An Abstract Machine for Computing the Well-Founded Semantics

Konstantinos Sagonas **Terrance Swift** **David S. Warren**
Department of Computer Science
State University of New York at Stony Brook

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.

This paper presents an implementation of the well-founded semantics in the SLG-WAM of XSB. To compute the well-founded semantics, the SLG-WAM adds three operations to its tabling engine — negative loop detection, delay and simplification — which serve to detect, to break and to resolve cycles through negation that may arise in evaluating normal programs. We describe fully the addition of these operations 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. Secondly, we present performance results that indicate that the overhead of delay and simplification to Prolog — or tabled — evaluations is minimal.

1 Introduction

The past decade of logic programming research has provided steady advancements 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 WAM-style Prolog engines. As proven by its widespread acceptance, the SLDNF/Prolog paradigm is extremely powerful for *programming*. However this method suffers from drawbacks which have prevented its extension into other areas such as databases or non-monotonic reasoning that need a higher level of declarativity 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 either positive or negative loops. Indeed, major areas of logic programming research can be viewed as efforts to formalize or implement evaluation methods that handle loops.

Handling positive loops has been addressed by many methods, with magic sets and tabling constituting the two main approaches. Although formulated differently, these methods turn out to treat positive loops in essentially the same way. Both assign failing values to SLD derivation paths which contain positive loops. As for negative loops, perhaps the critical insight behind the well-founded semantics is to assign the value *undefined* to derivations containing loops through negation [14]. It is natural, then, to extend tabling methods to handle negative loops and thereby execute the well-founded semantics, as several authors have noticed [2, 1]. For definite programs, tabling is amenable to an efficient, WAM-style implementation [13], so it is also natural to try to extend its implementation to normal programs. However, such an extension immediately leads to difficulties, based on fundamental aspects of computing normal logic programs. The first difficulty arises because an evaluation of a negative query has to wait until that query is completely evaluated before succeeding — conceptually the loop check must be extended from a check of a path of computation in the definite case, into a check of dependencies between subgoals. The second difficulty arises from the inability of a fixed-order computation rule to evaluate normal programs [8, 6]. Taking these difficulties at face value may lead to a conclusion that the well-founded semantics cannot be efficiently computed — in other words, that normal programs, unrestricted by stratification, are not suitable for programming but only for specification or knowledge representation.

This paper provides detailed evidence that queries to normal program under the well-founded semantics can be computed at the speed of Prolog programs, and integrated with Prolog. Further, we show a bound on the non-determinism of the computation rule required by normal programs (Theorem 2.1). In short, we demonstrate that logic *programming* is feasible over all normal programs. Specifically:

- We describe 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, including polynomial data complexity for ground queries to Datalog programs.
- We show that the features needed for the full SLG-WAM slow Prolog execution only minimally; also, that for Datalog programs with negation that terminate under Prolog, tabled execution in the SLG-WAM is competitive with Prolog.
- We show that the 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* [11]. This result indicates that the efficiency of the SLG-WAM accrues from its search strategy in addition to low-level implementation details.

1.1 Informal review of SLG resolution

A long development effort has preceded the results of this paper. To make this paper self-contained, we begin by summarizing previously reported aspects of SLG and the SLG-WAM.[1]

Like other tabling methods, SLG evaluates programs by keeping tables of subgoals and their associated answers, and by resolving repeated instances of subgoals against *answer clauses* from the table rather than against program clauses. By using answer resolution in this manner, rather than repeatedly using program clause resolution as in SLD, SLG avoids looping and terminates for all programs with the bounded term-depth property. In practice, not all subgoals need to be tabled for termination or efficiency: predicates may either be *tabled* in which case they are evaluated using SLG, or *non-tabled* in which case SLD evaluation is used.

Informally, an intermediate state of an SLG evaluation, called an *SLG system*, consists of a set of subgoals and their associated clauses, including answers. Such a system can be mapped to the usual forest-of-trees representation for tabling, in which each subgoal forms the root of a tree in the forest, the clauses form nodes of the tree, and answers are leaves with empty goal lists. Primitive operations for SLG have been presented elsewhere, and we review them briefly here. Assuming all predicates in a program are tabled, the evaluation of definite programs can be

[1] Full details of SLG can be found in [2] and of the variant we use in [11]. Details of the SLG-WAM for definite programs are in [13] and for stratified programs in [10].

modeled through four operational primitives. (1) Given a node N in a tree, NEW SUBGOAL checks to see if a the subgoal for a selected literal L already forms the root of a tree in the forest (or, equivalently is in the global table); if the subgoal is new, a new tree is created whose root is called a *generator* node and against which program resolution is used. Whether or not this is the case, the *status* of N becomes *active* with L as selected literal, and answer resolution will be used for L in N (for negative selected literals, the *status* of N is set to *suspended*: see below). (2) Because the derivation of answers may be asynchronous with their resolution against active nodes, a POSITIVE RETURN operation is formulated to explicitly perform the resolution. (3) When a resolution path produces an empty node, an answer has been derived, and a NEW ANSWER operation is performed which changes the status of the node to *answer* (operationally, this adds the answer to the table if it is not already there). (4) 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 and explicitly marks the subgoals as *completed*, and their *active* and *suspended* nodes as *disposed*.

Determining when subgoals can produce no more answers may involve finding a set of mutually dependent subgoals, and a *subgoal dependency graph (SDG)* is used to represent these dependencies. In the *SDG*, nodes consist of uncompleted subgoals in the system. A positive link from S_1 to S_2 occurs in the *SDG* if an *active* node in the tree for S_1 has S_2 as its selected literal. A negative link occurs in the same manner, but when the node of S_1 has status *suspended*. The mutual dependencies in the *SDG* are termed *Strongly Connected Components* (or *SCCs*). An SCC that depends on no other SCC is termed *independent*.

In the variant of SLG that we employ here, when a node has a selected negative literal for a subgoal that is not completed, the node's status becomes *suspended*. When the subgoal either succeeds (produces a fully evaluated answer) or fails, a NEGATIVE RETURN operation is performed to either remove the literal from the node, or to fail the derivation path.

In the evaluation of normal programs, a set of tabled subgoals may depend upon one another through negation in such a way that none can be determined to be completely evaluated before the rest. To allow tabled evaluation to proceed, SLG applies a DELAYING transformation to ground negative literals involved in negative loops, so that the remaining literals in the body of clauses can be resolved. The delayed literals are maintained in a *delay list* so that each node, including answers, is represented as: $answer\_template :- [delay\_list]\ goal\_list$. If the goal_list is empty, then the node is an answer which is added in the table through the NEW ANSWER operation. In SLG, each node is associated with the subgoal S that forms the root of the tree, so that *answer_template* is subsumed by S, and if the node is an answer, we will speak of the *answer substitution* which is the mgu of the *answer_template* and S. Answers in the system whose delay list is empty are termed *unconditional*, while answers with a non-empty delay list are termed *conditional*. An answer is called *supported* if none of the literals in its delay list is false; otherwise it is *unsupported*. Unconditional answers are always supported. There is a one-to-one mapping between an *answer_template* and an answer substitution, and we may group properties of answers according to answer substitution. Thus, we speak of an unconditional (supported) answer substitution η_S if there is an unconditional (supported) answer with answer substitution η_S. Otherwise the answer is conditional (unsupported). Finally, a subgoal S *succeeds* if it has an unconditional answer that is a variant of S, and it *fails* if it is completely evaluated with no answers.

Delayed literals within conditional answers may be removed from the body of clauses through SIMPLIFICATION operations. We further discuss DELAYING and SIMPLIFICATION, along with other aspects of SLG, through Example 1.1. Finally, we mention that throughout this paper we assume a left-to-right computation rule.

Example 1.1 We present in detail the evaluation of a query `?- t` with respect to the program of Figure 1(a) where all predicates are tabled. We depict an SLG

`t :- ¬p.`
`p :- q.`
`q :- ¬r.`
`r :- q, s.`
`s :- r.`
	`t:`*susp*`:t :- ¬p.`
	`p:`*act*`:p :- q.`
	`q:`*susp*`:q :- ¬r.`
	`r:`*act*`:r :- q, s.`
		`t:`*susp*`:t :- ¬ p.`
		`p:`*act*`:p :- q.`
		`q:`*act*`:q :- [¬r].`
		`r:`*act*`:r :- q, s.`
			`t:`*susp*`:t :- ¬p.`
			`p:`*ans*`:p :- [q].`
			`q:`*ans*`:q :- [¬r].`
			`r:`*act*`:r :- [q],s.`
			`s:`*act*`:s :- r.`
				`failed(t).`
				`p:`*ans*`:p.`
				`q:`*ans*`:q.`
				`failed(r).`
				`failed(s).`

	$t \xrightarrow{neg} p \longrightarrow q \xrightarrow{neg} r$	$t \xrightarrow{neg} p \longrightarrow q \quad r$	$t \xrightarrow{neg} p \quad r \longrightarrow s$	
(a)	(b)	(c)	(d)	(e)

Figure 1: A program, selected SLG systems, and their *SDG*s for the query `?- t`.

clause with status *status* of an SLG tree of subgoal S as S : *status* : A `:-` *Body*, where the head of the clause A is an *answer_template*. In the first stage of the evaluation, the system in Figure 1(b) is produced. In this system, the leftmost literal of the first four clauses has been selected. Clauses waiting for answers to be returned to them have the status *act* (*active*), those waiting for completion of a negative literal have the status *susp* (*suspended*). As can be seen from its accompanying *SDG*, 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. The negative literals involved in the independent *SCC* are delayed, producing the system in Figure 1(c) in which delayed literals are shown in a delay list. At this stage the clause `q:`*act*`:q :- [¬r]` has no more literals to resolve and becomes a conditional answer, `q:`*ans*`:q :- [¬r]` (through the NEW ANSWER operation). This conditional answer can be returned to the clause for `r` or to that for `p`. We assume that it is first returned to `p` (through the POSITIVE RETURN operation) producing another conditional answer, and to `r` which then calls `s` and afterwards, recursively, itself. The resulting system is shown in Figure 1(d). At this stage, `s` and `r`, which are in a positive loop, have been completely evaluated and can be completed, with both `s` and `r` failed. Upon the completion of `r`, the answer `q:`*ans*`:q :- [¬r]` should be made unconditional, since `¬r` is now true. SLG addresses situations such as this through the SIMPLIFICATION operation which simplifies away delayed literals of conditional answers. Using a SIMPLIFICATION operation, `¬r` is removed from the delay list for `q` making the answer `q` unconditional. This unconditional answer enables a further SIMPLIFICATION operation in the answer for `p`. Finally, since `p` has succeeded the suspended clause for `t` fails (through a NEGATIVE RETURN operation), leading to the final system of Figure 1(e). ■

We summarize the actions of Example 1.1 in handling negative loops: DELAYING can be thought of as a mechanism for dynamically changing the left-to-right computation rule. As seen from the *SDG* of Figure 1(c), delaying a literal may also avert dependencies through negation. As shown by the failure of `r`, DELAYING may also allow a clause that creates a cyclic negative dependency to fail based on the falsity of a literal in its body. Such failures trigger SIMPLIFICATION operations. In general, delayed literals that are true should be deleted from answers, and answers with delayed literals that are false should be deleted from the system.

When resolution occurs for an *active* or *suspended* node, any variable bindings accumulated in the answer head are propagated through unification. However, delayed literals themselves are not propagated. Instead, positive delayed literals are created as placeholders for propagating truth values of delayed ground negative literals. In Figure 1(d), the active clause for p uses the placeholder q, rather than propagating the delayed list [$\neg$r]. This use of positive delayed literals guarantees a polynomial representation of answers of queries that may otherwise have an exponential number of answers with only ground negative literals in answer bodies (see [2] for an example of that situation).

In a final system, each delayed literal in the body of an answer corresponds to the head of some *conditional* answer of that subgoal; i.e. an answer with delayed literals in its body. The set of answers resulting from the evaluation of a query Q with respect to a program P can be viewed as a *residual program* of the program-query pair $\langle P, Q\rangle$. This residual program consists of all answers of the query plus the rules of all head atoms upon which the answers of the query depend directly or indirectly through delayed literals in answer bodies.

Implementation Overview Example 1.1 brings up a number of issues that are involved in implementing the well-founded semantics in a WAM-style framework.

Suspending and Resuming Computations In a tabled evaluation, the derivation of an answer may be asynchronous with its consumption by an active node, and in a non-stratified program the resolution or DELAYING of a negative literal may be asynchronous with its completion or the derivation of a conditional answer. A mechanism must therefore be provided to *suspend* computation of active clauses, and to *resume* a computation at some point after an answer is derived, or after a non-successful subgoal is completed. [13] explains how a *forward trail* can be used to resume environments, and also discusses how suspend and resume affect memory management. It also discusses instructions for definite tabling which are presented in passing below: the tabletry instruction which performs much of the SLG NEW SUBGOAL operation, and the answer_return, new_answer, and completion SLG-WAM instructions.

Determining SCCs and Resolving Negative Literals Mechanisms are needed to determine when a subgoal has been *completely evaluated*: when every possible answer for that subgoal has been derived. The incremental stack-based algorithm of [13] was altered in [10] to efficiently compute fixed-order stratified programs. This involved a mechanism to detect negative loops by lazily constructing parts of *SDG*s. Section 2 discusses how this negative loop check is extended to normal programs, along with the tnot/1 predicate that resolves negative literals in normal programs.

Table Access Routines Table access routines must be developed to check whether an evaluation has called a subgoal or derived an answer, and to insert the subgoal or answer if not. A previous paper presented a solution using tries that was effective for answers with empty delay lists [7]. Section 3 addresses how tries can be extended to store delay lists and to support SIMPLIFICATION.

Manipulating Delay Lists Delay lists occur not only in answers, but in the heap for *active* and *suspended* clauses. Section 3 discusses the representation of delay lists, and their maintenance under backtracking, suspend and resume.

Simplification Operations The table space must also be extended to efficiently perform simplification when the truth value of a literal in the delay list of an answer becomes known, and so that these SIMPLIFICATION operations can be propagated if necessary. Section 4 discusses SIMPLIFICATION in detail.

2 Handling Negative Literals

Mechanisms for detecting loops through negation, and for executing non-stratified negation are based squarely on features of an engine that evaluates *fixed-order dynamically stratified programs* ([13, 10]). *Dynamically stratified* programs were introduced in [6], and differ from most stratification formalisms in that recursive components are determined in the course of a transfinite fixpoint 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* stratum. [11] introduces a natural restriction of dynamic stratification to a fixed-order computation rule, along with a restriction of SLG, SLG_{RD} (SLG with Reduced Delaying) which delays negative subgoals only if they have no left-to-right derivation. Specifically,

Theorem 2.1 ([11]) *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_{RD}. In principle, if a rule instance is used in a left-to-right well-founded evaluation, a literal of that rule should be 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 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.

Detection and handling of loops through negation As shown by Example 1.1, 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, in the SLG-WAM the same dependency mechanism that detects completion of tabled subgoals can also be used to detect SCCs containing loops through negation. First, a *tabled parent register* is added to the engine, and stored in choice points. The use of this register, along with other information about environments for calls to positive and negative subgoals, embeds the SDG in the WAM Choice Point stack, as described in [10]. The completion instruction starts with a stack-based approximation of the SDG. If there is a possibility that there is a negative loop involving a subgoal about to be completed, the completion instruction lazily constructs a subgraph of the SDG to find the independent SCC I, and determine if there is a negative loop among subgoals in I. Otherwise, the completion instruction can complete subgoals based on their stack-based approximation.

Resolving Negative Literals in a Well-Founded Model The predicate `tnot/1` resolves negative literals in the SLG-WAM. Figure 2 sketches its implementation using low-level builtins. When the negation involves a completed ground subgoal, the builtin 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 the subgoal is still incomplete. Then, the negation_suspend/1 builtin, introduced in [10], suspends the current computation, which will later be reinvoked when more is known about the truth value of the subgoal. At the engine level, the suspension is performed by placing a *negation suspension frame* onto the choice point stack to save the execution environment for the suspended computation; this is done in a manner similar to the way computations are suspended on creating active nodes. In addition to the usual information of WAM environments, this frame also contains a pointer to the subgoal that is being

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

Figure 2: The implementation of tabled negation (tnot/1) for normal programs.

negatively suspended, and a *Status* field that will be later filled by the completion instruction to indicate whether the literal is delayed.

Three outcomes can occur for a negatively suspended literal; it can succeed, fail, or its subgoal can complete with conditional answers only. Negation suspensions of subgoals that succeed are removed so that the tnot_resume/1 builtin will never be called and the associated suspended clauses effectively fail. Otherwise, the completion instruction will create *negation resume choice points* chained together on the choice point stack. Upon backtracking to these choice points, the engine resumes the environment of the suspended computation (using the negation suspension frame), and continues execution of the suspended clause.

Operationally, subgoals are resumed upon: (1) determining that the subgoal is completely evaluated, and (2) delaying the subgoal before its completion. In the first case, the subgoal will either be failed, or will have conditional answers only. If the subgoal is failed, tnot_resume/1 will do nothing, effectively resolving away a literal for a failed subgoal. If the subgoal has conditional answers, tnot_resume/1 will add the conditional negative literal to the delay list. In the second case, the negation_resume instruction was scheduled to break a loop through negation, and the *Status* field of the negation suspension frame is marked as *delayed*. Once again, tnot_resume/1 will add the negative literal to the delay list, as if it were completed with conditional answers.

3 Delaying and Handling of Conditional Answers

Delay lists and Delay elements As discussed in Section 1.1, delay lists represent unevaluated or undefined literals in the body of a clause. When and if the truth value of these literals becomes known, SIMPLIFICATION operations will either remove the literals from the delay list, or dispose of the clause. The representation of delay lists in the SLG-WAM must therefore support several operations: adding elements to the delay list in the course of resolution; copying the delay list into the table, and SIMPLIFICATION.

At an operational level, a delay list can be thought as an extra argument in tabled predicates. At subgoal call the argument is initialized to a variable representing an empty delay list; during resolution the variable may get bound to a list of literals. In terms of the table, the binding of the delay argument is treated as part of an answer and copied into the table if the derivation is successful. Obtaining a representation of the elements of delay lists, *delay elements*, to support the above operations is complicated by changes made to the variable bindings of a literal in the course of resolution. Consider the operational detail of delaying a literal for a clause C. First, a selected literal is called during the course of computation: for purposes of illustration let this literal be $p(X,Y)$. Then, suppose $p(X,Y)$ has a conditional answer: $p(a,Y)$: answer resolution propagates the binding $X = a$ into C, and $p(a,Y)$ is added to the delay list. Finally, suppose further resolution of C adds the binding $Y = b$. How should the delayed literal be represented within the delay list? As $p(X,Y)$, as $p(a,Y)$ or as $p(a,b)$? Efficient support of the above operations requires the delay element be represented by a triple: $\langle Subgoal\_Id, Literal, Answer\_Substr\_Id\rangle$ where:

Subgoal_Id indicates delayed literal as it was when it was a subgoal *e.g.* $p(X,Y)$;

Literal indicates the delayed literal with all bindings accumulated during the course of resolution *e.g.* $p(a,b)$; and

Answer_Subst_Id either indicates the answer substitution if the delayed literal is positive, *e.g.* $p(a,Y)$, or contains the marker $\neg$ if the delayed literal is negative.

In terms of data structures, *Subgoal_Id* is a pointer to a subgoal frame (a data structure in the table containing information on tabled subgoals: see Section 4.4), while *Answer_Subst_Id* is a pointer to a node of the answer trie (see below) that uniquely identifies the answer substitution. A null pointer is used to denote the $\neg$ indicator. Note that the only part of the *Literal* that has to be explicitly stored is the substitution for the variables in the answer, if any. Also note that even though SIMPLIFICATION may delete elements from the delay lists in tabled answers it will never change the bindings for these delay elements. For notational convenience, we will present answers by indicating their *answer substitution*; i.e. a set of substitutions for the variables in the subgoal, since this is how tabled answers are represented in the SLG-WAM [7]. Also, we will usually use the shorthand $\langle\neg L\rangle$ to denote a negative delay element of the form $\langle G, L, \neg\rangle$, since in a non-floundering evaluation, G must be ground and so $G = L$. Variations of the program in the following example will be used throughout this paper to describe the data structures used to implement the DELAY operations.

Example 3.1 Consider the evaluation of query `?- p(X)` against the program of Figure 3. This requires delay of both positive literals, and of negative literals. No simplification is possible, however. This query produces the subgoals and answers also shown in Figure 3. Both subgoals and answers are shown in chronological order of generation when a "depth-first" scheduling strategy is used. The answers

```
:- 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)).
```

Subg	*Answer*	*Ans_Id*
$p(X)$	`g(b)` []	$p(X)_1$
	`g(c)` $[\langle\neg \mathtt{p(g(c))}\rangle]$	$p(X)_2$
	`f(a)` $[\langle p(X), \mathtt{p(g(c))}, p(X)_2\rangle]$	$p(X)_3$
$p(g(c))$	$[\langle\neg \mathtt{p(g(c))}\rangle]$	$p(g(c))_1$
$p(g(b))$	[]	$p(g(b))_1$

Figure 3: A program requiring delay and the tables created for the query `?- p(X)`.

are shown as split into two parts; the *answer substitution* (e.g. $g(c)$), 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 (e.g. answers $p(X)_2$, 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. How the SLG-WAM supports this many-to-many relationship efficiently is illustrated below. ■

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 3.2 We extend the program of Example 3.1 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 unconditional answers of `p/1`. Figure 4 shows the extension, and some of the tables that are created by the evaluation of queries `pt(X)` and `u(X)`. The answers to `pt(X)` should be self-explanatory. Contrast, however, delay lists of

```
:- table pt/1, u/1.

pt(X) :- p(X).

pp(X) :- p(X).

u(X) :- pp(X), ¬ pt(X).
```

Subg	*Answer*	*Ans_Id*
$pt(X)$	`g(b)` []	$pt(X)_1$
	`g(c)` $[\langle p(X), \mathtt{p(g(c))}, p(X)_2\rangle]$	$pt(X)_2$
	`f(a)` $[\langle p(X), \mathtt{p(f(a))}, p(X)_3\rangle]$	$pt(X)_3$
$u(X)$	`g(c)` $[\langle p(X), \mathtt{p(g(c))}, p(X)_2\rangle, \langle \neg \mathtt{pt(g(c))}\rangle]$	$u(X)_1$
	`f(a)` $[\langle p(X), \mathtt{p(f(a))}, p(X)_3\rangle, \langle \neg \mathtt{pt(f(a))}\rangle]$	$u(X)_2$

Figure 4: Tables created for the queries `?- pt(X).` and `?- u(X).`

the answers of the `u(X)`. These lists contain delay literals `p(g(c))`, and `p(f(a))` which have been propagated through the non-tabled `pp/1`. ■

Delay list implementation and maintenance Delay lists are not only represented in the table, but also in the heap as a Prolog list, to which delay elements can be added in constant time. Once an answer is derived, the delay list is copied from the heap, interned, and stored in the answer table (all through the new_answer SLG-WAM instruction).

The WAM maintains information about the state of the computation in global registers. To keep track of the current delay list, we introduce a new register, *dreg*. Accordingly, all choice point frames used for program clause resolution, are expanded with an extra field that records the value of *dreg* at the time of choice point creation. Let S be the nearest tabled ancestor of a subgoal being evaluated. Then *dreg* is updated to point to the delay list of the generator subgoal for S, under the following three conditions. (1) When a new tabled subgoal is called or backtracked into, *dreg* is set to point to an empty delay list. (2) As is done for all other WAM registers, *dreg* is restored to its earlier value in the event of backtracking. (3) The value of *dreg* is also updated in forward continuations of tabled predicates to reflect the resolution of a conditional answer in SLG. The introduction of *dreg* requires changes to many SLG-WAM instructions, which are described in the full version of this paper.

Extending the SLG-WAM Table Space for Delay As discussed in [7], the table space of XSB is built around tries since these structures can avoid repeated rescanning of common subterms within subgoal and answer tables. However, the tries of [7] 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 SIMPLIFICATION, and, since delay lists do not need to be copied out of the table, it should not impose an overhead on the return of answers

The SLG-WAM associates with each tabled subgoal set an *answer trie*, which is organized using answer substitutions. Nodes of the trie consist of four fields: *atom*, *child*, *parent*, and *sibling*. The *atom* field records information about the answer substitutions. The outgoing transitions from a node are traced using its *child* pointer and by then following the list of *sibling* pointers of this child. An answer substitution may have many delay lists associated with it, corresponding to various partial deductions. We therefore access delay lists through the leaves of the answer substitutions, effectively factoring the answer substitutions. Access from an answer substitution η_S is through a *DelayInfo* record that contains information a pointer to a list of delay lists upon which an η_S is conditional, back pointers to positive delay elements *conditional on* η_S, and a pointer back to the subgoal with which η_S is associated. Naturally, leaf nodes of unconditional answer substitutions have null *child* pointers. Figure 5 provides a close look at these trie data structures by presenting the subgoal trie for predicate `p/1` of Example 3.1, and answer tables for two of these subgoals. The fields of the answer substitution trie nodes are shown in the order: *atom, child, parent,* and *sibling* (*parent* pointers are used to return answers, as explained below).

Subgoals:

$S_1 : p(X)$
$S_2 : p(g(c))$
$S_3 : p(g(b))$

Answers for $p(X)$:

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

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

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

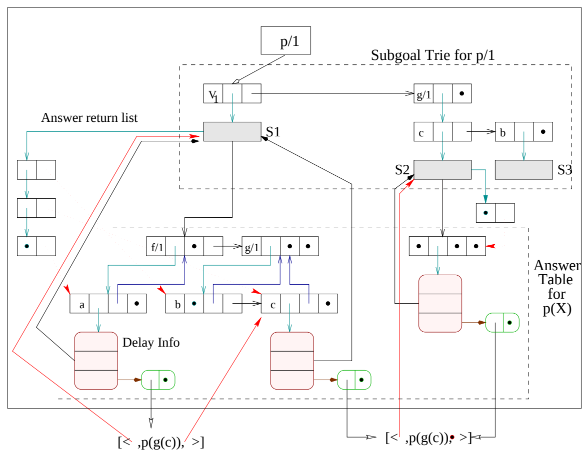

Figure 5: Subgoal and answer tables (using tries) for predicate `p/1` of Example 3.1.

Both delay elements and delay lists are interned in global data structures since, as illustrated by Examples 3.1 and 3.2, a given delay element may appear in more than one delay list. These are called *Interned Delay Elements (IDEs)* and *Interned Delay Lists (IDLs)* respectively. Interning elements usually saves space. However, the main advantage of interning is that simplification operations can now be shared: for example, one simplification operation for an interned delay element *ide* suffices to simplify all delay lists that contain the *ide*. Section 4 fully discusses this point.

A node may be delayed positively on a conditional answer, or negatively on a (ground) negative literal. The structure of IDEs reflect this fact. Positive IDEs are grouped together based on their *Answer_Subst_Id* field (which identifies their answer substitution), while negative IDEs are grouped based on their *Subgoal_Id*. We can therefore refer to the positive IDEs of an answer substitution, or the negative IDEs of a subgoal.

While these data structures are detailed at the level of instruction execution in Section 4.4, we broadly summarize their uses here. First there is a one-to-many relation between an answer substitution η_S and its (interned) delay lists. 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. Next, because the same delay list may occur for many answer substitutions, delay lists are interned as IDLs so that a single simplification operation will suffice for many delay lists (cf. in Example 3.1, the delay lists containing $\langle \neg \mathtt{p(g(c))} \rangle$). These IDLs in their turn, consist of delay elements which are interned as IDEs so that they can be grouped together. IDEs contain pointers back to the delay lists that contain them, explicitly maintaining a relation used to propagate simplification (cf. in Example 1.1, the propagation of the simplification of $\neg \mathtt{r}$ through the answer for q to the answer for p).

Answers from completed subgoals may be returned by backtracking through the answer trie [7]. However, since answers may be placed anywhere in the answer trie, (for example, in Figure 5 the answer substitutions for $p(X)$ are derived in the order p(g(b)), p(g(c)), and p(f(a))) the SLG-WAM maintains the *answer return list* to return answers to incomplete subgoals. This list chains together the leaves of the answer substitution trie (in chronological order) effectively avoiding the traversal of delay information when returning answers. To return answers accessed through the *answer return list*, every node of the answer substitution trie maintains a back pointer to its *parent* node.

4 Simplification

Simplification of delayed literals is necessary so that when subgoals are completed, a ground instance of a subgoal S is true in the well-founded model if and only if it is an instance of a unconditional answer, and a subgoal S is false if and only if it has no answers, conditional or otherwise.[2]

4.1 Simplification Principles

Example 1.1 provides instances of simplification of both positive and negative delayed literals. As mentioned above, not all delayed literals are undefined in the well-founded model, and some may become known to be true or false later during the process of SLG evaluation. When their truth value does become known, the literal or clause may be simplified away, but until then answers conditional on the delayed literal may be used to derive further answers and may unnecessarily open up the search space. We therefore adhere to the following principles:

Principle I *Conditions for simplification of delayed literals 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 substitution.*

Principle III *Unsupported answer substitutions (see Section 1.1) should be removed from the answer tables as soon as possible.*

4.2 Events that trigger simplifications

In order to follow Principle I, that is roughly, to simplify as soon as possible, the SLG-WAM performs simplification whenever the truth value of a subgoal or answer becomes known. SIMPLIFICATION is driven by the following three events:

Derivation of an unconditional answer signifies that all instances of the answer will be true in the well-founded model of the program. Unconditional answers for a subgoal S can be produced: (1) through program or answer clause resolution before the completion of S; the new answer is then inserted in the answer table by executing a new_answer instruction, and (2) through simplification of an existing conditional answer by removing the last delay element from its delay list. Note that in the latter case, the subgoal S might have already been completed.

Let A be a new unconditional answer of a subgoal S, and let η_S be its answer substitution. The derivation of A removes all conditional answers with η_S from the system (Principle II), and can trigger simplification of both positive and negative delay elements; (all positive elements of η_S become true, while negative elements of S become false). Note that simplification of positive delay elements takes place

[2] Technically, the truth of the above statement requires the implementation of the SLG ANSWER COMPLETION operation. Our experience has shown that this operation is rarely needed in practical programs, and its omission does not affect the soundness of our results.

only when conditional answers with answer substitution η_S have already been (conditionally) returned to consuming subgoals.

Completion of a subgoal with no answers The subgoal fails. As a result, all negative elements delayed on that subgoal now succeed, and can be removed from the delay lists that contain them. At an operational level, a step of the completion instruction checks whether there are any negative IDEs for the failed subgoal, and if so, triggers simplification. Note that no direct simplification of positive delay elements can be initiated by a failing subgoal.

Deletion of an unsupported answer substitution Simplification can change the status of an answer substitution η_S for a subgoal S from supported to unsupported when the last conditional answer is removed due to one of its delay elements becoming false. In this case, following Principle III, the engine must address two types of situations. First, all answers delayed (positively) on η_S must be removed. This can cause further simplifications to occur in answers delayed on these conditional answers. Secondly, if S is already marked as completed and η_S was its last answer substitution, then now S fails. Simplifications should take place similar to those initiated by completion of a subgoal with no answers.

4.3 Simplification procedures

The simplification procedures perform two types of actions:

Deletion of a delay element from delay lists is directly applicable either when a subgoal fails, or when an unconditional answer is derived for a conditional answer substitution. In this case, all delay lists that contain the simplifiable delay element are shortened by one element. More specifically, when a subgoal S fails all negative delay elements whose $Subgoal\_Id$ is S are removed from the lists where they appear. Similarly, all positive delay elements whose $Answer\_Subst\_Id$ is η_S are removed upon the derivation of an unconditional answer substitution η_S.

Deletion of unsupported answers takes place when an answer becomes unsupported because one of its delay elements is found to be false. This happens either when an unconditional answer is derived for a subgoal S (in which case all negative delay elements whose $Subgoal\_Id$ is S are false), or when a conditional answer substitution η_S changes status to unsupported (in which case all positive delay elements whose $Answer\_Subst\_Id$ is η_S become false). Note that because of the factoring of answers on their answer substitution, deletion of answers in this case may involve deletion only of delay lists. In the SLG-WAM, the following four procedures implement the abstract operations specified above:

simplify_neg_fails($Subgoal$) Removes the negatively delay elements of a failed subgoal from delay lists.

simplify_pos_unconditional($AnswerSubstitution$) Removes the positively delay elements of an unconditional answer substitution from delay lists.

simplify_neg_succeeds($Subgoal$) Deletes answers negatively delayed on a successful subgoal.

simplify_pos_unsupported($AnswerSubstitution$) Deletes answers positively delayed on an unsupported answer substitution.

Figure 6 contains representative pseudocode; the remaining procedures are similar and are presented in the full version. Note that all procedures that implement the simplification operations are mutually recursive. This reflects the fact that

Procedure simplify_pos_unconditional($AnswerSubstitution$)

foreach $PIDE$ in the IDE list of $AnswerSubstitution$
  foreach IDL in which this $PIDE$ appears
    $IDL := IDL - \{PIDE\}$; /* remove the positive IDE from the delay list */
    if ($IDL = [\,]$) then /* the DL is empty; answers are now unconditional */
      foreach answer A having answer substitution η_S and IDL as delay list
        i. Initiate a simplify_pos_unconditional(η_S) simplification;
        ii. if (the subgoal S containing A now succeeds) then
            Initiate a simplify_neg_succeeds(S) simplification;

Figure 6: Pseudocode for one of the four simplification procedures.

simplification operations usually have a cascading effect. To efficiently implement both categories of simplification operations mentioned above, we need information linking each interned delay element E to the set of interned delay lists that contain E, and this information is described in the following section.

4.4 SLG-WAM data structures that support simplification

Clearly, the efficiency of the simplification operations heavily depends on the data structures that support them. For example, to efficiently perform simplification triggered by the derivation of an unconditional answer for a conditional answer, the answer should be associated with its delay elements. The interned delay elements need to be associated with the delay lists that contain them, and in turn, each of these delay lists needs to be associated with the set of answers in which they are contained. Figure 7 shows the persistent data structures added to the WAM, and their relationships. Information about subgoals is recorded in *subgoal frames*;

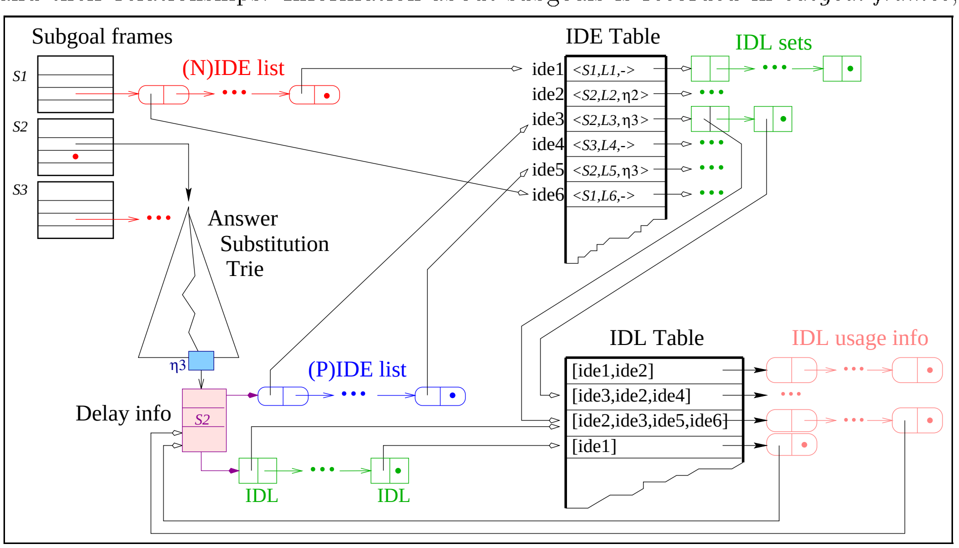

Figure 7: Relationships between Persistent SLG-WAM Data Structures

through which, a subgoal S is associated with an answer trie, the answer return list, and the list of negative IDEs of S. Note the $DelayInfo$ record of the conditional answer substitution η_3 contains a pointer to a list of positive IDEs of η_3, the backpointer to the subgoal for η_3, and the set of delay lists associated with η_3.

The delay lists and elements are interned and stored in global tables, the *Interned Delay List (IDL) Table*, and the *Interned Delay Element (IDE) Table*, respectively.

The information stored in the entries of these tables is actually in the form of pointers (to subgoal frames, answer substitutions, 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. Also, each entry of the IDL Table is associated with the conditional answer substitutions that point to this entry.

Entries in the IDE Table, once inserted, appear uniquely. Note, however, that through simplification, the IDL Table can contain duplicates (for example, by simplifying $ide2$ through a simplify_pos_unconditional simplification). Currently, XSB does not collapse these elements. Also, through simplification, IDLs may become empty (for example failure of $S1$ causes simplification of $ide1$, so that the last entry of the IDL Table becomes empty). Empty IDLs are removed from the table.

5 Performance

Overhead for the Evaluation according to the WFS As reported in [9], the SLG-WAM performs significantly better than other deductive database systems for in-memory computation of a variety of recursive queries. For definite programs, the SLG-WAM adds a 10-15% overhead to the WAM [13], while [10] reports that the additional overhead for the evaluation of fixed-order stratified programs is less than 1%. The full version of this paper also measures the cost that the additions for Well-Founded Semantics add to previous generations of engines, and finds that this overhead is minimal as well. In other words, the ability to compute the Well-Founded Semantics adds about 15% to Prolog execution (mostly for mechanisms to support tabling for definite programs), and adds about 2% overhead to the tabling engine for definite programs.

Performance of Tabled Negation To our knowledge, the `win/1` program is still the only standard benchmark in the well-founded semantics literature. Its SLG and SLD versions, are shown below. SLD resolution is sufficient for acyclic `move/2` graphs (like chains and trees). In this case the well-founded model is two-valued.

```
win(X) :- move(X,Y), tnot(Y).    win(X) :- move(X,Y), not(Y).
```

Figure 8 contains timing results (in seconds) for executing 50 iterations of the query `?- win(1), fail.` over different data structures. For chains and trees, each subgoal has at most one answer. For the case of the chain, the performance of SLG is comparable to SLDNF (less than 2 times slower). In the case of the complete binary tree, however, SLG is much slower. This is because tabled negation has to fully evaluate negative literals before completing them, and consequently, it traverses the entire search space which is exponential in the height of the tree (see [9]).

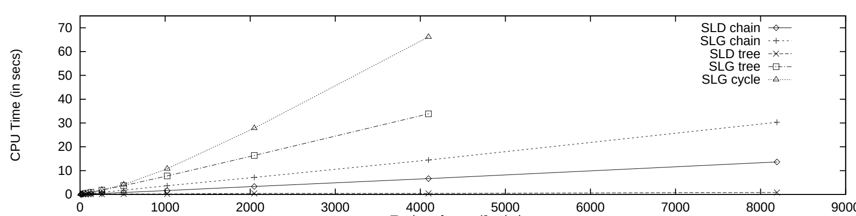

Figure 8: Performance of the `win/1` program over different data-structures.

The SLD version of `win/1` will not terminate if `move/2` contains cycles. Under the WFS, all answers of `win/1` over a cyclic data structure are *undefined*. The evaluation of the tabled version of `win/1` over a cyclic data structure requires the loop checking mechanism described in Section 2, and exact SCC detection to decide whether

DELAYING is needed. 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 and no SIMPLIFICATION is possible, performing exact SCC detection and interning information to support simplification, imposes an unnecessary overhead to the evaluation of this program.

6 Discussion

Tabling radically expands the practical uses of logic programming, even for definite programs. Indeed, many new uses have already been demonstrated. It has been shown in [3] that simple Horn representations of logic and functional program analysis methods, as executed by the SLG-WAM, can compete with or even surpass, special-purpose program analysis routines written in C. Furthermore, initial experiments indicate that, under the reduction to Horn clauses specified in [15], the SLG-WAM can efficiently perform model checking in the modal μ-calculus.

These applications concern definite programs only. We believe that the recent expansion of the SLG-WAM to compute the well-founded semantics will serve as a computational basis for applications in non-monotonic reasoning. For instance, the well-founded semantics, (perhaps expanded to include explicit negation) has been shown to be expressive for reasoning using non-monotonic inheritance, for diagnosis, for reasoning about actions [5], and as underpinnings for the behavior of collaborative agents [4]. The practical implementation of applications such as these forms the next avenue of research for tabling in logic programming.

References

[1] R. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In *Proceedings of the International Symposium on Logic Programming*, pages 199–219, 1993.

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

[3] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *ACM PLDI*, 1996.

[4] P. M. Dung. An Argumentation Semantics for Logic Programming with Explicit Negation. In *Proceedings of the 10th ICLP*, pages 616–630, 1993.

[5] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non-monotonic reasoning with logic programming. *JLP*, 17(2, 3, and 4):227–263, 1993.

[6] T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the 8th ACM PODS*, pages 11–21, 1989.

[7] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient tabling mechanisms for logic programs. In *Proceedings of the 12th ICLP*, pages 687–711, 1995.

[8] K. A. Ross. A procedural semantics for well-founded negation in logic programs. *JLP*, 13(1):1–22, 1992.

[9] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD Conference*, pages 442–453, 1994.

[10] K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Fixed-Order Dynamically Stratified Programs. In *Proceedings of the 13th CADE*, LNAI, 1996.

[11] K. Sagonas, T. Swift, and D. S. Warren. The Limits of Fixed-Order Computation. In *Proceedings of the First International Workshop on Logic in Databases*, LNCS, 1996.

[12] P. J. Stuckey and S. Sudarshan. Well-Founded Ordered Search. In *Proceedings of 13th Conference on FST-TCS*, LNCS, pages 161–171, 1993.

[13] T. Swift and D. S. Warren. An abstract machine for SLG resolution: Definite Programs. In *Proceedings of the ILPS*, pages 633–652, 1994.

[14] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

[15] S. Zhang, S. A. Smolka, and O. Sokolsky. On the parallel complexity of model checking in the modal mu-calculus. In *Proceedings of LICS'94*, pages 154–163. 1994.