ELSEVIER

The Journal of Logic Programming 45 (2000) 1–41

THE JOURNAL OF LOGIC PROGRAMMING

www.elsevier.com/locate/jlpr

An abstract machine for efficiently computing queries to well-founded models☆

Konstantinos Sagonas[a,*], Terrance Swift[b], David S. Warren[c]

[a] *Computing Science Department, Uppsala University, Box 311, S-751 05 Uppsala, Sweden*
[b] *Department of Computer Science, University of Maryland, College Park, MD 20742, USA*
[c] *Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, USA*

Received 15 October 1998; received in revised form 15 November 1999; accepted 8 February 2000

Abstract

The well-founded semantics has gained wide acceptance partly because it is a *skeptical* semantics. That is, the well-founded model posits as unknown atoms which are deemed true or false in other formalisms such as stable models. This skepticism makes the well-founded model not only useful in itself, but also suitable as a basis for other forms of non-monotonic reasoning. For instance, since algorithms to compute stable models are intractable, the atoms relevant to such algorithms can be limited to those undefined in the well-founded model. Thus, an engine that efficiently evaluates programs according to the well-founded semantics can be seen as a prerequisite to practical systems for non-monotonic reasoning. This paper describes the architecture of the Warren Abstract Machine (WAM)-based abstract machine underlying the XSB system. This abstract machine, called the SLG-WAM, uses tabling to efficiently compute the well-founded semantics of non-ground normal logic programs in a goal-directed way. To do so, the SLG-WAM requires sophisticated extensions to its core tabling engine for fixed-order stratified programs. A mechanism must be implemented to represent answers that are neither true nor false, and the delay and simplification operations – which serve to break and to resolve cycles through negation, must be implemented. We describe fully these extensions to our tabling engine, and demonstrate the efficiency of our implementation in two ways. First, we present a theorem that bounds the need for delay to those literals which are not dynamically stratified for a fixed-order computation. Second, we present performance results that indicate that the overhead of delay and simplification to Prolog – or tabled – evaluations is minimal.

Keywords: Tabling; WAM; SLG-WAM; SLG resolution; Well-founded semantics; Deductive databases; Non-monotonic reasoning; XSB; Implementation

☆ A short, preliminary version of this paper appeared in *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming* under the title "An Abstract Machine for Computing the Well-Founded Semantics", pp. 274–288.

* Corresponding author. Tel.: +46-18-4711056; fax: +46-18-511925.
E-mail addresses: kostis@csd.uu.se (K. Sagonas), tswift@cs.umd.edu (T. Swift), warren@cs.sunysb.edu (D.S. Warren).

0743-1066/00/$ - see front matter
PII: S0743-1066(00)00005-4

1. Introduction

The past decade of logic programming research has provided steady advances in the power of evaluation methods and in their implementation. Certainly the most popular resolution method to date is SLDNF and the most popular implementation method is based on WAM-style Prolog engines. As proven by its widespread acceptance, the SLDNF/Prolog paradigm is extremely powerful for programming. However this evaluation method suffers from serious drawbacks that have prevented its extension into other areas that benefit from a higher level of declaritivity than Prolog can offer. A strong claim can be made that these drawbacks stem from the fact that SLDNF (and its semantics as represented by Clark's Program Completion) does not adequately address problems related to loops in SLD(NF) trees. [1] Indeed, major areas of logic programming research can be viewed as efforts to formalize or implement evaluation methods that handle such loops.

Handling positive loops has been addressed by many methods, with magic sets/templates [4,26] and resolution based on *tabling* such as OLDT [33], SLD-AL [36], or SLG [7] constituting the two main approaches. Although formulated differently, these approaches turn out to treat positive loops in essentially the same way. Both assign failing values to derivation paths that contain infinite positive recursion. As for negative loops, perhaps the critical insight behind the well-founded semantics is to combine within a three-valued framework the foregoing method for positive loops with the assignment of the value *undefined* to derivations containing loops through negation [34]. It is natural, then, to extend the above evaluation methods to handle negative loops and thereby execute the well-founded semantics, as several researchers have noticed [5,7,31].

One such evaluation method, *SLG resolution* (Linear resolution with Selection function for General logic programs) uses tabling with delaying to resolve loops through negation [7]. SLG has an efficient WAM-style implementation in the SLG-WAM which forms the basis of the freely available XSB system [27,29].

The architecture of the SLG-WAM for fixed-order stratified programs is described in Refs. [16,28]; here, we consider the extensions to this engine needed to efficiently execute *all* non-floundering normal programs according to the well-founded semantics. The robustness and scalability of these extensions have led to concrete uses for tabled programming with non-stratified negation. For instance, non-stratified programs arise when logic programs are used to specify the temporal logics of verification of concurrent systems, and when logic programs are used for abductive diagnosis. These applications are overviewed in Sections 1.1 and 7.3, respectively.

This paper provides a detailed description of how queries to normal programs under the well-founded semantics can be computed at the speed of compiled Prolog, and tightly integrated with Prolog-style execution. Further, we discuss how the SLG-WAM realizes and makes use of a bound on the non-determinism of the computation rule required by normal programs. Specifically:

[1] An SLD tree has a *loop* if it contains an infinite branch B such that for i, l, $start > 0$, whenever $i > start$, then the ith node on B from the root and the $i + l$th node on B from the root, taken as terms, are variant.

- Based on the abstract machine for the tabled execution of fixed-order stratified logic programs of Ref. [28], we describe the implementation details of a publicly available engine, the full SLG-WAM, for computing the well-founded semantics. This engine inherits most of the properties of SLG resolution, including *polynomial data complexity* (as defined in Ref. [35]) for ground queries to Datalog programs with negation.
- We show that the features needed for the full SLG-WAM impose only a minimal overhead to Prolog (or tabled) execution; also, for Datalog programs with negation that terminate under Prolog, tabled execution in the SLG-WAM is competitive with their Prolog-style evaluation.
- Prolog's left-to-right computation rule need be broken only when the evaluation encounters a literal that is not *left-to-right dynamically stratified* [30]. We describe how SLG-WAM is designed to break Prolog's computation rule only in these cases, so that its efficiency accrues from its search strategy in addition to the data structures used by its instruction set.

A long development effort has preceded the results of this paper. Because non-stratified programs have not been commonly used for programming under the well-founded semantics, Section 1.1 provides a detailed motivational example of how non-stratified negation arises naturally in the verification of concurrent systems. Specifics of this example, however, are used in later sections of this paper. Next, in an effort to make this paper as self-contained as possible, Section 2 contains a new presentation of previously reported aspects of SLG resolution, [2] while Section 3 motivates the necessary additions for well-founded negation to a 'core' tabled engine by presenting a brief overview of the architecture and instruction set of the engine described in Ref. [28]. However, as the emphasis of this paper is on additions for well-founded negation (described in detail in Sections 4–6) as well as the performance of these additions (Section 7) we assume from the reader some familiarity with the Warren Abstract Machine (WAM [1,37]), and we refer to Ref. [28] for details of our tabled engine that mainly concern issues for tabling for fixed-order stratified programs.

1.1. A motivational example: using non-stratified negation for verification

Tabled logic programs can closely reflect underlying semantic definitions of a problem domain. As a simple instance, tabling allows efficient parsers to be constructed via their representation as context-free grammars [18]. More recently, it has been shown that tabling allows semantics-based program analyzers to be constructed by directly specifying the relationship of a given abstract and concrete domain [9,12]. Analyzers constructed using tabled logic programming have two advantages. First, they are concise and easy to program since they consist of a logic program that directly represents the semantic equations that form the basis of the analysis. Second, despite their declarativity, their performance is often competitive with analyzers written directly in C or C++. Negation is not used heavily in parsing or program analysis, because the basic grammars or semantics equations to be implemented do not themselves use negation. However, negation is used in the semantic definitions of other domains, such as the automated verification for concurrent

[2] Full details of SLG can be found in Ref. [7] and of the variant we use in Ref. [30].

systems, colloquially called *model checking*. We overview how tabled logic programming over the well-founded semantics can be used to represent the satisfiability definitions of concurrent temporal logics and thus form a basis of a concise and efficient model checker [10,19,24].

Typically model checkers consist of: (1) a transition system, usually specified by a process calculus which depicts states of processes and their communication methods; and (2) a temporal logic to represent possible properties of the transition system. The user of a model checker may query to see if a given temporal logic formula (e.g., one stating freedom from deadlock) is true in all states reachable from an initial state, or in all computation paths arising from an initial state. Thus if the transition system (process calculus) and temporal logic used in a model checking can be embedded into a normal logic program, model checking can perhaps be seen as an application of logic programming.

These embeddings turn out in fact to be possible, and we focus on how normal programs can be used to represent the *modal-mu calculus* a temporal logic into which most other temporal logics used in verification are themselves embeddible (see e.g., Refs. [8,15]). At a syntactic level, the modal-mu calculus includes the usual atoms and connectives of propositional logic formulas; the labelled modal constructors $\langle action \rangle$ and $[action]$, where *action* ranges over a set of (action) labels. In addition, there is also a least fixed point quantifier, μ, and a greatest fixed point quantifier ν. The following modal-mu calculus formula, whose meaning will be explained below, expresses the fairness property that along some a-labelled computation paths, the property p holds infinitely often:

$$\nu Z_1.\mu Z_2.\langle a_1 \rangle((p_1 \wedge Z_1) \vee Z_2). \tag{1}$$

At a semantic level, a model, $\mathscr{M}$, for the modal-mu calculus is a Kripke-like structure that consists of: (1) a labelled binary transition relation $\mathscr{M}_T$ between states $\mathscr{M}_S$ of a transition system $trans(state_1, label, state_2)$, and (2) a mapping $\mathscr{V}$ from atomic propositions and variables to sets of states in $\mathscr{M}_S$. $\mathscr{V}$ is extended to formulas by a set of satisfiability equations. The semantic equations relevant to Formula 1 are presented below. Throughout this section, ϕ_i ranges over the set of modal-mu calculus formulas, Z_i over the set of variables, p_i over the set of atomic propositions, a_i over the set of labels, and for $S' \in \mathscr{M}_S$, $\mathscr{V}_{[S'/Z_i]}$ agrees with $\mathscr{V}$, except that $\mathscr{V}_{[S'/Z_i]}(Z_i) = S'$

$$\mathscr{V}(\phi_1 \wedge \phi_2) = \mathscr{V}(\phi_1) \cap \mathscr{V}(\phi_2),$$
$$\mathscr{V}(\phi_1 \vee \phi_2) = \mathscr{V}(\phi_1) \cup \mathscr{V}(\phi_2),$$
$$\mathscr{V}(\neg\phi) = \mathscr{M}_S - \mathscr{V}(\phi),$$
$$\mathscr{V}(\langle a_i \rangle \phi) = \{s \in \mathscr{M}_S | \texttt{if } trans(s, a_i, s') \in \mathscr{M}_T, \texttt{ then } s' \in \mathscr{V}(\phi)\},$$
$$\mathscr{V}(\mu Z_1.\phi) = \cup\{S \subseteq \mathscr{M}_S | S \subseteq \mathscr{V}(\phi)_{[S/Z_1]}\},$$
$$\mathscr{V}(\nu Z_1.\phi) = \neg\mu Z_1.\neg\phi.$$

Given a model $\mathscr{M}$, a formula ϕ is true in a state S, written $S \models_{\mathscr{M}} \phi$, if $S \in \mathscr{V}(\phi)$.

The first-step of defining the satisfiability equations as normal clauses is to write a modal-mu calculus formula as a set of Prolog terms. In particular, each subformula that begins with a quantification symbol is defined as a Prolog term. Formula (1) is given the name `out` and defined as

```
:- table |=/2.

State_s |= prop(P):-
    has_prop(State_s,P).
State_s |= diam(Act_a,F):-
    trans(State_s,Act_a,State_t),
    State_t |= F.


State_s |= and(X_1, X_2)):-
    State_s |= X_1, State_s |= X_2.
State_s |= or(X_1, X_2)):-
    State_s |= X_1 ; State_s |= X_2.
State_s |= form(X):-
    formula_def(X, Y),
    ( Y = lfp(Z),
      State_s |= Z
    ;
      Y = gfp(Z),
      tnot(State_s |/= Z)
    ).
```

```
:- table |/=/2.

State_s |/= prop(P):-
    \+ has_prop(State_s,P).
State_s |/= diam(Act_a,F):-
    tfindall(State_t,
             trans(State_s,Act_a,State_t),
             State_ts),
    'list_|/='(State_ts, F).
State_s |/= and(X_1, X_2)):-
    State_s |/= X_1 ; State_s |= X_2.
State_s |/= or(X_1, X_2)):-
    State_s |/= X_1, State_s |/= X_2.
State_s |/= form(X):-
    formula_def(X, Y),
    ( Y = lfp(Z),
      tnot(State_s |= Z)
    ;
      Y = gfp(Z),
      State_s |/= Z
    ).
```

Fig. 1. Code for interpreting modal-mu calculus formulas.

```
form_def(out, gfp(diam(a, or(and(prop(p),out),
  form(in(form(out)))))))),
```

while the subformula

$$\mu X_2.\langle a\rangle((p \wedge X_1) \vee X_2) \tag{2}$$

is given the name `in` and defined as

```
form_def(in(form(out)), lfp(diam(a, or(and(prop(p),
  form(out)), form(in(form(out)))))))).
```

To determine whether a property such as `out` is true in a state S, the query `?- S |= out` is made. The predicate `|=/2` (see Fig. 1), uses XSB syntax, including a tabling declaration for `|=/2` and the predicate `tnot/1` which executes tabled negation. The clauses for `|=/2` – in which the underlying labelled transition relation is represented by the predicate `trans/3` – directly reflect the satisfiability equations. Note that in the last clause of `|=/2`, an explicit least fixed point quantification is evaluated directly through the tabling engine, while an explicit greatest fixed point quantification uses the identity $\nu X.\Phi = \neg\mu X.\neg\Phi$, and calls the predicate `|/=/2` which is designed so that the atom `State_s |= F` is true if and only if `State_s |/= F` is false under the well-founded semantics. Fig. 1 provides the relevant clauses for `|/=/2` again in XSB syntax, in which `tfindall/3` is a tabled version of `findall/3`.

Fig. 2 shows a calling sequence that gives rise to a non-stratified loop through negation. It is important to note that the loop arises from the fact that the μ-quantified Formula (2) contains a variable X_1 which lies in the scope of a ν quantifier. Such formulas are sometimes termed *alternating* formulas [15]. [3] In Ref. [24] it was shown that evaluation of an alternation-free modal-mu calculus formula could be

[3] Alternately a modal-mu calculus formula can be seen to be alternation-free if it is decomposable into subformulas each of which can be evaluated using a sequence of μ operators *or* a sequence of ν operators.

```
The original query is
   S |= form(out)
which calls
   tnot( S |/= diam(a,or(and(prop(p),out),form(in(form(out))))) )
which eventually calls
   S |/= form(in(form(out)))
which calls
   tnot( S |= form(in(form(out))) )
which eventually calls
   S |= form(out)
```

Fig. 2. Calling sequence for evaluation of Formula (1).

performed by a left-to-right dynamically stratified [30] program, but that alternating formulas gave rise to non-stratified programs. Alternating modal-mu calculus formulas can be executed directly using the well-founded engine described in this paper. In the case of non-alternating queries, the same engine produces a *residual program* consisting of rules whose bodies contain only those literals undefined under the well-founded semantics. This residual program is then sent to a stable model generator [22] to finish evaluation of the query. The core LMC model checker is a normal program containing a few hundred lines of code, including representation of a CCS-like process calculus for the underlying transition system. By applying to this program deductive database-style optimization techniques (literal reordering, factoring, clause resolution automata – see Ref. [24]), LMC is highly competitive with special purpose model checkers both in time and in space, and for both the alternation-free and the alternating case.

Several specific points may be drawn from this application. First, tabled logic programming, extended to handle non-stratified programs, forms a concise and efficient implementation technique for modal and temporal logics. Second, the process of using logic programming to implement such logics can lead to insights which are interesting in themselves, such as the relation between alternation-free modal-mu calculus formulas and stratification classes. More generally, it can be seen that tabling can provide a direct and efficient way to *implement* semantic equations: such as those for model checking, for program analysis, or for grammar specification. When these equations contain negation, the negation may be non-stratified, and implementations of tabled logic programs must address this case.

2. SLG resolution: review and terminology

We assume the basic terminology of logic programming from Ref. [20] and of the well-founded semantics from Ref. [34]. The version of SLG resolution presented below is based on SLG_{RD} (SLG with Reduced Delay) [30] which adds to the original formalism of SLG [7], the *delay minimality* property. This property improves the search of SLG for non-stratified programs by reducing the need for delay (see Section 2.1 for further discussion). Our presentation is also more operational than in Ref. [7] or [30], assumes a left-to-right literal selection strategy, and combines SLG with SLD resolution.

We define subgoals as atoms, and we will treat variant atoms as identical. A *tabled program* is a normal logic program augmented with tabling declarations of the form:

$$:- \texttt{table}\ p_1/n_1, \ldots, p_k/n_k,$$

where p_i is a predicate symbol and n_i is a non-negative integer. These declarations ensure that all calls to the predicate p_i of arity n_i will be executed using SLG resolution. These predicates are referred to as *tabled* predicates. All other predicates are implicitly assumed to be *non-tabled* in which case SLD resolution is used for their evaluation. For convenience, in this paper we also use the declaration `:- table_all.` to denote that all predicate symbols in the program are declared tabled. A tabled subgoal is a subgoal (i.e. an atom) whose predicate symbol is tabled; a tabled literal is either a tabled subgoal or its default negation. Also for simplicity, if a literal $\neg S$ is selected for resolution in a node of an SLG tree, we speak of S as the selected subgoal of the node. In order to compute the well-founded semantics, SLG may *delay* the evaluation of certain literals. [4] Delayed literals are defined as follows.

Definition 2.1 (*Delayed literal*). A *negative delayed literal* has the form $\neg S^S$ where S is a ground tabled subgoal; a *positive delayed literal* has the form B^S_{AT} where B, AT and S are atoms such that B is an instance of AT and AT is an instance of the tabled subgoal S. If η is a substitution, $(B^S_{AT})\eta = (B\eta)^S_{AT}$.

The annotations of both negative and positive delayed literals provide control information that may aid in establishing their truth value in the course of further evaluation. For a delayed literal $\neg S^S$ or B^S_{AT}, the superscript S is called the subgoal annotation of the literal and the subscript AT is sometimes called the answer annotation.

Like other tabling methods, SLG evaluates programs by maintaining tables of subgoals and their associated answers, and by resolving repeated instances of subgoals against *answers* from the table rather than against program clauses. By using answers in this manner, rather than repeatedly using program clause resolution as in SLD, SLG resolution avoids looping and terminates for all programs with the *bounded term-size property* (see e.g., Ref. [7]). States of an *SLG evaluation* of a query against a program are captured by *SLG systems* which are defined as follows.

Definition 2.2 (*SLG system*). An *SLG system* is a *forest of SLG trees*, along with an associated *table*. Root nodes of SLG trees are tabled subgoals. Non-root nodes either have the form *fail* or

$$\mathit{Ans\_Templ} :- [\mathit{Delay\_List}]\ \mathit{Goal\_List}.$$

The *Ans_Templ* is an atom, *Delay_List* is a (possibly empty) sequence of delayed literals, and *Goal_List* is a (possibly empty) sequence of literals. We assume, without

[4] Since the term *delay* is overloaded in logic programming, we note that this delay notion is different from the delay used for co-routining or for constraints.

loss of generality, that the leftmost element of *Goal_List* is the selected literal of the node.

The *table* is a set of ordered triples of the form

$$\langle S, Answer\_Set, State\rangle,$$

where the first element is a subgoal, the third either the constant *complete* or *incomplete*, and the second element is a set of ordered pairs of the form $\langle AT, [DL]\rangle$ where AT is an atom and DL is a (possibly empty) sequence of delayed literals.

As terminology, if $\langle S, Answer\_Set, State\rangle$ is an entry in the table for a system $\mathscr{S}$ we say that S is a *subgoal* in the table; that $A \in Answer\_Set$ is an *answer* in the table for S; and that *State* is the *state* of the subgoal. In an answer $A = \langle AT, [DL]\rangle$, AT and DL are called the *answer template* and the *delay list* of A, respectively. Informally, in nodes of a tree rooted by a subgoal S, *Ans_Templ* accumulates substitutions for the variables of S, *Goal_List* contains the currently selected literal and literals that remain to be selected in order to derive an answer, while *Delay_List* contains annotated versions of tabled literals that the evaluation has previously selected but has chosen to delay. Information in delay lists is propagated through *SLG answer resolution*.

Definition 2.3 (*SLG answer resolution*). Let N be a node of an SLG tree of the form $Ans\_Templ :- [Delay\_List]\ S, L_2, \ldots, L_n$, where $n > 0$ and S, the selected literal of the node, is both tabled and positive. Let $Ans = \langle AT, [DL]\rangle$ be an answer in the table of S whose variables have been standardized apart from N. N is *SLG resolvable* with *Ans* if S and AT are unifiable with a mgu η_S. The *answer resolvent* of N and *Ans* on S has the form

$$(Ans\_Templ :- [Delay\_List]L_2, \ldots, L_n)\eta_S$$

if DL is empty, and

$$(Ans\_Templ :- [Delay\_list, D]L_2, \ldots, L_n)\eta_S,$$

where $D = S^S_{AT}$, otherwise.

Definition 2.4 (*SLG evaluation*). Given a tabled program P, an *SLG evaluation* $\mathscr{E}$ for a subgoal G of a tabled predicate is a sequence of SLG systems $\mathscr{S}_0, \mathscr{S}_1, \ldots, \mathscr{S}_n$ such that

- $\mathscr{S}_0$ is the system whose forest consists of a single SLG tree containing the single root node G along with the table $\{\langle G, \emptyset, incomplete\rangle\}$;
- for each finite ordinal k, $\mathscr{S}_{k+1}$ is obtained from $\mathscr{S}_k$ by an application of one of the SLG operations in Definition 2.9.

If no operation is applicable to $\mathscr{S}_n$, $\mathscr{S}_n$ is called a *final system* of $\mathscr{E}$.

In our version of SLG, operations affect both forests and tables. Trees can be created and extended, and subgoals and answers copied into the table. If a subcomputation has derived all possible answers for a subgoal S and copied these answers into the table, the tree with root S is no longer needed and can be disposed. The subgoals in the table of a system $\mathscr{S}_k$ thus are root nodes of SLG trees in $\mathscr{S}_k$, or of trees that

were in previous systems and are now disposed [5] Before proceeding further with formal definitions, we review SLG operations informally and introduce some terminology.

The following five operational primitives are used to evaluate programs without negation: (1) Given a node N in a tree, a NEW SUBGOAL operation [6] checks to see if the tabled subgoal S for the selected literal L of N already forms the root of a tree in the forest (or, equivalently is a subgoal in the table); if S is new, it is explicitly added to the table and a new SLG tree is created whose root S is called a *generator* node. Independently of whether S is new, N is called a *positive* or *negative active* node of the SLG forest, based on the sign of L. (2) The children of generator nodes are created through PROGRAM CLAUSE RESOLUTION, as are the children of nodes whose positive selected literal is non-tabled. These latter nodes are called *interior* nodes. (3) The children of positive active nodes are created through answer resolution via the ANSWER RETURN operation. In general, the derivation of answers may be asynchronous with their resolution against active nodes. Thus, active nodes need to be preserved in the forest until they consume all possible answers. (4) If a node N in a tree with root S has an empty *Goal_List*, N is termed an *answer* for S. In such a case, a NEW ANSWER operation is applicable which explicitly adds the answer to the table if it is not already there. (5) Finally, when a subgoal (or set of subgoals) can produce no more answers, it is termed *completely evaluated*. Through the COMPLETION operation, an evaluation detects this condition, explicitly marks the table entries of the subgoals as *complete*, and disposes of their trees.

Determining when subgoals can produce no more answers may involve finding a set of mutually dependent subgoals; a *subgoal dependency graph* (SDG) of a system $\mathscr{S}_k$ is used to represent these dependencies. In $SDG(\mathscr{S}_k)$, vertices consist of subgoals in $\mathscr{S}_k$ whose state is *incomplete*; i.e., they form a root of a tree in the forest. A positive (negative) edge from S to S' occurs in the *SDG* if a *positive* (*negative*) *active* node in the tree rooted by S has $(\neg)S'$ as its selected literal. There might be both a positive and a negative edge between two (not necessarily distinct) vertices of the *SDG*. Because the *SDG* of an SLG system is a directed graph, the mutual dependencies between subgoals correspond to *Strongly Connected Components* (or SCCs) of the *SDG* and are defined in the usual manner. An SCC that depends on no other SCC is termed *independent*.

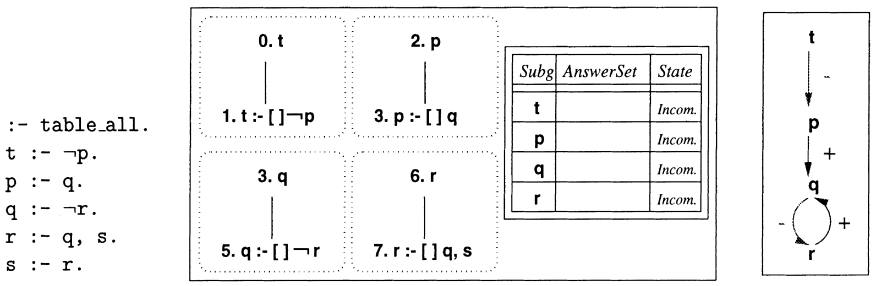
In the variant of SLG that we employ here, when a negative active node N has a selected negative literal $\neg S$ for a subgoal whose state is still *incomplete*, descendents of N in the tree cannot be immediately created. The node, however, is preserved in the forest, and computation in that branch of the tree is temporarily *suspended*. If the status of S changes to *complete* and its truth value becomes known, a NEGATION RETURN operation takes place to either *resume* that computation path by creating an immediate child of N with the selected literal removed from the node, or to fail the path by creating a *fail* child for N.

[5] In order for SLG to be complete for the well-founded semantics, SLG evaluation must also be defined for limit ordinals. The definition, which is somewhat technical, has been omitted from Definition 2.4 for presentation purposes.

[6] SLG operations are denoted in the font of NEW SUBGOAL throughout this paper, while engine-level instructions are denoted in the font of tabletry.

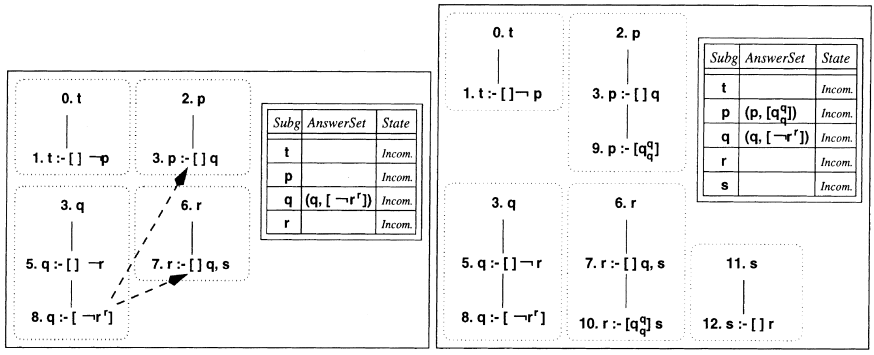
In the evaluation of normal programs, it is possible for tabled subgoals to depend upon one another through negation so that none can be determined to be completely evaluated before the rest. To allow these computation paths to proceed, SLG applies a DELAYING operation to negative literals involved in negative loops, transforming them to delayed literals and moving them to the *Delay_List* of their nodes. Doing so, the remaining literals in a node can then be resolved. As in definite programs, a node with an empty *Goal_List* is termed an answer, and literals in the *Delay_List* can be seen as conditions on the truth of these answers. Thus, we can call these answers *unconditional* or *conditional* depending on whether their *Delay_List* is empty or not. We can also speak of answers added conditionally into the table, in the following sense: by continuing the resumed computation path, information about the truth value of literals in a *Delay_List* might become known. In such a case, SIMPLIFICATION operations can either remove these delayed literals from conditional answers, or even remove the answers themselves. We further discuss DELAYING and SIMPLIFICATION, along with other aspects of SLG, through the following example.

Example 2.1. Consider the evaluation of a query `?- t.` with respect to the tabled logic program of Fig. 3(a). The evaluation begins by executing several NEW SUBGOAL and PROGRAM CLAUSE RESOLUTION operations, producing the SLG system in Fig. 3(b). In this system, the leftmost literal of the first four clauses of the program has been selected. There are four active nodes in the system, two of which are positive (3 and 7), and two of which are negative (1 and 5). As can be seen from its accompanying *SDG* in Fig. 3(c), the evaluation has encountered a negative loop containing `q` and `r`. Together, both subgoals form an *SCC* which is also independent. In order to determine the truth of `q` and `r` in the well-founded model, the computation rule must be broken so that other literals in clauses for `q` and `r` may be resolved. A DELAYING operation is applied to the negative literal involved in the independent *SCC* ($\neg$`r`), producing the forest shown in Fig. 4(a) in which node 5 has a non-empty *Delay_List*. At this stage, node 5 has no more literals to resolve, and becomes a conditional answer for `q`, which is entered into the table through the NEW ANSWER operation. As can be seen in Fig. 4(a) this conditional answer needs to be returned to active nodes 3 and 7. We assume for this example that it is first returned to node 3 (through the ANSWER RETURN operation). As stated in Definition 2.3, when

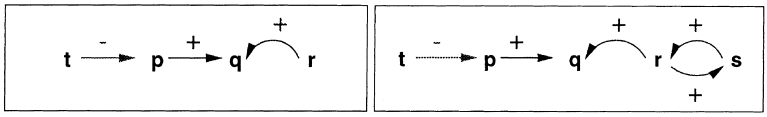


(a) Program (b) SLG System upon selecting the first literal of clauses 1–4 (c) SDG

Fig. 3. Program, SLG System created while evaluating the query `?- t.` and its induced SDG.



(a) SLG System (b) SLG System



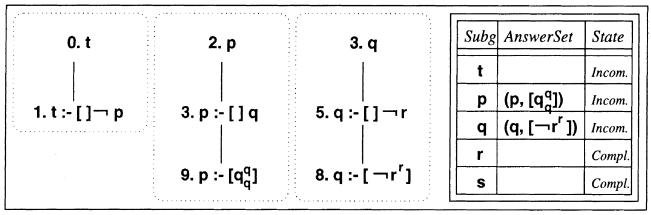
(c) SDG for the SLG System in Figure 4(a) (d) SDG for the SLG System in Figure 4(b)

Fig. 4. SLG Systems upon execution of the query `?- t.` against the program of Fig. 3a.

conditional answers are returned, SLG does not propagate the elements of delay lists, but rather uses positive delayed literals to indicate that a conditional answer was used for resolution. [7] In this case, answer resolution adds the delayed literal $\mathtt{q}^{\mathtt{q}}_{\mathtt{q}}$ to the delay list of node 3 producing a conditional answer. Similarly, answer resolution is used for node 7 which then calls `s` and afterwards, recursively, itself. The resulting system is shown in Fig. 4(b). At this stage, `s` and `r`, which are involved in a positive loop, can be determined to be completely evaluated and, through a COMPLETION operation, their trees are removed from the system and their state in the table becomes *complete*. The resulting system is shown in Fig. 5(a). Upon the completion of `r` with no answers, the answer $\langle \mathtt{q}, [\neg \mathtt{r}^{\mathtt{r}}] \rangle$ can be made unconditional, since $\neg\mathtt{r}$ is now known to be true in the well-founded model. As mentioned previously, the SLG SIMPLIFICATION operation is used in such situations. Using SIMPLIFICATION, the delayed literal $\neg\mathtt{r}^{\mathtt{r}}$ is removed from the delay list of `q`'s answer making this answer *unconditional*. The derivation of this unconditional answer enables a further SIMPLIFICATION operation in the answer $\langle \mathtt{p}, [\mathtt{q}^{\mathtt{q}}_{\mathtt{q}}] \rangle$ for `p`. Finally, since `p`'s truth value is now established, a NEGATION RETURN operation can be performed in node 1 of the forest, and through COMPLETION operations the remaining subgoals also become completed leading to the final system of Fig. 5(b).

We summarize the actions of Example 2.1 in handling negative loops: the DELAYING operation can be thought of as a mechanism for dynamically escaping from the left-to-right computation rule. As seen from the *SDG* of Fig. 4(c), delaying a

[7] This use of positive delayed literals guarantees a polynomial representation of answers to queries that may otherwise have an exponential number of answers (see Ref. [7] for an example of that situation).



Subg	AnswerSet	State
t		Incom.
p	(p, [q$^{q}_{q}$])	Incom.
q	(q, [¬r^{r}])	Incom.
r		Compl.
s		Compl.

(a) SLG System upon completion of subgoals r and s

Subg	AnswerSet	State
t		Compl.
p	(p, [ ])	Compl.
q	(q, [ ])	Compl.
r		Compl.
s		Compl.

(b) Final SLG System

Fig. 5. SLG Systems for Example 2.1.

literal may also avert dependencies through negation. As shown by the failure of s and r, DELAYING may also allow a clause that creates a cyclic negative dependency to fail based on the falsity of a literal further to the right of its body. Such failures trigger SIMPLIFICATION operations. In general, delayed literals that are *successful* should be removed from the delay lists of answers in the system, and answers with delayed literals that are *failed* should be deleted from the system. Concepts introduced in this example are now formally defined.

Definition 2.5 (*Success & failure of subgoals and delayed literals*). Let $\mathscr{S}_k$ be an SLG system. We say that a subgoal S *succeeds* in $\mathscr{S}_k$ if it has an answer of the form $\langle S, [] \rangle$, and that S *fails* in $\mathscr{S}_k$ if the state of the table entry of S is complete and the table contains no answers.

In a system $\mathscr{S}_k$, a negative delayed literal $\neg S^S$ is *successful* if subgoal S fails, and is *failed* if S succeeds. A positive delayed literal B^S_{AT} is *successful* if subgoal S has an answer of the form $\langle AT, [] \rangle$, and is *failed* if subgoal S does not have in its table an answer of the form $\langle AT, [DL] \rangle$ for any sequence of delayed literals DL.

Soundness and completeness of SLG are based on the following definition.

Definition 2.6. Let $\mathscr{S}$ be a system. Then the *interpretation induced by* $\mathscr{S}$, $I_{\mathscr{S}}$ has the following properties:

- A (ground) atom $A \in I_{\mathscr{S}}$ iff A is in the ground instantiation of some unconditional answer *Ans* in $\mathscr{S}$.
- A (ground) literal $\neg A \in I_{\mathscr{S}}$ iff A is in the ground instantiation of a completely evaluated subgoal in $\mathscr{S}$ (Definition 2.8), and A is not in the ground instantiation of any answer in $\mathscr{S}$.

Ref. [7] shows that for a final system $\mathscr{S}$ whose table contains the set of subgoals $\mathscr{Q}$, $I_{\mathscr{S}} = WFM|_{\mathscr{Q}}$, where $WFM|_{\mathscr{Q}}$ denotes the well-founded model of the program restricted to atoms that unify with subgoals in $\mathscr{Q}$.

Definition 2.7 (*Types of answers*). Let $\mathscr{S}$ be a system, S a subgoal in $\mathscr{S}$, and let $A = \langle AT, [DL] \rangle$ be an answer in the table for S. A is called an *unconditional* answer if DL is the empty sequence, otherwise it is called *conditional*.

A conditional answer $\langle AT, [DL] \rangle$ is *supported* by S in $\mathscr{S}$ if and only if

1. S is not completely evaluated (Definition 2.8); or
2. there exists an answer $\langle AT, [DL_1] \rangle$ of S such that for every positive delayed literal $B^{S_1}_{AT_1}$ in DL_1, AT_1 is supported by S_1.

Note that unconditional answers are always supported. The concept of a supported answer is used by the SLG ANSWER COMPLETION operation to handle cases in which delay propagation through answer resolution creates conditional answers that cannot later be simplified away. In general, ANSWER COMPLETION is necessary for the soundness and completeness of SLG. However, the need for ANSWER COMPLETION within an evaluation depends heavily on the scheduling strategy used to perform SLG operations. For scheduling strategies that try to minimize DELAYING or delay propagation (such as those in the SLG-WAM), ANSWER COMPLETION is rarely needed in practice. The issue is discussed further in Appendix A.

Recall that a subgoal (or set of subgoals) is termed *completely evaluated* when no more answers can be produced for it. For SLG evaluation defined over finite ordinals (see Definition 2.4) this condition can be defined as follows.

Definition 2.8 (*Completely evaluated set of subgoals*). Given an SLG system $\mathscr{S}_k$, a set Λ of subgoals is *completely evaluated* if either of the following conditions is satisfied:
1. Λ is an independent SCC of $SDG(\mathscr{S}_k)$, and for each subgoal S in Λ:
   1.1. All applicable SLG operations of Definition 2.9 (other than DELAYING, COMPLETION, SIMPLIFICATION, and ANSWER COMPLETION) have been performed for nodes in the tree rooted by S.
   1.2. The tree rooted by S contains no negative active node.
2. $\Lambda = \{S\}$ and S succeeds in $\mathscr{S}_k$.

Λ is *flummoxed* if conditions 1a and 2 hold but not 1b. A subgoal S is completely evaluated (flummoxed) in $\mathscr{S}_k$ if Λ is a completely evaluated (flummoxed) set of subgoals and $S \in \Lambda$.

We can now formally define the set of SLG operations that we employ.

Definition 2.9 (*SLG operations*). Given a system $\mathscr{S}_k$ of an SLG evaluation of a tabled program P and subgoal G, $\mathscr{S}_{k+1}$ may be produced by one of the following operations:

- NEW SUBGOAL. Given an *active* node N with selected tabled literal S or $\neg S$, where the subgoal S is not present in the table of $\mathscr{S}_k$, create a new SLG tree with root S and add the entry $\langle S, \emptyset, incomplete \rangle$ to the table.
- PROGRAM CLAUSE RESOLUTION. Let S be a subgoal and N be a node in $\mathscr{S}_k$ that is either a root node S or a node, $Ans\_Templ :- [Delay\_List]S, Goals$, whose selected atom S is non-tabled. Let $C = Head :- Body$ be a program clause such that $Head$ unifies with S with mgu θ and assume that C has not been used for resolution at node N.
  - if $N = S$ produce a child of N:

    $(S :- [\,]\ Body)\theta$,
  - if S is non-tabled, produce a child of N:

    $(Ans\_Templ :- [Delay\_List]\ Body, Goals)\theta$.
- NEW ANSWER. Let $Ans\_Templ :- [Delay\_List]$ be a node in a tree rooted by a subgoal S, such that the ordered pair $A = \langle Ans\_Templ, [Delay\_List] \rangle$ is not an answer in the table entry for S in $\mathscr{S}_k$. Then add A to the set of answers for S in the table.

- ANSWER RETURN. Let N be a *positive active* node $Ans\_Templ :- [Delay\_List]$ $S, Goals$, where S is a tabled subgoal. Let $A = \langle AT, [DL] \rangle$ be an answer for S in $\mathscr{S}_k$, let $\eta_S = mgu(S, AT)$ be the associated answer substitution, and assume that A has not been used for resolution against N. Then produce a child N' that is the SLG answer resolvent of N and A.
- NEGATION RETURN. Let N be a *negative active* node, $Ans\_Templ :- [Delay\_List]$ $\neg S, Goals$, such that the tabled subgoal S either *succeeds* or *fails* in $\mathscr{S}_k$. Then
  - if S succeeds in $\mathscr{S}_k$, then produce a *fail* node as the immediate child of N,
  - if S fails in $\mathscr{S}_k$, then produce a child of N: $Ans\_Templ :- [Delay\_List]$ $Goals$.
- DELAYING. Let N be a *negative active* node, $Ans\_Templ :- [Delay\_List]$ $\neg S, Goals$, such that the tabled subgoal S is ground [8] and flummoxed in S_k. Then produce an immediate child of N:

$$(Ans\_Templ :- [Delay\_List, \neg S^S]\ Goals).$$

- SIMPLIFICATION. Let $A = \langle AT, [DL] \rangle$ be an answer in the table for a subgoal S, and let D be either a successful or a failed delayed literal of DL. Then
  - if D is successful, then remove D from DL,
  - if D is failed, then remove A from the table.
- COMPLETION. If Λ is a set of subgoals that is *completely evaluated* (Definition 2.8), remove all trees whose root is a subgoal in Λ, and change the state of all table entries for subgoals in Λ from *incomplete* to *complete*.
- ANSWER COMPLETION. Given a set of non-supported answers $\mathscr{U}\mathscr{A}$, then remove each answer in $\mathscr{U}\mathscr{A}$ from the table.

In describing the SLG-WAM, it will often be convenient to speak of *answer substitutions*. If $\langle AT, [DL] \rangle$ is an answer for a subgoal S, its answer template AT is always subsumed by S, so that the answer substitution η_S is the *mgu* of AT and S. For a given subgoal S, there is a one-to-one mapping between the set of answer templates of its answers and the set of answer substitutions. Thus, we may group properties of answers according to answer substitutions.

2.1. Relevance in SLG and the SLG-WAM

As mentioned above, the version of SLG presented here is based on SLG_{RD} which has the property of *delay minimality* in that it does not use the DELAYING operation for *left-to-right dynamically stratified programs* [30]. *Dynamically stratified* programs were introduced in Ref. [23], and differ from most stratification formalisms in that recursive components are determined during the course of computation. The power of dynamic stratification can be seen from the fact that a normal program has a two-valued well-founded model iff it is dynamically stratified. Otherwise, in a partial well-founded model, the undefined atoms may be designated as belonging to the *ultimate* dynamic stratum. Ref. [30] introduces a natural restriction of dynamic stratification to a fixed-order computation rule. By defining the DELAYING operation to be applicable

[8] When the selected literal is negative and not ground, the SLG evaluation *flounders* and aborts through a FLOUNDERING operation. We restrict our attention to non-floundering SLG evaluations.

only for subgoals in flummoxed SCCs, SLG_{RD} delays negative subgoals only if they have no left-to-right derivation. Specifically, in Ref. [30] the following theorem is proven.

Theorem 2.1 (Ref. [30]). *Let $\neg S$ be a selected literal in a left-to-right SLG_{RD} evaluation of a ground program P. Then the* DELAYING *operation is applied to $\neg S$ if and only if S belongs to the ultimate left-to-right dynamic stratum of P.*

Theorem 2.1 can be seen as addressing the question of *relevance* in SLG. In principle, if a rule instance is used in a left-to-right well-founded evaluation, a literal of that rule is relevant only if it is preceded by a sequence of literals that are true or undefined in the well-founded model of the program. This criterion for relevance is quite strong and, as suggested by the inability of a non-ideal computation rule to evaluate normal logic programs, appears unobtainable in practice. Theorem 2.1 thus states our approximation to this ideal measure of relevance by using the notion of an ultimate left-to-right dynamic stratum. Mechanisms for executing non-stratified negation in the SLG-WAM make use of delay minimality, and can thus be based squarely on features of an engine to execute these stratified programs, to which we now turn.

3. A brief overview of the SLG-WAM

The SLG-WAM for the well-founded semantics is based on an existing abstract machine (denoted SLG-WAM$_{LRD}$) which is capable of evaluating left-to-right dynamically stratified programs; i.e., programs where DELAYING (and thus SIMPLIFICATION and ANSWER COMPLETION) are not needed. Full details of the machine model to support these tabled programs can be found in Ref. [28]. In this section we present a brief overview the SLG-WAM$_{LRD}$ and the extensions needed to evaluate the well-founded semantics.

3.1. Data structures and operations for fixed-order stratified programs

Most noticeably, the SLG-WAM$_{LRD}$ adds two new memory areas to the WAM: a *Table Space* where information about subgoals and their answers is persistently stored, and a *Completion Area* which is used to detect completely evaluated sets of subgoals. Besides the introduction of these new areas, for the tabled-based evaluation of programs, some modifications to the basic WAM data structures are needed which are summarized below.

An abstract machine for SLG resolution has to maintain information about a forest of trees rather than a single tree. In a WAM-like framework, a forest of SLG trees can be represented as a single *SLG search tree* in which the first positive active node for a subgoal is merged with its generator node. Also, as noted previously, tabling presents an asynchrony between the generation of answers and their consumption by (positive) active nodes. This asynchrony requires a mechanism to *suspend* computation of active nodes, and to *resume* these suspended computations at some point after the derivation of answers for the selected subgoals of the nodes. The suspend/resume functionality requires support for control of execution beyond that provided by the WAM: First, to ensure that the execution environments of active nodes are retained and that information in the SLG search tree is not lost, WAM stacks are

frozen when active nodes are suspended; all allocation occurs below the *freeze registers* for those stacks (assuming that the stacks grow downwards). Frozen segments in the stacks can be deallocated only upon determining that the subgoals associated with them are completely evaluated. Consider the effect of freezing segments of the choice point stack: choice points for nodes in the same branch of computation may not be contiguous. A given choice point may have been allocated below the freeze register so the choice point for its *parent* node in the SLG search tree may lie somewhere arbitrarily higher in the stack. So, unlike the WAM, information about parent nodes of the SLG search tree (or equivalently about the failure continuation on backtracking *out* of the choice point for the node) must be kept in an additional cell in each choice point.

To resume a suspended computation, the SLG-WAM needs to have a mechanism to reconstitute its environment. Besides the values of the WAM registers, the variable bindings at the time of suspension have to be restored. Thus, a *forward trail* is required. Given this trail, restoring the execution environment E from a current execution environment E_c, is a matter of untrailing from E_c to a common ancestor of E_c and E, and then using values in the forward trail to reconstitute the environment of E. The exact algorithm of this operation is presented in Ref. [28].

Finally, efficient implementation of stratified negation in the SLG-WAM$_{LRD}$ relies on detection of exact subgoal dependencies through lazy construction of parts of the Subgoal Dependency Graph. Mechanisms to support this are presented in Section 4.

3.2. Instruction set for fixed-order stratified programs

Besides extending the WAM data structures and using the instruction set of the WAM to implement PROGRAM CLAUSE RESOLUTION, the SLG-WAM$_{LRD}$ also adds a set of new instructions; we present them grouped by the SLG operation they perform:

- NEW SUBGOAL. This operation occurs when a tabled subgoal is called. In analogy to the WAM try, retry, and trust instructions, clauses of tabled predicates are compiled using tabletry, tableretry, and tabletrust SLG-WAM$_{LRD}$ instructions. The tabletry instruction checks whether the subgoal is new and if so, creates a *subgoal frame* (the engine-level representation of a table entry), sets up a *generator choice point*, and uses resolution against program clauses to obtain answers for the subgoal. If the subgoal is not new, it sets up a *consumer choice point* initiating the process of performing ANSWER RETURN operations for this instance of the subgoal. In either case, the tabletry instruction must fully traverse the subgoal to check if it is in the table or insert it if not. As it does so, the instruction factors out dereferenced variable occurrences from the subgoal and places them above the generator choice point. Later bindings to these variables will constitute *answer substitutions* which, if new, will be copied into the answer table.
  The tabletrust instruction, differs from its WAM counterpart in that it places a completion instruction in the failure continuation cell of the generator choice point. Recall that in the WAM this cell of the choice point contains the instruction to be executed upon backtracking out of the current program clause; thus, the completion instruction will be executed after all program clause resolution has been performed for the subtree stemming from this generator choice point.

- NEW ANSWER & ANSWER RETURN. Answers are added to the table via the new_answer instruction which is compiled as the last instruction of each clause of tabled predicates. In the SLG-WAM$_{LRD}$, where DELAYING is not needed to resolve loops through negation and where all answers are unconditional, ANSWER RETURNhas the following functionality. When an answer is derived for a particular subgoal, new_answer checks whether the answer substitution in the generator choice point has already been entered into the *answer table* associated with the subgoal. If it has, the derivation path fails, a vital step for ensuring termination. If not, the computation continues, and this answer together with any other unconsumed ones is scheduled to be returned to the applicable active nodes through answer_return instructions.
- COMPLETION. At an implementation level, COMPLETION is necessary not only for negation, but to reclaim stack space by disposing the trees of completed subgoals. Accordingly, SLG-WAM$_{LRD}$ uses two mechanisms to perform *incremental completion*. First, a safe over-approximation of each strongly connected component (SCC) of the subgoal dependency graph, termed a *scheduling component*, is maintained in the completion area. If there are no applicable operations for any subgoal in an independent scheduling component, and if no subgoal in this component is the selected literal of a negative active node, all subgoals in the scheduling component can be completed and their stack space reclaimed. Otherwise, exact dependencies of subgoals in the scheduling component are constructed in an *exact SCC detection phase* allowing incremental completion of exact SCCs.

3.3. Extensions for non-stratified normal programs

The features of the SLG-WAM$_{LRD}$ sketched above form a solid basis of an abstract machine for computing the well-founded semantics. However, as seen from Example 2.1, there are still a number of issues that need to be addressed by the SLG-WAM when loops through negation cannot be resolved by a fixed computation rule.

- *Implementation of the* DELAYING *operation*. While DELAYING is necessary to evaluate non-stratified programs, DELAYING should be minimized in order to restrict the search space of an evaluation. Section 4 discusses how the mechanism for exact SCC detection in the SLG-WAM$_{LRD}$ is extended to ensure a delay-minimal evaluation.
- *Representing and manipulating delay lists*. As seen in Section 2, delay lists are present both in nodes of SLG trees and in table entries. At the engine level, delay lists for nodes of SLG trees are represented in the heap, while delay lists for table entries are represented in table space. Sections 5.1 and 5.2 discuss the heap representation of delay lists, and their maintenance under backtracking, suspension and resumption. A previous paper [25] showed how *tries* formed a basis for efficiently implementing table access routines for the unconditional answers found in fixed-order stratified programs. Section 6.2.1 addresses how tries can be extended to store delay lists.
- *Simplification operations*. The table access routines of the SLG-WAM must also be extended to efficiently perform simplification when a delayed literal succeeds or fails, and so that these SIMPLIFICATION operations can be efficiently propagated if necessary. Section 6 describes the additional data structures and discusses SIMPLIFICATION in detail.

4. Implementation of the DELAYING operation

4.1. Detection and handling of loops through negation

As shown by Example 2.1, the evaluation of normal programs may encounter loops through negation. For the correct evaluation of such programs, a negative loop detection mechanism is required. Fortunately, the same dependency mechanism that detects completion of tabled subgoals can also be used to detect flummoxed SCCs containing loops through negation. As described in Ref. [28], a *root subgoal* register is added to the engine, and stored in choice points. This register effectively allows an active node in a tree T to point to the generator node constituting the root of T. This dependency information, along with information about whether calls are positive or negative, embeds the *SDG* in the WAM choice point stack. The SLG-WAM completion instruction starts with a stack-based over-approximation of the strongly connected components of the *SDG*. If there is no possibility of a negative loop involving a subgoal about to be completed, the completion instruction can complete subgoals based on the stack-based approximation of their SCCs. Otherwise, the engine lazily constructs the subgraph of the *SDG* restricted to the subgoals in the stack-based over-approximation to find an independent SCC I, and determine if there is a negative loop among subgoals in I. If so, DELAYING operations are applied to all negative active nodes suspended on subgoals in I. If not, the engine applies NEGATION RETURN operations to these negative active nodes.

4.2. Resolving negative literals in a well-founded model

The predicate `tnot/1`, together with the negation_resume instruction resolves negative literals of tabled subgoals in the SLG-WAM. Fig. 6 sketches the implementation of `tnot/1` using low-level built-ins. When the negation involves a completed ground subgoal, the built-in negate_truth_value/1 fails, succeeds, or delays depending on whether the subgoal succeeds, fails, or has only conditional answers, respectively.

A more complex case of tabled negation occurs when `tnot/1` is called when the subgoal is still incomplete. Then, the negation_suspend/1 built-in suspends the current computation, which may later be resumed when more is known about the truth value of the subgoal. At the engine level, the suspension is performed in a manner

```
tnot(S) :-
    ( ground(S) →
        ( subgoal_not_in_system(S), call(S), fail
        ; ( is_complete(S) → negate_truth_value(S)
          ; negation_suspend(S)
          )
        )
    ; error("Flounder: subgoal ", S, " is not ground")
    ).
```

Fig. 6. An implementation of tabled negation (`tnot/1`) for normal programs.

similar to the way computations are suspended on creating positive active nodes. A *negation suspension frame* (Fig. 7) is placed onto the choice point stack to save the execution environment for the suspended computation. When the computation resumes, a negation_resume instruction will be used to restore the suspended environment, to delay if necessary, and to continue the computation. In addition to the usual information of WAM choice points, the frame for a negative active node with selected literal ¬S contains information necessary in SLG-WAM choice points: the *RSreg* cell, whose value is used to maintain information about the root subgoal of the current SLG tree; and the *Dreg* cell, containing a pointer to the head of the delay list of its parent choice point (explained fully in Section 5.2). This latter cell is not necessary for stratified negation and all such cells are shown with an asterisk in Fig. 7. The *Dealloc_FailCont* cell of the negation suspension frame is used for scheduling and contains a pointer to possibly another negation suspension frame for ¬S. In addition, negation suspension frames contain a pointer to the subgoal frame of S that serves as its table entry; and a *Delay_Status* cell indicating whether a DELAYING operation is necessary for ¬S. The value of the *Delay_Status* cell is set by the completion instruction when the exact SCC detection phase of that instruction discovers a loop through negation.

Pseudo-code for the negation_resume instruction is shown in Fig. 8. The instruction must handle three cases for a negatively suspended literal ¬S: S can succeed or fail, in which case the NEGATION RETURN operation is applicable; or S is neither successful nor failed, in which case DELAYING is used. The negation_resume instruction itself is invoked by the completion instruction which can be thought of as performing a fixed point check on a scheduling component. If S does not succeed, the completion instruction backtracks into the chain of negation suspension frames for S, initiating a series of NEGATION RETURNor DELAYING operations for all (suspended) nodes whose selected literal is ¬*S*. Eventually, the engine will backtrack out of this chain and back to the completion instruction for the scheduling component containing S where the completion algorithm will be restarted.

Two other properties of the negation_resume instruction are worth noting. First, if an unconditional answer is derived for *S* before the negation_resume instruction is scheduled for *S*, the new_answer instruction will ensure that this instruction is never scheduled (line 2.6.3.1 of Fig. 16). However, to preserve delay minimality the negation_resume instruction itself checks whether S has got an unconditional answer between the moment of scheduling of the negation_resume instruction and its actual execution, and fails in this case. Otherwise if S has conditional answers or if the

FailCont	Pointer to negation_resume instruction
EBreg	Environment Backtrack Point
Hreg	Pointer to Top of Global Stack (Heap)
TRreg	Pointer to Top of (Forward) Trail Stack
CPreg	Return Point of Suspended Literal
Ereg	Pointer to Parent Environment
RSreg	Pointer to Choice Point of Root Subgoal
*Dreg**	Pointer to Parent Delay List
SubgFr	Pointer to Frame of Suspended Subgoal
*Delay_Status**	Set to true when the negative literal is determined not to be LRD-stratified
Dealloc_FailCont	Failure Continuation on Deallocation

Fig. 7. Format of negation suspension frames.

```
Instruction negation_resume
  CCP := breg;                          /* B register points to a negation suspension frame */
  Call restore_bindings(CCP);           /* restore environment of the suspended consumer */
  Restore values of WAM registers as saved in cells of the CCP;
  subgoal_ptr := NSF_SubgFr(CCP):   /* obtain pointer to suspended subgoal, S */
  if (has_answers(subgoal_ptr) or Delay_Status cell is set) {
    if (has_unconditional_answer(subgoal_ptr))
      fail;
    else  /* here either has_conditional_answer(subgoal_ptr) or Delay_Status cell is set */
      delay_negatively(subgoal_ptr, subgoal_ptr);          /* add literal ¬S^S to delay list */
  } /* else continue; */
  breg := Dealloc_FailCont(CCP);
  deallocate environment;
  proceed;
```

Fig. 8. The negation_resume instruction.

Delay_Status cell is set, ¬S is delayed. If neither of these cases is true, `tnot/1` succeeds without delaying. In all cases, the engine will reset the **B** register of the WAM, deallocate the environment of `tnot/1`, and proceed.

5. Representing delayed literals and delay lists in active nodes

Delay lists and delay elements are accessed in SLG resolution in the following ways:

- *Delay propagation.* Delay elements can be added to the delay list of an active node either by SLG answer resolution during the ANSWER RETURN operation or by the DELAYING operation.
- *Answer check/insert.* When the evaluation has encountered an answer node $AT :- [DL]$ in a tree rooted by a subgoal S and the NEW ANSWER operation is about to insert it into the table, a check must be made to determine whether the (conditional) answer $\langle AT, [DL] \rangle$ exists in the table for S, and the answer template and delay list must be copied into the table if not.
- *Simplification.* As mentioned previously, SIMPLIFICATION affects delay lists of tabled answers as the truth value of their delay elements becomes known.

From the perspective of the SLG-WAM, active nodes and their delay lists are represented in WAM stacks, while tabled answers are represented in the table space. The delay lists themselves have a different representation depending on whether they are in the heap or in table space. In this section we discuss how delay elements are represented in the SLG-WAM along with how delay propagation is performed. The representation of delay lists in conditional answers in the table space is intimately tied with support for simplification, and its detailed discussion is deferred until Section 6.2.1.

5.1. Representing delay elements in the SLG-WAM

As discussed in Section 2, a positive delayed literal requires both a subgoal and an answer annotation, while a negative delayed literal requires only a subgoal

annotation. As an example of how this works in SLG resolution, suppose a literal `p(X,Y)` is selected in a node N, and that `p(X,Y)` has in its table entry a conditional answer: $\langle$`p(a, f(Z))`, $[DL]\rangle$ for some non-empty delay list DL. Then by the answer resolution of an ANSWER RETURN operation the answer substitution

$$\eta_{\mathtt{p(X,Y)}} = \{\mathtt{X/a}, \mathtt{Y/f(Z)}\}$$

is propagated into N, and the delayed literal $\mathtt{p(a, f(Z))}^{\mathtt{p(X,Y)}}_{\mathtt{p(a,f(Z))}}$ is added to N's delay list. Finally, suppose further resolution in the remaining literals of N produces the binding `Z/b`. In this case the delayed literal now becomes $\mathtt{p(a, f(b))}^{\mathtt{p(X,Y)}}_{\mathtt{p(a,f(Z))}}$.

It turns out that a slightly simpler form of SLG answer resolution is sufficient to compute the well-founded model of a program. To see this, consider the SLG operations in Definition 2.9. First, note that only tabled literals end up in delay lists. Second, SIMPLIFICATION propagates information about the success or failure of delayed literals, and to propagate this information only the annotations of a delayed literal are required (cf. Definition 2.5). Indeed, even though SIMPLIFICATION may delete elements from the delay lists in tabled answers, it will never change the bindings of these delayed literals.

The implementation of the SLG-WAM in version 1.8 of XSB uses this variant of answer resolution. A positive delayed literal of the form B^S_{AT}, where $AT = S\eta_S$ is represented by a pair

$$\langle \mathrm{S_{ID}}, \eta_{\mathrm{S_{ID}}} \rangle,$$

where

S_{ID}: identifies the subgoal annotation of the delayed literal, e.g. `p(X,Y)`;

$\eta_{S_{ID}}$: either identifies (1) the answer annotation if the delayed literal is positive, e.g. `p(a,f(Z))`, or (2) contains the marker $\neg$ if the delayed literal is negative.

In terms of data structures, the $\eta_{S_{ID}}$ is a pointer to a leaf node of the answer trie for S. In the SLG-WAM$_{LRD}$, answer tries store only the substitutions of an answer template, rather than the entire answer template, due to an optimization called *substitution factoring* (see Ref. [25]). Thus an answer template is represented in the SLG-WAM by an answer substitution, and $\eta_{S_{ID}}$ is actually a pointer to a particular answer substitution η_S for a subgoal S. Thus, for discussion purposes, $\eta_{S_{ID}}$ uniquely determines a positive delay element, and we will use the two terms more or less interchangeably. S_{ID} is a pointer to a subgoal frame (the SLG-WAM data structure representing a table entry). A null pointer is used to denote the $\neg$ indicator, so that S_{ID} uniquely determines a negative delay element. S_{ID} and $\eta_{S_{ID}}$ are further discussed in Section 6.2.

Variations of the program in the following example will be used throughout this paper to describe the representation of delay lists and their elements. In these examples, a subgoal `p(X)` has the subgoal identifier $p(X)$, and its n answer substitutions each have an answer substitution identifier of the form $p(X)_1, \ldots, p(X)_n$. We will use the shorthand $\langle \neg S_{ID} \rangle$ to denote a negative delay element of the form $\langle S_{ID}, \neg \rangle$.

Example 5.1. Consider evaluation of query `?- p(X)` against the program of Fig. 9. This evaluation creates both positive and negative delayed literals. No simplification is possible, however. Fig. 9 also shows subgoals and answers in chronological order of generation under *Batched Evaluation* (the default scheduling strategy of XSB

```
:- table p/1.
p(f(X)) :- p(Y), r(X,Y), ¬q(X).
p(g(X)) :- q(X).
p(Y) :- r(X,Y), ¬p(Y).
q(X) :- r(X,g(X)).
r(a,g(c)).
r(b,g(b)).
```

S_{ID}: Subgoal	Answer		: η_{SID}
$p(X)$: p(X)	{X/g(b)}	[]	: $p(X)_1$
	{X/g(c)}	[⟨¬p(g(c))⟩]	: $p(X)_2$
	{X/f(a)}	[⟨p(X),p(X)$_2$⟩]	: $p(X)_3$
$p(g(c))$: p(g(c))	∅	[⟨¬p(g(c))⟩]	: $p(g(c))_1$
$p(g(b))$: p(g(b))	∅	[]	: $p(g(b))_1$

Fig. 9. A program requiring DELAYING and the tables created for the query ?- p(X).

version 1.8 and prior; see e.g., Ref. [16]). The answers are shown as split into two parts; the *answer substitution* (e.g. $\{X/\mathtt{g(c)}\}$), and the *delay list* (e.g. $[\langle \neg p(g(c))\rangle]$). Answers for different subgoals may be conditional on the same sets of delayed literals (as e.g. is the case of answers with answer substitution identifiers $p(X)_2$ and $p(g(c))_1$). Note that in general the same answer substitution can also have many different delay lists corresponding to different derivations of the answer substitution.

One of the goals of the SLG-WAM is to cleanly integrate SLG and SLD resolution. The introduction of conditional answers gives rise to an extra complication: delay lists must be propagated through the "unfoldings" of SLD resolution as illustrated by the following example.

Example 5.2. We extend the program of Example 5.1 to include both a tabled (pt/1), and a non-tabled (pp/1) alias for predicate p/1, along with a predicate u/1 which produces the conditional answers of p/1. The extension to the program is shown below:

```
:- table pt/1, u/1.
pt(X) :- p(X).
pp(X) :- p(X).
u(X) :- pp(X), ¬pt(X).
```

Fig. 10 shows the tables for subgoals of those predicates that are created by the evaluation of queries pt(X) and u(X). The answers to subgoals of the pt/1 predicate are derived in the manner explained in Example 5.1. Contrast, however, delay lists of the answers of the subgoal u(X). The first element of these lists contains representations of positive delayed literals $\mathtt{p(g(c))}^{\mathtt{p(X)}}_{\mathtt{p(g(c))}}$, and $\mathtt{p(f(a))}^{\mathtt{p(X)}}_{\mathtt{p(f(a))}}$ that have been propagated through the non-tabled predicate pp/1. Again note that the same delay list (e.g. $[\langle p(X), p(X)_2\rangle]$) is present in different answers.

5.2. Implementing delay propagation for active nodes

The SLG-WAM maintains information about the state of a computation path in global registers, as does the WAM. To keep track of the delay list for the current computation path, the SLG-WAM represents delay lists on the heap, and introduces a new register, **D** register , to maintain them. During DELAYING or resolution of a conditional answer by ANSWER RETURN, delay propagation allocates new list cell on the top of the heap, sets the tail to point to the old value of the delay list (as pointed to by **D** register), and the head to the new delay element. Observe that this

S_{ID}:	Subgoal	Answer		: $\eta_{S\,ID}$
$pt(X)$:	`pt(X)`	$\{X/$`g(b)`$\}$	[]	: $pt(X)_1$
		$\{X/$`g(c)`$\}$	$[\langle p(X), p(X)_2\rangle]$	: $pt(X)_2$
		$\{X/$`f(a)`$\}$	$[\langle p(X), p(X)_3\rangle]$	: $pt(X)_3$
$pt(g(b))$:	`pt(g(b))`	$\emptyset$	[]	: $pt(g(b))_1$
$pt(g(c))$:	`pt(g(c))`	$\emptyset$	$[\langle p(X), p(X)_2\rangle]$	: $pt(g(c))_1$
$pt(f(a))$:	`pt(f(a))`	$\emptyset$	$[\langle p(X), p(X)_3\rangle]$	: $pt(f(a))_1$
$u(X)$:	`u(X)`	$\{X/$`g(c)`$\}$	$[\langle p(X), p(X)_2\rangle, \langle \neg pt(g(c))\rangle]$	: $u(X)_1$
		$\{X/$`f(a)`$\}$	$[\langle p(X), p(X)_3\rangle, \langle \neg pt(f(a))\rangle]$	: $u(X)_2$

Fig. 10. Tables created for the queries `?- pt(X).` and `?- u(X).`

method of delay propagation represents delay lists in reverse order. To maintain the correspondence between the original and the residual program, a delay list is reversed when copied into the table space during the new_answer instruction. This mechanism imposes no overhead, since the delay list has to be traversed to be copied from the heap.

The **D** register is maintained in the following manner: (1) When a new tabled subgoal is called or failed into (i.e., when a generator choice point is created or backtracked into), the **D** register is set to point to an empty delay list. (2) When backtracking into an active or interior node N, the **D** register is restored to its value upon creation of N (as is done for all other WAM registers). (3) The value of the **D** register is also updated in forward continuations of tabled predicates to reflect the propagation of delayed literals in SLG answer resolution. The introduction of the **D** register requires small modifications to many SLG-WAM$_{LRD}$ instructions, which we indicate below.

- tabletry & tabletrysingle. Execution of this instruction corresponds to creation of a new SLG tree by the NEW SUBGOAL operation. Accordingly, this instruction saves the current value of the **D** register in the *Dreg* field of the tabled choice point, and then sets the **D** register to point to an empty delay list.
- tableretry & tabletrust. In this case evaluation has backtracked to the root of an SLG tree. Accordingly this instruction resets the **D** register to point to the representation of the empty delay list.
- try. Execution of the try instruction corresponds to creation of an interior node of an SLG tree. The current value of the **D** register is saved in the *Dreg* field of the choice point for future use when the node is backtracked into.
- retry & trust. Both instructions restore the **D** register using the value saved in the choice point.
- answer_return. If the answer returned is conditional, delay propagation is performed by adding a new (positive) delay element to the head of the delay list referred to by the **D** register.
- negation_resume. As mentioned in Section 4 the negation_resume instruction performs delay propagation when the DELAYING operation is performed. In this case a new (negative) delay element is added to the head of the delay list referred to by **D** register.
- new_answer. As described in Ref. [28], when using Batched Evaluation the SLG-WAM$_{LRD}$ performs a *first-call* optimization in which a generator node shares a choice point with the first active node whose selected literal is S. Furthermore,

the new_answer instruction is compiled as the last instruction of each clause of a tabled predicate. When a new answer is added to the table and the forward continuation of a tabled clause is followed, the engine implicitly returns the answer for S to the first active node for S. For this reason, the new_answer instruction must perform delay propagation similar to that of the answer_return instruction when a new answer is derived. The introduction of conditional answers imposes many changes to the new_answer SLG-WAM$_{LRD}$ instruction which are discussed in Section 6.3.2.

6. Simplification

When the truth value of a delayed literal becomes known, a SIMPLIFICATION operation becomes applicable. If SIMPLIFICATION operations are not performed as soon as they become applicable, more conditional answers may be derived and propagated through answer resolution. Doing so, unnecessarily expands the search space that is explored. The SLG-WAM therefore adheres to the following principles in initiating and propagating simplification.

Principle I. Conditions for simplification of delay elements should be detected, and the SIMPLIFICATION operation should be applied, as early as possible.

Principle II. Derivation of an unconditional answer for a subgoal S should immediately remove from the table for S all conditional answers with the same answer template.

At the implementation level, it is useful to specialize the SIMPLIFICATION operation depending on whether a delayed literal is successful or failed, and whether the delayed literal is negative or not. Thus, the SLG-WAM performs four separate simplification instructions:

- simplify_pos_successful ($\eta_{S_{ID}}$) which removes the positive delay element referred to by $\eta_{S_{ID}}$ from the delay lists of all answers conditional on it.
- simplify_neg_successful (S_{ID}) which removes the negative delay element referred to by $\langle \neg S_{\mathrm{ID}} \rangle$ from the delay lists of all answers conditional on it.
- simplify_neg_failed (S_{ID}) which removes all answers conditional on the negative delay element referred to by $\langle \neg S_{\mathrm{ID}} \rangle$.
- simplify_pos_failed ($\eta_{S_{ID}}$) which removes all answers conditional on the positive delay element referred to by $\eta_{S_{ID}}$.

We first discuss how these simplification instructions are initiated in the SLG-WAM. We then discuss in detail how the SLG-WAM table space represents conditional answers and supports simplification. Finally, we present pseudo-code instructions that manipulate conditional answers including the new_answer instruction, which interns conditional answers in the table and initiates simplification, as well as pseudo-code for simplification instructions themselves.

6.1. Events that trigger simplifications

In order to follow Principle I, the SLG-WAM executes a simplification instruction whenever the truth value of a subgoal or answer becomes known (see

Definition 2.5). In the SLG-WAM, this occurs through one of the following three events:

Derivation of an unconditional answer via the new_answer *instruction*. If an unconditional answer $AT :- []$ is derived for a subgoal S, then the new_answer instruction removes all conditional answers with answer template AT in the answer trie for S (Principle II). In addition, the derivation of the unconditional answer will spark simplify_pos_successful instructions if conditional answers with template AT have been returned to active nodes (Principle I).

Completion of a subgoal with no answers. When a subgoal fails, all negative elements delayed on that subgoal become successful, and can be removed from the delay lists that contain them. At an operational level, a step of the completion instruction calls a simplify_neg_successful instruction to perform this. Note that no direct simplification of positive delay elements can be initiated by a failing subgoal.

Failure or success of a delayed literal due to simplification. As initiated by either of the preceding two events, simplifications may have cascading effects. If a simplification instruction removes the last conditional answer for $\eta_{S_{ID}}$, then simplify_pos_failed instructions should be initiated for $\eta_{S_{ID}}$. If the removal of $\eta_{S_{ID}}$ causes S to be failed, and S is ground, then simplify_neg_successful instructions should be initiated for S_{ID}. In a similar manner creation of an unconditional answer can also spark new simplification instructions.

6.2. Data structures to support conditional answers and SIMPLIFICATION

6.2.1. Representing conditional answers in table space

As discussed in Ref. [25], the table space of the SLG-WAM$_{LRD}$ is built around tries since these structures can avoid repeated rescanning of common subterms within subgoal and answer tables. However, the tries of Ref. [25] allow storage and manipulation of unconditional answers only, and an extension is necessary to store delay lists in answer tables. This extension should efficiently support not only the answer check/insert and delay propagation operations mentioned at the beginning of Section 5, but the simplification instructions as well.

We begin by briefly reviewing the data structures used for unconditional answers as presented in Ref. [25]. As mentioned previously, associated with each tabled subgoal is an *answer trie*, which stores answer substitutions. Nodes of the answer trie consist of four fields: *symbol*, *first child*, *parent*, and *sibling*. The *symbol* field records information about the elementary bindings (i.e., constants, functor symbols, and variables) of the answer substitutions. The outgoing transitions from a node in the trie are traced using its *first child* pointer and by then following the list of *sibling* pointers of this child. [9] Answers are returned by backtracking through the answer trie (via the ANSWER RETURN operation). However, backtracking through the trie of an incomplete subgoal cannot be done efficiently by a simple trie structure. New answers may be added anywhere in the trie, while answer backtracking requires a sequential list of answers that have not been consumed via backtracking. Accordingly an answer trie for an incomplete subgoal requires an *answer return list* to support answer backtracking. This list chains together the leaves of the answer

[9] Ref. [25] also describes how hashing can be used to find outgoing transitions from a node.

substitution trie in chronological order. To return answers accessed through the *answer return list*, every node of the answer substitution trie maintains a back pointer to its *parent* node. When a subgoal is completed, its answer return list is reclaimed; future calls to the subgoal will backtrack through the trie itself.

Fig. 11 provides a close look at these trie data structures along with those added to support DELAYING and SIMPLIFICATION. The figure presents subgoal trie for predicate `p/1` of Example 5.1, and answer tries for two of these subgoals. The fields of the answer trie nodes are shown in the order: *symbol*, *first child*, *parent*, and *sibling*. Note that the answer substitutions for $p(X)$ are derived in the order `p(g(b))`, `p(g(c))`, and `p(f(a))` so that the order of answers in the *answer return list* differs from that of the trie.

An answer substitution may have many delay lists associated with it, corresponding to various clauses of the residual program. Accordingly, we access delay lists through the child pointers of the leaves of the answer substitutions, maintaining factoring of answer substitutions for conditional answers. Specifically, access to a delay list from an answer substitution η_S with identifier $\eta_{S_{ID}}$ is through a *Delay Info* record that contains:

$p(X)$:	`p(X)`
$p(g(c))$:	`p(g(c))`
$p(g(b))$:	`p(g(b))`

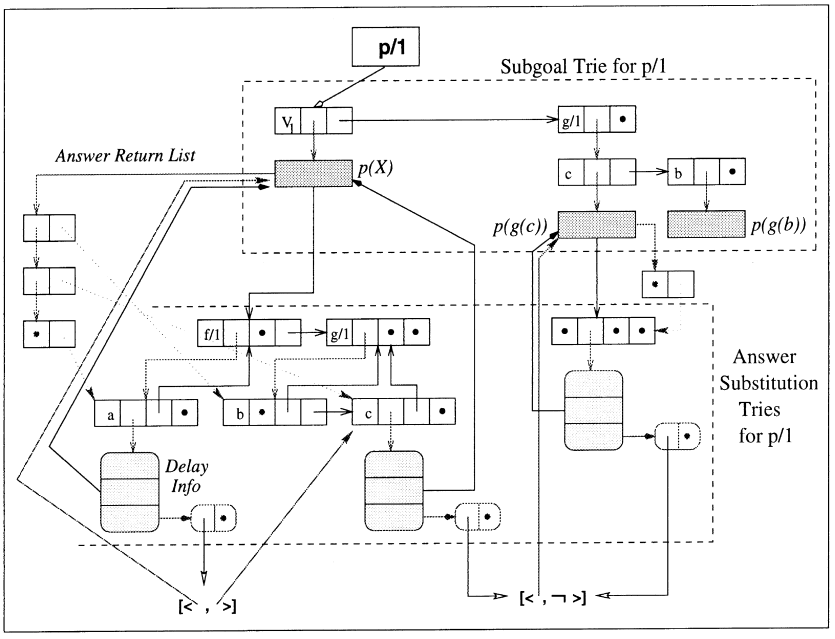
(a) Subgoals

Ans. Subst.	*Delay List*
$\{X/\texttt{f(a)}\}$	$[\langle p(X), p(X)_3\rangle]$
$\{X/\texttt{g(b)}\}$	[]
$\{X/\texttt{g(c)}\}$	$[\langle \neg p(g(c))\rangle]$

(b) Answers for $p(X)$

Ans. Subst.	*Delay List*
$\emptyset$	$[\langle \neg p(g(c))\rangle]$

(c) Answers for $p(g(c))$



(d) Table Space organized using *tries*

Fig. 11. Subgoal and answer tables for predicate `p/1` of Example 5.1.

1. An *IDE pointer* to the *Interned Delay Element* (IDE) of $\eta_{S_{ID}}$. The IDE in turn contains a pointer to a list of *Interned Delay Lists* (*IDLs*) containing $\eta_{S_{ID}}$.
2. A *subgoal pointer* back to the subgoal frame for S.
3. A *conditionality pointer* to a list of IDLs upon which η_S is conditional.

The fields of the delay info record are illustrated in Fig. 11 by the conditional answer substitution $\eta_{S_2} = X/f(a)$ which contains a pointer to the associated positive IDE of η_{S_2}, a back-pointer to the subgoal frame for η_{S_2}, a pointer to a chain of pointers to delay lists associated with η_{S_2}. Finally, we note that delay lists need never be copied out of the table upon answer resolution (cf. Definition 2.3). Accordingly, the answer return list points to the leaf node of each answer substitution as in Ref. [25].

Before presenting the internal structure of IDEs and IDLs, we note the rationale for the elements of Fig. 11. First there is a one-to-many relation between an answer substitution η_S and the (interned) delay lists upon which it depends. It is important to maintain this relation explicitly since the addition of an unconditional answer for η_S means that all its conditional answers are unnecessary and should be removed (Principle II). Next, because the same delay list may occur for many answer substitutions, delay lists are interned as IDLs so that a single simplification instruction will suffice for many delay lists of the same form (cf. Example 5.1, the delay lists containing $\langle \neg p(g(c)) \rangle$). These IDLs in their turn, consist of delay elements, which are interned as IDEs, in a manner that can efficiently propagate simplification instructions (cf. in Example 2.1, the propagation of the simplification of $\neg \mathtt{r}^{\mathtt{r}}$ through the answer for q to the answer for p).

Clearly, the efficiency of the simplification instructions heavily depends on the data structures that support them: subgoals, answer substitutions, interned delay elements, and interned delay lists must all be interconnected. Fig. 12 shows the persistent data structures of the SLG-WAM's *Table Space*, and their relationships. Connections from subgoals to simplification structures are recorded in *subgoal frames* through a pointer to the (negative) IDE of the subgoal. As Fig. 12 indicates, subgoal frames also associate a subgoal with elements such as its answer trie and its

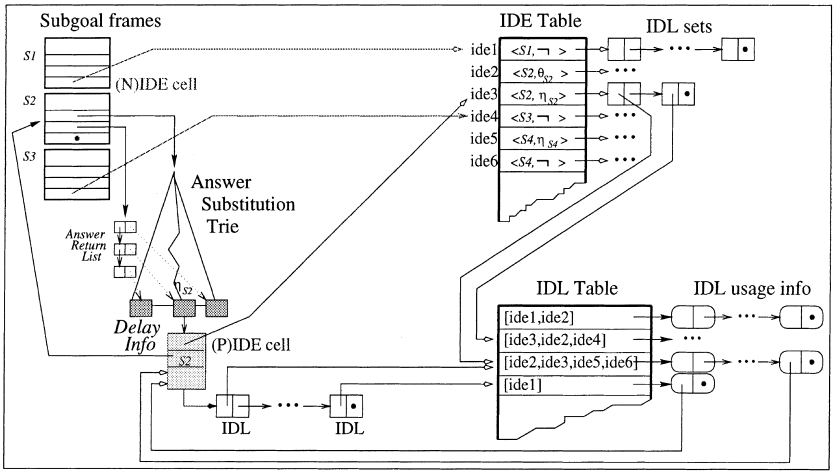


Fig. 12. Relationships between elements of the SLG-WAMs table space.

```
Instruction simplify_pos_successful(η_SID)
    Conditionality_Cell(AS_DelayInfo(η_SID)) = null;
    PIDE := PIDE_Cell(AS_DelayInfo(η_SID));     /* PIDE: Pointer to a Positive IDE */
    foreach IDL in which this PIDE appears
      IDL := IDL − {PIDE};          /* remove the positive IDE from the delay list */
      if (IDL = [])                 /* the DL is empty; answers are now unconditional */
        foreach answer A having answer substitution θ_SID and IDL as delay list
          i. simplify_pos_successful(θ_SID);
          ii. if (the subgoal S_ID containing A now succeeds)
                simplify_neg_failed(S_ID);
```

Fig. 13. Pseudo-code for a simplification instruction that deletes delay elements.

answer return list. A connection from an answer substitution to its positive delay element is made through the delay info record accessible through the leaf node of an answer substitution. Entries in IDE and IDL tables actually have the form of pointers (to subgoal frames, answer substitution trie leaves, and entries of the IDE Table); they are not shown as such for readability of the figure. Each entry of the IDE Table is associated with the set of delay lists that contains this entry as an element (the IDL set of Fig. 12). Conversely, each entry of the IDL Table is associated with the conditional answer substitutions that point to this entry (IDL usage info). [10]

6.3. *Instructions to support conditional answers and* SIMPLIFICATION

6.3.1. *Simplification instructions*

We now illustrate how the data structures of Figs. 11 and 12 are used by representative SLG-WAM simplification instructions.

- simplify_pos_successful ($\eta_{S_{ID}}$) (Fig. 13). This instruction begins by accessing the delay info record for $\eta_{S_{ID}}$ and setting the conditionality pointer to null. It then accesses the IDE for $\eta_{S_{ID}}$, and using that IDE, obtains a list of IDLs that contain the delay element, *PIDE*, for $\eta_{S_{ID}}$. *PIDE* is then removed from each of these delay lists. Two cases must be considered for propagation of simplification. The first is the case where the removal of *PIDE* creates an empty delay list. In this case, the IDL usage info of the newly empty IDL is used to find those answer substitution identifiers that are now unconditional and to execute simplify_pos_successful instructions for each of these. The second case occurs when the newly successful answer causes a subgoal to succeed. In this case, a backpointer from the delay element to the subgoal frame is followed, and a simplify_neg_failed instruction is called to remove all negative delayed literals of the subgoal that now succeeds.
- simplify_neg_failed (S_{ID}) (Fig. 14). This instruction deletes answers containing failed negative delay elements of a subgoal S that succeeds. As discussed, the S_{ID} points to a subgoal frame, and through this pointer the list of IDLs containing S_{ID} is obtained. Each IDL in this list is removed from the IDL table. As the IDL is removed, the answer substitution back-pointer is followed to determine if the answer substitution is associated with any remaining delay lists. If not, the answer

[10] An alternative representation of delay lists within the trie-based framework of Ref. [25] can be found in Ref. [11].

```
Instruction simplify_neg_failed(S_ID)                    /* S_ID succeeds */
    NIDE := SF_NIDE_Cell(S_ID);                          /* NIDE: Pointer to a Negative IDE */
    foreach IDL in which this NIDE appears
      Remove IDL from the IDL Table;                     /* IDL contains a failed element */
      foreach answer substitution η_S' such that there is an answer A = ⟨S'η_S', [IDL]⟩
        Remove IDL from the set of delay lists of η_S';
        if (the resulting set is empty)                  /* the status of η_S' is */
          i. Delete η_S';                                /* now unsupported */
         ii. simplify_pos_failed(η_S'_ID);
        iii. if (the subgoal S' that had η_S' as an answer substitution now fails)
               simplify_neg_successful(S'_ID);
```

Fig. 14. Pseudo-code for a simplification instruction that deletes answers with failed IDEs.

substitution $\eta_{S'}$ is deleted from the answer trie, and a simplify_pos_failed instruction is called for $\eta_{S'_{ID}}$. The deletion of $\eta_{S'}$ may now cause S' to fail, in which case a simplify_neg_successful instruction is called for S'_{ID}.

6.3.2. *The* new_answer *instruction*

In SLG resolution, SIMPLIFICATION operations are never applied to active nodes. The SLG-WAM follows this rule, but applies simplification instructions to answers as they are interned into the table by the NEW ANSWER operation, whose instruction is presented in Fig. 16. As discussed in Ref. [28], the new_answer instruction is compiled as the last instruction of each clause of a tabled predicate. It is thus executed immediately upon deriving an answer node. The following example illustrates the need for initiating SIMPLIFICATION by the new_answer instruction.

Example 6.1. Consider the evaluation of a query `?- p.` against the program of Fig. 15 where all predicates are tabled. The table in the same figure presents a chronological listing of the main events that occur while evaluating this query. After detecting two loops through negation, the selected literal ¬p in the active node for t gets delayed. The delay element $\langle \neg p \rangle$ representing the negative delayed literal $\neg \mathrm{p}^{\mathrm{p}}$ is added to the front of the delay list of t's node and is stored on the heap. Execution continues by selecting literal ¬q in the node of t, this literal and the new node of t suspend, and after several steps the completion instruction detects subgoals p, q, and r forming an *unfounded set* [34] and completes them. Since they have no answers, they fail. q's completion resumes the suspended negative literal ¬q in the body of t; this literal is true, and, since there are no more literals in this node, the SLG-WAM executes a new_answer instruction for t. Following the rules for **D** register described

	Negative loop involving p, r; literal ¬r gets delayed
`:- table_all.`	Negative loop involving p, s, t; literals ¬p, ¬t get delayed
	Delay element $\langle \neg p \rangle$ resides on the heap; ¬q, r are called and both suspend
`p :- ¬r, s, q.`	Subgoal s gets a conditional answer s_1 with delay list $[\langle \neg t \rangle]$
`q :- r.`	Answer s_1 returned to s in p's body, q is called
`r :- p.`	Subgoals $\{p, q, r\}$ get completed and fail; ¬q is resumed and succeeds
`s :- ¬t.`	Subgoal t is about to get an answer t_1 with delay list $[\langle \neg p \rangle]$
`t :- ¬p, ¬q.`	However, $\langle \neg p \rangle$ is now successful; t_1 is unconditional; t succeeds
	Answer s_1 is now unsupported and is deleted by SIMPLIFICATION; s fails

Fig. 15. A program requiring SIMPLIFICATION, and an outline of the evaluation of `?- p.`

```
/* Let A = ⟨AT, [DL]⟩ be the newly derived answer of subgoal S having answer substitution η_S */
Instruction new_answer(Arity,GCP)          /* GCP points to the generator choice point for S */
1   Get the delay list DL by using the current value of D register;
2   If (no D ∈ DL is failed)
2.1     S_ID := GCP_SubgFr(GCP);           /* pointer to the subgoal frame for S */
2.2     IDL := intern_delay_list(DL);      /* also removes all successful delay literals */
2.3     TR_S := SF_AnsTrieRoot(S_ID);      /* the answer trie for S */
2.4     η_S := locate_answer_substitution(Arity,GCP);
                                           /* η_S now points to the answer substitution*/
2.5     η_SID := answer_check_insert(η_S,TR_S,IDL,NewFlag);
                                           /* insert A (if needed) and get its ID */
2.6     if (NewFlag = TRUE)                /* η_S is new */
            /* implicit return of answer on forward continuation */
2.6.1       Restore the value of the D and RS registers as saved in GCP;
2.6.2       If (DL ≠ [])                   /* A is conditional */
2.6.2.1         Add a positive delay element ⟨S_ID, η_SID⟩ to the head of the restored delay list;
2.6.3       else
2.6.3.1         SF_NS_Chain(S_ID) := NULL; /* remove negation suspensions of S */
2.6.3.2         if (η_SID = ∅)             /* S succeeds; see Definition 2.5 */
2.6.3.2.1           Mark S_ID as complete; /* perform early completion */
2.6.3.2.2           simplify_neg_failed(S_ID);
2.6.4       Deallocate local environment;
2.6.5       Set the program pointer P to the continuation pointer CP;
2.7     else   /* η_S was already present in the answer table for S */
2.7.1       if (DL = [])
2.7.1.1         simplify_pos_successful(η_SID);
2.7.2       fail;
3   else fail;     /* A was simplified and failed */
```

Fig. 16. The new_answer instruction (for normal programs).

in Section 5.2, the delay list for t is $[\langle \neg p \rangle]$. Note however that since subgoal p has been completed with no answers, the delay element $\langle \neg p \rangle$ is removable from the delay list, and the answer of t is actually unconditional and is added to the table as such. After the execution of a simplify_pos_successful instruction, the computed well-founded model is two-valued: $T = \{t\},\ F = \{p, q, r, s\}$.

Upon the derivation of an answer node $AT :- [DL]$ in the tree rooted by a tabled subgoal S, the new_answer instruction works in the following manner: The instruction begins by simplifying the delayed literals in DL. If any of these literals is failed, new_answer effectively performs a SIMPLIFICATION operation by failing without inserting $\langle AT, [DL] \rangle$ in the table. Otherwise, the instruction interns the sequence of delayed literals DL at the same time removing from DL all successful delay literals (line 2.2). If the resulting DL is empty no insertion in the IDL table is made, and a null pointer is returned. The instruction then prepares to check whether the answer template is present in the system by locating a pointer, η_S, to the answer substitution for AT. This pointer is obtained via a pointer, GCP, to the generator choice point for S (Section 3.2), offset by the arity of S (line 2.3). [11] The new_answer

[11] The pointer to the generator choice point is kept in the local environment for each clause of a tabled predicate; see Ref. [28].

instruction also locates the root of the answer trie, TR_S, and then invokes the routine answer_check_insert which traverses the answer to see whether it is in the table, and inserts it if not, setting *NewFlag* to TRUE only if AT is new for S. If AT is not new, the computation will fail, but if AT is now unconditional, a simplify_pos_successful instruction will be initiated by the unconditional answer. When AT is new, the forward continuation will be taken (lines 2.6.4 – 2.6.5). The **D** and **RS** registers are reset, since the computation will be leaving the tree for S (see also Sections 3 and 5). If the new answer is conditional, a new element is added to the delay list (line 2.6.2.1). Otherwise, if AT is unconditional, any negative active nodes are removed from the data structures that schedule them, (lines 2.6.3.2.1 – 2.6.3.2.2) effectively performing a NEGATION RETURN operation. In addition, if the answer substitution is $\emptyset$, an answer has been derived that is a variant of S. S is completely evaluated according to Definition 2.8, and can be completed through a mechanism called *early completion* in Ref. [30] that corresponds to Condition 2 of Definition 2.8. In this last case, a simplify_neg_failed instruction for the completed subgoal will be triggered.

7. Performance

In this section, we first present detailed analyses of the costs and overheads of the operations described in this paper, and then present benchmark information for a commercial application in which non-stratified negation is used.

7.1. Overhead for Prolog execution

The addition of tabling mechanisms – both those described in this paper and those for fixed-order stratified programs [28] – adds little overhead to Prolog execution in XSB. For the D.H.D. Warren suite of Prolog benchmarks (see e.g., Ref. [28]), these additions add an overhead of about 8–10% to Prolog execution compared to the XSB version 2.0 emulator in which support for tabling has been removed. [12]

The engine described in this paper requires changes to many WAM instructions in order to maintain environments for all active nodes in an SLG search tree. As described in Section 3.1, this is done by sharing these environments in the WAM stacks and maintaining freeze registers, a forward trail, and other modifications to the WAM. A recent alternative approach to maintaining active nodes in an SLG search tree is called the Copy-Hybrid Approach to Tabling (CHAT) [13], and is designed to minimize changes to the WAM by copying environments for active nodes into and out of the execution stacks. Careful optimization for Prolog execution, such as omitting freeze registers as suggested in Ref. [13], can further reduce the overhead of tabling for Prolog execution.

[12] Overhead reported here is less than that reported in Ref. [28]; the difference is due to emulator optimizations between XSB version 2.0 and the version used in Ref. [28].

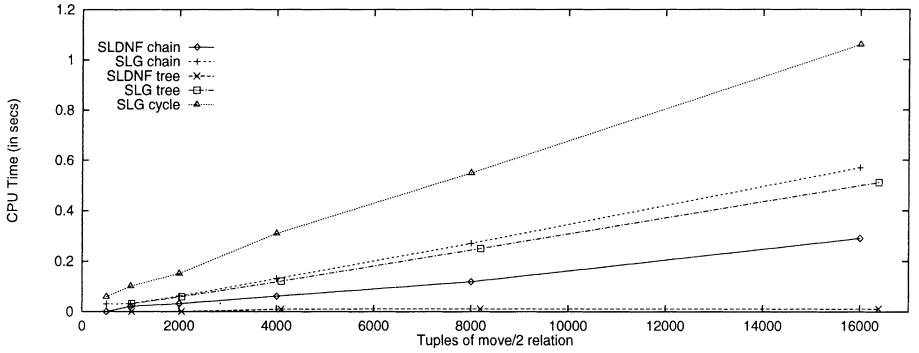


Fig. 17. Performance of the `win/1` program over different data-structures.

7.2. Analysis of performance of tabled negation

As reported in Ref. [6], an early version of the XSB system performs significantly better on non-monotonic queries than other systems for programming with the well-founded semantics. We begin by analyzing the performance of tabled negation on programs that are variants of the `win/1` predicates shown below.

SLG	`:- table win/1.` `win(X) :- move(X,Y), tnot(win(Y)).`
SLDNF	`win(X) :- move(X,Y), not(win(Y)).`

Fig. 17 demonstrates the scalability of negation in the SLG-WAM through timing results (in seconds) for executing the query `?- win(1),fail.` when the `move/2` relation represents chains, complete binary trees and cycles varying from 500 to 16k nodes. [13] SLDNF-resolution is sufficient for executing queries to `win/1` over acyclic `move/2` relations such as chains and trees. In this case the well-founded model of `win/1` is two-valued, while for the cycle all answers of `win/1` are undefined. On the other hand, the SLDNF version of `win/1` will not terminate if `move/2` contains cycles. As seen from Fig. 17, SLG evaluation of `win/1` over a cycle is about 80% more expensive than SLG evaluation of `win/1` over a chain of the same length. When `win/1` is evaluated over a cycle all answers are conditional and contain one element in their delay list; these elements and delay lists are interned and stored in global tables. Since DELAYING is always needed when `win/1` is executed over a chain, and since no SIMPLIFICATION is possible, performing exact SCC detection and interning information to support simplification imposes an unused overhead to the evaluation of this program.

When the `move/2` relation represents a complete binary tree or a chain, SLG evaluation can be compared to SLDNF evaluation. For binary trees, there is an operational difference between negation in SLG and in SLDNF resolution: a negated goal is fully evaluated in SLG (in order to ensure polynomial data complexity of an SLG

[13] A SPARCstation 20/55 running Solaris 2.5.1 was used for this test.

evaluation); in SLDNF, sub-evaluations are cut away after obtaining the first solution to a negated goal. For win/1 on a binary tree, SLG negation is exponentially slower than SLDNF; see Ref. [6] for further details. For chains, however, it can be seen that SLG evaluation of win/1 is about 2.5 times that of SLDNF evaluation of win/1, a comparison that we now consider in detail.

Consider the times for variants of the SLG and SLDNF win/1 predicate for a chain of length $2k$ in Table 1. In addition to executing negation under a different resolution method, the operator tnot/1 performs a floundering check which not/1 does not, while not/1 performs a dynamic transformation in case its argument contains a (Prolog) cut, an operation which tnot/1 does not. When each of these operations is factored out, the cost of using tnot/1 for win/1 becomes seven times slower than that of not/1.

Conceptually, the overhead of executing tabled negation when executing a query to win/1 over a cycle is due to:

- *The cost of the exact SCC detection phase*. In executing win/1 over a cycle, the engine must perform exact SCC detection to decide whether DELAYING is needed (Section 4). This exact SCC detection involves traversing the entire cycle of win/1 subgoals created by the move/2 relation.
- *The cost of* DELAYING. The engine must perform a DELAYING operation for each subgoal in the cycle.
- *The cost of interning conditional answers*. The engine must intern conditional answers creating the appropriate IDL and IDE data structures as discussed in Section 6.2.1.
- *Other overheads of executing* tnot/1 – see Fig. 6 – such as the cost of checking whether the subgoal is currently an entry in the table, the cost of suspending, and other bookkeeping operations.

Estimates of the proportion of times spent in these operations can be obtained by a testing tnot/1 without the floundering check on variants of the win/1 predicate. For instance, to measure the cost of interning conditional answers, consider the program:

```
:- table failing_win/1.
failing_win(X) :- move(X,Y), tnot(failing_win(Y)), fail.
```

This program will delay subgoals if necessary – however failing_win/1 will never succeed. Thus failing_win/1 essentially acts like win/1 except that no answer will ever be obtained. In this way, the cost of adding answers to the table is factored out. In the case of a chain of 2048 elements, failing_win/1 avoids interning 1024 unconditional answers, and presents a small speedup, as Table 2 indicates. In the case of the cycle, the difference is the addition of 2048 conditional answers and a much larger

Table 1
Benchmarking variants of win/1 for SLDNF and SLG

Variation	Chain	Cycle
Full tnot/1 (Fig. 6) executing win/1	1	1
tnot/1 without floundering check	0.83	0.90
(SLDNF) not/1 (with cut check) executing win/1	0.4	*
(SLDNF) not/1 (without cut check) executing win/1	0.12	*

Table 2
Benchmarking variants of win/1

Variation	Chain	Cycle
Full tnot/1 (Fig. 6) executing win/1	1	1
tnot/1 without floundering check executing win/1	0.83	0.90
tnot/1 without floundering check executing failing_win/1	0.76	0.53
positive_win/1	0.21	0.12
scc_win/1	0.22	0.13
simp_win/1	1.02	1.11
(SLDNF) not/1 executing win/1	0.12	*

speedup is obtained indicating a substantial cost due to the additional data structures needed to store conditional answers and to handle simplification.

For purposes of comparison, the definite program:

```
:- table positive_win/1.
positive_win(X) :- move(X,Y), positive_win(Y).
```

behaves like failing_win/1 in that it will create a table for every node reached through edges of the move/2 relation and in that it never succeeds. However, positive_win/1 will not incur the overhead of delaying, of exact SCC detection or of the other overheads of tnot/1. As can be seen from Table 2 these combined overheads are substantial. Finally, by executing a query to scc_win/1

```
scc_win(X) :- tnot(positive_win(X)).
:- table positive_win/1.
positive_win(X) :- move(X,Y), positive_win(Y).
```

over a cyclic move/2 relation exact SCC detection can be forced. To see this, note that the positive_win/1 subgoals for elements of the cycle are mutually dependent and will be completed together. However, upon execution of the COMPLETION operation, a negative subgoal dependency will be detected. While this negative dependency is from a subgoal, scc_win(1), outside the SCC to a subgoal positive_win(1) within the SCC, the exact SCC detection phase must be invoked to determine whether a loop through negation exists (see Ref. [28] for details). As can be seen from Table 2 – the normalization of the times is done separately for the chain and for the cycle – the cost of the check is minimal.

Table 3 summarizes these. By comparing failing_win/1 to win/1, it can be seen that interning conditional answers takes a large fraction of the time to execute win/1 over the cycle, while interning unconditional answers requires a smaller fraction for

Table 3
Distribution of overheads of win/1 over cycle and chain

Factor	% of Overhead	
	Chain	Cycle
Floundering check in tnot/1	17	10
Interning of answers in tables	7	37
Exact completion detection (SCC check)	1	1
negation_resume and other tnot/1 overheads	54	40

the chain.[14] By comparing `positive_win/1` to `scc_win/1`, it can be seen that exact SCC detection is extremely efficient for the cycle. Finally by comparing `failing_win/1` to `scc_win/1` it can be seen that the overhead of `tnot/1` together with the cost of the negation_resume instructions is fairly substantial, and accounts for most of the overhead of `tnot/1` over `not/1`. We note that a rough idea of the cost of negation_resume instructions themselves can be obtained by comparing the times for `failing_win/1` over a cycle, where 2048 negation_resume instructions are executed to that of the chain where 1027 such instructions are executed (negation suspension frames for negative literals whose subgoal is successful are pruned away by early completion operations).

Thus, scope for optimization remains by better compilation of `tnot/1`. [15] The routines for interning conditional answers, on the other hand, also cause a substantial amount of overhead, but this overhead is designed to reduce the time for SIMPLIFICATION operations that are not executed by the `win/1` benchmark. The cost of simplification can be measured by using the program `simp_win/1`:

```
:- table simp_win/1.
simp_win(X) :- move(X,Y), tnot(simp_win(Y)), fail_one(X).

fail_one(X) :- X \== 1.
```

Execution of the query `simp_win(1)` on the cycle of $2k$ elements will intern 2047 conditional answers, and then perform 2047 simplifications once it is determined that `simp_win(1)` fails. As can be seen from Table 2 the extra cost of these simplifications is small, and adds only about 11% cost to `win/1`.

7.3. A case study: diagnosis via abduction over the well-founded semantics

Standard diagnostic procedures about psychiatric disorders have been codified by the American Psychiatric Association in the fourth edition of its reference book *Diagnostic and Statistical Manual of Mental Disorders*, or *DSM-IV* [14]. DSM-IV is widely used in the United States to ensure accurate and standard diagnoses, to control medical costs, and to conduct research into the effects of social and economic factors on mental disorders. However while most psychiatrists use DSM-IV, few use it to its full advantage since DSM-IV is nearly 1000 pages long and contains 618 different, but often closely related, diagnoses. Typically, clinicians err in using DSM-IV by not considering all possible diagnoses, while researchers err by not excluding diagnoses quickly enough.

The *Diagnostica* system, which is based on XSB, attempts to help psychiatrists, psychologists, and psychiatric social workers use DSM-IV to treat patients. [16] DSM-IV can be represented as a directed acyclic graph (DAG) of propositions and their dependencies. As such, it contains 2553 diagnostic, symptom, and other

[14] Reducing the cost of this overhead has been recently addressed in Ref. [11].

[15] In version 2.0 of XSB, the built-ins that constitute `tnot/1` are shared with other system predicates such as tabled findall, `tfindall/3`.

[16] Diagnostica is available commercially through Medicine Rules, Inc. (See `http://www.medicine-rules.com`.)

nodes along with 4364 positive and negative links between these nodes. A practitioner might interact with Diagnostica by using a visual interface to assert symptoms about a patient. Diagnostica can then respond in a number of ways. It can inform the practitioner what criteria, if any, for diagnoses are met and which diagnoses are excluded. However, Diagnostica also allows practitioners to determine which diagnoses a patient is close to meeting, and could be met by knowledge of one, two, or *n* other symptoms. Furthermore, the system also allows a practitioner to query about symptoms that can be used to decide between two closely related diagnoses.

Using logic programming, diagnostic solutions can be found by constructing a small interpreter to determine when diagnoses in the DSM-IV DAG are satisfied by symptoms asserted to a database. Furthermore, information about close and differential diagnoses can be determined by abduction. Because Diagnostica is designed to fit onto a physican's personal computer, it is infeasible to maintain all possible abductive solutions for all possible sets of symptoms. Rather, abductive solutions must be computed efficiently on the fly.

Tabling is used for this abductive procedure in two ways. First, the use of tabling improves efficiency when retraversing the DAG. Second, the abduction itself can be stored in a delay list by making abducible literals undefined via the predicate:

```
:- table abduce_pos/1, abduce_neg/1.
abduce_pos(Symptom) :- tnot(abduce_neg(Symptom)).
abduce_neg(Symptom) :- tnot(abduce_pos(Symptom)).
```

Using the mechanisms described in this paper, a conditional answer thus represents an abductive diagnosis, whose *Delay_List* contains the abducibles required to prove the diagnosis. By adding integrity constraint checking to this mechanism, the mechanism described can be extended to construct abductive scenarios for all normal programs. See Ref. [3] for further information.

As a rough measure of the efficiency of this use of abduction, a benchmark was created to find every unconstrained abductive solution for all 618 diagnoses in the DSM-IV DAG, given a database in which no symptoms are asserted. On a Pentium 200 MHz laptop computer, this search required about 15 seconds to record all abductive diagnoses. Analysis of the abductive solutions indicates that there were 1680 abductive solutions in all, and that each abductive solution contained an average of 3.63 abducibles, making the total number of abducibles used in abductive solutions as 6115. As interned in the tables, the 1680 abductive solutions became 440 distinct IDLs, containing a total of 1547 elements, indicating about a 300% reduction in the number of (non-distinct) delayed literals in *Delay_Lists*, when the *Delay_Lists* are interned.

8. Discussion

The treatment of closed-world negation for normal logic programs as described in this paper surpasses the traditional treatment in Prolog in its ability to compute finitely meaningful answers to programs for which Prolog would loop infinitely. However, there are two aspects in which this treatment seems inferior to modern Prolog implementations: its inability to terminate all subcomputations at the point at which an answer is first found for a negative goal, and the fact that it flounders when a nonground

negative subgoal is first encountered with its left-to-right selection rule. Prolog's cut easily handles the first, and a co-routining mechanism as introduced in Mu-Prolog permits would-be floundering goals to be delayed to allow them to become further instantiated, and perhaps ground permitting a nonfloundering call. The lack of these capabilities in the engine described may seem like an oversight, but it is not.

A major design goal of the SLG-WAM was for it never to do redundant computation. Maintaining this goal and also introducing a pruning operator like the cut is problematic. In Prolog, where each subcomputation is on behalf of a single consumer, pruning a subcomputation when the consumer is satisfied is straightforward. However, in the SLG-WAM, a subcomputation may be generating results for many consumers. One consumer may need only one answer but another might need them all. Even deallocating a generator that has no other consumers may introduce an infinite loop if that generator is later needed again. Interesting and efficient pruning operators may well be possible in the SLG-WAM but they will be different from those in Prolog. Finding and implementing them efficiently requires more research.

As to the second limitation, one might think that the mechanism the SLG-WAM now has for delay is very similar to the mechanism used in Mu-Prolog to delay nonground negative goals to reduce floundering; see e.g., Ref. [21]. After all, they both are called "delay" and the purpose of both is to relax the strict left-to-right search order of classical Prolog. However, on closer examination, the two mechanisms differ in two respects. The more important one is that in the SLG-WAM a delayed element has undergone clause resolution and its subgoal is in the process of being computed; in co-routining delay, however, the computation of the delayed subgoal has not started. So implementing co-routining delay would require another delay list (or a bit distinguishing the type of delayed element) and significantly different processing. The other aspect in which they differ is the time at which goals are removed from the delay list. Removal of a delayed goal in a co-routining setting is triggered when a variable becomes bound. In the SLG-WAM a delayed goal is removed from its delay list during simplification. These require quite different mechanisms. Also the power of co-routining delay depends significantly on the ability of a goal to be delayed past the end of the clause in which it syntactically appears. A similar treatment in the SLG-WAM would require that the entire delay list be returned as part of the answer to each consumer. But that would lead to exponential behavior (and was, in fact, the reason why positive delayed literals were introduced in SLG-resolution in the first place; see Ref. [7, Example 3.1].) Again, adding a kind of co-routining behavior to the SLG-WAM is an interesting research problem.

9. Concluding remarks

Computation of the well-founded semantics is sometimes thought to be only of theoretical interest. At this time, Prolog programmers rarely write non-stratified programs, but the assumption behind the SLG-WAM is that because the well-founded semantics is a natural extension to normal programs an efficient implementation will encourage programmers to think in new ways and to discover uses for non-stratified programs. Two non-trivial examples of these new uses have been discussed: the verification system mentioned in Section 1.1 and the diagnosis system of Section 7.3. The SLG-WAM has been used to create a machine learning system, LIVE [17].

LIVE makes use of the extended logic programs under the well-founded semantics with explicit negation (WFSX [2]) to distinguish inferences that are explicitly disproved by a training set (explicitly false) from those that are not implied by a training set (default false). At an implementation level, extended logic programs are transformed into normal logic programs that can be efficiently executed by the SLG-WAM.[17] Transformations into the well-founded semantics exist for many other formalisms as well (see e.g., Ref. [32]). Examples such as these suggest that use of non-stratified negation will become increasingly important for logic programming.

Acknowledgements

Many people have made important contributions to the design and implementation of the SLG-WAM. In particular, we thank Prasad Rao for implementing trie-based table management routines for definite programs and Juliana Freire for implementing the batched scheduling strategy. Also, sincere thanks to anonymous reviewers for comments that improved the presentation of this paper. This research was supported in part by NSF grants CCR-9702681, ESS-9705998, INT-9600598, CCR-910259, CCR-9404921, and by a fellowship from the K.U. Leuven Research Council. Most of the research reported in this paper was conducted while authors were affiliated with the Computer Science Department of SUNY at Stony Brook.

Appendix A. Implementation of ANSWER COMPLETION

As mentioned in Section 2, the SLG ANSWER COMPLETION operation can be implemented as a post-processing step acting on conditional answers of completed subgoals in a table. The ANSWER COMPLETION operation is necessary for SLG to compute the well-founded model (see Ref. [7]) and ensures that, once all applicable SIMPLIFICATION operations have been performed, each delayed literal in the residual program is either involved in or depends on a literal that is involved in a loop through negation. In the terminology of Definition 2.7, ANSWER COMPLETION ensures that all literals are *supported*.

Example A.1. To illustrate the need for ANSWER COMPLETION, consider the execution of query `?- p.` against the program:

```
:- table p/0, s/0, r/0.

p :- p.
p :- ¬s.

s :- ¬r.
s :- p.

r :- ¬s, r.
```

[17] This preprocessor was written by Alferes and Pereira, and is included in Versions 1.8 of XSB and beyond.

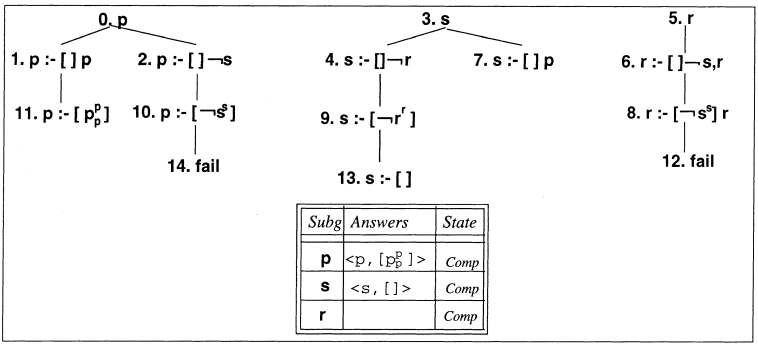


Fig. 18. An SLG System corresponding to a derivation that will require ANSWER COMPLETION.

The well-founded model of the above program is two-valued: r and p are both false, while s is true. Fig. 18 illustrates how the need for ANSWER COMPLETION arises during the course of evaluation by the SLG-WAM (for illustration purposes, the figure displays trees for completed subgoals). After the creation of node 7, there are no applicable operations for the nodes of the SCC {p, s, r} other than applying DELAYING operations. These DELAYING operations then create nodes 8, 9, and 10, with nodes 9 and 10 copied into the table as conditional answers. The conditional answer corresponding to node 10, p :- [¬s^s], is returned to node 1 to create node 11, which is also copied into the table. Finally, when execution resumes from node 8, subgoal r is determined to be completely evaluated and fails. Its failure causes a SIMPLIFICATION operation to simplify the delay list of the answer s :- [¬r^r], depicted in Fig. 18 by the creation of node 13. This simplification then is propagated to the answer p :- [¬s^s] (corresponding to node 10). Note that p should now fail, but does not, because of the conditional answer p :- [p_p^p]. The table of Fig. 18 illustrates the final answers if ANSWER COMPLETION were not used.

As the above example shows, the SIMPLIFICATION operation alone is not sufficient to remove all conditional answers that are false in the well-founded model. In particular, the existence of non-supported answers (Definition 2.7) must be addressed.

In the SLG-WAM, non-supported conditional answers are detected through the predicate `tc_unsupported/1` which is built on the builtin predicate `get_residual(?Atom,-Delay_List)`. This latter predicate, given an atom, succeeds if there is an answer $\langle AT, [DL]\rangle$ in the table such that AT unifies with `Atom`. In this case, the predicate unifies the second argument with DL represented as a Prolog list. Thus, conditional answers in the table can be treated as a residual program. Using `get_residual/2`, an *answer dependency graph* can be constructed whose vertices are answers, and whose directed edges are obtained using the elements in the `Delay_List` argument of `get_residual/2`. A sufficient condition for detecting non-supported answers in the table of a final system is the check for SCCs of the answer dependency graph that are independent and that contain only positive edges. The non-supported answers can be deleted from the table using the builtin predicate `delete_return/2` and the appropriate SIMPLIFICATION operations can then be performed.

Because the SLG-WAM is built using an evaluation strategy, SLG_{RD}, that eliminates the need for DELAYING in ground LRD-stratified programs (Theorem 2.1), the ANSWER COMPLETION operation is never required for such programs. In fact, even for programs that require DELAYING, ANSWER COMPLETION is rarely needed in practice. However, it remains an open question of what further classes of programs never need ANSWER COMPLETION given the evaluation (and scheduling) strategy of the SLG-WAM.

References

[1] H. Aït-Kaci, Warren's Abstract Machine: A Tutorial Reconstruction, The MIT Press, Cambridge, MA, 1991. See also: http://www.isg.sfu.ca/~ hak/documents/wam.html.
[2] J.J. Alferes, C.V. Damásio, L.M. Pereira, A logic programming system for non-monotonic reasoning, Journal of Automated Reasoning 14 (1) (1995) 93–147.
[3] J.J. Alferes, L.M. Pereira, T. Swift, Well-founded abduction via tabled dual programs, in: D. De Schreye (Ed.), Proceedings of the Joint International Conference and Symposium on Logic Programming, The MIT Press, Cambridge, MA, 1999, pp. 426–440.
[4] F. Bancilhon, D. Maier, Y. Sagiv, J.D. Ullman, Magic-sets and other strange ways to implement logic programs, in: Proceedings of the Fifth ACM Symposium on Principles of Database Systems, ACM, New York, 1986, pp. 1–15.
[5] R. Bol, L. Degerstedt, Tabulated resolution for the well founded semantics, Journal of Logic Programming 34 (2) (1998) 67–110.
[6] W. Chen, T. Swift, D.S. Warren, Efficient top-down computation of queries under the well-founded semantics, Journal of Logic Programming 24 (3) (1995) 161–199.
[7] W. Chen, D.S. Warren, Tabled evaluation with delaying for general logic programs, Journal of the ACM 43 (1) (1996) 20–74.
[8] R. Cleaveland, J. Parrow, B. Steffen, The concurrency workbench: a semantics-based tool for the verification of concurrent systems, ACM Transactions on Programming Languages and Systems 15 (1) (1993) 36–72.
[9] M. Codish, B. Demoen, K. Sagonas, Semantics-based program analysis for logic-based languages using XSB, Springer International Journal of Software Tools for Technology Transfer 2 (1) (1998) 29–45.
[10] B. Cui, Y. Dong, X. Du, K.N. Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, D.S. Warren, Logic programming and model checking, in: C. Palamidessi, H. Glaser, K. Meinke (Eds.), Principles of Declarative Programming, 10th International Symposium, PLILP'98, number 1490 in LNCS, Springer, Berlin, 1988, pp. 1–20.
[11] B. Cui, T. Swift, D.S. Warren, From tabling to transformation: implementing non-ground residual programs, in: Proceedings of the International Workshop on Implementation of Declarative Languages, Paris, France, 1999.
[12] S. Dawson, C.R. Ramakrishnan, D.S. Warren, Practical program analysis using general purpose logic programming systems – a case study, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, 1996, pp. 117–126.
[13] B. Demoen, K. Sagonas, CHAT: the Copy-Hybrid Approach to Tabling, in: G. Gupta (Ed.), Practical Aspects of Declarative Languages: First International Workshop, number 1551 in LNCS, Springer, Berlin, 1999, pp. 106–121 (extended version in: Journal of Future Generation Computer Systems, 16 (7) (2000) 809–830).
[14] Diagnostic and Statistical Manual of Mental Disorders, Americal Psychiatric Association, Washington, DC, 4th ed., 1994, Prepared by the Task Force on DSM-IV and other committees and work groups of the American Psychiatric Association.
[15] E.A. Emerson, C.-L. Lei, Efficient model checking in fragments of the propositional mu-calculus, in: Proceedings of the IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 1986, pp. 267–278.
[16] J. Freire, T. Swift, D.S. Warren, Beyond depth-first strategies: improving tabled logic programs through alternative scheduling, Journal of Functional and Logic Programming 1998 (3) (1998).

[17] E. Lamma, F. Riguzzi, L.M. Pereira, Strategies in combined learning via logic programs, Machine Learning Journal 38 (1&2) (2000) 63–87.
[18] R. Larson, D.S. Warren, J. Freire, K. Sagonas, Syntactica, The MIT Press, Cambridge, MA, 1996.
[19] X. Liu, C.R. Ramakrishnan, S.A. Smolka, Fully local and efficient evaluation of alternating fixed points, in: B. Steffen (Ed.), Proceedings of TACAS-98: Tools and Algorithms for the Construction and Analysis of Systems, number 1384 in LNCS, Springer, Berlin, March/April 1998, pp. 5–19.
[20] J.W. Lloyd, Foundations of Logic Programming, second ed., Springer, Berlin, 1987.
[21] L. Naish, Negation and Control in Prolog, number 238 in LNCS, Springer, New York, 1986.
[22] I. Niemelä, P. Simons, Smodels – An implementation of the stable model and well-founded semantics for normal LP, in: J. Dix, U. Furbach, A. Nerode (Eds.), Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning, number 1265 in LNAI, Dagstuhl Castle, Germany, Springer, 1997, pp. 420–429.
[23] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model, in: Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM, New York, 1989, pp. 11–21.
[24] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, D.S. Warren, Efficient model checking using tabled resolution. in: O. Grumberg (Ed.), Proceedings of the Ninth International Conference on Computer-Aided Verification, number 1254 in LNCS, Springer, Berlin, 1997, pp. 143–154.
[25] I.V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, D.S. Warren, Efficient access mechanisms for tabled logic programs, Journal of Logic Programming 38 (1) (1999) 31–54.
[26] R. Ramakrishnan, Magic templates: a spellbinding approach to logic programs, Journal of Logic Programming 11 (3&4) (1991) 189–216.
[27] P. Rao, K. Sagonas, T. Swift, D.S. Warren, J. Freire, XSB: a system for efficiently computing WFS, in: J. Dix, U. Furbach, A. Nerode (Eds.), Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning, number 1265 in LNAI, Dagstuhl Castle, Germany, Springer, 1997. pp. 430–440.
[28] K. Sagonas, T. Swift, An abstract machine for tabled execution of fixed-order stratified logic programs, ACM Transactions on Programming Languages and Systems 20 (3) (1998) 586–634.
[29] K. Sagonas, T. Swift, D.S. Warren, XSB as an efficient deductive database engine, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, ACM, New York, 1994, pp. 442–453.
[30] K. Sagonas, T. Swift, D.S. Warren, The limits of fixed-order computation, Theoretical Computer Science, to appear.
[31] P.J. Stuckey, S. Sudarshan, Well-founded ordered search: goal-directed bottom-up evaluation of well-founded models, Journal of Logic Programming 32 (3) (1997) 171–206.
[32] T. Swift, Using tabling for non-monotonic reasoning, Annals of Mathematics and Artificial Intelligence 25 (3&4) (1999) 201–240.
[33] H. Tamaki, T. Sato, OLD resolution with tabulation, in: E. Shapiro (Ed.), Proceedings of the Third International Conference on Logic Programming, number 225 in LNCS, Springer, Berlin, 1986, pp. 84–98.
[34] A. Van Gelder, K.A. Ross, J.S. Schlipf, The well-founded semantics for general logic programs, Journal of the ACM 38 (3) (1991) 620–650.
[35] M. Vardi, The complexity of relational query languages, in Proceedings of the 14th Annual ACM Symposium on the Theory of Computing, ACM, San Fransisco, CA, 1982, pp. 137–146.
[36] L. Vielle, Recursive query processing: the power of logic, Theoretical Computer Science 69 (1) (1989) 1–53.
[37] D.H.D. Warren, An abstract Prolog instruction set, Technical Report 309, SRI International, Menlo Park, USA, 1983.