION operations is at most $size(P_Q(\mathcal{E}))$.
— COMPLETION. The cost of the COMPLETION operation is non-constant; rather, its cost is the number of trees it marks as complete. Since each tree is marked as completed only once, the combined cost of the COMPLETION operations is at most $|atoms(\mathcal{F}_{fin})|$.
— ANSWER COMPLETION. The cost of the ANSWER COMPLETION operation is again non-constant. The cost of an ANSWER COMPLETION operation is at most $size(P_Q(\mathcal{E}))$, however each ANSWER COMPLETION operation must produce failure nodes for all answers whose head is some unsupported atom A. Afterwards, A will no longer be subject to an ANSWER COMPLETION operation. Since P is ground, A corresponds to an element of $atoms(\mathcal{F}_{fin})$. Thus there are thus at most $|atoms(\mathcal{F}_{fin})|$ ANSWER COMPLETION operations.

The total worst-case cost of $\mathcal{E}$ is thus the cost of the constant time operations $\mathcal{O}(size(P_Q(\mathcal{E})))$, plus the cost of the PROGRAM CLAUSE RESOLUTION operations $\mathcal{O}(size(P_Q(\mathcal{E})))$, plus the cost of the COMPLETION operations $\mathcal{O}(|atoms(\mathcal{F}_{fin})|)$, plus the cost of the ANSWER COMPLETION operations $\mathcal{O}(|atoms(\mathcal{F}_{fin})| \times size(P_Q(\mathcal{E})))$ so that the total cost of $\mathcal{E}$ is $\mathcal{O}(|atoms(\mathcal{F}_{fin})| \times size(P_Q(\mathcal{E})))$. □