XSB: A System for Efficiently Computing Well-Founded Semantics

Prasad Rao,[1] Konstantinos Sagonas,[2] Terrance Swift,[1]
David S. Warren[1] and Juliana Freire[1]

[1] Department of Computer Science, SUNY at Stony Brook,
Stony Brook, NY 11794-4400, U.S.A.
[2] Department of Computer Science, K.U. Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Abstract. The well-founded model provides a natural and robust semantics for logic programs with negative literals in rule bodies. We implemented the well-founded semantics in the SLG-WAM of XSB [19]. Performance results indicate that the overhead of delay and simplification to Prolog — or tabled — evaluations is minimal. To compute the well-founded semantics, the SLG-WAM adds to an efficient tabling engine for definite programs three operations — negative loop detection, delay and simplification — which serve to detect, to break and to resolve cycles through negation that might arise in evaluating normal programs. XSB is a full Prolog system that closely approximates the ISO standard; additionally, it supports a tight integration of tabled predicates with nontabled predicates.

1 Introduction

XSB is a research-oriented logic programming system for Unix- and Windows-based systems. In addition to providing all the functionality of the Prolog language, XSB contains several features not usually found in logic programming systems, including

- Evaluation according to the Well-Founded Semantics [24] through SLG resolution [3];
- A compiled HiLog implementation;
- A variety of indexing techniques, along with a novel transformation technique called *unification factoring* [5] that can improve program speed through deep indexing of terms;
- Source code availability for portability and extensibility.

Though XSB can be used as a Prolog system, we avoid referring to XSB as such, because of the availability of SLG resolution and the handling of HiLog terms. These facilities, while seemingly simple, significantly extend its capabilities beyond those of a typical Prolog system. In fact, we feel that these capabilities justify viewing XSB as a new paradigm for logic programming.

To understand the implications of SLG resolution [3], recall that Prolog is based on a depth-first search strategy (SLD) which uses program clause resolution. As such, Prolog is susceptible to getting lost in an infinite branch of a search tree, where it may loop infinitely. SLG evaluation, available in XSB, can correctly evaluate many such logic programs. To take the simplest of examples, any query to the program (together with a database of facts for `parent/2`):

```
:- table ancestor/2.
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
```

will terminate in XSB, since `ancestor/2` is compiled as a tabled predicate; Prolog systems, however, would go into an infinite loop. The user can declare that SLG resolution is to be used for a predicate by using `table` declarations, as here. Alternately, an `auto_table` compiler directive can be used to direct the system to invoke a simple static analysis to decide what predicates to table. This power to process recursive queries has proven very useful in a number of areas, including deductive databases, natural language processing [11, 12], meta-interpreters, and program analysis [6, 4]. For efficiency, we have implemented SLG at the abstract machine level so that tabled predicates are executed with the speed of compiled Prolog. We finally note that for definite programs SLG resolution is similar to other tabling evaluation methods such as extension tables, OLDT, or SLD-AL resolution [7, 23, 25].

The use of tabling also makes possible the evaluation of programs with negation. When logic programming rules have negation, paradoxes become possible. As an example consider one of Russell's paradoxes — the barber in a town shaves every person who does not shave himself — written as a logic program.

```
shaves(barber,P) ← person(P), ¬shaves(P,P)
person(barber) ←
person(mayor) ←
```

Logically, it should be possible to deduce that the barber shaves the mayor, while in a *skeptical* semantics it is undefined whether the barber shaves himself or not. The well-founded semantics does not treat this conclusion as either true or false, but as *undefined*. Prolog, of course, would be trapped in an infinite loop in this program. XSB's treatment of negation is discussed extensively in Section 2.

The second important extension in XSB is efficient support of HiLog programming. However due to space limitations HiLog is not discussed here, but is discussed in detail in [2, 21].

XSB is available on a wide variety of platforms: DOS, Microsoft Windows, Windows 95, Windows NT, PowerMacs running MkLinux, Sun SPARCstations (running SunOS 4.x.x or Solaris 2.x), NeXT machines, Intel-based machines running Linux, 386BSD, or Solaris, Silicon Graphics machines running IRIX, Hewlett-Packard machines running HP-UX, DECstations running Ultrix, Amiga machines running AmigaDOS, IBM RS/6000, and Sony NEWS workstations.

Finally, we note that XSB is under continuous development, and this document reflects the current status (Version 1.7 as of March 97) of our system.

While we have taken great effort to create a robust and efficient system, we would like to emphasize that XSB is also a research system and is to some degree experimental. Some of the research features of XSB such as HiLog, indexing techniques and efficient tabled access mechanisms, or the SLG-WAM implementation are not discussed in detail in this document. Documents related to them [2, 21, 5, 16] can be found via the world-wide web or anonymous ftp from `{www/ftp}.cs.sunysb.edu`, the same host from which XSB can be obtained.

The rest of this paper is organized as follows: Section 2 describes SLG evaluation and the computation of the well-founded semantics in XSB. Some other noteworthy features of XSB are mentioned in Section 3. Section 4 briefly comments on the performance of evaluating WFS in XSB. We end with a discussion.

2 SLG Evaluation

XSB differs from Prolog systems in its ability to use SLG resolution, which uses a type of *tabling* in contrast to SLD resolution, which does not. The difference in evaluation strategy means that, with appropriate declarations, XSB is complete under the well-founded semantics [24] for non-floundering normal programs, and terminates when these programs have the bounded term-size property, as will be explained below. Tabling allows a different, more declarative programming style than Prolog that can be of use for a number of problems. The following sections provide a brief introduction to SLG resolution as it is implemented in XSB.

2.1 Informal Description of Tabling for Definite Programs

Consider again the `ancestor/2` predicate of the introduction, together with the query `?- ancestor(david,Y)`. This program has a simple, declarative meaning: `Y` is an ancestor of `X` if `Y` is a parent of `X` or if `Y` is a parent of an ancestor of `X`. Prolog, however, cannot compute the answer to this query, but enters an infinite loop. The inability of Prolog to answer such queries, which comes up frequently, comprises one of its major limitations as an implementation of logic.

A number of approaches have been developed which solve programs like ancestor by reusing partial answers to the query `ancestor(1,Y)` [7, 23, 1, 25, 26]. Briefly, the ideas behind these algorithms can be described in the following manner. First, the program keeps track of nodes where calls to `ancestor(1,Y)` have been made. By maintaining this information, the repeated call to `ancestor(1,Y)` in the body of the second clause can be detected. The next idea is to store partial solutions to the call in a *table*, which consists of calls associated with their answers. These answers are substituted into the call to `ancestor(1,Y)` in the body of the second clause. Once they are joined to the second literal of the second clause these substitutions will in general produce new solutions to the query, and these new solutions in their turn are fed into the body of the second clause until a fixpoint (if any) is reached. In SLG terminology, the first call of a goal is called a *generator* node, and is expanded using program clauses as in SLD resolution. Correct answer substitutions to the solution node are interned

in the *table entry* (call & returns) for a call (for details see [16]). Subsequent calls which are identical under variable renaming are referred to as *active* nodes and are expanded using answers derived for that call instead of repeatedly using the program clauses. The computation ends when all possible program clauses have been used to expand all *generator* nodes, and when all answers have been returned to all *active* nodes.

Because *active* nodes are expanded using answers rather than program clauses, tabling will terminate whenever the set of answers to all queries in a derivation is finite – a situation which occurs when the program has a finite model, for instance. Indeed, it can be proven that for any program with the *bounded term-size property* where all terms have a maximum size, SLG computation will terminate. These programs include the important class of *Datalog* programs.

It is important to notice two differences between tabling and SLD resolution beyond termination properties. First, termination requires that each distinct up to variable renaming solution to a call be returned only once. Second, because of the way that solutions are stored in tables and returned to *active* nodes in XSB, answers may be returned to the user in an unaccustomed order.

2.2 Left-to-Right Dynamically Stratified Negation

Most frameworks for tabled resolution considered only definite programs, though some (e.g. [10, 22]) extended tabled evaluation to programs with restricted forms of *stratified* negation. Broadly, a program uses stratified negation whenever there is no recursion through negation. Refining this intuition based on predicates, (ground) atoms or modules can lead to a variety of stratification classes. While XSB evaluates all programs, whether stratified or not, its evaluation is especially efficient for LRD-stratified programs [20], which we now explain.

Consider a simple approach to incorporate negation into tabling, and assume, to avoid complications of floundering, that the entire program is ground; i.e., that no clause of the program contains variables. Then, if all predicates for negative literals are tabled, we could evaluate a query to a program like

```
a ← ¬b
b ← c, d
c ← b
c ←
```

in a simple manner. Each time a negative goal is called, a separate table is opened for the negative call. This evaluation is carried on to termination. If it terminates, as it will for the classes of programs in the previous subsection, its answers if any, are used to determine the success of failure of the calling goal.

The method just sketched is correct for certain classes of stratified programs, but is not terribly efficient. Every time a new negative goal is called, a new table must be started, and run to completion. We would like to use information already derived from the computation to answer a new query, if at all possible.

XSB addresses this problem by keeping track of the *state* of each tabled call. Calls can have the state *incomplete*, or *complete*. A call is *complete* only when

its table contains all solutions that are implied by the program, otherwise it is *incomplete*.[3] XSB efficiently detects when calls may be marked *complete*.

Using these concepts, we can now explain left-to-right dynamically stratified negation (LRD-stratified negation). Assume we have a tabled evaluation like that described in the previous subsection: given a subgoal S of predicate p the evaluation uses information from all clauses of p and those which S calls to solve S. However, literals within each clause are searched using a fixed left-to-right computation rule as in Prolog. When a derivation encounters a negative literal, say $\neg A$, the behavior of the evaluation varies depending on the state of A. If A is completed then depending on its truth value, the path either fails or $\neg A$ is removed from the list of subgoals to be resolved. Otherwise the path is *suspended*. If A has not yet been called, the evaluation calls it; otherwise, if A is incomplete the evaluation proceeds to other clauses that may help determine the truth of A. In either case, if an answer is later found for A, the suspended derivation path is failed, while if A is determined to be completely evaluated with no answers, $\neg A$ is removed from the derivation path which then resumes its computation.

Informally, if a program can be evaluated using the method sketched above it is called LRD-stratified(formal definitions are provided in [20]). The adjective "left-to-right" comes from the computation rule. The adjective "dynamic" arises from the fact that run-time information about the truth value of atoms is central to whether the derivation path remains suspended or not. The following simple program is LRD-stratified, but does not fit into other stratification classes (e.g. the more familiar class of *(left-to-right) modularly stratified programs* [17]).

```
p ← q, ¬r, ¬s
q ← r, ¬p
r ← p, ¬q
s ← ¬p, ¬q, ¬r
```

Before leaving the subject of stratification, we note that the use of completion also forms the basis of evaluation of programs with stratified uses of aggregation.

2.3 Dynamically Stratified Negation

A simple rearrangement of the program of the previous section causes it not to be LRD-stratified. Consider the program

```
p ← ¬s, ¬r, q
q ← r, ¬p
r ← p, ¬q
s ← ¬p, ¬q, ¬r
```

which cannot be evaluated by the method sketched above. However, the well-founded model of this program is *total*. Such programs, are called *dynamically stratified*, but may not be stratified for any particular evaluation order [15, 17].

[3] This use of completion allows ground queries to Datalog programs with negation to be computed with polynomial data complexity.

XSB handles dynamically stratified programs through delaying of negative literals when it becomes necessary to look to their right in a clause, and then simplifying away the delayed literals when (and if) their truth value becomes known. However, to ensure efficiency, literals are delayed *only if* the engine determines them not to be stratified under the LRD-stratified evaluation method.

2.4 Non-Stratified Negation

Russell's paradox, presented in Section 1, illustrates that when negation appears in a logic program, recourse to a *three-valued logic* may be necessary to describe its semantics. That is, the value of a ground atom may be *undefined* in addition to being true or false. From the perspective of stratification, these programs are not even dynamically stratified. XSB, in fact, handles such programs, evaluating them according to the well-founded semantics [24]. This semantics is a natural extension from a programmer's perspective. Furthermore, more elaborate forms of non-monotonic reasoning, such as stable models [9], can be computed starting from the well-founded model as a basis. Indeed, XSB provides a residual program which, as explained below, can be used to compute stable models.

The intuition behind the well-founded semantics can be explained through the operations of tabling. First, consider how tabling acts on a ground definite program. Any ground atom is true if there is a successful derivation of that atom, and false if every derivation path ends in one of two cases. In the first case, the path ends in a node with a selected *generator* literal and there is no program clause that can resolve the literal away. In the second, the path ends in a node with a selected *active* literal and there is no answer clause which can resolve the literal away. If all paths are failed by the first case, the goal is said to be finitely failed, and the goal would be failed by Prolog as well. If any paths were failed through the second case, it is possible that there would be infinite loops in the Prolog derivation, and that Prolog would not fail the literal properly.

In a definite program, the action of failing infinite positive recursion makes intuitive sense. In a normal program, though, recursion can also occur through negation, and simply failing such recursion is inappropriate. For instance, if recursion through negation were failed in a clause like $\mathtt{p} \leftarrow \neg\mathtt{p}$. the atom p would fail. But then p would be in any model of the program making p both true and false. This makes the program inconsistent (among other problems) so that reasoning in any program with rules as the above becomes difficult at best.

The well-founded semantics can be seen as an extension of dynamic stratification to handle "unavoidable" recursion through negation. First, if an atom, A, has a successful derivation, A is true in the well-founded model. (With stratified negation, the notion of a successful derivation may involve the full evaluation of negative literals along with their completion, but remains intuitively clear). If every clause defining A fails (perhaps by dynamically breaking the left-to-right computation rule and checking all literals in the clause) A is false. If neither of these cases occur A is undefined. Thus, A is undefined if there is no successful derivation of it and it is not failed either through finite failure or infinite positive recursion. That is, A is undefined if after removing all successful literals

in clauses that define A, and after failing all clauses for A with failed literals, recursion through negation persists.

2.5 Beyond Well-Founded Models

As a final point, Version 1.7 of XSB also provides a basis for computing (partial) stable models of programs through what is called the *residual program* (of a program P and a set of queries Q). Consider the simple, non-stratified program:

```
p ← ¬q, s
q ← r
q ← ¬p
r ← ¬s
s ←
```

There are two stable models of this program: one in which p and s are true, and another in which q and s are true. In the well-founded model, s is true, r is false, and q and p are undefined. As mentioned above, SLG delays literals whose truth value cannot be determined and later simplifies the delayed literals away when and if their truth value becomes known. Those delayed literals which are not simplified away, therefore, may either be true or false in a given stable model of a program, and the computation of the well-founded model performed by XSB can be seen as a pre-processor for stable models. Given a set of queries Q, the pre-processed program consists of all answers of atoms in Q whether they have a delay list (are *conditional*) or not (are *unconditional*). This program, consisting of true and undefined atoms under the well-founded model, is called the *residual program* (for P and Q). In such a program delayed literals show dependencies between undefined atoms and are treated as body literals of a clause. The residual program for the above program and $Q = \{\leftarrow \texttt{p}\}$ would be:

```
p ← ¬q
q ← ¬p
s ←
```

Computing stable models of even propositional programs is an intractable problem [14]. The residual program (for queries to all predicates) may have evaluated away clauses and body literals from the original program. Thus using the residual program may bring a considerable performance gain since the search space of algorithms to compute stable models will be correspondingly reduced.[4]

3 Other Noteworthy Features of XSB

Indexing Traditional Prolog systems are unsuitable for data-oriented applications such as data mining not only because of the limitations of SLD resolution

[4] Because computation in XSB is goal-oriented, the models computed from a residual program for P and Q are the *partial stable models* of P which are total on all atoms on which atoms in Q depend. For computing total stable models, Q has to be appropriately chosen.

but because accessing (large amounts of) data efficiently may require indices other than Prolog's first argument indexing. XSB offers a variety of indexing techniques for asserted code. Clauses can be indexed on groups of arguments or on alternative arguments. A variant of `assert/1` that utilizes the full prefix indexing provided by *tries* is also available. For compiled code, XSB offers *unification factoring* [5], which not only offers complete indexing through nondeterministic indexing automata, but can also *factor* elementary unification operations. For a detailed description of XSB's indexing schemes see the manual.

Using different scheduling strategies Given the asynchronicity of answer generation and consumption, tabling systems face an important scheduling choice not present in traditional top-down evaluation: How does the order of returning answers to active subgoals affect performance? The scheduling of answer returns is an important issue for the efficiency of tabled evaluation. Indeed, the scheduling strategy of XSB was thoroughly revised in version 1.5 to use a strategy that is more efficient in terms of time and space (cf. [8]). Finally, we note that Version 1.7 allows experimentation with more than one scheduling strategies.

XSB as a Prolog system XSB can also be used simply as a Prolog system. As such, it offers several interesting features: (i) Fast compilation of dynamic code. (ii) Object file compilation. (iii) Module system and inference of symbol usage at compilation. (iv) Dynamic loading of Prolog and C code.

A goal of XSB is to provide an efficient engine for both logic programming and in-memory deductive database queries [18]. One prerequisite for this functionality is the ability to load large amounts of data very quickly. We have taken care to code in C a compiler for asserted clauses. The result is that the speed of asserting and retracting code is faster in XSB than in any other Prolog system that we know of. At the same time, because asserted code is compiled into SLG-WAM code, the speed of executing asserted code in XSB is usually faster than that of other Prologs as well. We note however, that XSB does not follow the semantics of dynamic code specified in [13].

XSB does not compile programs directly into memory but rather generates SLG-WAM object code (called *byte code*) into files. The object files can be executed directly by the emulator or, as is more usual, loaded by the interpreter (which itself is a byte code file executed by the emulator).

The module system (users can choose not to use it) of XSB is *atom based* (where all non-predicate symbols are by default assumed to be global to all modules), and *flat* (modules cannot be nested). The dynamic loader lazily loads modules containing imported predicates when these are first called, which can be useful for reducing program space and start-up time.

4 Performance

The SLG-WAM overhead for SLD resolution is minimal. Thus, when XSB is used simply as a Prolog system, it is reasonably competitive with other WAM-

based emulated Prolog systems. For example, XSB is about two to three times slower than Quintus or emulated SICStus.

The bottom-up aspect of SLG aids in filtering out redundant subcomputations and can considerably improve performance. The same generation program furnishes a case of the usefulness of tabled-based evaluation for optimizing Prolog programs.

Example 1. The query `?- sg(1,X),fail.` was executed against the program

```
:- table sg/2.
sg(X,Y) :- cyl(X,X1), sg(X1,Y1), cyl(Y,Y1).
sg(X,X).
```

using a 24x24x2 randomly generated cylinder, as the base relation `cyl/2`. A cylinder can be thought of as a rectangular matrix of elements where each element in row N has links to a certain number of elements in row $N+1$. The 24x24x2 cylinder then, is an array of 24x24 nodes, where each of the nodes in each row (except the last) is connected to two elements in the next higher row. Executing this query under SLG is more than three orders of magnitude faster than under SLD, since SLD will search a complete binary tree of depth 24.

While the judicious use of tabling can make some programs faster, its indiscriminate use can make other programs slower. For instance, tabling `append/3` makes a linear algorithm into a quadratic one. A detailed analysis of the performance of XSB can be found in [18].

4.1 Performance of Well-Founded Negation

To our knowledge, the `win/1` program is still the only standard scalable benchmark in the well-founded semantics literature. Its SLG and SLDNF versions, using XSB, are shown below. SLDNF resolution is sufficient for acyclic `move/2` graphs (like chains and trees). In this cases the well-founded model is total.

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

Figure 1 contains timing results (in seconds) for executing on a SPARCstation 20 running Solaris 2.5.1 the query `?- win(1),fail.` over different `move/2` data structures of sizes varying from 500 to 16K. The tabled evaluation of this query creates many distinct subgoals and suspended nodes. For chains and trees, the program is modularly stratified and each subgoal either has one answer, or fails. Figure 1 shows that for the case of the chain, the performance of tabled negation is comparable to that of the negation of Prolog (less than 2 times slower). This difference is due to the recording of subgoals and answers in tables, of suspending computations and of performing completion. In the case of the complete binary tree, however, tabled negation is much slower. This is because tabled negation has to fully evaluate negative literals before completing them, and consequently, it traverses the entire search space which is exponential in the height of the tree (see [18]).

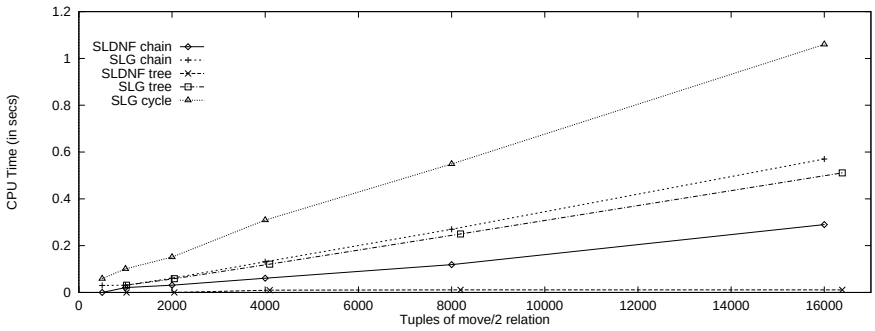

Fig. 1. Performance of the `win/1` program over different data structures.

5 Conclusion

The goal of the XSB system is to provide a framework and environment that gracefully integrates Logic Programming, in-memory Deductive Database query processing, and Nonmonotonic Reasoning with respect to the well-founded semantics. The fact that it produces a residual program for non-stratified normal logic programs means that it can be used as a preprocessor for many nonmonotonic formalisms other than just the well-founded semantics. XSB provides extensive builtin support for accessing generated tables (including the residual program) and general programmability, so that XSB can be used as an environment to build more complex NMR systems. XSB is quite robust as a logic programming and deductive database query processing system and has been used in this way for various applications. It has not as yet been used so extensively as a nonmonotonic reasoning system, so our confidence in its robustness for these applications is not as great. One goal is to find users who wish to use it for such applications so that we can work with them to make XSB a powerful and general tool for nonmonotonic reasoning.

References

1. F. Banchilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*. ACM, 1986.
2. W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *J. Logic Programming*, 15(3):187–230, 1993.
3. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
4. M. Codish, B. Demoen, and K. Sagonas. XSB as the natural habitat for general purpose program analysis. Technical report, KU Leuven, 1996.
5. S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *Proc. of the 22nd POPL*, pages 247–258. ACM, 1995.

6. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *ACM PLDI*, pages 117–126, 1996.
7. S. Dietrich. *Extension Tables for Recursive Query Evaluation*. PhD thesis, SUNY at Stony Brook, 1987.
8. J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *8th International PLILP Symposium*, number 1140 in LNCS, pages 243–258. Springer-Verlag, 1996.
9. M. Gelfond and V. Lifshitz. The stable model semantics for logic programming. In *Joint International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.
10. D. Kemp and R. Topor. Completeness of a top-down query evaluation procedure for stratified databases. In *Logic Programming: Proc. of the Fifth International Conference and Symposium*, pages 178–194, 1988.
11. R. Larson, D. S. Warren, J. Freire, and K. Sagonas. *Syntactica*. MIT Press, 1995.
12. R. Larson, D. S. Warren, J. Freire, K. Sagonas, and P. Gomez. *Semantica*. MIT Press, 1997. To Appear.
13. T. Lindholm and R. O'Keefe. Efficient implementation of a defensible semantics for dynamic PROLOG code. In *Proceedings of the 4th ICLP*, pages 21–39, 1987.
14. W. Marek and M. Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, July 1991.
15. T. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS*, pages 11–21, 1989.
16. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient table access mechanisms for logic programs. In L. Sterling, editor, *International Conference on Logic Programming*, pages 697–711, 1995.
17. K. Ross. *The Semantics of Deductive Databases*. PhD thesis, Department of Computer Science, Stanford University, 1991.
18. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conference*. ACM, 1994.
19. K. Sagonas, T. Swift, and D. S. Warren. An abstract machine for computing the well-founded semantics. In *Joint International Conference and Symposium on Logic Programming.*, pages 274–289, 1996.
20. K. Sagonas, T. Swift, and D. S. Warren. The Limits of Fixed-Order Computation. In *Proceedings of the First International Workshop on Logic in Databases*, number 1154 in LNCS, pages 343–363, Italy, 1996. Springer-Verlag.
21. K. Sagonas and D. S. Warren. Efficient execution of HiLog in WAM-based Prolog implementations. In *Proceedings of the 12th International Conference on Logic Programming*, pages 349–363, Japan, 1995.
22. H. Seki. On the power of Alexandrer templates. In *Proc. of 8th PODS*, pages 150–159. ACM, 1989.
23. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, 1986.
24. A. van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *JACM*, 38(3):620–650, 1991.
25. L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.
26. A. Walker. Backchain iteration: Towards a practical inference method that is simple enough to be proved terminating, sound, and complete. *J. Automated Reasoning*, 11(1):1–23, 1993. Originally formulated in New York University, 1981.