An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs

KONSTANTINOS SAGONAS
Katholieke Universiteit Leuven
and
TERRANCE SWIFT
University of Maryland at College Park

SLG resolution uses tabling to evaluate non-floundering normal logic programs according to the well-founded semantics. The SLG-WAM, which forms the engine of the XSB system, can compute in-memory recursive queries an *order of magnitude faster* than current deductive databases. At the same time, the SLG-WAM tightly integrates Prolog code with tabled SLG code, and can execute Prolog code with minimal overhead compared to the WAM. As a result, the SLG-WAM brings to logic programming important termination and complexity properties of deductive databases. This article describes the architecture of the SLG-WAM for a powerful class of programs, the class of *fixed-order dynamically stratified programs*. We offer a detailed description of the algorithms, data structures, and instructions that the SLG-WAM adds to the WAM, and a performance analysis of engine overhead due to the extensions.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.4 [**Programming Languages**]: Processors—*compilers*; *interpreters*; *optimization*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*logic programming*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming*; *nonmonotonic reasoning and belief revision*; *resolution*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: WAM, tabling, memoing, SLG, Prolog, stratification theories

---

Most of the research reported in this article was conducted while both authors were affiliated with the Computer Science Department of SUNY at Stony Brook. Preliminary papers presenting initial designs of this abstract machine appeared in the following conference proceedings. In *Proceedings of the 1994 International Symposium on Logic Programming* (The MIT Press): "An Abstract Machine for SLG resolution: Definite Programs" pp. 633–652; and "Analysis of SLG Evaluation of Definite Programs" pp. 219–235. In *Proceedings of the Thirteenth Conference on Automated Deduction*, (Springer-Verlag): "An Abstract Machine for Fixed-Order Dynamically Stratified Programs".
Authors' present addresses: K. Sagonas, Departement Computerwetenschappen, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium. E-mail: kostis@cs.kuleuven.ac.be. T. Swift, Department of Computer Science, University of Maryland at College Park, A.V. Williams Building, College Park, MD 20742, U.S.A. E-mail: tswift@cs.umd.edu.

1. INTRODUCTION

The lack of termination of SLDNF resolution, even on simple programs such as transitive closure, is a well-known problem. So is the fact that SLDNF may repeatedly evaluate the same subgoal, leading to unacceptable complexity and performance. Much research has aimed at addressing these issues. Until recently, the majority of such research has focussed on set-at-a-time strategies such as those based on magic-style evaluation. Tabled resolution offers an alternative approach to the limitations of SDLNF. One such formulation, *SLG resolution* (Linear resolution with Selection function for General logic programs [Chen and Warren 1996]), offers advantages in its ability to evaluate queries to programs in accordance with the well-founded semantics [Van Gelder et al. 1991], and to do so with polynomial data complexity (as defined in [Vardi 1982; Van Gelder et al. 1991]) if these queries are ground and the programs restricted to Datalog with negation.

Despite the limitations of SLDNF and the availability of newer evaluation methods such as magic and tabling, Prolog is still by far the most popular logic programming language. The persistent popularity of Prolog arguably arises from two causes. First, robust implementations are available for Prolog which are suitable for practical, and even commercial purposes. Secondly, Prolog offers a number of well-known programming constructs, along with a proven programming environment.

One would like to have the best of both worlds: to handle termination and negation according to the well-founded semantics, but with the speed of Prolog and within its environment. Research on the XSB system [Sagonas et al. 1994] is geared exactly towards these goals. The termination and complexity properties of XSB have been central to its use for program analysis [Dawson et al. 1996; Codish et al. 1996], for natural language processing [Larson et al. 1996; Larson et al. 1997] and for concurrency analysis [Ramakrishna et al. 1997]. Furthermore inclusion of these termination and complexity properties adds little performance overhead to the engine underlying most Prolog systems, the WAM (Warren Abstract Machine [Warren 1983; Aït-Kaci 1991]). As a result, XSB has also been used by thousands of people around the world to develop Prolog as well as tabled logic programs.

XSB is based on an extended WAM-style engine, the SLG-WAM. This article describes the data structures, algorithms, instruction set and performance of the SLG-WAM on the important class of *left-to-right dynamically stratified* (LRD-stratified) programs [Sagonas et al. 1996]. This class properly includes other stratification classes such as (left-to-right) modular stratification [Ross 1994], and may be the largest class of normal logic programs that can be evaluated using a fixed-order computation rule.

The structure of the article is as follows. Section 2 reviews a variant of SLG suitable for definite programs. Section 3 presents in detail an abstract machine for definite programs. Sections 4 and 5 define the class of LRD-stratified programs and tabling operations needed to evaluate this class; Section 6 presents extensions to the definite engine that are needed to evaluate this class of programs. Finally, Section 7 presents performance results on the overhead incurred by the tabling extensions and on the speed of tabled evaluation compared to SLDNF evaluation.

We point out that, while we use the terminology of SLG, the differences between SLG and other table-based evaluation strategies such as OLDT of Tamaki and Sato [1986] or SLD-AL of Vielle [1989] are minor for definite programs. We also note that while much of this article assumes a knowledge of the WAM, whenever possible we have tried to present algorithms and data structures of the SLG-WAM in a manner that assumes only a modest familiarity with the WAM. We thus hope that this detailed description will enable other implementors to incorporate various types of tabling in their own systems.

Related Implementations of Tabling. It is natural to ask whether engine modifications are really required to implement tabling, or whether an SLG interpreter (or preprocessor) could be written in Prolog. If so, then Prolog itself could compute SLG. Such interpreters can and in fact have been written by using Prolog's dynamic database as a table store, (for instance the *extension tables* of Dietrich [1987] preprocess tabling operations for definite programs), but their speeds and robustness have usually turned out to be unacceptable for general programming. As will be described in Section 2, certain tabled subgoals resolve against answers rather than against program clauses. The branches of the search tree corresponding to these subgoals must either be maintained or reconstructed. Subgoals that are to be resolved against answers must be retained until the fixpoint is reached: until all applicable answers have been derived and resolved against the subgoals. Likewise, newly derived answers must be queued to resolve against subgoals arbitrarily far away in the search tree. These actions require scheduling and suspension features that are not easily implementable without appropriate extensions to the WAM. A recent alternative approach implements SLG by transforming a program using a continuation passing style and then employs foreign function calls from SICStus Prolog to access tables [Ramesh and Chen 1997]. This approach has the advantage of portability — foreign function calls are less system-dependent than engine redesign — but may compromise on speed, flexibility, and robustness.

2. BASIC DEFINITIONS AND NOTATION OF SLG RESOLUTION

In this section we present the terminology and basic definitions of SLG resolution [Chen and Warren 1996]. We do so through a simplified version which is sufficient to model finite computations of definite and fixed-order stratified logic programs. In general, we assume the usual terminology of logic programs from [Lloyd 1987]. We also assume that programs are evaluated using a fixed left-to-right literal selection strategy. We define subgoals as atoms, and treat variant atoms as identical. In our version of SLG a *tabled program* is a program augmented with tabling declarations of the form

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

where p_i is a predicate symbol and n_i is an integer. These declarations ensure that all queries to the predicate p_i of arity n_i will be executed using SLG. Other predicates are implicitly assumed as *non-tabled* in which case SLD resolution is used for queries to these predicates. Slightly abusing terminology, we will speak of tabled subgoals and literals as well as tabled predicates. Also for simplicity, if a literal $(not)S$ is selected for resolution in node of an SLG tree, we will speak of S as the selected subgoal of a node.

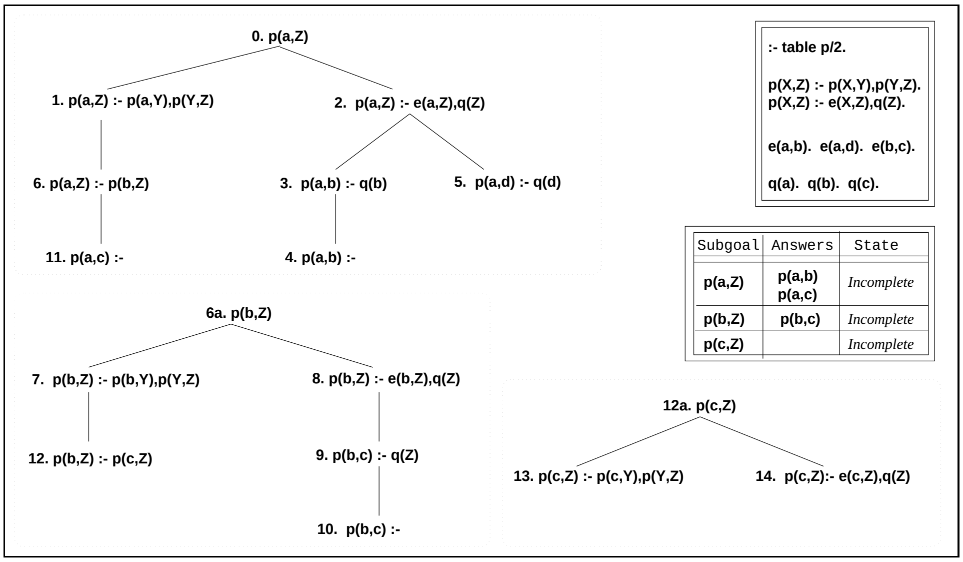


Fig. 1. Program and SLG System for the query `?- p(a,Z).`

Tabling methods such as SLG evaluate programs by maintaining tables of subgoals and their answers, and by resolving repeated occurrences of subgoals against *answers* from the table rather than against program clauses. By resolving answers in this manner, rather than repeatedly using program clause resolution as in SLD, SLG avoids looping and thus terminates for all programs with the *bounded term-size property* (see e.g. [Van Gelder 1989; Chen and Warren 1996]). *SLG systems* capture the states of an *SLG evaluation* of a query against a program and have two components: an *SLG forest*, which is a set of *SLG trees*, and a *table*. Before providing formal definitions, we introduce some aspects of SLG evaluation informally through an example.

EXAMPLE 2.1. Consider the evaluation of the query `?- p(a,Z)` with respect to the program in Figure 1. The declaration `:- table p/2` indicates that SLG resolution is to be used for calls to predicate `p/2`. An SLG system consisting of a forest of SLG trees and a table is depicted in Figure 1 near the end of the evaluation. A root node of a tree in the forest consists of a tabled subgoal, and, for definite programs, a non-root node consists of a clause: *Answer_Template* :− *Goal_List*, where *Answer_Template* accumulates substitutions for the variables of the subgoal, while *Goal_List* contains literals that remain in order to derive an answer.

Let us examine operations of the SLG evaluation in detail. The evaluation begins with a system containing a tree with root node `p(a,Z)` and an entry

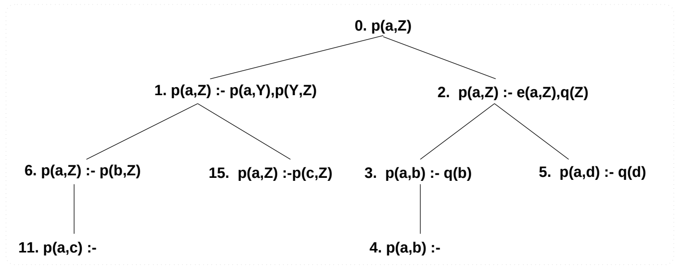
$$\langle p(a, Z), \emptyset, incomplete\rangle$$

in the table. In the above table entry, the first argument represents the tabled subgoal, the second its current set of answers, and the third its state. This system initialization can be thought of as being performed by the NEW SUBGOAL operation which is applicable to a subgoal S if no entry for S exists in the table. In this case, NEW SUBGOAL creates a tree with root S, and an entry for S in the table.[1] The evaluation of query `p(a,Z)` then uses PROGRAM CLAUSE RESOLUTION to generate children for this subgoal. The program clause `p(X,Z) :- p(X,Y),p(Y,Z)` is first resolved against the new subgoal, creating node 1 in Figure 1. In node 1, the selected literal `p(a,Y)` is tabled, so the node is termed *active*, and its selected literal will be resolved away using answers. Since (a variant of) `p(a,Y)` has an entry in the table, the NEW SUBGOAL operation is not applicable. If answers for this subgoal were present, children for node 1 could be produced via ANSWER RETURN operations. However, since there are no answers, the only alternative is to *suspend* this branch of the computation to wait for their possible generation. The only applicable operation for the forest at this point is to resolve the second program clause (`p(X,Z) :- e(X,Z),q(Z)`) against `p(a,Z)` in node 0. This resolution produces node 2. Since the selected literal for node 2 is non-tabled, node 2 is termed an *interior* node, and SLD-style program clause resolution is used on this literal. SLD-style resolution continues, eventually producing node 4 which contains no further literals to resolve. The NEW ANSWER operation adds `p(a,b)` to the table as an answer for `p(a,Z)`. Further program clause resolution is performed for the subtree rooted at node 2, leading to node 5. Next, the answer produced in node 4, `p(a,b)`, is returned to all active nodes suspended on `p(a,Z)` via the ANSWER RETURN operation. In this example, the only such node is node 1 and through ANSWER RETURN node 6 is created.

The evaluation eventually gives rise to two other tabled subgoals, `p(b,Z)` and `p(c,Z)`, each of which is entered in the table and forms the root of its own SLG tree. In general, the process of expanding nodes, adding new answers and returning them to consuming subgoals, continues until further resolution will produce no new answers for a mutually dependent set of tabled subgoals, called *Strongly Connected Components* (or *SCC*s). At such a stage, the subgoals in the SCC are *completely evaluated*: no NEW SUBGOAL, ANSWER RETURN, PROGRAM CLAUSE RESOLUTION, or NEW ANSWER operations are applicable for any node of their trees. Because answers for a completely evaluated subgoal S are in the table, the tree for S is of no further use to a computation and can be *disposed*. In the SLG system of this example there are no mutual dependencies among subgoals, and so there are three singleton *SCC*s {`p(a,Z)`}, {`p(b,Z)`}, and {`p(c,Z)`}. Using the SLG COMPLETION operation the trees for `p(c,Z)` and `p(b,Z)` can be disposed once it is determined that they are *completely evaluated*. In this example that condition occurs after node 14 was created. The SLG system after completing these subgoals is shown in Figure 2. After node 15 has been created, {`p(a,Z)`} is also completely evaluated, and all subgoals can be completed and their trees safely disposed.

From Example 2.1, it can be seen that over the class of definite programs, SLG resolution does not greatly differ from other tabled-based formulations. SLG, however, is a *variant-based* tabling method: a tree for a new subgoal is created, or an answer added to the table depending whenever the subgoal or answer is different (up to variance) from those previously derived. Other tabling methods, such as OLDT [Tamaki and Sato 1986] check whether a new subgoal or answer is subsumed

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



Subgoal	Answers	State
p(a,Z)	p(a,b) p(a,c)	*Incomplete*
p(b,Z)	p(b,c)	*Complete*
p(c,Z)		*Complete*

Fig. 2. SLG System for the query `?- p(a,Z)` on creation of node 15.

by one previously derived in the evaluation. A variant-based tabling method preserves observables for Prolog; while a subsumption-based method may have better termination or complexity properties for certain programs and queries.

We now present the formal definitions of terms used in the example.

DEFINITION 2.1 (SLG SYSTEM). An *SLG system* is a *forest of SLG trees*, along with an associated *table*. Root nodes of SLG trees are subgoals of tabled predicates. Non-root nodes either have the form *fail* or

$$Answer\_Template :- Goal\_List.$$

The *Answer_Template* is an atom, and *Goal_List* is a possibly empty sequence of literals.

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

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

where the first element is a subgoal, the second a set of atoms, and the third either the constant *complete* or *incomplete*.

As terminology, if $\langle S, AS, St\rangle$ is an entry in the table and $A \in AS$, we say that S is a *subgoal* in the table, that A is an *answer* in the table for S and St is the *state* of the subgoal.

DEFINITION 2.2 (SLG EVALUATION). Given a tabled program P, an *SLG evaluation* $\mathcal{E}$ for a subgoal G of a tabled predicate is a sequence of systems $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n$ such that:

—$\mathcal{S}_0$ is the forest consisting of a single SLG tree rooted by G and the table $\{\langle G, \emptyset, incomplete\rangle\}$;

—for each finite ordinal k, $\mathcal{S}_{k+1}$ is obtained from $\mathcal{S}_k$ by an application of one of the operations in Definitions 2.3 or 5.1.

If no operation is applicable to $\mathcal{S}_n$, $\mathcal{S}_n$ is called a *final system* of $\mathcal{E}$. In a final SLG system $\mathcal{S}_n$ of a non-floundered evaluation $\mathcal{E}$ (i.e., where no non-ground negative literal of a tabled predicate is selected), if all its subgoals are completely evaluated, we say that $\mathcal{S}_n$ (and $\mathcal{E}$) is *complete*; otherwise we say that $\mathcal{S}_n$ is *flummoxed*.

In our version of SLG, tabling 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 to the table, the tree with root S is no longer needed and can be disposed. The subgoals in the table of a system $\mathcal{S}$ thus are root nodes of SLG trees in $\mathcal{S}$, or of trees in a predecessor of $\mathcal{S}$ that are now disposed.

It is convenient to describe a node of an SLG tree by its *status*. The root node of an SLG tree has status *generator*. Non-root nodes may have status *interior* if its selected literal is non-tabled, *answer*, if its *Goal_List* is empty, or *active* if its selected literal is tabled and the node does not have *fail* as an immediate child. In the last case, we will speak of positive or negative active nodes, depending on whether the selected literal is positive or negative. We call a subgoal S a *consumer subgoal* in a system $\mathcal{S}$ if it is the selected subgoal of a positive active node, and the state of S in the table is not complete. *fail* nodes are used only in programs with negation and we postpone their discussion until Section 5. Using this terminology, we define tabling operations for definite programs.

DEFINITION 2.3 (SLG OPERATIONS FOR DEFINITE PROGRAMS). Given a system $\mathcal{S}_k$ of an SLG evaluation of a tabled program P and subgoal G, $\mathcal{S}_{k+1}$ may be produced by one of the following operations.

$\boxed{\text{NEW SUBGOAL}}$. Given an *active* node N with selected subgoal S, where S is not present in the table of $\mathcal{S}_k$, create a new SLG tree with root S and add the entry $\langle S, \emptyset, incomplete\rangle$ to the table.

$\boxed{\text{PROGRAM CLAUSE RESOLUTION}}$. Let N be a node in $\mathcal{S}_k$ that is either a *generator* node S or *interior* node $Answer\_Template$:– $S, Goals$. 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. Then

—if N is a generator node, produce a child of N: $(S$:– $Body)\theta$.

—if N is an interior node, produce a child of N: $(Answer\_Template$:– $Body, Goals)\theta$.

$\boxed{\text{ANSWER RETURN}}$. Let N be a positive *active* node $Answer\_Template$:– $S, Goals$. Let A be an answer for S in $\mathcal{S}_k$ and assume that A has not been used for resolution against N. Then produce a child of N: $(Answer\_Template$:– $Goals)\theta$ where θ is the mgu of S and A.

$\boxed{\text{NEW ANSWER}}$. Let A :– be a node in a tree rooted by a subgoal S, such that A is not an answer in the table entry for S in $\mathcal{S}_k$. Then add A to the set of answers for S in the table.

$\boxed{\text{COMPLETION}}$. If *Set* is a set of subgoals that is completely evaluated (according to Definition 2.5), remove all trees whose root is a subgoal in *Set*, and change the state of all table entries for subgoals in *Set* from *incomplete* to *complete*.

Further operations to handle negative literals are presented in Section 5.

Returning to Example 2.1 it can be seen that the operation NEW SUBGOAL is used to create nodes 6a and 12a. PROGRAM CLAUSE RESOLUTION is used to create nodes 1, 2, 7, 8, 13 and 14 via resolution against generator nodes, and to create nodes 3, 4, 5, 9 and 10 via resolution with selected literals of interior nodes. ANSWER RETURN creates nodes 6, 11, 12 and 15 through resolution against selected atoms of active nodes. NEW ANSWER is used to intern answer nodes 4, 10 and 11 into the table.

The COMPLETION operation in Definition 2.3 relies on the notion of a set of subgoals being *completely evaluated.* In order to define this latter notion we introduce the notion of subgoal dependencies in an SLG system.

DEFINITION 2.4 (SUBGOAL DEPENDENCY GRAPH). Let $\mathcal{S}_k$ be an SLG system and $\mathcal{F}$ its SLG forest. We say that a tabled subgoal S *directly depends on* a tabled subgoal S' iff the tree rooted by S contains an active node whose selected literal is $(not)S'$. If $(not)S'$ is a positive (negative) literal, then we say that S *directly depends positively (negatively) on* S'. The dependence may be both positive and negative at the same time.

The *Subgoal Dependency Graph* $SDG(\mathcal{S}_k) = (V, E)$ of $\mathcal{S}_k$ is a directed graph in which V is the set of root goals for trees in $\mathcal{F}$ and $(S, S') \in E$ iff subgoal S directly depends on subgoal S'. The edges are labeled positively, negatively, or both depending on the sign of the direct dependencies.

Because the subgoal dependency graph of a given system is a directed graph, strongly connected components can be defined on it in the usual manner. Throughout this article, we denote a set of SCCs as an Approximate SCC, or ASCC. An ASCC A is termed *independent* if no subgoal in A depends on any subgoal outside of A. Using these notions, we can provide an operational definition of when a set of subgoals has been completely evaluated.

DEFINITION 2.5 (COMPLETELY EVALUATED SET OF SUBGOALS). Given an SLG system $\mathcal{S}_k$, a set *Set* of subgoals is *completely evaluated* iff either of the following conditions is satisfied:

(1) *Set* is an independent ASCC of $SDG(\mathcal{S}_k)$, and for each subgoal S in *Set*:
  —All applicable SLG operations other than COMPLETION have been performed for nodes in the tree rooted by S according to Definition 2.3 (and to Definition 5.1 for programs with negation).
  —No active node in the tree rooted by S contains a selected negative literal.

(2) *Set* $= \{S\}$ and S contains an answer identical to itself in the table entry for S.

We say that a subgoal S is *completely evaluated* iff *Set* is a completely evaluated set of subgoals and $S \in$ *Set*.

The second condition, introduced in [Sagonas et al. 1996], will sometimes be referred to as *early completion* of subgoals. In a given system, a subgoal S may have an answer S, but there could be SLG operations such as PROGRAM CLAUSE RESOLUTION which would otherwise be applicable to the tree for S. S would thus be completely evaluated according to condition 2, but not to condition 1. Early completion is necessary to evaluate certain stratified programs using a fixed computation rule and will be further discussed in Section 5.

3. THE ABSTRACT MACHINE FOR DEFINITE PROGRAMS

Having introduced basic tabling definitions and operations, we now consider the main extensions made by the SLG-WAM to the WAM to support tabling of definite programs.

(1) The engine must be able to *suspend* a computation when encountering a consumer subgoal and *resume* the consumer subgoal at a later point to return

answers (e.g. nodes 1, 7 and 13 in Example 2.1). The need to resume computations requires that the environment corresponding to an active node of an SLG tree be efficiently restored. Section 3.1 describes extensions to the WAM that support the ability to suspend and resume computations.

(2) A space for tables themselves must be designed, and their access methods must be tightly integrated with WAM data structures. These issues are covered in Section 3.2.

(3) The choice of when to return an answer to an active node gives rise to several possible scheduling strategies. Naturally, different scheduling strategies require different amounts of time and space, and influence the architecture of the abstract machine. We discuss issues related to scheduling of SLG operations in Section 3.3.

(4) The preceding features must be compiled into WAM-like code. The design of the SLG-WAM instruction set is described in detail in Sections 3.4 and 3.5.

(5) Since environments are needed for the ANSWER RETURN operation, space for active nodes cannot be reclaimed upon backtracking, but only when the strongly connected component to which they belong is completed, i.e., only when it is known that no more answers will be produced. A mechanism must be developed to detect completion of subcomputations in order to reclaim space. Section 3.5 describes how this is done for definite programs.

3.1 Suspending and Resuming Computations

A tabled evaluation like that of Example 2.1 cannot be implemented using the pure depth-first search of the WAM. Rather, the computation path of an active node may have to *suspend* when it has exhausted all answers in the table, and *resume* when new answers have been derived. Suspension is performed in the WAM framework by creating a choice point to represent the suspended environment, and then failing to a previous choice point without reclaiming any stack space. Suspended choice points thus freeze the stack, prohibiting memory reclamation before completion of a subgoal. Resuming uses a *forward trail* to restore variable bindings along the path to the suspended subgoal. We consider data structures and algorithms to support suspending and resuming computations.

3.1.1 *SLG Search Trees.* Rather than a forest of trees, the memory layout of the SLG-WAM resembles a single *SLG search tree* which can be constructed by using a *first-call optimization.* This optimization has the effect of inserting the tree with root goal G as a subtree below the first node N_G whose selected literal is G, and sharing their environments.[2] Figure 3 illustrates how the SLG search tree for the program of Example 2.1 is constructed from the forest of SLG trees shown in Figures 1 and 2 with the first-call optimization occurring at nodes 6 and 12. First-call optimization merges a generator and an active node; the resulting node produces answers like a generator and does not require an explicit instruction to perform an ANSWER RETURN operation.

[2]First-call optimization is also used implicitly in the OLDT dichotomy of solution and lookup nodes (see [Tamaki and Sato 1986]).

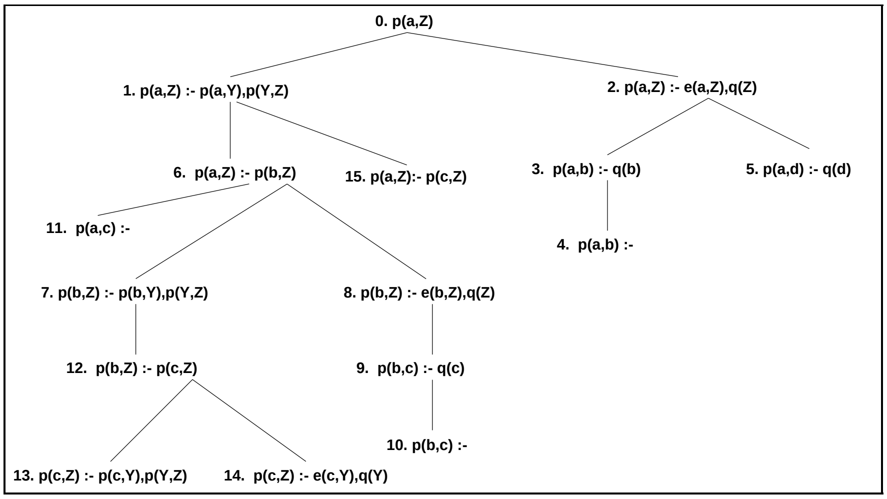
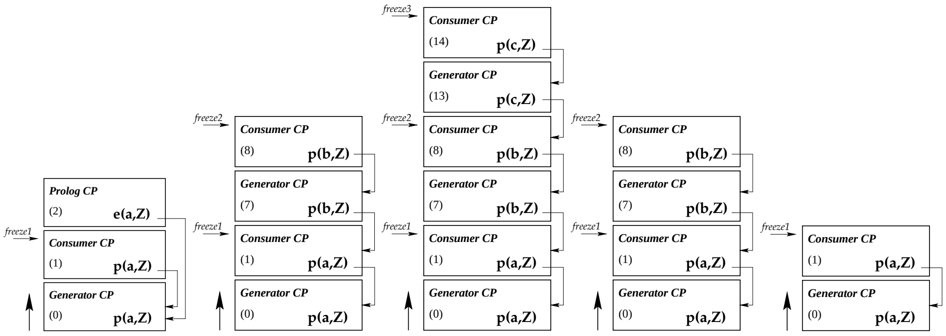


Fig. 3. The SLG Forest of Figures 1 and 2 as a single SLG Search Tree.

3.1.2 *Preserving Environments through Freeze Registers.* To ensure that environments for suspended active nodes of the SLG tree may be later resumed, the SLG-WAM freezes the stacks using a *freeze register* for each stack of the WAM [3]. Space is not reclaimed below these freeze registers until completion of the appropriate generator node. In definite programs, stacks are frozen whenever a consumer subgoal is encountered, since consumer subgoals need to suspend either to obtain new answers, or to ensure the consumption of all relevant answers. Figure 4 shows states of the choice point stack while executing the program of Example 2.1, where choice points for generator and consumer subgoals are denoted explicitly. Note that on calling the consumer subgoal `p(a,Y)` in node 1, the computation is suspended, a freeze point is set (denoted as *freeze1* in Figure 4(a)), computation continues with node 2 of the tree, and the next choice point (for `e(a,Z)`) is allocated above the choice point freeze register.

The introduction of freeze registers affects the placement of choice points by the WAM `try` and `trust` instructions: a choice point is placed at the maximum of the **B** register and the choice point freeze register (**BF** register). Similarly, for local environments, freeze registers affect the `allocate` instruction which must determine the greatest of the environment register (**E** register), the environment backtrack register (**EB** register), and the environment freeze register (**EF** register). Likewise, the allocation of new trail entries requires a check of the trail freeze register (**TRF** register), as well as the WAM trail register (**TR** register). In addition to their use for allocation, both the **B** register and the **TR** register are used to store information about the environment of a node. The **B** register points to the continuation to take upon failure, and the **TR** register is used to untrail appropriate variables when that failure continuation is taken. However, because the **H** register is used *only* for

[3]Throughout this article, we assume a WAM model with environment and choice point stacks separated rather than combined as in the original WAM. We also assume throughout this article that stacks grow upwards.



(a) At node 4 (b) At nodes 7–10 (c) At nodes 13–14 (d) After completing p(c,Z) (e) After completing p(b,Z)

Fig. 4. Choice Point Stack States for Program of Figure 1.

allocation, the effects of freezing the heap on the SLG-WAM can be reduced. Upon backtracking and execution of a trust instruction, the **H** register is reset to the **HB** register only if the **HB** register is greater than the heap freeze (**HF**) register; otherwise no change is made to the **H** register. This ensures that the **H** register is always above the **HF** register and points to an unfrozen portion of the heap. Thus heap information is not overwritten, and instructions can build information on the heap just as in the WAM. Note that with this updating scheme, the significant overhead of checking two registers at every write to the heap is avoided.

The introduction of freeze registers necessitates another change in stack management from the WAM. Consider the SLG system represented by the choice point stack in Figure 4(a). The parent of node 2 (`p(a,Z) :- e(a,Z),q(Z)`) is the generator node, `p(a,Z)`. However, due to the use of freeze registers, the (Prolog) choice point for the `e(a,Z)` call does not lie immediately above the generator choice point for `p(a,Z)`. To handle cases such as this, each choice point must maintain an explicit pointer to the proper failure continuation to take upon backtracking *out* of the choice point (e.g. when all applicable program clauses have been resolved against a subgoal). Freeze registers also add an extra pointer to trail frames, as will be seen in the next section.

Frozen segments in the stacks can be deallocated only when it is known that a set of consumer subgoals has no further need to be resumed. This condition holds when it is determined that the subgoals are completely evaluated, and that their SLG trees, as represented in the SLG-WAM stacks as generator choice points and consumer choice points, can be safely disposed. Deallocation of freeze registers after completion of subgoals `p(c,Z)` and `p(b,Z)` is represented in Figures 4(d) and (e).

3.1.3 *Resuming Suspended Computations by Restoring Environments.* To *resume* computation at an active node, all variable bindings and WAM register values are restored to their state at the time that the node was suspended (as saved using a *consumer choice point*, Section 3.2, or a *negation suspension choice point*,

Parent	Pointer to Parent trail frame
Value	Value to which the variable was bound
Addr	Address of the trailed variable

Fig. 5. Format of (Forward) Trail Frames.

```
Algorithm restore_bindings(new_breg)
    start_trreg := trreg;                         /* current TR register */
    end_trreg := choice_point_trreg(new_breg);
    trreg := choice_point_trreg(new_breg);
    while (start_trreg != end_trreg)
        while (start_trreg > end_trreg)
            untrail(trail_addr(start_trreg));
            start_trreg := trail_parent(start_trreg);
        while (end_trreg > start_trreg)
            end_trreg := trail_parent(end_trreg);
    end_trreg := trreg;
    while (start_trreg < end_trreg)
        * trail_addr(end_trreg) := trail_value(end_trreg);
        end_trreg := trail_parent(end_trreg);
```

Fig. 6. The **restore_bindings** procedure.

Section 6.3). The appropriate action is then taken (e.g. returning an answer) and execution continues with the success continuation (as represented by the **CP** register) of the suspended computation.

Restoring variable bindings for a resumed computation is done in the SLG-WAM through a *forward trail* whose frame format is shown in Figure 5 (see also the papers by D. S. Warren [1984] and D. H. D. Warren [1987]). Recall that the WAM trail contains (local or global stack) variables that must be unbound upon backtracking. In fact, only *conditional bindings* that affect a variable existing before the creation of the current choice point need to be trailed[4]. The SLG-WAM trail must keep addresses of conditionally bound variables as in the WAM. However, the trail must also contain information about the value to which the variable was bound so that bindings of suspended nodes can be restored. Furthermore, as the trail is also a tree rather than a stack, each trail frame has to maintain an explicit pointer to the previous trail frame (using its *Parent* cell). The overhead incurred by the forward trail, compared to the simple trailing of the WAM, is measured in Section 7.1.

The algorithm restore_bindings (Figure 6) uses the forward trail to reconstitute the environment of an active node, as represented by a consumer choice point. Specifically, restore_bindings starts from the current environment, and switches variable bindings to those of the active node represented by the choice point designated by *new_breg*. Both *start_trreg* and *end_trreg* follow their parent chains until a common ancestor is reached, with *start_trreg* untrailing as it goes. Then, variables on the path from *end_trreg* to the common ancestor are rebound. The bindings are applied in the opposite order in which they happened. This is safe since no local

[4]In **get**-style instructions, the WAM checks the **E** register and the **HB** register to determine whether a binding is conditional. The SLG-WAM must also check the corresponding freeze registers, **EF** and **HF**. Other than that, these instructions remain unchanged.

FailCont	The Failure Continuation
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of Trail
CPreg	Success Continuation for Subgoal
Ereg	Parent Environment
*RSreg**$\neg$	Root Subgoal Choice Point
*Breg_Chain**	Failure Continuation on Backtracking *out* of this CP
*SubgFr**	Pointer to the Subgoal Frame
*BFreg**	Choice Point Freeze Register
*HFreg**	Heap Freeze Register
*TRFreg**	Trail Freeze Register
*EFreg**	Local Stack Freeze Register
A_n	Argument Register n
$\vdots$	$\vdots$
A_1	Argument Register 1
*VarNum**	Number of Variables: m
V_m*	Substitution Factor Variable m
$\vdots$	$\vdots$
V_1*	Substitution Factor Variable 1

Fig. 7. Format of Generator Choice Points.

stack or heap variable can have more than one entry on each branch of the trail. Note that since restore_bindings is used to reconstitute environments for returning answers, each tabled predicate is compiled using a choice point, even if the predicate is defined by a single clause.

3.1.4 *Generator and Consumer Choice Points.* We end our discussion of mechanisms to suspend and resume computations by presenting the format of generator and consumer choice points. The format of a *generator choice point* is depicted in Figure 7. Cells that are not found in WAM choice points are marked with an asterisk, while cells marked with a $\neg$ symbol are not necessary for definite programs (we will use these conventions throughout the rest of this article). Figure 7 is divided into three sections. The top section contains state information that the SLG-WAM must restore on backtracking for any subgoal, whether tabled or not. This information includes the cells of a WAM choice point along with an explicit pointer of the failure continuation to take upon backtracking out of the choice point (*Breg_Chain*), and a cell, *RSreg*, that records the value of a new global register, called the **RS** register [5]. The middle section is not found in Prolog choice points in the SLG-WAM. It contains a pointer to the table entry of the subgoal (the *Subgoal Frame*, Section 3.2), and the values of the freeze registers at the time of choice point creation. Although creating a generator choice point frame does not require freezing the stacks, the values of the freeze registers must be recorded so that they can be properly reset when the subgoal associated with a choice point is completed. The bottom section contains argument registers of the subgoal along

[5]This new register and choice point cell are used to determine exact subgoal dependencies for programs with negation (see Section 6.4).

FailCont	Pointer to **answer_return** instruction
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of Trail
CPreg	Success Continuation
Ereg	Parent Environment
*RSreg**¬	Root Subgoal Choice Point
*Breg_Chain**	Failure Continuation on Backtracking *out* of this CP
*LastAnswer**	Pointer to Last Consumed Answer
*PrevCCP**	Pointer for Consumer Choice Point Chain
*VarNum**	Number of Variables: m
V_m*	Substitution Factor Variable m
⋮	⋮
V_1*	Substitution Factor Variable 1

Fig. 8. Format of Consumer Choice Points.

with its *substitution factor*, the set of free variables which exist in the terms in the argument registers. As explained in Section 3.2, the substitution factor is used to reduce copying of information into and out of answer tables.

Consumer choice points are created to store environments for consumer subgoals, and their format is shown in Figure 8. As their name implies, these frames are stored on the choice point stack and contain the same WAM state registers as any choice point. However, answers are resolved using a substitution factor (Section 3.2) which replaces the usual argument registers for consumer choice point frames. A consumer choice point for a tabled subgoal also maintains the following information.

(1) A pointer, *LastAnswer*, to the last answer resolved by the consumer choice point (using the *answer return list* of Section 3.2).
(2) A pointer *PrevCCP*, used to chain together all consumer choice points for the same subgoal. For instance, in Figure 1 of Example 2.1 active nodes 12 and 13 have selected literal `p(c,Z)`. In the SLG-WAM, consumer choice points for these nodes would be chained together using the *consumer choice point chain* for `p(c,Z)`, as would nodes 6 and 7 for `p(b,Z)`. The consumer choice point chain is needed for scheduling the return of answers and will be discussed fully in Section 3.3.

3.2 Interfacing Table Space to Run-Time Stacks

The SLG-WAM adds two memory areas to those of the WAM: a *completion stack* and *table space*. The *completion stack* is used to detect when a set of subgoals has been completely evaluated and is described in Section 3.5. The *Table Space* stores information about tabled subgoals and their answers. The design and implementation of data structures and algorithms for efficient access to table space is a critical issue for the performance of any implementation of tabling. In this article, we provide only a brief description of the layout of the table space; full details are presented in [Ramakrishnan et al. 1995].[6] Elements of the table space may need to be repeatedly accessed in several different ways during the course of evaluation.

[6] Implementation of the table access routines is primarily due to Prasad Rao.

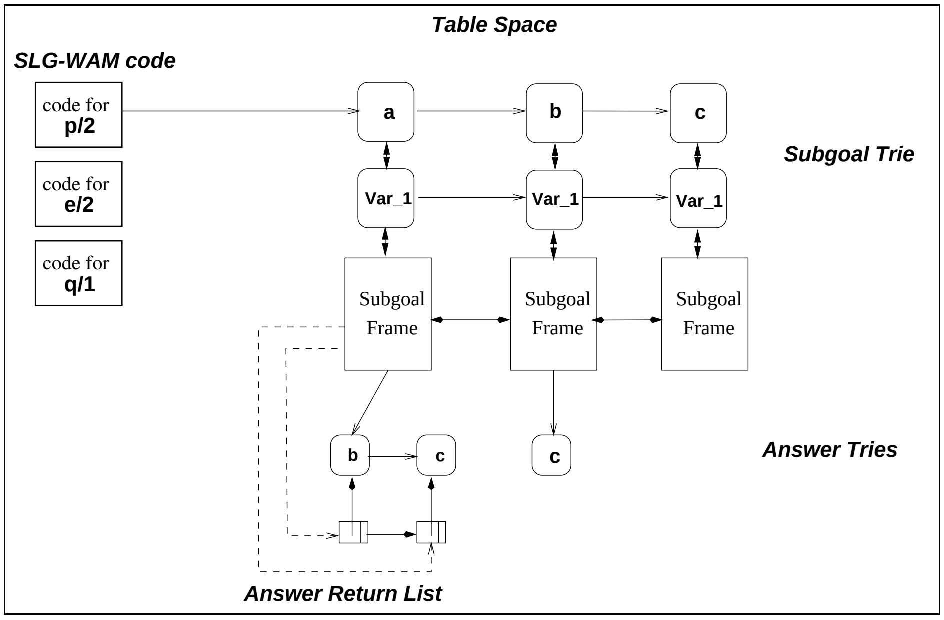


Fig. 9. Relationships between elements of the Table Space.

First, to implement the NEW SUBGOAL operation, a check must be made to determine whether each tabled subgoal is present in the table, and the subgoal must be inserted if not; this mode of access is called *subgoal check/insert*. An analogous mode, called *answer check/insert*, is needed to implement NEW ANSWER. Furthermore, the mode of *answer backtracking* is also needed during the course of ANSWER RETURN. In principle, tables can be implemented using any data structure that supports these three types of access: such as hashing, tries, or discrimination nets. Experience has demonstrated the superiority of *tries* as the basis for table space. Tries not only provide complete discrimination of terms, but also permit a check and possible insertion to be performed in the same pass through a term. Subgoals and answers are copied from the execution stacks to the table space during subgoal check/insert and answer check/insert, while answers are copied from the table to the execution stacks during answer backtracking. This copying is performed so that (i) variables in subgoals and in answers do not share bindings when they are used in different nodes of the search forest; and (ii) information about subgoals and answers may survive the effects of backtracking and possible space reclamation (i.e. so that tabled information is *persistent*).

Figure 9 represents elements of the table space for the SLG system in Figure 2. At the entry point for `p/2` an operand of a `tabletry` SLG-WAM instruction (discussed in Section 3.4.2) points to a node of its subgoal trie which is designated as the trie's root. In our example, the subgoal trie of `p/2` contains subgoals `p(a,Z)`, `p(b,Z)`, and `p(c,Z)`. Each of these subgoals may have an associated answer trie, although that of `p(c,Z)` is empty. Each root-to-leaf path through a subgoal trie corresponds to a single subgoal, and leaf nodes of the subgoal trie have a special form and are called *subgoal frames*. Root-to-leaf paths through an answer trie also correspond to an answer. However, answer tries for incomplete subgoals also have their leaves chained together via an *answer return list*. The need for the answer return list arises to support the mode of answer backtracking. Since the generation and consumption of answers are asynchronous, and new answers may be inserted *anywhere* in a trie, it is not possible to perform answer backtracking by sequentially backtracking through an answer trie of an *incomplete* subgoal. To address this, the elements of the answer return list point to answers (identified by leaf nodes of the answer trie), in the order of their creation times. Using this list, it is guaranteed that no answer is skipped, and that no answer is returned to the same consumer choice point more than once.

Substitution Factoring. As Figure 9 shows, an answer trie stores only bindings that are not present in the associated tabled subgoal. This optimization is called *substitution factoring* [Ramakrishnan et al. 1995]. Substitution factoring uses the following observation to optimize the answer check/insert and answer backtracking access modes. In a variant-based tabling method, all answers to a tabled subgoal are subsumed by the subgoal itself. For instance, `p(a,Z)` subsumes both `p(a,b)` and `p(a,c)`, while `p(b,Z)` subsumes `p(b,c)`. Thus, each answer A of a tabled subgoal G can be represented as $G\eta_A$, where η_A is an *answer substitution* for G. The core idea of substitution factoring is to store only the answer substitution, and to create a mechanism of returning answers to consumer subgoals that is proportional to the size of η_A rather than the size of A. The set of unbound variables of a tabled subgoal is determined as part of the subgoal check/insert procedure. This procedure must fully traverse the subgoal either to check if it is in the table or to insert it if not. As the procedure traverses the subgoal, it factors out dereferenced pointers to variables from the subgoal and places them in the choice point stack. We refer to this set of dereferenced variable pointers as the *substitution factor* (see Figures 7 and 8). The values in cells of the substitution factor thus point to variables on the local or global stack. The substitution factor is used by generator choice points to add answer substitutions to the table, and by consumer choice points to backtrack through answers. Furthermore, because the subgoal check/insert procedure must be performed to determine whether a subgoal is new to an evaluation (and by extension, whether a generator or consumer choice point is to be created), the substitution factor is placed before the rest of a generator choice point or consumer choice point.

Subgoal Frames. Subgoal frames contain general information about the state of a tabled subgoal, and their format is shown in Figure 10. To access answers for a subgoal, subgoal frames contain a pointer to the root of the associated *answer trie*. To facilitate the COMPLETION operation, subgoal frames have a *ComplSF* cell which points to a *completion stack frame* (described in Section 3.5) when a subgoal is incomplete, and when set to null, indicates that the subgoal is complete. To facilitate memory management of subgoal frames and of the answer tries which are accessed through them, subgoal frames are maintained in a doubly linked list (see Figure 9). The foregoing information must persist after subgoals are determined as completely evaluated, but the subgoal frames also contain information that can be reclaimed after completion of their subgoals. This consists of the following pointers: (1) a pointer (*CCP_Chain*) to the head of the consumer choice point chain for the

AnsTrieRoot	Pointer to the Root of the Answer Trie
ComplSF	Pointer to the Associated Completion Stack Frame
NextSF	Pointer to Next Subgoal Frame
PreviousSF	Pointer to Previous Subgoal Frame
AnsRetListH	Pointer to the Head of the Answer Return List
AnsRetListT	Pointer to the Tail of the Answer Return List
CCP_Chain	Pointer to the Head of the Consumer Choice Point Chain
NS_Chain$\neg$	Pointer to the Head of the Negation Suspension Chain

Fig. 10. Format of Subgoal Frames.

subgoal; (2) a pointer (*NS_Chain*) to an analogous *negation suspension chain* of choice points for negative active nodes (the negation suspension chain is discussed in Section 6.3); (3) a pointer to the head of the *answer return list* in the answer trie, which is used for answer backtracking when a consumer choice point is created; and (4) a pointer to the tail of the answer return list, used in the new_answer instruction to efficiently add answers in their proper generation sequence.

3.3 An Overview of Batched Evaluation for Definite Programs

It is usually possible to apply more than one operation to a particular SLG system. For instance, there may be program clauses to resolve with generator or interior nodes, answers to return to active nodes, or completion operations to be performed on sets of trees. The decision of when to perform such operations is determined by a *scheduling strategy.* This section overviews a particular scheduling strategy, called *Batched Evaluation* [Freire et al. 1996], which forms the default scheduling strategy of version 1.7 of XSB[7]. Later sections provide instruction-level details of the implementation of Batched Evaluation, as well as its extension to programs with negation.

Batched Evaluation takes its name because it tries to avoid resuming an active node until there are several answers to return to that node. For in-memory Datalog queries, Batched Evaluation has been shown to be superior in terms of time and space to three other scheduling strategies (see [Freire et al. 1996]). As an aside, we note that it is unlikely that a single scheduling strategy can be uniformly faster that all others for all applications. For instance, the breadth-first evaluation described in [Freire et al. 1997] is extremely efficient for queries to disk-resident data, giving disk-access properties comparable to those of the semi-naive evaluation of a magic-transformed program. Batched Evaluation is a highly optimized scheduling strategy, which we present through the series of rules in Figure 11. We begin by considering actions of Batched Evaluation in Example 2.1, where the numbers associated with the nodes in Figure 1 correspond to the order of generation by that strategy.

Batched Evaluation schedules PROGRAM CLAUSE RESOLUTION in a depth-first manner as does the WAM as can be seen (from e.g, nodes 2 and 8) in Example 2.1. The advantages of this strategy are well-known: for instance backtracking can be used to reclaim space, reducing the need for garbage collection. Furthermore, the WAM's strategy gives a good locality of reference so that cache misses are also

[7]This scheduling strategy was primarily implemented by Juliana Freire.

In the following, let N be an *active* node, $Answer\_Template$:– $S, Goal\_List$, in an SLG tree.

(1) The SLG-WAM schedules PROGRAM CLAUSE RESOLUTION as does the WAM: clauses are resolved according to their textual order and literals are selected by a fixed left-to-right rule. This applies to both *interior* and *generator* nodes.

(2) If S is *complete*, the node N need not be suspended and, answers can be returned to it as if they were program clauses. Support for this strategy is called the *completed table optimization*.

(3) If S is *incomplete*, any answers for S are returned to N under a model that approximates program clause scheduling: the first answer is immediately returned upon the creation of N, and a consumer choice point frame is set up to return any further answers to N. Only when N exhausts all answers (currently) in the table for S will it suspend.

(4) Answers may also be scheduled for return to N during the procedure fixpoint_check performed by the *leader* S_L of the *scheduling ASCC* of S. This fixpoint_check is executed during the completion instruction for S_L.

Fig. 11. Rules of the *Batched Evaluation* scheduling strategy.

reduced (see e.g. [Tick 1988; Van Roy 1994]). This design decision is shown in Rule 1 of Figure 11.

When an active node, N, is created with selected subgoal S, scheduling of ANSWER RETURN operations varies depending on whether S is complete or incomplete. In the case where S is complete, a *completed table optimization* can be performed (Rule 2 of Figure 11). The node can be treated as if it were an interior node, and need not be suspended; rather, the engine backtracks through answers for S as if they were unit program clauses. An example of this optimization occurs on node 15 in Example 2.1. In this case, node 15, whose selected subgoal is completed, immediately fails. We mention in passing that nodes of the trie data structure are in fact SLG-WAM instructions which are directly executed for completed tables. Surprisingly, execution of unit clauses compiled into an answer trie can sometimes outperform that of unit clauses compiled into standard WAM code mainly due to factoring of common prefixes and possible avoidance of unnecessary bindings and unbindings (see [Ramakrishnan et al. 1995] for further explanation).

If the table for S is incomplete, then N might not be able to consume all answers for S in a depth-first manner. This situation is portrayed in Example 3.1.

EXAMPLE 3.1. Figure 12 presents an example of two mutually recursive predicates `a/2` and `b/2` each of which produces answers consumed by the other. An SLG system is shown for the evaluation of the query `a(0,X)`. While details of this evaluation will be presented below, note in particular that answers for `b(0,Y)` are returned to node 3 only when no other operations are applicable in the tree for `b(0,X)`.

In Example 3.1, the children of node 6 cannot be derived in a depth-first manner because the answer `a(0,2)` is not derived until `a(0,1)` has been been resolved against the selected atom of node 6. Rather, node 6 needs to suspend so that answers may be derived, and later to resume to return those answers. Suspension is performed using the mechanisms described in Sections 3.1 and 3.2: a consumer choice point is created and stacks are frozen. If no answers for S are present in the table when N is created, the engine goes on to other resolution by picking up

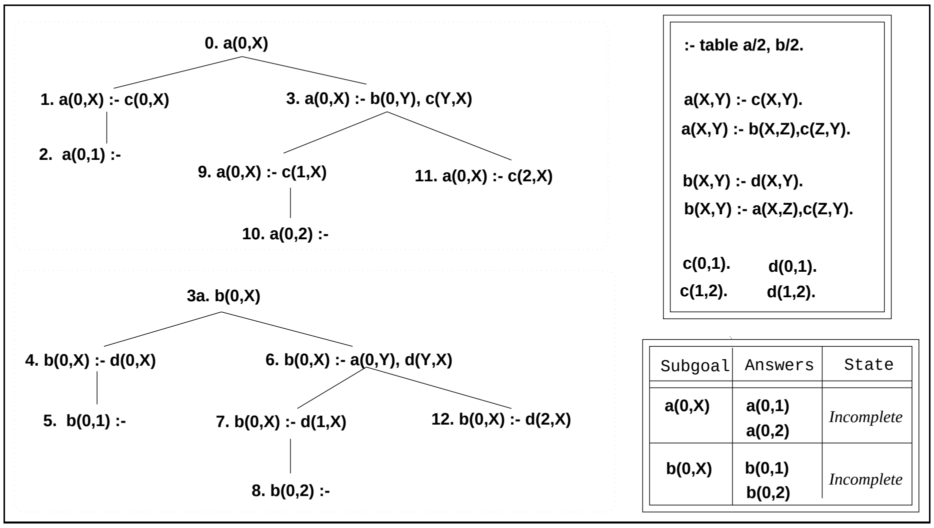


Fig. 12. Illustration of Batched Evaluation

the *Breg_Chain* failure continuation in the consumer choice point of N. If there are answers for S, the consumer choice point of N will backtrack through them, approximating a depth-first search. At an operational level, answer backtracking is done using the *answer return list* (Section 3.2) which causes the set of answers to be traversed in the order of their derivation and helps ensure that each answer is returned exactly once to an active node. Whether there are answers present in the table or not, N will be suspended when it has exhausted all answers present in the table. Rule 3 of Figure 11 summarizes these actions.

In order to completely evaluate subgoals, the engine must ensure that all appropriate answers are returned to all consumer subgoals in an (Approximate) SCC. The subgoals `a(0,X)` and `b(0,X)` in Example 3.1 form a non-trivial ASCC. In evaluating this ASCC, the first batch of answers for `a(0,X)`, {`a(0,1)`} is returned to node 6 creating node 7. Later, in nodes 9 and 11, the first batch of answers for `b(0,X)`, {`b(0,1)`,`b(0,2)`} is returned to node 3. Finally, the second batch of answers for `a(0,X)`, {`a(0,2)`} is returned to node 6, this time creating node 12. It can thus be seen that the process of resuming an active node, backtracking through answers, performing program clause resolution, suspending and then resuming another active node is an iterative process which repeats until a fixpoint is reached for a set of subgoals. Precisely, this fixpoint is reached when an ASCC is completely evaluated (Definition 2.5). At a general level, the fixpoint is controlled by backtracking into generator choice points which causes a fixpoint_check to schedule resumption of active nodes via consumer choice points (Rule 4 of Figure 11).

Specifically, fixpoint_check is part of the SLG-WAM completion instruction which is invoked for a subgoal S when the engine backtracks into the generator choice point for S after all applicable PROGRAM CLAUSE RESOLUTION steps for S have been applied. The completion instruction actually executes a fixpoint_check only when a given subgoal is designated as *leader*, or oldest subgoal, of its *scheduling ASCC*. Scheduling ASCCs are oriented toward space reclamation in a stack-based system and their representation and maintenance is presented in Section 3.5. For now, a scheduling ASCC can be thought of as a unique ASCC to which every incomplete tabled subgoal belongs.

The fixpoint_check procedure determines whether the subgoals in a scheduling ASCC have been completely evaluated or whether further answers need to be returned to consumer choice points for subgoals in the scheduling ASCC. This determination is made by calling the procedure schedule_resumes for each subgoal in the scheduling ASCC. Given a subgoal S, the procedure schedule_resumes traverses the consumer choice point chain (Section 3.5) to find the first consumer choice point for S with unresolved answers (if any). If there is such a choice point, say C, it is resumed by setting the **B** register to point to C and failing. After failing, the engine executes answer_return instructions for C for as long as there are unconsumed answers for C, and then suspends C as in Rule 3, failing into the next choice point on the consumer choice point chain for S. The consumer choice point chain is set so that the engine will backtrack to fixpoint_check after returning answers to the last consumer choice point on the chain. The schedule_resumes procedure is presented in Figure 18 and is discussed fully in Section 3.4; the fixpoint_check procedure is discussed in Section 3.5.

3.4 Extending the Abstract Machine Instruction Set

We present the set of SLG-WAM tabling instructions in two steps: first we motivate a naive instruction set from the SLG operations for definite programs, and then we present the actual instruction set in detail.

3.4.1 *A Reconstruction of the Instruction Set for Tabling.* Consider the tabling instructions that need to be generated for the k clauses of a tabled predicate $\mathtt{t}/n$. Using the program transformation shown below, WAM indexing code is pushed one level down to a *new* predicate $\mathtt{p}/n$ which is evaluated using Prolog-style resolution.

`:- table t/n.` $\mathtt{t}(t_{1,1},\ldots,t_{1,n})$ `:-` B_1. ⋮ $\mathtt{t}(t_{k,1},\ldots,t_{k,n})$ `:-` B_k.	$\longrightarrow$	`:- table t/n.` $\mathtt{t}(X_1,\ldots,X_n)$ `:-` $\mathtt{p}(X_1,\ldots,X_n)$. $\mathtt{p}(t_{1,1},\ldots,t_{1,n})$ `:-` B_1. ⋮ $\mathtt{p}(t_{k,1},\ldots,t_{k,n})$ `:-` B_k.

Notice that these two programs are equivalent with respect to observables.

The Prolog predicate $\mathtt{p}/n$ can be compiled using the instruction set of the WAM. We concentrate on the instructions needed for the tabled evaluation of predicate $\mathtt{t}/n$ defined by the single rule:

$$\mathtt{t}(X_1,\ldots,X_n) :- \mathtt{p}(X_1,\ldots,X_n).$$

A pseudo-compilation of such a tabled predicate is shown in Figure 13. Roughly, the first portion of this pseudocode, instructions labeled L_1–L_7, checks whether subgoals are in the table and inserts them if not, derives answers for these subgoals by performing program clause resolution, and records these answers into the table.

```
L1 : try_me_else                 L8
L2 : new_subgoal_check_insert    n   Trie_Root
L3 : allocate
L4 : call                        1   p/n
L5 : new_answer_check_insert     n   v1
L6 : deallocate
L7 : proceed

L8 : retry_me
L9 : schedule_answer_returns
L10: completion_check
```

Fig. 13. SLG-WAM pseudocode for tabled predicates.

The second portion schedules the return of answers to consumer choice points, essentially performing the functionality of the fixpoint_check procedure, and completes subgoals once the fixpoint is reached (Rule 4 of Figure 11).

The pseudo-instructions new_subgoal_check_insert, new_answer_check_insert, schedule_answer_returns, and completion_check perform functions of the SLG operations NEW SUBGOAL, NEW ANSWER, ANSWER RETURN, and COMPLETION, respectively. The procedures that implement these instructions rely on information that is *dynamic in nature* (checking whether a particular subgoal or answer is new or already exists in the table, whether all answers have been returned to appropriate active nodes, and whether it can be determined that subgoals are completely evaluated).

Finally, note that $\mathtt{t}/n$ requires both a choice point and a local environment, even though the predicate consists of a single clause and none of the variables in the clause are permanent, in the WAM classification.

Need for a Choice Point. Choice point creation is necessary since checking for fixpoint and completion may require information from the choice point frame in order to schedule the return of answers or to mark the table for a subgoal as *complete*. This requirement explains the unorthodox use of a retry_me in the second block of code, followed by an explicit deallocation of the choice point once fixpoint is reached.

Need for an Environment. The local environment in the first block of code is used by the new_answer_check_insert instruction which needs access to the generator choice point, GCP, of the subgoal for which the answer is derived. As shown in Figure 7, this choice point contains both the substitution factor (which provides the answer substitution) and a pointer to the subgoal frame — and through the subgoal frame, a pointer to the corresponding answer trie. It is not possible in general to efficiently find GCP at the time of new_answer_check_insert because any number of choice point frames may have been placed between the top of the choice point stack and GCP. To address this, a local environment is created for all tabled subgoals. This environment contains a *GCP_pointer* to GCP, denoted as v_1 in Figure 13 — or in general v_{m+1} for a clause with m permanent variables. Because the *GCP_pointer* is required whenever an answer is derived, the deallocate instruction has to occur *after* the new_answer_check_insert instruction. Consequently, the *last call optimization* is not applicable to tabled predicates; other optimizations like *environment trimming*, however, can be applied.

3.4.2 *Optimizing the Instruction Set for Tabling.* Note that using the transformation and instruction set presented in the previous section, tabled predicates defined by more than one clause require two choice points instead of one. Also, this initial set contains fixed sequences of instructions: a new_subgoal_check_insert instruction is always preceded by a try-type instruction and followed by an allocate; similarly, a new_answer_check_insert instruction is always followed by a deallocate and a proceed instruction. The SLG-WAM provides the following optimizations:

—Tabled predicates defined by a single rule are compiled using a tabletrysingle tabling instruction rather than the transformation presented above. tabletrysingle includes the functionality of a try_me_else, new_subgoal_check_insert, and allocate sequence of instructions.

—Tabled predicates defined by more than one clause are compiled using the tabletry, tableretry, and tabletrust SLG-WAM instructions, rather than the transformation presented above. The tabletry includes the functionality of the try_me_else, new_subgoal_check_insert, and allocate sequence. The tableretry and tabletrust differ from the WAM retry and trust instructions in that they restore a generator choice point and substitution factor rather than a WAM-style choice point.

—The functionality of the new_answer_check_insert, deallocate, and proceed sequence of instructions is folded into a single SLG-WAM instruction called new_answer.

—During run-time, upon execution of tabletrysingle and tabletrust instructions for a subgoal S, the *FailCont* cell of the generator choice point for S is made to point to a completion instruction which includes the functionality of the schedule_answer_returns and completion_check sequence of Figure 13. This instruction determines whether S is the leader (oldest subgoal) of its scheduling ASCC, and

 —if S is the leader of its scheduling ASCC

  —calls the procedure schedule_resumes as part of performing the fixpoint_check for all subgoals in the scheduling ASCC of S; and

  —if the ASCC of S is completely evaluated, deallocates the generator choice points for subgoals in the ASCC of S along with any frozen portions of the stack that are associated with the ASCC.

 The completion instruction and space reclamation is presented fully in Section 3.5.

Following these principles, the compiled SLG-WAM code for the predicate p/2 in the program of Example 2.1 is shown in Figure 14. As mentioned in Section 3.1.2, the allocate instruction must now use the **EF** register to check for the top of the local stack, in addition to the **E** and **EB** registers. Also, get- and unify- instructions must be changed to use a forward trail and to allocate trail frames above freeze registers. We note, however, that the substitution factor, which is used to efficiently access answer substitutions, does not affect the unification instructions. This is because the substitution factor does not contain variables, but only pointers to variables which occur in the tabled subgoal. These variables are constructed as part of constructing the call to the subgoal and afterwards reside in the local and global stack. We now turn to the newly introduced instructions.

3.4.3 *Instructions for the SLG Operations.*

L_1 :	tabletry	2	L_3	TR	%	
L_2 :	tabletrust	2	L_{10}		%	
L_3 :	getpvar	v_1	r_2		% p(X,Z) :-	
L_4 :	putpvar	v_2	r_2		%	p(X,Y
L_5 :	call	3	p/2		%	),
L_6 :	putpval	v_2	r_1		%	p(Y,
L_7 :	putpval	v_1	r_2		%	Z
L_8 :	call	3	p/2		%	)
L_9 :	new_answer	2	v_3		%	.
L_{10}:	getpvar	v_1	r_2		% p(X,Z) :-	
L_{11}:	call	2	e/2		%	e(X,Z),
L_{12}:	putpval	v_1	r_1		%	q(Z
L_{13}:	call	2	q/1		%	)
L_{14}:	new_answer	2	v_3		%	.

Fig. 14. SLG-WAM code for predicate p/2 of Figure 1.

NEW SUBGOAL. The pseudocode for the tabletrysingle instruction is shown in Figure 15. The arguments of the subgoal are in the WAM argument registers (the *Arity* of the subgoal is a parameter). Using a pointer to the root of the trie for an input subgoal S, as a second parameter, the instruction first checks whether S already exists in table or is new to the evaluation. If S is new (case α in Figure 15), the instruction creates a *subgoal frame* for the subgoal, pushes a *generator choice point* onto the choice point stack, and a *completion stack frame* onto the completion stack, initializing all cells in these frames. tabletrysingle also allocates a local environment and initializes the appropriate permanent variable as the *GCP_pointer*. Furthermore, tabletrysingle places a completion instruction in the *FailCont* cell of the generator choice point. Recall that in the WAM the *FailCont* cell points to the instruction to be executed upon failure of the current 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. After setting up bookkeeping, tabletrysingle branches to the appropriate instructions for program clause resolution.

On the other hand if S already exists in the table the instruction checks whether S is *completed.* If so, (case β), execution immediately branches to the root of the subgoal's answer trie to begin backtracking through answers implementing the *completed table optimization* (Rule 2 of Figure 11). As mentioned in Section 3.3, these answers have been dynamically compiled into SLG-WAM code. On the other hand, if the subgoal is still incomplete (case γ), a *consumer choice point* is added to the head of the consumer choice point chain, dependency information is updated for maintenance of scheduling ASCCs, and the stacks are frozen. The computation then fails into the consumer choice point, which will execute answer_return instructions as long as any unconsumed answers for S are available,[8] and then will suspend by failing into the choice point designated by the *Breg_Chain* cell of the consumer choice point.

The tabletry instruction is similar to the tabletrysingle instruction, but it also has

[8]This step is slightly optimized in XSB version 1.7 by returning the first answer, if any, directly by the **tabletry(single)** instruction.

```
Instruction tabletrysingle(Arity, Subgoal_Trie_Root)     /* Subgoal is in argument registers */
   If (subgoal_check_insert(Subgoal,Subgoal_Trie_Root) == new)
(α)   /* Subgoal is new and added */
      Create and set up a subgoal frame SF for the Subgoal;
      Set up a generator choice point GCP to perform program clause resolution;
      Set the failure continuation FailCont cell of GCP to point to a completion instruction;
      Push a new completion stack frame ComplSF onto the Completion Stack;
      Associate ComplSF with SF;                    /* see Section 3.5 */
      Allocate a local environment and initialize the GCP_pointer, v_{m+1};
      Branch to the next instruction to perform program clause resolution;
   else  /* Subgoal was not new to the evaluation — already existed in the Subgoal_Trie */
      If (SF_ComplSF(Subgoal) == complete)
(β)   /* The subgoal frame has been marked as complete */
         Answer_Root := SF_AnsTrieRoot(Subgoal);
         Branch to Answer_Root to perform answer clause resolution
            by executing the code in the answer trie;
      else
(γ)   /* Subgoal was not new, but its subgoal frame has not yet been marked as complete */
         Create a consumer choice point CCP for Subgoal,
         and add CCP to the head of Subgoal's consumer choice point chain;
         Set the failure continuation FailCont cell of CCP to point to an answer_return instruction;
         Call update_dependencies(Subgoal);     /* for scheduling ASCCs: see Figure 20 */
         Freeze stacks and fail into CCP to execute answer_return instructions;
```

Fig. 15. The **tabletrysingle** instruction.

```
Instruction new_answer(Arity,v_{m+1})  /* v_{m+1} is the GCP_pointer */
   answer_table := SF_AnsTrieRoot(GCP_SubgFr(v_{m+1}));
   η_A := locate_substitution_factor(Arity,v_{m+1});  /* η_A is a pointer to an answer substitution */
   if (answer_check_insert(η_A,answer_table) == new)  /* the answer substitution was inserted */
      Deallocate local environment;
      Set the program pointer P to the continuation pointer CP; /* continue forward execution */
   else
      fail;  /* the answer substitution pointed by η_A was already present in the answer table */
```

Fig. 16. The new_answer instruction (for Definite Programs).

a *Label* as an argument (cf. Figure 14) which is used to branch to the next program clause for the predicate.

NEW ANSWER. The new_answer instruction (Figure 16) is the final instruction of each clause of a tabled predicate. When this instruction is reached, the body of the clause has been resolved away and the dereferenced values of the substitution factor constitute an *answer substitution*, which uniquely identifies an answer for a subgoal. The instruction begins by using the *GCP_pointer* of the local environment to access the answer substitution and the root of the answer trie. The answer substitution (i.e. the substitution factor) can be found as the value of the *GCP_pointer* minus an offset (*Arity* plus the size of a generator choice point, see Figure 7). The generator choice point also provides access to the *subgoal frame* which, in turn, contains pointers to the root of the subgoal's *answer trie* and to the *answer return list* (see Figure 10). Using the answer substitution, η_A, and the root of the answer trie, new_answer checks whether η_A already exists in the answer trie and inserts

```
Instruction answer_return
  CCP := breg;                          /* B register points to a consumer choice point */
  Call restore_bindings(CCP);           /* restore environment of the suspended consumer */
  Restore values of WAM registers as saved in cells of the CCP;
  if (the last answer consumed by this CCP is not the last answer of the answer return list)
     /* let answer be the first unconsumed answer of the answer return list */
     CCP_LastAnswer(CCP) := answer;  /* mark answer as consumed by CCP */
     Load answer from the answer trie into the substitution factor of CCP;
     Set the program pointer P to the continuation pointer CP; /* continue forward execution */
  else      /* backtrack to another choice point */
     breg := CCP_Breg_Chain(CCP);     /* backtrack */
     fail;                            /* Suspend the node to await further answers */
```

Fig. 17. The answer_return instruction.

it (in the same pass) if not. If the η_A exists in the trie, the derivation path fails, a vital step for ensuring termination. On the other hand, if η_A is new, a new element is added to the end of the *answer return list* which points to the leaf of the answer trie whose path corresponds to η_A, a step which will support answer backtracking by consumer choice points. The new_answer instruction will then deallocate the environment and proceed, by setting the WAM program register to the local environment continuation pointer. This action effectively returns the new answer to the generator node. It is in this manner that the SLG-WAM executes *first-call optimization* and avoids freezing stacks for generator choice points.

ANSWER RETURN. As mentioned, derived answers are immediately returned to the generator node. They also need to be returned to active nodes of the SLG search tree, an action which is performed by the answer_return instruction of consumer choice points. The answer_return instruction is shown in Figure 17 and is executed by failing into a consumer choice point. The instruction begins by restoring the computation state of a consumer choice point, CCP, (i.e., restoring the WAM registers and variable bindings) using information in the consumer choice point and forward trail. If the last answer consumed by this active node (identified by the *LastAnswer* cell of CCP) is not the last element of the answer return list, the next unconsumed answer substitution, η, is loaded into the substitution factor of CCP, and the *LastAnswer* cell is updated, implicitly marking η as consumed by this consumer choice point. The computation then continues by taking the forward continuation of the consumer choice point. Whenever the engine backtracks into CCP, if an unconsumed answer is present in the table, it is returned to the active node; otherwise, if there are no more answers for the active node at the time of backtracking, execution fails to the choice point designated by the *Breg_Chain* cell of CCP.

Conceptually, the *Breg_Chain* cell of a consumer choice point, CCP, can designate two types of information: the choice point of the parent node in the SLG search tree, and a choice point on the consumer choice point chain. A consumer choice point is originally created by a tabletry(single) instruction, and the parent of CCP is initialized to the value of the **B** register when CCP is created. In this case, the engine returns answers to CCP upon backtracking into it in accordance with Rule 3 of Figure 11, until no more answers remain to be returned in this man-

```
Procedure schedule_resumes(SubgFr)
        /* SubgFr is a pointer to the subgoal frame for subgoal S */
  CCP_Head := SF_CCPChain(SubgFr); /* consumer choice point chain for S */
  First_CCP := NULL;
  Starting from CCP_Head traverse the consumer choice point chain and
     set First_CCP to point to the first consumer choice point with unconsumed answers, if any;
  if (First_CCP != NULL)
     Create a consumer choice point backtracking chain, and
        set the Breg_Chain cell of its last element to point
        to the choice point currently pointed by breg;    /* breg is B register */
     breg := First_CCP;
     fail; /* to execute answer_return instructions by picking up the failure continuation */
```

Fig. 18. Pseudo-Code to implement Schedule_Resumes.

ner. As discussed in Section 3.3, a fixpoint-style computation may be necessary in order to completely evaluate all subgoals in a scheduling ASCC. If so, CCP may be resumed after backtracking through its initial batch of answers, and in this case its *Breg_Chain* cell will contain a pointer to a choice point on the consumer choice point chain set by the procedure schedule_resumes.

The functionality of schedule_resumes was introduced in Section 3.3 as part of the fixpoint_check routine in the completion instruction; its pseudocode is shown in Figure 18. The schedule_resumes procedure for a subgoal S checks whether any consumer subgoal of S has unconsumed answers. Recall that consumer subgoals are represented via consumer choice points, maintained in a consumer choice point chain. The head of this chain is accessed via the *CCP_Chain* cell of the subgoal frame, and links of the chain are maintained by the *Prev_CCP* cell of the consumer choice points. Also recall from the description of the tabletrysingle instruction that new consumer choice points are added to the head of the list during tabletry(single). Procedure schedule_resumes begins by constructing a *consumer choice point backtracking chain* for S. The backtracking chain contains all consumer choice points for S that have unresolved answers at the time of fixpoint_check. The elements of the consumer choice point backtracking chain are linked by their *Breg_Chain* so that a new consumer choice point is resumed upon backtracking out of another (as opposed to the consumer choice point chain which uses the *Prev_CCP* cells to link consumer choice points). Figure 18 shows the first element of the consumer choice point backtracking chain as *First_CCP*, and indicates that the *Breg_Chain* failure continuation of the last element on the chain points back to the choice point that initiated the fixpoint_check. Thus, once schedule_resumes has performed an iteration for a subgoal S, fixpoint_check is reinvoked to determine whether another iteration of schedule_resumes is needed for any subgoals in the scheduling ASCC. As a final point, note that performing schedule_resumes for a subgoal does not have any effect on the computation state unless some consuming choice points for that subgoal have unresolved answers.

3.5 Completion in Definite Programs

In this section we first present the algorithms that the SLG-WAM uses to maintain scheduling ASCCs, and then turn to a detailed description of the completion

SubgFr	Pointer to Subgoal Frame
Subg#	Unique Subgoal Number
DirLink	Deepest Direct Dependency

Fig. 19. Format of Completion Stack Frames.

instruction for definite programs.

Implementing Incremental Completion by Approximating Subgoal Dependencies. Incremental completion is necessary for the SLG-WAM to be efficient in terms of space and to be effective on large programs. Incremental completion was first introduced in [Chen et al. 1995] to reclaim the stack space occupied by sets of subgoals when they are determined to be completely evaluated. For example, incremental completion affects the choice point stack of Example 2.1 shown in Figure 4, unfreezing and reclaiming stack space for the subgoals `p(c,Z)` and `p(b,Z)`. Furthermore, incremental completion of subgoals enables the *completed table optimization* described in Section 3.3.

To efficiently perform incremental completion, the SLG-WAM contains an area of memory new to the WAM, the *Completion Stack*, which is used to efficiently keep track of scheduling ASCCs. Specifically, the completion stack maintains, for each subgoal S, a representation of the deepest subgoal S_{dep} upon which S or any subgoal on top of S *may* depend. When S and all subgoals on top of S have exhausted all program and answer clause resolution, S is checked for completion. If S depends on no subgoals deeper than itself, S and all subgoals on top of S are completely evaluated. Otherwise, if S_{dep} is deeper in the completion stack than S, S may depend upon subgoals that appear below it in the completion stack, and cannot be completed. As an aside, we note that for the program of Figure 1, each tabled subgoal can be completed after fixpoint_check and failure over its generator choice point since each component consists of a singleton set of subgoals, but this situation is not the case in general, as will be shown in Example 3.2.

A *completion stack frame* is pushed onto the completion stack when a new tabled subgoal is added to the system (see Figure 15), and is popped off when that subgoal is determined to be completely evaluated by execution of a completion instruction. There is thus a one-to-one correspondence between completion stack frames and generator choice point frames. For definite programs, the format of the completion stack frame is shown in Figure 19 and its cells can be described as follows:

The *Subg#* is a unique number representing the chronological order of encountering the subgoal (assigned through a global counter), *DirLink* keeps track of the deepest direct subgoal dependency (information which is propagated when a consumer choice point is created). In addition, we define for a given state of an SLG evaluation, the function $MinLink(S)$, which is the minimum *DirLink* value for all subgoals on the completion stack whose *Subg#* is greater or equal to $Subg\#(S)$. We briefly present how fields of the completion stack are updated:

—When a *new* tabled subgoal S is called, a unique number is assigned to $Subg\#(S)$, a new frame is pushed onto the completion stack, and $DirLink(S)$ is initialized to $Subg\#(S)$.

—When a tabled subgoal S is called and S is neither new to the evaluation nor

```
Procedure update_dependencies(Subgoal)
        /* Let ComplSF_top be the topmost frame of the completion stack */
    ComplSF := SF_ComplSF(Subgoal);
        /* Completion stack frames are accessed through the corresponding subgoal frame */
    ComplSF_DirLink(ComplSF_top) :=
            min(ComplSF_Dirlink(ComplSF_top), ComplSF_DirLink(ComplSF));
```

Fig. 20. Updating ASCC information on encountering Consumer Subgoals.

complete, let S_{top} represent the subgoal whose frame is on top of the completion stack, and set

$$DirLink(S_{top}) := min(DirLink(S_{top}), DirLink(S))$$

Figure 20 shows the steps performed by the tabletry and tabletrysingle instructions when creating a consumer choice point (cf. Figure 15). Based on these rules and the format of the completion frame, we define Scheduling ASCCs through their leaders as follows.

Definition 3.1 (Leader of a Scheduling ASCC). A subgoal S is called a *leader of a scheduling ASCC* iff the completion frame associated with S is either the deepest one in the completion stack, or satisfies the condition:

$$Subg\#(S_{prev}) < min(DirLink(S), MinLink(S))$$

where S_{prev} is the predecessor of S on the completion stack.

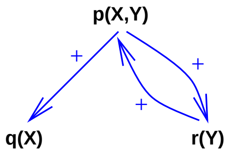
The completion stack can thus be partitioned into scheduling ASCCs, $A_1, \ldots, A_n$, with the property that no subgoal in a given scheduling ASCC depends on any subgoal in a scheduling ASCC deeper in the stack. As a result, the leader of the topmost scheduling ASCC can be used to determine when subgoals in that ASCC can be completed. This property is the basic idea behind the SLG-WAM's implementation of incremental completion. Example 3.2 indicates a further property of incremental completion.

Example 3.2 [Chen et al. 1995]. For the program in Figure 21(a) and query `?- p(X,Y).`, Figure 21(b) depicts the subgoal dependency graph and completion stack at the time of completion of `p(X,Y)`. The order of entries in the Completion Stack reflects the *Subg#* of the subgoals. Subgoal `q(X)`, with *Subg#* 2, is trapped below `r(Y)` with *Subg#* 3, because its *MinLink* is low due to the *DirLink* value from subgoal `r(Y)`. As a result, `p(X,Y)` is the only leader, all three subgoals end up in the same scheduling ASCC and will be completed simultaneously.

Example 3.2 illustrates both a disadvantage and an advantage of scheduling ASCCs. Clearly, `q(X)` is not detected to be completely evaluated as soon as it can be. However, in terms of space reclamation, the detection of completion of `q(X)` is not useful for a stack-based engine. To see this, recall that the order of subgoals in the completion stack reflects that of generator nodes in the choice point stack. Thus if `r(Y)`, which is above `q(X)`, depends on `p(X,Y)` below `q(X)`, there must be an active node with selected literal `p(X,Y)` above `q(X)` in the choice point stack. As a result, all WAM stacks remain frozen by the active node regardless of whether `q(X)` is completed. Space frozen by `q(X)` therefore cannot be unfrozen until the leader

```
:- table p/2, r/1, q/1.

p(X,Y) :- q(X), r(Y).
p(c,a).

q(a).
q(b).

r(c).
r(X) :- p(X,Y).
```

(a) Program



Subgoal	Subg#	DirLink	MinLink(S)
r(Y)	3	1	1
q(X)	2	2	1
p(X,Y)	1	1	1

(b) Subgoal Dependency Graph and Completion Stack

Fig. 21. A Trapped Component (consisting of a single subgoal).

of the scheduling ASCC, `p(X,Y)`, is completed. Scheduling ASCCs are efficiently maintainable, and have good space reclamation properties. Section 6.5 will discuss how to extend the rules presented here so that exact detection of SCCs can be performed when necessary for stratified programs.

The Completion Instruction for Definite Programs. Figure 22 presents the completion instruction for definite programs. Section 3.4 discussed how the scheduling of answer_return instructions is performed by the fixpoint_check procedure as part of the completion instruction for the leader of a scheduling ASCC. A call to this procedure is made in step 1.1 of Figure 22. The fixpoint_check procedure, shown in Figure 23, simply traverses completion stack frames to call schedule_resumes for subgoals in a scheduling ASCC. If there are unconsumed answers for a particular subgoal, schedule_resumes breaks the loop of fixpoint_check by causing the engine to fail and return answers by backtracking through consumer choice points for that subgoal. When this batch of answers has been consumed, the engine once again backtracks to the completion instruction for *Subgoal*. Thus, step 1.2.1 is reached only if all answers have been returned to each subgoal in the scheduling ASCC. In this case, stacks are unfrozen and space is reclaimed (step 1.2.2). More precisely, the stacks are restored to their state at the time *Subgoal* was first called by adjusting the WAM stack and freeze registers (i.e., **B**, **BF**, **E**, **EF**, ...) to their values as saved in the generator choice point of *Subgoal* (see Figure 7). In addition, subgoals in the *ASCC* of *Subgoal* are removed from the completion stack. When this is done execution fails to the previous choice point, as is also the case when *Subgoal* is not a leader.

4. A REVIEW OF LEFT-TO-RIGHT DYNAMIC STRATIFICATION

Stratification theories share a common thread: that a program can be broken up into strata, and that elements of a given stratum may depend negatively only on elements in lower strata. These elements may be predicates or atoms or a mixture of both; and their division into strata may take place either statically or during the program's evaluation. In *Dynamic Stratification* [Przymusinski 1989], the elements are atoms and their division into strata takes place dynamically during a program's evaluation. The power of dynamic stratification arises from a theorem that a program has a two-valued well-founded model if and only if it is dynamically

```
Instruction completion
0      SubgCSF := SF_ComplSF(GCP_SubgFr(breg));
                 /* B register (breg) points to the Generator CP of Subgoal */
1      If (Subgoal is the leader of a scheduling ASCC, A)
                 /* using SubgCSF according to Definition 3.1 */
1.1         Call fixpoint_check(SubgCSF);
1.2.1       Mark as complete all subgoals in A;
1.2.2       Reclaim the stack space of subgoals in A and adjust the freeze registers;
2      breg := GCP_BregChain(breg);
3      fail;
```

Fig. 22. The **completion** instruction (for Definite Programs).

```
Procedure fixpoint_check(SubgCSF)  /* SubgCSF is a pointer to the completion stack frame */
  while (SubgCSF is less than or equal to the top of the completion stack)
    SubgFr := CSF_SubgFr(SubgCSF);
    Call schedule_resumes(SubgFr); /* a failure continuation is taken if any consumer choice */
                   /* point associated with SubgFr has unconsumed answers (cf. Figure 18) */
    Increment SubgCSF by the size of a completion stack frame;
```

Fig. 23. The Fixpoint_Check Procedure.

stratified.

Evaluation of dynamically stratified programs cannot be done using a fixed computation rule as shown by Przymusinski [1989]. Within SLG, the ability to alter a computation rule is addressed by DELAYING and SIMPLIFICATION operations. These operations can be expensive and can deeply affect the SLG-WAM. However, by restricting the definition of dynamic stratification to fixed-order computations, the useful subclass of *Left-to-Right Dynamically Stratified* programs (*LRD-stratified* programs) arises. As we will show, this class can be efficiently evaluated without elaborate modifications of the definite engine. LRD-stratified programs were introduced in [Sagonas et al. 1996], along with the variant of SLG, SLG_{strat} that we use throughout the remaining part of this article. It can be shown that the class of LRD-stratified programs properly contains the class of left-to-right weakly stratified programs, which in turn properly contains the class of left-to-right modularly stratified programs. Further, it was shown in [Ross 1994] that *all* modularly stratified programs are statically reorderable into this later class. Figure 24 provides an example of a left-to-right dynamically stratified program and a dynamically stratified (but not LRD-stratified) program. We note that the LRD-stratified program in Figure 24(a) is neither modularly nor weakly stratified.

Intuitively, LRD-stratified programs are those with two-valued well-founded models that can be evaluated using a fixed left-to-right literal selection strategy. Formally, these programs are defined by adapting Przymusinski's iterated fixed point for the well-founded semantics [Przymusinski 1989] to a fixed left-to-right computation rule. Our single modification is the introduction of the *failing prefix* constraint in Definition 4.1. This constraint restricts false facts from being included in $\mathcal{F}_I(F)$, unless their falsity can be established by a left-to-right examination of literals.

DEFINITION 4.1. For sets T and F of ground atoms

(a) LRD-Stratified	(b) Dynamically Stratified

`p :- q, not r, not s.`
`q :- r, not p.`
`r :- p, not q.`
`s :- not p, not q, not r.`
	`p :- not s, not r, q.`
	`q :- r, not p.`
	`r :- p, not q.`
	`s :- not p, not q, not r.`

Fig. 24. Program Examples for Dynamically Stratified Negation.

$\mathcal{T}_I(T) = \{A \mid$ there is a clause $B \leftarrow L_1, ..., L_n$ in P and a ground substitution θ such that $A = B\theta$ and for every $1 \leq i \leq n$ either $L_i\theta$ is true in I, or $L_i\theta \in T\}$;

$\mathcal{F}_I(F) = \{A \mid$ for every clause $B \leftarrow L_1, ..., L_n$ in P and ground substitution θ such that $A = B\theta$ (1) there is some i $(1 \leq i \leq n)$, such that $L_i\theta$ is false in I or $L_i\theta \in F$, and (2) there exists a *failing prefix*: for all j $(1 \leq j \leq i-1)$, $L_j\theta$ is true in $I\}$.

In $\mathcal{T}_I$ and $\mathcal{F}_I$, I represents facts shown to be true or false in a previous fixpoint derivation. These operators serve as primitives upon which inner fixpoint operators T_I and F_I can be built.

DEFINITION 4.2. Let $I = \langle T; F\rangle$ be a partial interpretation,

$$T_I^{\uparrow 0} = \emptyset \text{ and } F_I^{\downarrow 0} = H_P$$
$$T_I^{\uparrow n+1} = \mathcal{T}_I(T_I^{\uparrow n}) \text{ and } F_I^{\downarrow n+1} = \mathcal{F}_I(F_I^{\downarrow n})$$
$$T_I = \bigcup_{n<\omega} T_I^{\uparrow n} \text{ and } F_I = \bigcap_{n<\omega} F_I^{\downarrow n}.$$

Further, define $\mathcal{I}(I)$ as: $\mathcal{I}(I) = I \cup \langle T_I; F_I\rangle$.

The outer (transfinite) fixpoint is based, according to the usual definitions, on the operator $\mathcal{I}$ which extends the interpretation I to $\mathcal{I}(I)$ by adding to I: (1) new atomic facts T_I which can be derived from P knowing I, along with (2) negations of atoms in unfounded sets based on the interpretation I. Using this framework a LRD-stratified program is defined as one in which the iterated fixpoint of $\mathcal{I}$ produces a two-valued model (i.e. one in which no atom is undefined). When this model exists, it is equal to the well-founded model for P.

5. TABLING OPERATIONS FOR LRD-STRATIFIED PROGRAMS

At the level of tabling operations, the intuition behind the evaluation of LRD-stratified programs is that nodes with selected negative literals are suspended using mechanisms similar to those of Section 3.1, and are resumed only when the subgoals for those literals are *failed*: i.e., when they are completed with no answers. Along with the SLG operations for definite programs of Definition 2.3, the following operations are used in LRD-stratified programs.

DEFINITION 5.1 (SLG OPERATIONS FOR LRD-STRATIFIED PROGRAMS). Let N be an *active* node of an SLG tree of the form $Answer\_Template$:− not $S, Goals$ where S is a subgoal of a tabled predicate.

$\boxed{\text{FLOUNDERING}}$. If S is non-ground, then the evaluation is *floundered.*

$\boxed{\text{NEGATION FAILURE}}$. If S is ground and has an answer, then create a *fail* node as the immediate child of N in its SLG tree.

$\boxed{\text{NEGATION SUCCESS}}$. If S is ground and is failed, then produce an immediate child of N of the form: *Answer_Template* :– *Goals*.

Creating a *fail* node in an SLG tree effectively fails the computation path to the *fail* node. If an evaluation encounters a literal not S and S is not yet in the system, the NEW SUBGOAL operation takes place; i.e., a new SLG tree rooted by a *generator* node is created for S, and PROGRAM CLAUSE RESOLUTION is used to derive answers for it. No operations are applicable for node N containing a ground literal not S until either an answer is derived for S (at which time a NEGATION FAILURE operation would be applicable), or until S is failed, when a NEGATION SUCCESS operation would become applicable. The following theorem, slightly modified from Sagonas et al. [1996], indicates the validity of the approach outlined.

THEOREM 5.1. *Let P be a ground LRD-stratified program, and let $\mathcal{E}$ be an SLG evaluation of P consisting of the operations* NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, ANSWER RETURN, NEW ANSWER, COMPLETION, NEGATION FAILURE, *and* NEGATION SUCCESS. *Then $\mathcal{E}$ will reach a final state that is* not *flummoxed.*

Together with the correctness of SLG, Theorem 5.1 implies that the above set of operations suffices to evaluate LRD-stratified programs without the SLG DELAYING, SIMPLIFICATION and ANSWER COMPLETION operations (see [Chen and Warren 1996]). As will be discussed in Section 6, the engine makes direct use of this result.

Other evaluation mechanisms are of course possible. For instance the approach of Chen et al. [1995] applies the SLG DELAYING operation whenever there is a node with a selected negative literal in a *ASCC* that is being checked for completion. Such an approach has the disadvantage that DELAYING breaks the fixed order of computation for N, perhaps unnecessarily expanding the search space of the program. As implied by Theorem 5.1, in LRD-stratified programs this search space expansion can be avoided.

6. THE ABSTRACT MACHINE FOR LRD-STRATIFIED PROGRAMS

In order to evaluate LRD-stratified programs, five main changes are made to the definite engine: 1) implementation of early completion, 2) implementation of a stratified negation operator, 3) a suspend and resume operation for selected negative literals, 4) explicit maintenance of subgoal dependencies, and 5) a completion instruction with the ability to determine SCCs precisely and complete them independently of their stack-based order. We discuss each of these changes in turn.

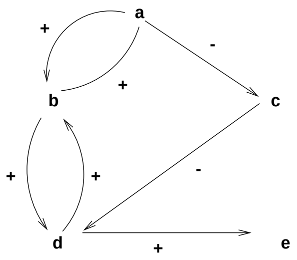
6.1 Implementation of Early Completion

Early completion (condition 2 in Definition 2.5), or the ability to complete subgoals whose truth value has been established without taking into account their possible dependence on other subgoals, is necessary to evaluate ground LRD-stratified programs without breaking a fixed literal selection strategy. Example 6.1 illustrates one case of this.

EXAMPLE 6.1. Let P be the LRD-stratified program in Figure 25(a) for which

```
:- table a/0,b/0,c/0,d/0,e/0.

a :- b, not c.

b :- a.
b :- d.
b.

c :- not d.

d :- b, e.
```

(a) A LRD-stratified Program



(b) Subgoal Dependency Graph

Fig. 25. Program showing the need for early completion.

the query ?- a is to be evaluated. The execution of this query against P causes cascading suspensions, of a on c and c on d, as seen from the subgoal dependency graph shown in Figure 25(b). Observe that b has been completely evaluated. If b were explicitly completed, it could be removed from the SDG, and the apparent loop through negation (subgoals a, b, c, and d) could be broken.

To perform early completion the engine must check if a subgoal's answer is a variant of the subgoal itself. The SLG-WAM of XSB currently implements early completion in the case where the subgoal is ground. Early completion is thus easily implemented during the answer check/insert step of the new_answer instruction of Figure 16. Recall from Section 3.4 that new_answer is compiled as the last instruction of each clause in a tabled predicate and it uses the substitution factor from the generator choice point to add an answer to the table. If the number of variables in the subgoal is equal to 0, the subgoal is ground and it may be (early) completed upon addition of its answer. In such a case, the subgoal frame, which is accessible through the generator choice point, is marked as complete, and its pointer to the nodes depending negatively on the subgoal set to null (the *NS_Chain* cell: see Figure 10). In addition, the *FailCont* cell of the generator choice point (see Figure 7 in Section 3.1.4) is made to point to a completion instruction. This action bypasses any possible remaining tableretry and tabletrust instructions for that subgoal [9]. The completion instruction will return the answer to any consuming choice points through the fixpoint_check procedure, and will revise dependency information to take account of subgoals that have been early completed (see Section 6.5).

6.2 Implementation of a Predicate for Fixed-Order Stratified Negation

Because negation is restricted to ground literals, whenever an answer is derived for a subgal S, a NEGATION FAILURE operation becomes applicable to any active nodes with not S as their selected literal. As mentioned in the previous section the SLG-WAM removes pointers to such active nodes upon early completion of S; these active nodes will never be resumed, so that NEGATION FAILURE operations are executed implicitly upon early completion. These considerations lead to the

[9] It will be shown in Section 7 that early completion can also benefit certain definite programs because of this action.

```
tnot(S) :-
    ( ground(S) →
        ( subgoal_not_in_system(S), call(S), fail
        ; (is_complete(S) → has_no_answers(S)
          ; negation_suspend(S), true    /* if execution reaches here, S */
          )                              /* is completed with no answers */
        )
    ; error("Flounder: subgoal S is not ground")
    ).
```

Fig. 26. An implementation of tabled negation (tnot/1) for LRD-stratified programs.

FailCont	Pointer to **negation_resume** instruction
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of (Forward) Trail Stack
CPreg	Return point of suspended literal
Ereg	Parent Environment
RSreg	Root Subgoal Choice Point
SubgFr	Frame of Suspended Subgoal
PrevNS	Pointer for Negation Suspension Frame Chain

Fig. 27. Format of Negation Suspension Frames.

following invariant:

INVARIANT 1. *In the SLG-WAM for LRD-stratified programs, the completion of subgoals initiates only* NEGATION SUCCESS *operations.*

The predicate tnot/1 implements negation for LRD-stratified programs, and, as shown in Figure 26, makes use of this invariant. tnot/1 is implemented using low-level builtins. Since any ground subgoal with an answer is marked as complete by early completion, tnot/1 calls the negation_suspend/1 builtin only if the subgoal is incomplete (and has no answer). Later, according to Invariant 1, the computation resumes (to true) only if the completed subgoal has no answers. The exact mechanisms of suspending and resuming negative literals can now be described.

6.3 Suspending and Resuming Negative Literals

The operation of suspending negative literals is implemented through a C-level builtin negation_suspend/1 (cf. Figure 26). This builtin pushes a *negation suspension frame* onto the choice point stack and then suspends the computation by freezing the stacks and failing. The *negation suspension frame* (whose format is shown in Figure 27) saves the execution environment for the suspended computation in a manner similar to saving suspended environments for consumer choice points. Like consumer choice points, negation suspension frames of the same subgoal are chained together in a chain (using the *PrevNS* cell) and can be accessed from the subgoal frame (through their *NS_Chain* cell: see Figure 10).

The completion instruction schedules negation_resume instructions in a manner similar to the way it schedules answer_returns. For each subgoal, its *negation suspension frames* are chained together using their *PrevNS* cells, and these subchains are chained together into one chain upon completion of corresponding subgoals. The engine then backtracks to execute negation_resume instructions for each negative active node suspended on one of the completed subgoals. Upon executing a negation_resume instruction, the engine will use the forward trail to resume the suspended computations and continue execution. Because, the SLG-WAM calls negation_suspend/1 builtins only through `tnot/1`, continued execution will immediately succeed out of the `tnot/1` predicate, implicitly performing a NEGATION SUCCESS operation.

6.4 Maintenance of the Subgoal Dependency Graph

As Definition 2.4 implies, vertices of the SDG are incomplete subgoals, and edges are drawn between the root subgoals of incomplete SLG trees and the selected subgoals of their active nodes. Within the SLG-WAM, the SDG can be effectively represented by maintaining pointers from consumer choice points to their root subgoals, and — if first call optimization is used — from generator choice points to the appropriate root subgoals. [10]

In the WAM, global information, such as the root subgoal of the node currently under execution is kept in registers, and we therefore introduce a global **RS** register (short for *Root Subgoal register*) to keep track of this dependency. All choice point frames, including those for interior nodes, need to maintain the value of this register, and do so in their *RSreg* cell (see Figures 7, 8 and 27). The **RS** register is updated as follows:

—First, the **RS** register is modified upon creating the generator node for a new SLG tree. This is performed by the tabletry and tabletrysingle instructions, after the creation of the generator choice point frame. The value of the **RS** register is set to the address of that choice point.

—Secondly, the **RS** register must also be restored when the computation successfully exits an SLG tree by, say, deriving an answer. This restoration of the **RS** register during forward execution is performed by the new_answer instruction. Note that restoration during forward execution is unnecessary for interior nodes since the SLG tree in which computation takes place is not affected by executing PROGRAM CLAUSE RESOLUTION in the forward direction.

—Thirdly, the **RS** register must be restored when the computation executes a failure continuation, potentially switching to a new tree. This can occur either when executing PROGRAM CLAUSE RESOLUTION by the retry, trust, tableretry, and tabletrust instructions; when returning an answer by the answer_return instruction; or when executing a negation_resume instruction.

6.5 Completion in LRD-Stratified Programs

In an LRD-stratified program there is nothing to prevent a given subgoal in an ASCC, A, from depending negatively on another subgoal in A. If an engine is to evaluate LRD-stratified programs using a fixed computation rule, it must correctly order the completion of subgoals and the execution of NEGATION SUCCESS operations. We first discuss how exact SCC detection is done in our framework, and then present the completion instruction for LRD-stratified programs.

[10] As a technical point, these pointers maintain the transpose of the SDG (SDG^T) rather than the SDG itself.

```
:- table p/1.

p(a) :- p(b), not p(d).

p(b) :- p(c).
p(b) :- not p(d).
p(b).
p(b) :- not p(a).

p(c) :- p(b), p(e).

p(d) :- not p(c), p(d)

p(e) :- p(c).
p(e) :- not p(b), not p(e).
```

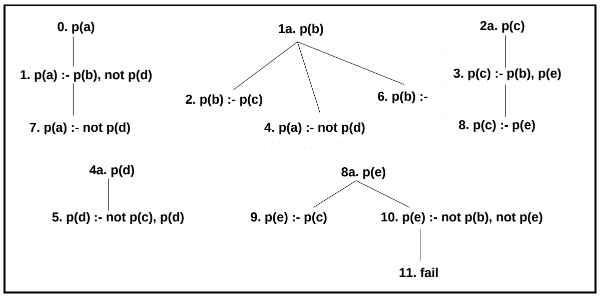
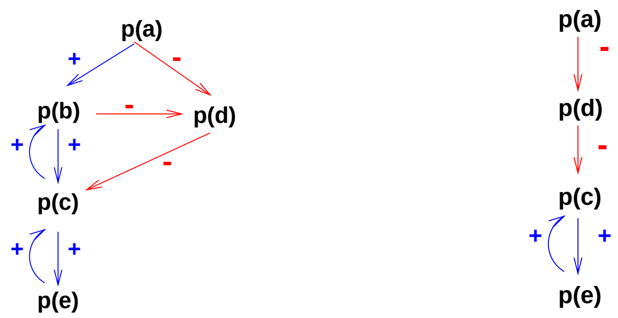


Fig. 28. A LRD-stratified Program and the SLG Forest created for the query ?- p(a).



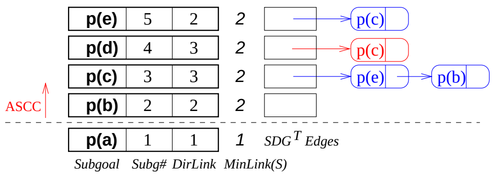
(a) For all subgoals in the forest (b) When disregarding early completed subgoals

Fig. 29. Subgoal Dependency Graphs for the query ?- p(a).

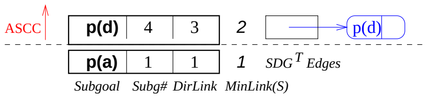
6.5.1 *Performing* COMPLETION *based on Exact Subgoal Dependencies*

EXAMPLE 6.2. Let P be the program in Figure 28 for which the query ?- p(a). is to be evaluated. Note that since there is only one predicate p/1, P is not modularly stratified for any selection order. It is, however, LRD-stratified. The SLG forest of Figure 28 depicts a state of the evaluation of p(a) in which there are apparent cycles through negation, as can be seen from the associated *SDG* in Figure 29(a). Note that in this state a PROGRAM CLAUSE RESOLUTION step has not been applied using the last clause of p(b). Because the subgoal p(b) is ground and contains an answer, p(b) may be early completed, producing the SDG of Figure 29(b), which contains no loops through negation. The SCC {p(c),p(e)} is then found to be completely evaluated according to Definition 2.5, and a COMPLETION operation is applicable to the subgoals of this SCC.

In order to describe how the SLG-WAM performs the computation described in Example 6.2, we first consider how the completion stack may be augmented



(a) Before completion of p(b), p(c) and p(e).



(b) After their completion.

Fig. 30. Completion Stack states when evaluating the program of Example 6.2.

to perform exact SCC detection. Figure 30(a) shows the completion stack and $MinLink(S)$ values at the state of computation depicted in the SLG forest of Figure 28. According to the definitions given in Section 3.5, `p(b)` is the leader of a scheduling ASCC containing `p(c)`, `p(d)`, `p(e)` and itself. In order for the SLG-WAM to determine the order of completion for subgoals in the scheduling ASCC, it augments the completion stack with reverse dependency links. As Figure 30 illustrates, this augmentation effectively constructs the transpose of the SDG restricted to incomplete subgoals in the scheduling ASCC. [11] At this point, an independent SCC is obtained by performing a combination of a topological sort and an SCC computation of a directed graph [Cormen et al. 1990].

Example 6.3. Continuing Example 6.2, the completion operation for the scheduling ASCC led by `p(b)` finds subgoals `p(c)` and `p(e)` to be an independent SCC, and completes them. The completion frames of these subgoals, as well as that of `p(b)`, which was early completed, are removed from the completion stack. Also, their completion initiates a negation success operation for the node `p(d) :- not p(c), p(d)`. When computation resumes for this node, the literal `p(d)` is selected, and the subgoal dependency graph is modified. The resulting completion stack of the new computation state is depicted in Figure 30(b).

Only completion operations are applicable at this point. A completion operation for `p(d)` is performed and `p(d)` is found to be the leader of its scheduling ASCC and is completed. Finally, literal `not p(d)` in the body of `p(a)` is resumed (using a negation_resume instruction) and succeeds, which in turn activates the early completion of subgoal `p(a)` upon the derivation of its answer.

As the example shows, the approximation of the strongly connected components kept by the completion stack may considerably change as a result of negation success operations, and fresh dependency information may have to be added to the completion stack whenever exact SCC detection is required.

6.5.2 *The* completion *instruction for LRD-stratified Programs.* The completion instruction for LRD-stratified programs is shown in Figure 31. With the exception of the test in step 1.2, up to line 1.2.2, and in steps 2–3 the actions of the completion instruction for LRD-stratified programs are the same as for definite programs. The

[11] We note that as an optimization, links do not need to be created for subgoals that are completed, but whose frames remain on the completion stack.

Instruction completion

```
0       SubgCSF := SF_ComplSF(GCP_SubgFr(breg));
                        /* B register (breg) points to the generator CP of Subgoal */
1       if (Subgoal is a leader of a scheduling ASCC, A)
                        /* using SubgCSF according to Definition 3.1 */
1.1         Call fixpoint_check(SubgCSF);
1.2         if (there are no negation suspensions on subgoals in A)
1.2.1           Mark as complete all subgoals in A;
1.2.2           Reclaim the stack space of subgoals in A and adjust the freeze registers;
            else
1.3             I := independent_scc(Subgoal);
1.4             For each subgoal S ∈ I
1.4.1               Mark the subgoal frame of S as complete;
1.4.2               if (there are negation suspensions on S)
1.4.3                   if (there exists a subgoal S' ∈ I that is suspended on S)
1.4.4                       Abort: the program is not LRD-stratified;
1.4.5               else Schedule negation_resume instructions for S by chaining together
                            the negation suspension frames for all completed subgoals;
1.5             Let E be the set of subgoals of A that were early completed and let C := E ∪ I;
1.5.1           Compact the completion stack by removing the frames of subgoals in C;
1.5.2           If possible, reclaim the stack space of subgoals in C and adjust the freeze registers;
2       breg := GCP_BregChain(breg);
3       fail;
```

Fig. 31. The **completion** instruction (for LRD-stratified Programs).

instruction is scheduled on the choice point stack either by tabletrysingle or tabletrust when PROGRAM CLAUSE RESOLUTION is no longer applicable for a subgoal *Subgoal*, or by the new_answer instruction in the case of early completion. Upon execution, if *Subgoal* is the leader of its scheduling ASCC, the completion instruction for *Subgoal* will first access the subgoal frame and perform a fixpoint_check to ensure that all ANSWER RETURN operations have been performed for active nodes in the scheduling ASCC of *Subgoal*. If the *Subgoal* is not a leader, the action of the completion instruction is simply to backtrack to the previous choice point. If *Subgoal* is the leader of a scheduling ASCC, a check is made whether there are negative dependencies on any members of the ASCC (the *NS_chain* pointers of each subgoal frame are used for this check). If no such negative dependencies are present, all subgoals in the ASCC can be completed and their space reclaimed, just as in the definite case. If there is a negative dependency on some subgoal of the ASCC, the engine refines the approximation of the scheduling ASCC by finding an independent SCC as explained in the previous section (step 1.3). Once an independent SCC, I, is obtained, its subgoals are completed and a check made for whether the program is LRD-stratified using the property that the relevant portion of a program is LRD-stratified iff, after disregarding early completed subgoals, no independent SCC contains negative dependencies among its subgoals. If it is sound to continue, negation_resume instructions are scheduled for all nodes that were suspended on the completion of the subgoals in I (step 1.4.5). This implementation of this scheduling is analogous to that of the fixpoint_check procedure. Finally, the completion stack is compacted to remove frames of complete subgoals, the remaining choice points are (re)chained through their *Breg_Chain* cell, and, if possible, stack space is reclaimed for subgoals in the independent SCC. This is always possible when the bottom of the completion stack is reached.

The correctness of the completion algorithm follows from the proposition below, which can be proven using the formalism of SLG_O automata in [Swift 1994].

PROPOSITION 6.1 (CORRECTNESS OF completion ALGORITHM). *In an evaluation of a ground LRD-stratified program, let L be the leader of a scheduling ASCC A in the completion stack. Furthermore, assume that all applicable SLG operations of Definitions 2.3 and 5.1 (but for* COMPLETION *itself) have been performed for subgoals in A. Finally, let C be the set of subgoals in step 1.5 of Figure 31. Then:*

(*1*) *C will be non-empty;*

(*2*) *No subgoal will be in C unless it is completely evaluated.*

7. PERFORMANCE

Previous sections have described how the SLG-WAM extends the WAM so that tabling can be intermixed with Prolog execution. We adopt two ideal criteria for judging the success of the engine.

(1) *Performance overheads should be minimized.* Prolog programs should not pay a penalty for tabling mechanisms that they do not use. Likewise, definite programs that use tabling should not pay a penalty for mechanisms added for stratified negation.

(2) *Performance times of tabled and non-tabled code of similar complexity (cf. Section 7.2) should be compatible.* Performance times of both types of predicates should be similar if tabled evaluation is to be used to solve practical problems.

This section measures the performance of the SLG-WAM using these criteria. Additional comparisons of the SLG-WAM against other tabling systems and deductive databases can be found in [Sagonas et al. 1994; Chen et al. 1995; Swift and Warren 1994; Ramesh and Chen 1997; Rao et al. 1996].

7.1 Measuring Performance Overheads

7.1.1 *Overheads Imposed on Prolog Programs.* When the SLG-WAM executes Prolog code, performance differences with the WAM can arise from several factors: from the forward trail, from the introduction of freeze registers, and from other miscellaneous factors such as the addition of words to Prolog choice points (the *Breg_Chain* cell and the *RSreg* cell, whose uses were explained in previous sections). Of these differences, the forward trail affects every trailed binding, and each environment restoration at backtracking. The freeze registers affect the allocate and backtracking instructions, but moreover the values of **EF** and **HF** registers need to be checked at every variable binding in order to determine whether the variable has been created since the last choice point.

In summary, the differences with the WAM are:

—The try, retry, and trust instructions are changed due to freeze registers, to the forward trail, and due to the addition of extra cells in choice points.

—The allocate instruction is changed due to freeze registers.

	deriv	qsort	nreverse	serialise	query	Mean
WAM	1	1	1	1	1	1
SLG-WAM: WAM-trail	1.10	1.10	1.04	1.04	1.09	1.08
SLG-WAM: Definite	1.16	1.11	1.05	1.09	1.13	1.13
SLG-WAM: LRD	1.16	1.11	1.05	1.09	1.13	1.13

Table I. Normalized CPU times for executing standard Prolog benchmarks.

—The get and unify instructions are changed due to augmented trail frames, and due to the incorporation of freeze registers in the check for whether trailing is necessary.

To measure the effect of these the following versions of the engine were created along with an unmodified ***WAM*** engine.

SLG-WAM: WAM-trail. Contains freeze registers, but WAM-style trail.

SLG-WAM: Definite. Performs SLG evaluation for definite programs only. It contains a forward trail as well as freeze registers.

SLG-WAM: LRD. Performs SLG evaluation for LRD-stratified programs. It contains all additions and changes to the WAM described in this article.

Normalized CPU times of all emulators are compared for five standard benchmarks from the D.H.D. Warren test suite in Table I.[12]

For `qsort`, `nreverse`, and `query` the increase in time appears to be due to the addition of the freeze registers, while for `serialise` it is due to writing trail cells. `query`, and to some extent `qsort` also test the efficiency of shallow backtracking. However `query`, `qsort`, and `nreverse` rarely actually trail variables either because the predicates are called with instantiated arguments, or because the variables that are bound do not lie below a choice point. The `serialise` benchmark, on the other hand, builds a structure which is successively instantiated at a progressively deeper level, creating trail frames.

For the `deriv` benchmark, the performance of compiled cuts is also tested. Due to the complications stemming from environment switching, cuts can be expensive in the SLG-WAM. To measure the effect of the SLG-WAM cuts on `deriv`, a version of the emulator was created with cuts compiled as in the WAM. This version had a normalized time of 1.11, indicating a sensitivity to the cut extensions.

The addition overhead of changes to evaluate LRD-stratified programs was negligible (less than 1%) for these benchmark programs, so that it is probably safe to conclude that on average the changes to the WAM described in this article add about a 10-15% overhead to Prolog CPU times.

In order to test memory usage, the LRD engine was tested against a vanilla WAM engine. For Prolog programs, the SLG-WAM consumes more memory than the WAM due to its larger choice point and to trail frames which consist of three words rather than one word. Surprisingly, for the above benchmarks, memory usage is only about 5% higher than in the WAM. For these benchmark programs, bindings to variables usually occurs in deterministic predicates: those for which

[12] All tests in this section were done on a SPARC 20 running SunOS 5.4. The compilation of XSB was done with gcc 2.7.0 (using the `-O4` option)

	TC-chain	TC-cycle	TC-tree	same gen.
SLG-WAM: Definite	1	1	1	1
SLG-WAM: LRD	0.947	0.945	0.979	0.961
SLG-WAM: No-EC	1.008	1.009	1.012	1.010

Table II. Normalized CPU times for executing tabling benchmarks using XSB.

	TC-chain	TC-cycle	same gen.
CORAL SemiNaive	1	1	1
CORAL Ordered Search	1.42	1.45	1.30

Table III. Normalized CPU times for Ordered Search compared to seminaive evaluation in CORAL.

only one clause can succeed due to indexing or to the use of cuts. As has been noted in other, more detailed studies (e.g. [Taylor 1991; Tick 1988; Van Roy 1990]), the actual creation of trail frames can usually be avoided.[13]

7.1.2 *Overhead for the Evaluation of LRD-stratified Programs.* The previous section measured the overhead of the engine for stratified negation on Prolog programs. In this section, we further measure the performance of this engine on definite programs that use tabling (Figure 32). Table II contains normalized execution times for left-recursive transitive closure (over a chain, a cycle, and a full binary tree data structure, all of size 8k), and a same generation program (over a randomly generated $24 \times 24 \times 2$ cylinder). A cylinder can be thought of as a rectangular matrix of elements where each element in row i has links to a certain number of elements in row $i+1$. The $24 \times 24 \times 2$ cylinder then, is an array of 24×24 nodes, where each of the nodes in each row (except the last) is connected to two elements in the next row. None of these programs contains negative literals. It is somewhat surprising that the engine for stratified programs outperforms (if slightly) the engine for definite programs. The third line of the table measures the performance of an engine with all changes for negation *except* early completion. With this information it can be seen that the advantage of early completion outweights the overheads of other changes to implement LRD-stratified programs.

Comparison with Other Evaluation Strategies for Stratified Negation. To put the numbers of the previous section in perspective, we compared the overhead of the SLG-WAM's algorithm for stratified negation with Ordered Search [Ramakrishnan et al. 1992a], a magic-oriented strategy implemented in the CORAL system [Ramakrishnan et al. 1992b]. The default strategy for CORAL is *Supplementary Magic Rewriting* which correctly evaluates definite programs. This default strategy was not designed for stratified programs, and if such programs are to be evaluated Ordered Search, which correctly evaluates left-to-right modularly stratified programs, should be used. Table III compares performance of Ordered Search with *Supplementary Magic Rewriting* on definite programs. The results in Table III show that Ordered Search is considerably less efficient than ordinary seminaive fixpoint evaluation (around 40% slower). CORAL provides many annotations that affect the

[13] This memory comparison was obtained using a SLG-WAM with trail compaction added. Without trail compaction (as in XSB version 1.7), the memory overhead is 18%.

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

(a) Left-Recursive Transitive Closure

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

(b) Right-Recursive Transitive Closure

```
sg(X,Y) :- cyl(X,X1), sg(X1,Y1), cyl(Y,Y1).
sg(X,X).
```

(c) Same Generation

```
even(0).
even(X) :- X > 1, Y = X-1, not even(Y).
```

(d) Even

```
path(X,Y) :- path(X,Z), edge(Z,Y), not congested(Y).
path(X,Y) :- edge(X,Y).
```

(e) Congested

Fig. 32. Test Programs (versions with Prolog-style negation).

	even	congested
CORAL SemiNaive	1	1
CORAL Ordered Search	1.71	12.33

Table IV. Normalized CPU times for CORAL's evaluation strategies on programs with negation.

performance of programs; for both evaluation strategies the timings reported are the best that could be obtained by setting these options.

We also measured the performance overhead of both methods on programs that contain negation, but no negative loops, and which can be evaluated using SLDNF or a simple semi-naive search strategy. The benchmarks `even`, and `congested` are shown in Figures 32(d) and 32(e) (these programs can be found in the examples directory of in CORAL manual). In the case of the `congested` program, predicate `congested/1` contains recursion but no negation and serves as a test of whether a particular path is valid. As can be seen from the results of Table IV, Ordered Search can impose a performance penalty on the execution of stratified programs that do not need its power. To determine the overhead of tabled negation in XSB for these programs, the first two rows of Table V compare the performance of tabling using (`tnot/1`) and Prolog-style negation (`not/1`). As a further comparison, the last row of the same table represents the performance of SLDNF evaluation (the left-recursive transitive closure of `congested` was manually transformed to right recursion for the SLDNF test).

These performance numbers indicate a small overhead for the additional functionality of tabled-based SLG negation relative to Ordered Search. We believe that

	even	congested
SLG-`not/1`	1	1
SLG-`tnot/1`	1.19	1.23
SLD-`not/1`	0.72	0.71

Table V. Normalized CPU times for different types of negation in XSB.

Length	8	16	32	64	128	256	512	1k	2k
SLD	.56	.53	.67	.78	.71	.78	.78	.75	.73
SLG-cycle/chain	1	1	1	1	1	1	1	1	1

Table VI. Normalized CPU Times for SLD and SLG Transitive Closure on Chains

these results reflect fundamental aspects of the computation strategies involved, rather than accidents of implementation. As Section 6.4 indicates, it is a simple matter to use the SLG search forest to maintain dependencies between subgoals. On the other hand, such a structure does not naturally follow from a semi-naive evaluation. Rather a set of *context nodes*, which together serve as an analogue to the subgoal dependency graph, must be built from scratch, leading to the observed overheads. The SLG-WAM's small overhead is especially striking since, XSB has been shown to be about an order of magnitude faster than CORAL for definite Datalog queries [Sagonas et al. 1994].

7.2 Measuring Performance Compatibility

As shown in Section 7.1, SLG-WAM overhead for SLD resolution is minimal. When XSB is used simply as a Prolog system (i.e., no tabling is used), it is reasonably competitive with other Prolog implementations based on a WAM emulator written in C or assembly. For example, XSB is slightly faster than NU-Prolog and is around three times slower than Quintus 3.1.1 or emulated SICStus Prolog 2.1.9.

In general, performance times of tabled and non-tabled predicates may vary widely: certain tabled predicates may not terminate in SLD or their complexity may become exponential, while simple Prolog predicates, such as `append/3` with the first two arguments instantiated, usually become quadratic when tabled. Datalog programs with no redundant subcomputations form one class of programs for which the complexity of both methods is the same. Two examples of this are transitive closure over trees and chains, and Tables VI and VII show the normalized times for the query `?- path(1,X),fail.` using XSB. In these tables the right-recursive form of transitive closure was used for SLD (Figure 32(b)) against its left-recursive version for SLG (Figure 32(a)). The left-recursive, SLG derivation is only slightly slower than SLD for the chains and trees. Relative times for the tree are closer than for the chain because SLD evaluations over the tree execute backtracking instructions to traverse the immediate children of a given node, and these are less efficient operations in the WAM. For example, a choice point is set up at the subgoal `edge(1,X)` because it unifies with both `edge(1,2)` and `edge(1,3)`. The similarities in the speed of SLD and SLG on the chain and tree are especially significant since the SLG times include time to copy answers to and from the table space.

Memory usage for Prolog execution of the transitive closure in Figure 32(b) over a chain will be constant. Assuming 32-bit addresses and a split-stack WAM, 60 bytes

Height	6	7	8	9	10	11
SLD	.89	.82	.87	.88	.85	.84
SLG	1	1	1	1	1	1

Table VII. Normalized CPU Times for SLD and SLG Transitive Closure on Complete Binary Trees

of stack space will be required to backtrack through all solutions of the transitive closure (The 60 bytes is comprised of 1 local environment frame, one trail cell, and one choice point frame). We present a detailed analysis of memory usage of SLG transitive closure in Appendix A.1, and summarize the results here. Like Prolog, tabled execution will require a constant stack space of 192 bytes. In addition, tabled execution requires space for tabled subgoals and answers. The subgoal trie for `p(1,Y)` requires 92 bytes, while the answer trie requires 28 bytes per answer for this subgoal. The order of the clauses does not affect memory usage for the tabled program, but if the order of the clauses in Figure 32(b) are interchanged, the Prolog program creates a choice point (of 32 bytes) for each `path/2` subgoal. Surprisingly, in this latter case, Prolog becomes less efficient in terms of memory than tabling.

Memory usage of tabled evaluation is the same when transitive closure is executed over trees as when executed over a chain: stack space is constant and table space grows linearly with the number of answers. On the other hand, regardless of the order of clauses in Figure 32(b), the size of the choice point stack for Prolog execution will grow with the depth of the tree.

8. DISCUSSION

Extending the SLG-WAM to evaluate non-stratified programs according to the well-founded semantics has already begun, with version 1.7 of XSB offering an initial implementation of this engine. The main components of this extension are the introduction of the SLG DELAYING and SIMPLIFICATION operations to allow the engine to evaluate body literals in a flexible order [Sagonas 1996]. As the LRD-stratified extensions avoided slowing down SLD resolution and tabled evaluation of definite programs, one goal of these new extensions is to avoid slowing down SLD and tabled evaluation of LRD-stratified programs. Although the basic components of the SLG-WAM are similar to those described in this article, SIMPLIFICATION and DELAYING necessitate deep changes to them, and experimentation is underway to determine the best data structures for these operations.

Another extension to this work is to incorporate more sophisticated compilation techniques into the engine. As indicated in Appendix A.1, the tabling instructions are large for byte-code instructions, but are amenable to specialization based on mode and type. As mentioned in the introduction, the tabling engine can efficiently perform mode and type analysis (among many others), and the results of such analysis can be fed back into the XSB compiler. Tabling thus makes possible an engine which can analyze itself declaratively, and using this analysis, can improve its performance. We believe that advantages such as this, combined with the power to evaluate normal logic programs, will make tabling a common component of logic programming systems of the future.

ACKNOWLEDGEMENTS

Many people have made important contributions to the design and implementation of the SLG-WAM. The authors are deeply indebted to David S. Warren without whose ideas, guidance and encouragement stemming from his conviction to the importance of tabled execution of logic programs, the SLG-WAM would never have been implemented. Thanks also to Prasad Rao for his work on the trie-based tabling and to Juliana Freire for her work on batched scheduling. In addition, we thank Bart Demoen for his comments on an earlier draft of this article and the anonymous referees for their detailed comments which have significantly improved the presentation of this article.

This research was supported in part by NSF grants CCR-9702681, CCR-9705998, INT-9600598 and by a fellowship from the K.U. Leuven Research Council.

REFERENCES

AÏT-KACI, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts.

CHEN, W., SWIFT, T., AND WARREN, D. S. 1995. Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *J. of Logic Program. 24,* 3 (Sept.), 161–199.

CHEN, W. AND WARREN, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *J. of the ACM 43,* 1 (Jan.), 20–74.

CODISH, M., DEMOEN, B., AND SAGONAS, K. 1996. Semantics-Based Program Analysis for Logic-Based Languages using XSB. Extended version of K.U. Leuven Tech. Rep. CW 245. Available at: http://www.cs.kuleuven.ac.be/~kostis/Papers/papers.html.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts.

DAWSON, S., RAMAKRISHNAN, C. R., AND WARREN, D. S. 1996. 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 Press, Philadelphia, Pennsylvania, 117–126.

DIETRICH, S. 1987. Extension Tables for Recursive Query Evaluation. Ph.D. thesis, Department of Computer Science, SUNY at Stony Brook.

FREIRE, J., SWIFT, T., AND WARREN, D. S. 1996. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *Proceedings of the 8th International Symposium on PLILP*, H. Kuchen and S. D. Swierstra, Eds. Number 1140 in LNCS. Springer-Verlag, Aachen, Germany, 243–258.

FREIRE, J., SWIFT, T., AND WARREN, D. S. 1997. Treating I/O Seriously: Resolution Reconsidered for Disk. In *Proceedings of the 14th International Conference on Logic Programming*, L. Naish, Ed. The MIT Press, Leuven, Belgium, 198–212.

LARSON, R., WARREN, D. S., FREIRE, J., GOMEZ, O. P., AND SAGONAS, K. 1997. *Semantica.* The MIT Press, Cambridge, Massachusetts.

LARSON, R., WARREN, D. S., FREIRE, J., AND SAGONAS, K. 1996. *Syntactica.* The MIT Press, Cambridge, Massachusetts.

LLOYD, J. W. 1987. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag, Berlin.

PRZYMUSINSKI, T. C. 1989. Every Logic Program has a Natural Stratification and an Iterated Least Fixed Point Model. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM Press, Philadelphia, Pennsylvania, 11–21.

RAMAKRISHNA, Y. S., RAMAKRISHNAN, C. R., RAMAKRISHNAN, I. V., SMOLKA, S. A., SWIFT, T., AND WARREN, D. S. 1997. Efficient Model Checking Using Tabled Resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification*, O. Grumberg, Ed. Number 1254 in LNCS. Springer-Verlag, Haifa, Israel, 143–154.

RAMAKRISHNAN, I. V., RAO, P., SAGONAS, K., SWIFT, T., AND WARREN, D. S. 1995. Efficient Tabling Mechanisms for Logic Programs. In *Proceedings of the 12th International Conference* *on Logic Programming*, L. Sterling, Ed. The MIT Press, Tokyo, Japan, 687–711. Extended version to appear in J. of Logic Program.

RAMAKRISHNAN, R., SRIVASTAVA, D., AND SUDARSHAN, S. 1992a. Controlling the Search in Bottom-Up Evaluation. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, K. Apt, Ed. The MIT Press, Washington D.C., 273–287.

RAMAKRISHNAN, R., SRIVASTAVA, D., AND SUDARSHAN, S. 1992b. CORAL — Control, Relations, and Logic. In *Proceedings of the 18th Conference on Very Large Data Bases*. Morgan-Kaufmann, Vancouver, Canada, 238–249.

RAMESH, R. AND CHEN, W. 1997. Implementation of Tabled Evaluation with Delaying in Prolog. *IEEE Transactions on Knowledge and Data Engineering 9,* 4 (July/Aug.), 559–574.

RAO, P., RAMAKRISHNAN, C. R., AND RAMAKRISHNAN, I. V. 1996. A thread in time saves tabling time. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, M. Maher, Ed. The MIT Press, Bonn, Germany, 112–126.

ROSS, K. A. 1994. Modular Stratification and Magic Sets for Datalog programs with Negation. *J. of the ACM 41,* 6 (Nov.), 1216–1266.

SAGONAS, K. 1996. The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs. Ph.D. thesis, Department of Computer Science, SUNY at Stony Brook.

SAGONAS, K., SWIFT, T., AND WARREN, D. S. 1994. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM Press, Minneapolis, Minnesota, 442–453.

SAGONAS, K., SWIFT, T., AND WARREN, D. S. 1996. The Limits of Fixed-Order Computation. In *Proceedings of the International Workshop on Logic in Databases*, D. Pedreschi and C. Zaniolo, Eds. Number 1154 in LNCS. Springer-Verlag, San Miniato, Italy, 343–363.

SWIFT, T. 1994. Efficient Evaluation of Normal Logic Programs. Ph.D. thesis, Department of Computer Science, SUNY at Stony Brook.

SWIFT, T. AND WARREN, D. S. 1994. Analysis of SLG-WAM Evaluation of Definite Programs. In *Proceedings of the 1994 International Symposium on Logic Programming*, M. Bruynooghe, Ed. The MIT Press, Ithaca, New York, 219–235.

TAMAKI, H. AND SATO, T. 1986. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming*, E. Shapiro, Ed. Number 225 in LNCS. Springer-Verlag, London, 84–98.

TAYLOR, A. 1991. High Performance Prolog Implementation. Ph.D. thesis, Basser Department of Computer Science, University of Sidney, Australia.

TICK, E. 1988. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers.

VAN GELDER, A. 1989. Negation as Failure using Tight Derivations for General Logic Programs. *J. of Logic Program. 6,* 1 & 2 (Jan./Mar.), 109–134.

VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1991. The Well-Founded Semantics for General Logic Programs. *J. of the ACM 38,* 3 (July), 620–650.

VAN ROY, P. 1990. Can Logic Programming Execute as Fast as Imperative Programming? Ph.D. thesis, Computer Science Division, University of California at Berkeley.

VAN ROY, P. 1994. 1983–1993: The Wonder Years of Sequential Prolog Implementation. *J. of Logic Program. 19/20*, 385–441.

VARDI, M. 1982. The Complexity of Relational Query Languages. In *Proceedings of the 14th Annual ACM Symposium on the Theory of Computing*. ACM, San Fransisco, California, 137–146.

VIELLE, L. 1989. Recursive Query Processing: The Power of Logic. *Theor. Comput. Sci. 69,* 1 (Dec.), 1–53.

WARREN, D. H. D. 1983. An Abstract Prolog Instruction Set. Tech. Rep. 309, SRI International, Menlo Park, U.S.A. Oct.

WARREN, D. H. D. 1987. The SRI Model for OR-Parallel Execution of Prolog — Abstract Design and Implementation Issues. In *Proceedings of the 1987 Symposium on Logic Programming*. IEEE Computer Science Press, San Fransisco, California, 92–102.

L_1 :	tabletry	2	L_3	TR	%	
L_2 :	tabletrust	2	L_{10}		%	
L_3 :	getpvar	v_1	r_2		% path(X,Y) :-	
L_4 :	putpvar	v_2	r_2		%	path(X,Z
L_5 :	call	3	path/2		%	),
L_6 :	putpval	v_2	r_1		%	edge(Z,
L_7 :	putpval	v_1	r_2		%	Y
L_8 :	call	3	edge/2		%	)
L_9 :	new_answer	2	v_3		%	.
L_{10}:	call	2	edge/2		% path(X,Y) :-	edge(X,Y)
L_{11}:	new_answer	2	v_3		%	.

Fig. 33. SLG-WAM Code for Left Recursive path/2.

WARREN, D. S. 1984. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the 1984 Symposium on Logic Programming.* IEEE Computer Science Press, Atlantic City, New Jersey, 198–202.

APPENDIX

A.1 Analysis of Left-Recursion

In order to understand better why the execution overhead and stack space usage of SLG transitive closure is so low we analyze the behavior of the left-recursive path/2 predicate (Figure 32(a)) on a chain of 1024 elements. The byte-code for path/2 is shown in Figure 33.

Given a query path(1,Y),fail, the predicate path/2 is entered through instruction labeled L_1 in Figure 33. Conceptually, the tabletry instruction begins by checking whether a variant of path(1,Y) exists in the table, and copying the subgoal into the table if not. Assuming the evaluation starts from a system with an empty table, the subgoal path(1,Y) is new to the evaluation, so a generator choice point and a completion stack frame are created for the subgoal, and the tabletry instruction will branch to the instruction labeled as L_3 after it executes. In instruction L_5 the subgoal path(1,Y) is called again: a tabletry instruction is executed a second time, but the subgoal path(1,Y) is now located in the table. Stacks are frozen and a consumer choice point is created, whose substitution factor which will serve as the template for bindings in the fixpoint computation. This second execution of the tabletry instruction also sets up pointers to backtrack through any existing answers in the table. There currently are none, so the evaluation suspends by failing. At this point the necessary structures to evaluate the fixpoint have been constructed.

The suspension and failure described in the previous paragraph causes backtracking to the generator choice point of path(1,Y) and subsequent execution of the tabletrust instruction. This instruction places a completion instruction in the failure continuation cell of the generator choice point and then branches to the instruction labeled L_{10}. The second clause calls edge(1,Y), whose instructions do not differ from those of the WAM. edge(1,2) succeeds, causing the new_answer instruction to be invoked. Recall from Section 3.4.3 that the second operand of new_answer is the *GCP_pointer*, through which the subgoal frame of p(1,Y) can be accessed. The new_answer instruction checks for the existence of the binding

Instructions of path/2	Other instructions	Dynamic execution count	Percent	SPARC instructions	Percent
answer_return		1023	10%	259	34%
putpval		2046	20%	14	5%
call		1024	10%	34	5.5%
	switchonbound	1024	10%	65	9.4%
	getnumcon	1024	10%	15 (bb)	3.1%
	getnumcon	1024	10%	43 (bf)	6.7%
	proceed	1024	10%	3	1.7%
new_answer		1023	10%	253	32.9%
	fail	1024	10%	4	1.8%

Table VIII. Instruction Counts for Left-Recursive Transitive Closure in SLG

$\{Y \leftarrow 2\}$ as an answer for the subgoal, `path(1,Y)`. Since the binding does not exist in the table, the answer is inserted (as with subgoals, this check/insert operation is done in a single pass). At this point the fixpoint computation contains its seed.

By returning the answer to the generator node in the query, the evaluation hits the `fail` predicate. Because the `edge/2` predicate is a chain, backtracking is not possible for the goal `edge(1,Y)`, and the engine fails to the completion instruction in the generator choice point. This instruction performs the fixpoint_check operation which determines that a consumer subgoal of `p(1,Y)` has not consumed all answers. The **B** register is set to the consumer choice point for `p(1,Y)` (whose *Breg_Chain* cell is set to point to the generator choice point — see Figure 18) and through schedule_resumes the engine fails. Failure invokes the answer_return instruction which returns the answer `p(1,2)` to the consumer and proceeds. The subgoal `edge(2,Y)` is then called, succeeds, the answer `p(1,3)` is derived, and the binding $\{Y \leftarrow 3\}$ is added to the answer table of `p(1,Y)` and returned to the generator. Once again the predicate `fail` is encountered, and again the answer_return instruction is executed, this time returning `p(1,3)`. The engine stays in this loop throughout the transitive closure, executing answer_return instructions, calling `edge/2`, adding the new answers through new_answer, and failing. When the transitive closure is exhausted, the engine finally fails out of the consumer choice point and into the completion instruction for `path(1,Y)`. The completion instruction determines that `path(1,Y)` is the leader of its SCC, and that all its answers have been returned to all its consumers. `path(1,Y)` can therefore be completed, and the evaluation ends. Table VIII contains a dynamic count of SLG-WAM instructions for the fixpoint loop of the query `path(1,Y),fail` over a chain of 1024 elements, along with an estimate for the number of SPARC instructions needed for each SLG-WAM instruction.[14]

Several points can be made about this evaluation. First, the substitution factoring of Section 3.2 allows execution of the fixpoint of `path/2` to be equivalent to execution of the fixpoint for

```
path(Y) :- path(X), edge(X,Y).
```

[14]The SPARC instruction count factors out operations which are not usually done in each instruction, such as memory management, and hash table reconfiguration for answers, although the counts do include overheads for determining whether these operations are needed. Assembly code for the count was produced using the `-O2` option when compiling the ***SLG-WAM: Definite*** emulator.

A second point is that the same local environment is reused throughout the fixpoint, and so is the consumer choice point, so that transitive closure is a tight, failure driven loop. Optimizations could, however, be made to specialize the answer_return instruction, which must copy bindings out of the table and the new_answer instruction, which must copy bindings into the table. These instructions necessarily have an interpretive flavor, and could be made more efficient by using information about modes or types.

Memory usage of the query can be accounted for as follows. The original query `?- p(1,Y)` requires a local environment with two permanent variables. Each local environment for a tabled subgoal requires a three word overhead (a pointer to the parent of the environment, a pointer to the **CP** register, and a pointer to the generator choice point) for a total of 20 bytes. A three-word trail frame is needed, a 24 byte completion stack frame, along with a generator choice point of 72 bytes including argument cells and substitution factor, and a consumer choice point of 64 bytes, for a total of 192 bytes. 92 bytes are needed for the subgoal trie (including a 32 byte subgoal frame), while 28 bytes are needed per answer: 20 to store the binding $\{Y \leftarrow n\}$, and 8 for the answer list cell for each node.

Received December 1996; revised September 1997; accepted January 1998