Tabling for Non-monotonic Programming

Terrance Swift

Department of Computer Science SUNY at Stony Brook, Stony Brook, NY 11794-4400, U.S.A.
E-mail: tswift@cs.sunysb.edu

Non-monotonic extensions add power to logic programs. However, the main logic programming language, Prolog, is widely recognized as inadequate to implement these extensions due to its weak termination and complexity properties. By extending Prolog's SLD resolution with tabling, Prolog can be improved in several ways. Tabling can allow a logic programming system to compute the well-founded semantics for programs with bounded term depth, and to do so with polynomial data complexity. By exploiting these properties, tabling allows a variety of non-monotonic extensions to be efficiently implemented, and used to solve practical problems. In this paper we describe tabling as it is implemented in the XSB system and show how it can be used to construct meta-interpreters (or preprocessors) for two sample formalisms: the Well Founded Semantics with Explicit Negation, and Generalized Annotated Logic Programs. We also describe how non-monotonic extensions are used in practical applications such as psychiatric diagnosis, extraction of information from poorly structured textual data, and model checking.

Keywords: AI languages, AI logics, Knowledge Representation

AMS Subject classification: Primary 68T25; Secondary 68T27, 68T30

1. Introduction

Over the last ten years, the fields of logic programming (LP) and non-monotonic reasoning (NMR) have become closely intertwined. Both rely on logical formalisms — NMR to model aspects of common-sense reasoning such as the consistency of views of an agent; LP to model the declarative and operational semantics of computer programs. The similarities in formalism have spurred cross-fertilization. The LP community has studied how to include non-monotonic extensions in logic programs in a computationally tractable way; the NMR community has studied how aspects of common-sense reasoning can be viewed as programs: for instance by viewing auto-epistemic expansions as stable models.

Tabled logic programming [55,58,8,10] has become recognized as an important computational basis for including various non-monotonic extensions in logic programming. Attractive properties of tabling are that it allows direct implementation of constructs such as well-founded negation [57]. Indeed, tabled evaluation of Datalog programs with well-founded negation terminates with polynomial data complexity [10]. Furthermore, tabled evaluation can be closely integrated with Prolog-style control and meta-predicates, and can also access disk-resident data in the same manner as deductive databases [25].

These properties of tabling provide a basis for *programming* with non-monotonic constructs. Consider implementation of a class of programs that is embeddable into the well-founded semantics, for instance the well-founded semantics with explicit negation (WFSX) [2]. Using the embedding, a tabled meta-interpreter can be written for the program, and partial deduction applied to the meta-interpreter to produce an efficient implementation. Thus the representational power of non-monotonic extensions like explicit negation becomes available to programmers solving practical problems.

It has been the purpose of the XSB system [49] to explore the addition of tabling to a Prolog system. At a language level, XSB maintains full support for Prolog while including tabling declarations, operators, and predicates. At a level of operational semantics XSB is based on SLG resolution [10] which formulates tabling for the well-founded semantics. At an implementational level XSB has extended the most common Prolog engine, the WAM, to produce an engine , the SLG-WAM [46,47,26,44] in which tabling can be mixed with Prolog. At a practical level the mixture of tabling and Prolog has been used in numerous applications: in grammar processing [34,15]; in program analysis [18,12,9]; in model checking [43,14]; in machine learning [33]; in diagnosis [22]; and as an implementation platform for logical updates [36]. While XSB was originally developed at SUNY Stony Brook, current versions are developed in close collaboration between developers at SUNY Stony Brook, Universidade Nova de Lisboa, and Katholieke Universiteit Leuven.

For the purposes of this paper, non-monotonic programming is considered to be the use of constructs that are not typically available in logic programming or deductive database systems, and to include well-founded (default) negation as well as a construct called tabled aggregation. This view is perhaps broader than is sometimes taken, but is justified by the variety of formalisms that can be implemented using these constructs, as discussed in Section 3. Within this viewpoint, this paper concentrates on aspects of non-monotonic programming that can be performed using the well-founded semantics and its extensions as implemented in version 1.8 of XSB, and thus covers only a fragment of non-monotonic programming. Section 2 provides a background for non-monotonic programming using tabling in XSB by reviewing the SLG resolution upon which XSB is based, along with aspects of its implementation of that are of relevance for practical programming. Section 3 then provides examples of how XSB can be used for non-monotonic programming by means of two meta-interpreters: one for WFSX, and another for Generalized Annotated Programs [32]. Section 4 discusses three large-scale applications that use tabled non-monotonic programming: psychiatric diagnosis, information extraction from poorly structured textual data, and verification of concurrent systems.

2. Extending Prolog with Tabling

In order to productively use non-monotonic extensions to logic programs, a full programming environment is needed to support the extensions. The XSB system extends Prolog with tabling at a low level so that the well-developed Prolog environment — including Prolog's command-line interpreter, debuggers, module system and other features — can be used. SLG, the resolution method underlying XSB, is described in Section 2.1; general issues involved in programming with SLG are overviewed in Section 2.2, and the XSB system itself described in Section 2.3.

2.1. Evaluating Normal Programs through SLG Resolution: an Overview

We present the main definitions of SLG [10] (Linear Resolution with Selection with Selection function for General logic programs) using a variant described in [53] that uses forests of trees to model the state of a tabled evaluation, and explicitly combines SLG with SLD evaluation. We assume the basic terminology of logic programming from [39] and of the well-founded semantics from [57], with the following exceptions. An interpretation, I of a normal program is a set of literals, such that for a ground atom A, $I \models A$ iff $A \in I$, and $I \models not\ A$ iff $not\ A \in I$. As does [10], we define subgoals as atoms, and treat variant atoms as identical. If I is an interpretation, and S is a set of atoms,

$$I|_S = \{A|\ (A \in I \text{ or } not\ A \in I) \text{ and } A \text{ is an instance of } S_1 \text{ for some } S_1 \in S\}$$

We present first the ideas behind SLG informally by example, and postpone formal definitions until Section 2.1.1.

Like other tabling methods, SLG evaluates programs by keeping track of subgoals and their associated answers, and by resolving repeated instances of subgoals against stored *answers* rather than against program clauses. By using answers in this manner, rather than repeatedly using program clause resolution as in SLD, SLG resolution avoids looping and terminates for all programs with the *bounded term-size property*[1] such as Datalog programs. In order to handle non-stratified programs, SLG allows evaluation of literals to be *delayed* if their truth value cannot be determined; *simplification* operations may later act to propagate truth values to delayed literals once the truth values have been determined. Finally, SLG evaluation keeps track of subgoals for which no further evaluation is necessary: such subgoals are termed *complete*.

[1] Intuitively, if a program has the bounded term-size property then there is a computable function f such that if the size of the argument of query Q is n, then the maximum size of any argument to any subgoal encountered in the evaluation of Q is less than $f(n)$. For the formal definition see e.g. [10].

If a predicate is to be evaluated using SLG operations other than program clause resolution, it is declared to be *tabled*; all other predicates are *non-tabled* and evaluated as in SLD. The form of a tabling declaration is *:- table* $p_1/a_1, ..., p_n/a_n$ where for $1 \leq i \leq n$, p_i is the name of a predicate symbol and a_i its arity. Slightly abusing terminology, we speak of tabled subgoals and literals as well as tabled predicates. In SLG, the literal selection rule is fixed within an evaluation. Without loss of generality we assume throughout this paper that a left-to-right literal selection strategy is used. Also for simplicity, if a literal *not* S is selected for resolution in a node of an SLG tree, we speak of S as the *selected subgoal* of the node.

Example 1. Consider the evaluation of a query *?-* t with respect to the tabled logic program of Figure 1. The evaluation is modeled as a sequence of forests of *SLG trees* to which various operations are applied. Nodes of SLG trees either have the form

$$Answer\_template \text{ :- } Delay\_list | Goal\_list$$

or *fail*. In the first form, *Answer_template* is used to represent bindings made to a tabled subgoal in the course of resolution along a computation path; the *delay list*, *Delay_list*, contains a set of literals that have been selected by a fixed-order selection strategy, but whose evaluation has been delayed; the *goal list*, *Goal_list*, is a sequence of unresolved literals. The evaluation begins with a forest containing a single tree, whose single node is t :- $|t$. Children of a root node of an SLG tree are formed by PROGRAM CLAUSE RESOLUTION [2] against the literal of the goal list. Given a non-root node N in a tree, whose selected subgoal S is tabled, a NEW SUBGOAL operation checks whether S :- $|S$ forms the root of a tree in the forest; if not, a new SLG tree is created consisting of the root note S :- $|S$. The evaluation of t begins by executing several NEW SUBGOAL and PROGRAM CLAUSE RESOLUTION operations, producing the SLG forest, $\mathcal{F}_1$ in Figure 1. Also in Figure 1 is a *Subgoal Dependency Graph* induced by $\mathcal{F}_1$, ($SDG(\mathcal{F}_1)$), which is used to represent dependencies among subgoals. Given a forest $\mathcal{F}$, nodes in $SDG(\mathcal{F})$ consist of subgoals in $\mathcal{F}$ that have not been marked as *complete* (see below); while an edge (N_1, N_2) lies between two nodes in $SDG(\mathcal{F})$ if N_2 is the selected literal of some node in the tree with root N_1 :- $|N_1$. If N_2 is a positive (negative) literal, the sign of (N_1, N_2) is positive (negative). As can be seen from Figure 1, the evaluation of the query t has encountered a negative loop containing q and r. In order to determine the truth value of q and r in the well-founded model, the fixed left-to-right literal selection strategy must be broken so that other literals in the clause for r may be resolved. A DELAYING operation is applied to node 5, moving the selected literal *not* r from the goal list

[2] Throughout this article, operations for SLG and its extensions are denoted in the font of PROGRAM CLAUSE RESOLUTION.

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

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

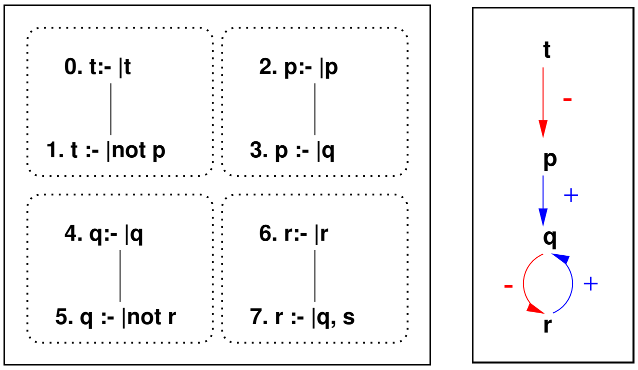


Figure 1. Program; SLG Forest $\mathcal{F}_1$ created while evaluating the query *?- t*; and $SDG(\mathcal{F}_1)$.

to the delay list, and producing the forest shown in Figure 2. At this stage, since node 8 is a leaf that has no more unresolved literals in its goal list it is termed an *answer* (for q), but because its delay list is non-empty it is termed a *conditional* answer. As can be seen in Figure 2 this conditional answer can be resolved against the selected literals of nodes 3 and 7. We assume for this example that it

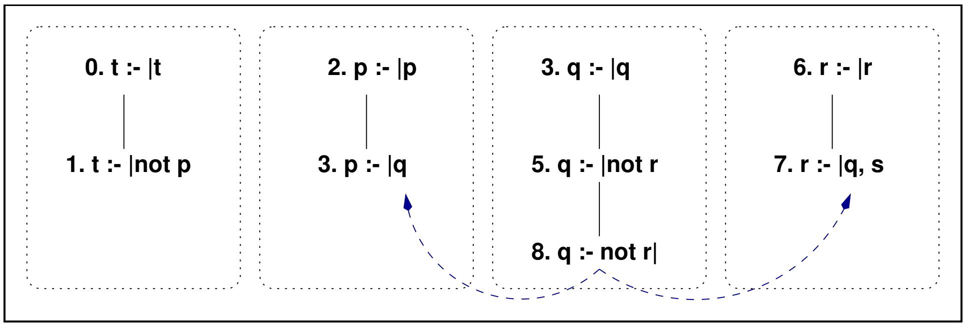


Figure 2. SLG Forest upon execution of the query *?- t.* against the program of Figure 1

is first returned to node 3 (through the SLG ANSWER RETURN operation). When a conditional answer A :- $DL|$ is resolved against a selected literal L, SLG does not propagate the elements of the delay list DL, but rather delays the selected literal, L, to indicate that a conditional answer was used for resolution. In non-propositional programs, this delayed literal may be subject to further unification operations so that positive delayed literals are annotated with the answer used for resolution, A, along with the root subgoal of the tree to which the answer belongs. As will be seen, these annotations provide control information for SIMPLIFICATION operations[3]. In this case, ANSWER RETURN adds the delayed literal q^q_q to the

[3] This use of positive delayed literals guarantees a polynomial representation of answers to queries that may otherwise have an exponential number of answers (see [10] for an example of that situation).

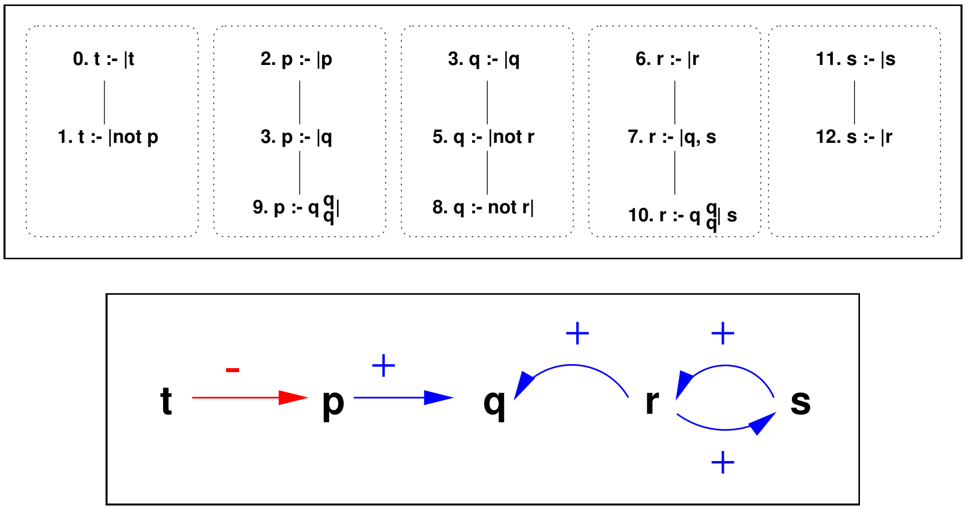


Figure 3. SLG Forest and SDG upon execution of the query *?- t.* against the program of Figure 1

delay list of node 3 producing a conditional answer for p (node 9 of Figure 3). Similarly, ANSWER RETURN of the conditional answer q :- *not* $r|$ is used for the selected literal of node 7 producing node 10. The evaluation then calls s, and recursively, r. The resulting forest is shown in Figure 3. As can also be seen from the *SDG* in Figure 3, s and r are involved in a positive loop, and can be determined to be *completely evaluated*, meaning that they can produce no further answers. Through a COMPLETION operation their trees are marked as *complete*. Upon the completion of r with no answers, since r is known to be false in the well-founded model. so the answer q :- *not* $r|$ can be made unconditional, As mentioned previously, the SLG SIMPLIFICATION operation is used to propagate truth values to delayed literals. Using SIMPLIFICATION, the delayed literal *not* r is removed from the delay list of q's answer making this answer *unconditional* (node 13 of Figure 4). The derivation of this unconditional answer enables a further SIMPLIFICATION operation to remove the delayed literal in the answer p :- $q^q_q|$ for p. Finally, since p's truth value is now established, a NEGATION RETURN operation can be performed to fail the computation path leading to node 1 of the forest, and through COMPLETION operations the remaining subgoals also become completed leading to the final forest represented in Figure 4.

The above example describes a *variant-based* tabling method, in which the NEW SUBGOAL operation creates a new tree if a subgoal is not a variant of the root of a tree already in a forest, and the ANSWER RETURN operation is applicable for an answer and a node N if a variant of that answer has not already been returned to N. Either of these operations could in principle be based on a relation of subsumption between terms, rather than variance. We restrict our

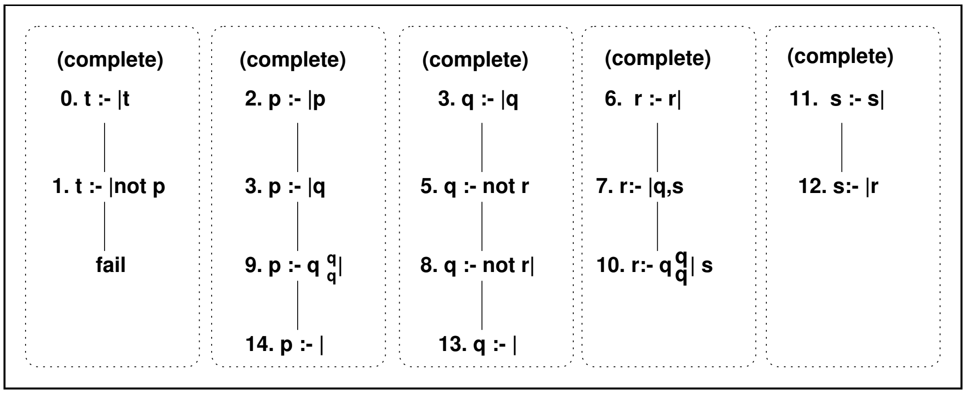


Figure 4. Final SLG Forest for the query *?-t* against the program of Figure 1

attention to variant-based tabling through out this section. Section 3.2 however, describes how an alternative version of the ANSWER RETURN operation is used to evaluate Generalized Annotated Logic Programs.

We summarize the actions of Example 1 in handling negation. Given a forest $\mathcal{F}$ and node N with a selected ground negative literal *not* S, the particular operation applicable for N depends on information known about the truth value of S in $\mathcal{F}$. If *not* S is *successful* — S is completely evaluated with no answers — a NEGATION RETURN operation can be used to remove the literal *not* S from the goal list of N. If *not* S is *failed*, — there is an unconditional answer S :- | in $\mathcal{F}$ — a NEGATION RETURN operation creates a child $fail$ of N indicating that the computation path of N leads to failure. If neither of these cases is true in $\mathcal{F}$, a DELAYING operation is applicable to N. As seen from the SDG of Figure 1, delaying a literal may avert dependencies through negation, perhaps selecting a false literal further in the body of a node as the false node s was selected in node 10. When such failures break loops through negation, they trigger SIMPLIFICATION operations. In general, delayed literals that are *successful* should be removed from the delay lists of answers in the forest, and failure nodes should be created as children of answers with delayed literals that are *failed* (thereby causing the nodes no longer to be answers).

2.1.1. Definitions of SLG Resolution

We now formalize notions from the previous section beginning with the notion of an SLG forest.

Definition 2 (SLG Forest).An *SLG forest* is a set of SLG trees. Nodes of an SLG tree either have the form

$$Answer\_Template \text{ :- } Delay\_List | Goal\_List.$$

or

$$fail$$

In the first form, the *answer template*, $Answer\_template$, is an atom, the *delay list*, $Delay\_List$, is a sequence of delayed literals, and the *goal list*, $Goal\_List$ is a sequence of literals. The second form is called a *failure node*. The root node of an SLG tree may be marked with the token *complete*. A node N is an *answer* when it is a leaf node whose goal list is empty. If the delay list of an answer is also empty, the answer is termed *unconditional*; otherwise it is a *conditional* answer.

A *negative delayed literal* has the form *not* S where S is a ground tabled subgoal; a *positive delayed literal* has the form S_{Call}^{Answer} where S, $Answer$ and $Call$ are atoms such that S is an instance of $Answer$ and $Answer$ is an instance of $Call$. If θ is a substitution, then $S_{Call}^{Answer}\theta = S\theta_{Call}^{Answer}$.

As terminology, if $\mathcal{F}$ contains a tree T with root S :- $|S$, we say that $\mathcal{F}$ *contains* S, and that T is the *tree for* S. If the root node of a tree for S is marked complete, we say that S is *complete*, and is *incomplete* otherwise. As mentioned above, in nodes of a tree rooted by a subgoal S, the answer template accumulates substitutions for the variables of S, the goal list contains literals that remain to be selected in order to derive an answer, while the delay list contains delayed literals that the evaluation has previously selected but has chosen to delay. The annotations of delayed literals provide control information that may aid in establishing their truth value in the course of further evaluation. Specifically, if an answer $Answer$ in the tree for $Call$ becomes unconditional or is failed, a simplification operation is applied to the literal $S_{Call}^{Answer}\theta$ $(= S\theta_{Call}^{Answer})$. Information in delay lists is propagated through *SLG answer resolution*.

Definition 3 (SLG Answer Resolution). Let N be a node of the form Ans_Templ :- $DL|\ S, L_1 \ldots, L_n$, where $n \geq 0$ such that S is a tabled subgoal. Let $Ans = AT$:- $DL_A|$ be an answer in the tree for S whose variables have been standardized apart from N. N is *SLG resolvable* with Ans if S and AT are unifiable with an mgu η. The *resolvent* of N and Ans on S has the form:

$$(Ans\_Templ \text{ :- } DL|\ L_1, \ldots, L_n)\eta$$

if DL_A is empty, and

$$(Ans\_Templ \text{ :- } DL, D|\ L_1, \ldots, L_n)\eta$$

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

Definition 4 (Success & Failure of Subgoals & Delayed Literals). Let $\mathcal{F}$ be an SLG forest. A subgoal S *succeeds* in $\mathcal{F}$ iff the tree for S has an answer of the form S :- $|$, and *fails* in $\mathcal{F}$ if the tree for S is completed, but has no answers.

In a forest $\mathcal{F}$, a negative delayed literal *not* S is *successful* iff subgoal S fails, and is *failed* iff S succeeds. A positive delayed literal S^{Call}_{Answer} is *successful* iff subgoal $Call$ has an answer of the form $Answer$:- $|$, and is *failed* iff the tree for $Call$ is completed, but does not have an answer $Answer$:- $DL|$ for any sequence of delayed literals DL.

Recall that a subgoal (or set of subgoals) is termed *completely evaluated* when no more answers can be produced for it.

Definition 5 (Completely Evaluated). A set $\mathcal{S}$ of subgoals in a forest $\mathcal{F}$ is *completely evaluated* if at least one of the conditions holds for each $S \in \mathcal{S}$

1. The tree for S contains an answer S :- $|$; or
2. For each node N in the tree for S:
   (a) The selected literal of N is completed or in $\mathcal{S}$; or
   (b) There are no applicable New Subgoal, Program Clause Resolution, Answer Return, Delaying, or Negative Return operations (Definition 6) for N.

We now define the set of SLG operations.

Definition 6 (SLG Operations). Given a forest $\mathcal{F}_n$ of an SLG evaluation of program P and query Q, $\mathcal{F}_{n+1}$ may be produced by one of the following operations.

1. New Subgoal: Let $\mathcal{F}_n$ contain a non-root node

   $$N = Ans \text{ :- } Delay\_List|G, Goal\_List$$

   where G is the selected literal S or *not* S. Assume $\mathcal{F}_n$ contains no tree with root S :- $|S$. Then add the tree S :- $|S$ to $\mathcal{F}_n$.
2. Program Clause Resolution: Let $\mathcal{F}_n$ contain a node N such that N is a root node S :- $|S$, or such that the selected literal S of N is non-tabled. Let C be a program clause $Head$:- $Body$ whose variables are standardized apart from those in N and such that $Head$ unifies with S with mgu θ. Assume that in $\mathcal{F}_n$, N does not have a child $N_{child} = (S \text{ :- } |Body)\theta$. Then add N_{child} as a child of N.
3. Answer Return: Let $\mathcal{F}_n$ contain a non-root node N whose selected literal S is tabled and positive. Let Ans be an answer node for S in $\mathcal{F}_n$ and N_{child} be the SLG resolvent of N and Ans on S. Assume that in $\mathcal{F}_n$, N does not have a child N_{child}. Then add N_{child} as a child of N.
4. Negative Return: Let $\mathcal{F}_n$ contain a leaf node

   $$N = Ans \text{ :- } Delay\_List|not\ S, Goal\_List$$

whose selected literal *not* S is ground.

(a) If S is failed in $\mathcal{F}_n$, then create a child for N of the form

$$Ans \text{ :- } Delay\_List|Goal\_List.$$

(b) If S succeeds in $\mathcal{F}_n$, then create a child for N of the form *fail*.

5. Delaying: Let $\mathcal{F}_n$ contain a leaf node

$$N = Ans \text{ :- } Delay\_List|not\ S, Goal\_List$$

whose selected literal *not* S is ground, and assume that $\mathcal{F}_n$ contains S, but S is neither successful nor failed in $\mathcal{F}_n$. Then create a child for N of the form Ans :- $Delay\_List, not\ S|Goal\_List$.

6. Simplification: Let $\mathcal{F}_n$ contain a leaf node $N = Ans$:- $Delay\_List|$, and let $L \in Delay\_List$

(a) If L is failed in $\mathcal{F}_n$ then create a child *fail* for N.

(b) If L is successful in $\mathcal{F}_n$, then create a child Ans :- $Delay\_List'|$ for N, where $Delay\_List' = Delay\_List - L$.

7. Completion: Given a completely evaluated set $\mathcal{S}$ of subgoals (Definition 5), mark the trees for all subgoals in S as *complete*.

8. Answer Completion: Given a set of unsupported answers $\mathcal{UA}$ (Definition 7) create a child of the form *fail* for each answer $Ans \in \mathcal{UA}$.

The definition of a supported answer is required by the Answer Completion operation, but is not otherwise used in this paper.

Definition 7 (Supported Answer). Let $\mathcal{F}$ be a SLG forest, S a subgoal in $\mathcal{F}$, and *Answer* be an atom the occurs in the head of some answer of S. Then *Answer* is supported by S in $\mathcal{F}$ if and only if:

1. S is not completely evaluated; or
2. there exists an answer node $Answer$:- $Delay\_list|$ of S such that for every positive delay literal D_{Ans}^{Call} in $Delay\_list$, Ans is supported by $Call$.

Note that unconditional answers are always supported. The concept of a supported answer is used by the SLG Answer Completion operation to handle cases in which delay propagation through answer resolution creates conditional answers that cannot later be simplified away. Unsupported answers appear rarely in practical evaluations: see [10] or [53] for details of how this case may arise.

It is well known that rooted trees can be represented by sets (see e.g. [20]). Furthermore, if nodes of SLG trees are taken as terms, then two nodes of are considered identical if they are variant terms. Based on these observations, the limit of a monotonically increasing sequence of forests can be formulated using set union operations, so that transfinite computation can be formulated using SLG forests (see [53] for details).

Definition 8 (SLG Evaluation). Given a program P, and an atomic query Q, a tabled evaluation $\mathcal{E}$ is a sequence of SLG forests $\mathcal{F}_0$,$\mathcal{F}_1$,...,$\mathcal{F}_\beta$, such that:

- $\mathcal{F}_0$ is the forest containing a single tree Q :- | Q
- For each successor ordinal, $n+1 \leq \beta$, $\mathcal{F}_{n_1}$ is obtained from $\mathcal{F}_n$ by an application of a tabling operation (Definition 6)
- For each limit ordinal $\alpha \leq \beta$, $\mathcal{F}_\alpha$ is defined as the set of trees T such that
  - ∗ The root of T, S :- $|S$ is the root of some tree in a forest F_i, $i < \alpha$; and
  - ∗ $T = \cup\{T_i | T_i \in \mathcal{F}_i, i < \alpha$ and T_i has root S :- $|S\}$

If no operation is applicable to $\mathcal{F}_\alpha$, $\mathcal{F}_\alpha$ is called a *final forest* of $\mathcal{E}$. If $\mathcal{F}_\beta$ contains a leaf node with a non-ground selected negative literal, it is *floundered.*

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

Definition 9. Let $\mathcal{F}$ be a forest. Then the *interpretation induced by* $\mathcal{F}$, $I_\mathcal{F}$ has the following properties.

- A (ground) atom $A \in I_\mathcal{F}$ iff A is in the ground instantiation of some unconditional answer Ans :- | in $\mathcal{F}$.
- A (ground) literal *not* $A \in I_\mathcal{F}$ iff A is in the ground instantiation of a completely evaluated subgoal in $\mathcal{F}$ (Definition 5), and A is not in the ground instantiation of any answer in $\mathcal{F}$.

Using the above definition, soundness and completeness of SLG evaluations with respect to the well-founded model (WFM) is stated by the following theorem.

Theorem 10 [10]. Let Q be an atomic query to a normal program P. Then any SLG evaluation of Q to P will reach a final forest, $\mathcal{F}$. Let $\mathcal{Q}$ denote the set of root subgoals of trees in $\mathcal{F}$. Then if $\mathcal{F}$ is not floundered, then $I_\mathcal{F} = WFM|_\mathcal{Q}$.

Theorem 10 is extended in [10] to show that SLG preserves stable models in the sense that if all atoms in a ground program P were evaluated using SLG, the resulting (conditional and unconditional) answers can be considered as a *residual program* P_{final} and that the stable models of P_{final} are the stable models of P. SLG can thus be used as a preprocessor which produces a residual program which is then sent on to a stable model generator. Section 4.3 discusses how the residual program is used in model checking.

As a final definition, we present the *Subgoal Dependency Graph* of an SLG forest, a construct that is used to describe strategies for scheduling SLG operations.

Definition 11. Given a forest $\mathcal{F}$, $SDG(\mathcal{F})$, is a directed graph such that

- V is a vertex in $SDG(\mathcal{F})$ iff V :- $|V$ is the root node of a tree in $\mathcal{F}$ that is not marked as *complete*.
- (V_1, V_2) is an edge in $SDG(\mathcal{F})$ iff V_2 is the selected literal in the goal list of a node in the tree for V_1. If V_2 is a positive literal the *sign* of (V_1, V_2) is positive, and is otherwise negative.

Because the SDG of an SLG forest is a directed graph, the mutual dependencies between subgoals correspond to *Strongly Connected Components* (or *SCCs*) of the SDG and are defined in the usual manner. An SCC that depends on no other SCC is termed *independent*.

2.2. Programming with SLG

As mentioned, SLD(NF) resolution is used as a default for predicates unless a predicate is declared as tabled. As implemented in the XSB system, this allows Prolog programs to be executed without requiring modifications. We discuss several other features of SLG as implemented in XSB that support programming with tabling. Details of the engine-level implementation of XSB can be found in [46,47,26,44].

A feature of Prolog worth preserving within tabling is the fixed-order selection strategy for literals within a clause. Programmers (and compiler optimizations) make use of a fixed-order literal selection strategy so that, a test literal say, will be chosen as soon in an evaluation as possible. As described above, SLG uses a fixed-order selection strategy in the goal list of a node, but a DELAYING operation can move a negative literal from the goal list of a node to the delay list. In this way, DELAYING effectively breaks the fixed-order strategy and can thus open up search space in a program that was unintended by the programmer/optimizer. To address this problem, XSB implements a version of SLG so that it has a property termed *delay minimality* [48]. The definition of delay minimality depends on the definition of dynamic stratification [42] of normal programs. A program is dynamically stratified if and only if it has a two-valued well-founded model: thus the class of dynamically stratified programs includes programs in all other stratification classes. A restriction of dynamic stratification, called *fixed-order dynamic stratification* denotes those programs that are dynamically stratified under a fixed literal selection strategy. An SLG evaluation is delay minimal only if it delays negative literals that are involved in a component that is not *fixed-order dynamically stratified.* It can be shown [48] that a delay minimal evaluation preserves a fixed-order literal selection strategy as much as is possible under the well-founded semantics.

Delay minimality can be ensured by, among other actions, postponing DELAYING operations until it is determined that they are absolutely needed. Delay minimality can thus be seen as a property of the *scheduling strategy*, or rule for ordering operations, of an evaluation method. Indeed, Theorem 10 states that correctness of SLG is independent of the *scheduling strategy* or the order of operations within an evaluation, giving a great deal of flexibility for determining useful scheduling strategies.

Efficiently scheduling the COMPLETION operation can be of great importance to practical evaluations. To see this, note that the only information required from a completed SLG tree are its answers. In an actual implementation, of course, goals and their answers are stored in a global table in addition to being maintained in an SLG tree. Once a tree is completed only its answers are of use. Accordingly, memory for the tree can be freed, and answers read directly from the global table. By scheduling COMPLETION operations as early as possible (or performing *incremental completion*), tabled evaluations become efficient enough to solve large-scale practical problems.

In addition to choosing when to perform COMPLETION and DELAY, The choice of when to perform an ANSWER RETURN operation can be of great importance to practical evaluations. [26] introduces a particular evaluation strategy called *batched evaluation* which cycles through nodes that consume answers, returning batches of answers to each node by backtracking through the answers as they are demanded. Experimental results show that within the context of the XSB engine, batched evaluation has good space management properties over other strategies for performing the ANSWER RETURN operation. Batched evaluation is an efficient way to find the first answer to a query and is the default strategy used by XSB. Furthermore, it is implemented so that the delay minimality property holds. However, as shown in [26], a second evaluation strategy, called *local evaluation* can be useful when certain answers are preferred over others — such as in tabled aggregation (a use of tabled aggregation is discussed in Section 3.2). The idea behind local evaluation is to allow an ANSWER RETURN operation to be applicable in a forest $\mathcal{F}$ only if the node to which an answer is returned belongs to a tree whose root subgoal is in an independent SCC of $SDG(\mathcal{F})$ (Definition 11). In other words, local evaluation completely evaluates all subgoals in an independent SCC before returning any answers out of the SCC. In the case of tabled aggregation only maximal or minimal answers need be returned out of an SCC. Use of local evaluation in such cases can lead to improvements in the asymptotic complexity of such evaluations [26]. In XSB, local evaluation is built an extension of batched evaluation so that it shares the space management properties of batched evaluation, and is delay minimal in almost all cases. In summary, when all answers to a query are desired, local evaluation may be more efficient than batched evaluation, but when it is important to reduce the time to the first answer of a query or when multiple answers are not required, batched evaluation is generally more efficient than local evaluation.

2.3. System overview of XSB

As mentioned, XSB adds tabling to Prolog, and as a Prolog system, XSB approximates the ISO standard [30]. When no tabling is used, XSB executes standard Prolog benchmarks 2-3 times slower than emulated SICStus or Quintus. This difference in speed is not primarily due to the overhead of tabling — for Prolog computations this overhead is about 10-15% [46]. The difference in speed rather is due to the fact that XSB does not have a threaded interpreter, as does Quintus, and does not contain many of the sophisticated compiler and emulator optimizations of SICStus. Much of the non-research oriented development effort of XSB has been to make XSB robust and portable rather than to improve its speed. For instance, XSB has been ported to a variety of 32 and 64 bit UNIX platforms, as well as Windows 95/8, and Windows NT. It supports interfaces to other languages, (C, Java and Perl), and to databases (Oracle and general ODBC connectivity). To enhance performance of Prolog code, however, XSB contains a fast reader for canonical clauses, and fast assert and execution of dynamic facts. Various mechanisms for indexing clauses have also been developed and implemented, such as *trie* indexing, which can be used to index on information nested within asserted facts [44].

However, it is the tabling constructs of XSB that are most pertinent to this paper. In addition to the `:- table` declaration introduced in Section 2.1, XSB also provides a declaration to automatically table predicates for termination. The declaration `:- auto_table` causes the XSB compiler to ensure that each loop on the predicate dependency graph of a program in a given file will be broken. Several operators make use of tabling. In particular the operator `tnot/1` is used to denote well-founded negation, evaluated via SLG, while the operator `\+/1` evaluates Prolog's negation by failure. Thus, in subsequent programming examples, the operator `tnot/1` will denote well-founded negation. Tabled aggregation predicates are also available, such as `bagPO/3` and `bagReduce/3`. The predicate `bagPO(Unary_pred,Result,Partial_order)` finds all solutions to `Unary_Pred` that unify with `Result` and that are maximal according to `Partial_order`. For example solutions to the goal `bagPO(p,X,my_greater_than)` are those solutions *p(Sol)* such that

$$\neg\exists New\_sol\ (p(New\_sol) \text{ and } my\_greater\_than(New\_sol, Sol))$$

`bagPO/3` uses tabling not only for termination, as with normal programs, but also to materialize the set of maximal solutions as they are derived. The related predicate, `bagReduce/3` uses tabling to maintain the value of an associative function applied to a set of answers: for instance, the sum or minimum of a set of answers. In terms of applications, the predicate `bagPO/3` is used to constuct preference grammars for data standardization in Section 4.2; while a predicate closely related to `bagReduce/3` is used to create a meta-interpreter for generalized annotated programs in Section 3.2.

XSB also provides several predicates to view tables. The predicate `get_residual(?Answer,-Delay_list)` unifies its first argument with an answer in a complete table and its second argument with the delay list of that answer. Thus, on completed tables, `get_residual/2` serves much as Prolog's `clause/2`, so that meta-programming can be performed on a (completed) table itself. XSB also provides routines for abolishing and reclaiming space for the table of a particular subgoal, for tables of a given predicate, and for all tables present in an evaluation.

3. Programming Non-Monotonic Logics using XSB

3.1. Programming the Well-Founded Semantics with Explicit Negation

As a first example, we consider how tabling can be used to extend the well-founded semantics with explicit negation — that is, with negative literals that are derived explicitly rather than implicitly as is done using negation by failure. A well-known example of the use of explicit negation concerns instructions to school bus drivers [28]:

A school bus may cross railway tracks only if there is no approaching train.

Using the well-founded semantics, the rule could be encoded as

```
cross :- not train.
```

but this encoding is unsatisfactory, since the bus driver could cross if the driver had no evidence that a train was approaching. In fact, what is intended here is surely something stronger: the driver should cross only if he or she could prove that a train was not approaching. This can be written as

```
cross :- -train.
```

Where the operator `-/1` denotes explicit negation.

Explicit negation has been studied by a number of authors (see [16] for a summary of this work), here we consider programming the *Well-Founded Semantics with Explicit Negation* or *WFSX* [2]. The key idea behind WFSX is the enforcement of the *coherence principle*, which, informally, states that the default negation of a statement is entailed by its explicit negation. WFSX has well-understood relations with other semantics for explicit negation, such as answer sets [28]. Furthermore, the use of explicit negation under WFSX is well-motivated by applications in defeasible reasoning including taxonomic reasoning [3], in reasoning about actions and program updates, [37,36], in diagnosis [50] and in machine learning [33].

Before presenting how WFSX can be programmed in XSB, we review relevant definitions. With the following exceptions, terminology for programs with explicit negation is the same as for normal programs. An *objective literal* is either an atom A, or its explicit negation $-A$. If an objective literal O is an atom A, the *complement* of O is $-A$; otherwise if O has the form $-A$, then its complement is A. A (default) literal, either has the form O, where O is an objective literal, or *not* O, the default negation of O. A rule has the form

$$Head \text{ :- } Body.$$

where $Head$ is an objective literal and $Body$ is a conjunction of literals. An *extended* program is comprised of rules of this form. An interpretation of an extended program is simply a set of literals, as with the well-founded semantics. In addition, any objective literal $-A$ can be understood as an atom not_A that is unrelated to A, and several definitions below will be based on this understanding.

We first consider a fixed-point definition for a paraconsistent version of WFSX, termed $WFSX_P$ that allows an objective literal to be both true and (default or explicitly) false. This fixed-point definition relies on the notion of a semi-normal version of a program.

Definition 12 [2].The semi-normal version of an extended program P is the program P_s, obtained from P by adding to the (possibly empty) *Body* of each rule

$$Head \text{ :- } Body$$

the default literal *not Head'*, where *Head'* is the complement of *Head*.

Next, we let Γ_P denote the stability operator [27] applied to an interpretation of P, where every objective literal $-A$ is understood as a new atom unrelated to A. Then

Definition 13 [27].Let P be an extended program and I be a two-valued interpretation. Then $\frac{P}{I}$ denotes the program obtained by removing from P all rules containing a default literal *not* A such that $A \in I$, and by then removing all the remaining default literals from bodies of rules in P. $\Gamma_P(I)$ is the least two-valued model of the program $\frac{P}{I}$ obtained by considering all objective literals of the form $-A$ as atoms unrelated to A.

Accordingly Γ_{P_s} denotes the stability operator applied to an interpretation of the semi-normal rewrite, P_s. Note that the operator $\Gamma_P \Gamma_{P_s}$ will be monotonic, and hence will have a least fixed point. In the following definition $\mathcal{H}_P$ is the Herbrand Base of a program P in which objective literals are understood as atoms.

Definition 14. Let P be a program, and let the least fixed point of $\Gamma_P\Gamma_{P_s}$ be T. The paraconsistent well-founded model of P ($WFSX_P(P)$) is defined as $T \cup not(\mathcal{H}_P - \Gamma_{P_s}(T))$,

In the special case where an extended program contains no explicitly negated objective literals, the semi-normal rewrite of P is equal to P, so that $WFSX_P(P) = WFS(P)$.

Example 15. The program P^1 and its semi-normal rewrite, both shown in Figure 5, illustrate features of $WFSX_P$. From Definition 14 it can be seen

`a:- b.`	`a:- b,not -a.`
`b:- not b.` `-b.`	`b:- not b,not -b.` `-b:- not b.`
`c:- d`	`c:- d,not -c.`
`d` `-d`	`d:- not -d.` `-d:- not d.`

Figure 5. Program P^1 and its Semi-normal Rewrite.

that true objective literals in $\Gamma_P\Gamma_{P_s}(\emptyset)$ are $\{a, -b, c, d, -d\}$. Continuing until fixed point, it can be seen that the paraconsistent $WFSX_P$ model of P is $\{-b, c, d, -d, not\ a, not\ -a, not\ b, not\ -c, not\ c, not\ d, not\ -d\}$. The coherence principle ensures that the derivation of *not b* holds (and by extension *not a*), by ensuring that the derivation of b fails in the semi-normal rewrite. It also causes the objective literals c, d, and *-d* to have inconsistent truth values. In a practical system, this paraconsistency would be temporary, and removed via a technique for belief revision [2]. Alternatively, if paraconsistency was not deemed useful, one could simply retain only the consistent models, i.e those that do not contain the pair L and $-L$ for any objective literal L.

Given the fact that $WFSX_P$ is defined as the least fixed point of the operator $\Gamma\Gamma_s$, and that WFS itself can be defined as the least fixed point of the operator $\Gamma\Gamma$, it is not surprising that $WFSX_P$ is embeddable into WFS, by the following transformation, first published in [16].

Definition 16 (T-TU Transformation). Let P be an extended logic program. Then a normal logic program P^{T-TU} is derived from P by replacing each rule R in P with two new rules.

- In the first rule, all default literals of the form *not* L in R are replaced by literals of the form *not* L_{TU}, while all other literals of the form L are replaced by literals of the form L_T.
- In the second rule, all default literals of the form *not* L in R are replaced by literals of the form *not* L_T, while all other literals of the form L are replaced by literals of the form L_{TU}. In addition, if O is the objective literal in the head of R, then the literal *not* O' is added to the body of R, where O' is the complement of O.

Example 17. The T-TU transformation of P^1 is shown in Figure 6. Note that at this point, explicit negation can be treated simply as a function symbol, and the transformed program can be directly executed by any evaluation method for WFS.

```
a_t :- b_t.
a_tu :- b_tu,not -a_t.

b_t :- not b_tu.
-b_t.
b_tu :- not b_t,tnot -b_t.
-b_tu :- not b_t.

c_t :- d_t.
c_tu :- d_tu,not -c_t.

d_t.
-d_t.
d_tu :- not -d_t.
-d_tu :- not d_t.
```

Figure 6. T-TU Transformation Applied to P_1.

Theorem 18. Let P be an extended logic program. Then the following equivalences hold for an objective literal O (considered as an atom):

- $O \in WFSX_P(P)$ iff $O_T \in WFM(P^{T-TU})$;
- *not* $O \in WFSX_P(P)$ iff *not* $O_{TU} \in WFM(P^{T-TU})$;

3.1.1. A Meta-Interpreter for $WFSX_P$

Figure 7 shows both an unoptimized and an optimized meta-interpreter for the T-TU transformation. The unoptimized meta-interpreter, shown on the left of Figure 7, simply encodes the T and TU transformations as two tabled demo-style predicates which call each other. In the final clause of the unoptimized meta-interpreter, the predicate *exchange(?Olit,?Complement)* helps to perform the semi-normal transformation by replacing an objective literal with its complement. The unoptimized meta-interpreter is simple, but unnecessarily tables conjunctions of body literals. This can easily be remedied by tabling only calls to literals as on the right of Figure 7. The optimized meta-interpreter has two differences from the unoptimized meta-interpreter. First, the predicates `demo_t/1` and `demo_tu/1` are no longer tabled. Instead, calls to literals are performed by the tabled predicates `tdemo_t/1` and `tdemo_tu/1`. The result is that only answers to meta-interpreter calls to objective literals need be entered into the table, leading to practical gains in time and space efficiency.

```
demo(A):- demo_t(A).
demo(A):- demo_tu(A).

:- table demo_t/1, demo_tu/1.
demo_t(true).
demo_tu(true).
demo_t(',’(A,B)):-!,
    demo_t(A),demo_t(B).
demo_tu(',’(A,B)):-!,
    demo_tu(A),demo_tu(B).
demo_t(not A):-!,
    tnot(demo_tu(A)).
demo_tu(not A):-!,
    tnot(demo_t(A)).
demo_t(A):- clause(A,B),
    demo_t(B).
demo_tu(A):-clause(A,B),
    demo_tu(B),
    exchange(A,A_opp),
    demo_tu(not A_opp).
```

```
demo(A):- demo_t(A).
demo(A):- demo_tu(A).


demo_t(true).
demo_tu(true).
demo_t(',’(A,B)):-!,
    demo_t(A),demo_t(B).
demo_tu(',’(A,B)):-!
    demo_tu(A),demo_tu(B).
demo_t(not A):-!,
    tnot(tdemo_tu(A)).
demo_tu(not A):-!
    tnot(tdemo_t(A)).
demo_t(A):- tdemo_t(B).

demo_tu(A):- tdemo_tu(A).



:- table tdemo_t/1,tdemo_tu/1.
tdemo_t(A):- clause(A,B),
    tdemo_t(B).
tdemo_tu(A):- clause(A,B),
    demo_tu(B),
    exchange(A,A_opp),
    demo_tu(not A_opp).
```

Figure 7. Unoptimized and Optimized Meta-Interpreters for $WFSX_P$

Several features of both meta-interpreters are worthwhile to note. First, the variant-style tabling of Section 2.1 means that the meta-interpreter will make all calls that Prolog makes and return all answers that Prolog returns. Also note that the Prolog cut operator can be used with tabled predicates, as long as a tabled predicate does not occur within the scope of the cut. Finally, we note that even if the underlying program to be interpreted is left-to-right modularly stratified, the meta-interpreted program may not be since interpretations of the meta-interpreted programs consist only of demo-style predicates. However, the meta-interpreted program will be fixed-order stratified (Section 2.2). Thus, the property of delay minimality — the ability to evaluate fixed-order stratified programs without using SIMPLIFICATION or DELAYING operations — can lead to a practical performance gain when implementing tabled meta-interpreters over traditional stratification classes.

For a particular objective literal O, the meta-interpreters of Figure 7 can indicate whether `demo_t(A)` or `demo_tu(A)` is true or false in the well-founded model. An additional predicate is needed to indicate whether O is true or false in the paraconsistent $WFSX_P$ model. In XSB if an answer `A` is undefined, then both `A` and `tnot(A)` will succeed, and that A will also have a non-empty delay list. Accordingly, the simplest way to determine whether A is true is to check the table entry for A once its table is completed. The predicate `wfsxp_value/2` does just this:

```
wfsxp_value(Obj_lit,Value):-
        (demo(Obj_lit),fail ; true),
        (   get_residual(demo_t(Obj_lit),[]),Value = true)
        ;
            \+ get_residual(demo_t(Obj_lit),_),Value = false
        ;
            Value = undefined
        ).
```

By calling `demo/1` within a failure-driven loop, `wfsxp_value/1` ensures that the XSB predicate `get_residual/2` is called on a completed table. If the objective literal has an unconditional answer for its T-evaluation, the literal will be true in the paraconsistent $WFSX_P$ model according to Theorem 18. Otherwise, if its TU-evaluation is failed, the literal will be false. Of course, once a transformation such as that of Definition 16 is provided a preprocessor can be written rather than a meta-interpreter, which is the approach taken by the XSB WFSX library. This library allows users to preprocess programs according to the T-TU transformation so that they can be run directly using XSB's engine [4].

We also mention that the technique of defining a transformation, and writing a meta-interpreter (or preprocessor) can be used to implement many different

[4] The XSB WFSX library was implemented by J. Alferes and L.M. Pereira.

logics using tabling. For instance, [16] illustrates how several other logics can be transformed into WFSX — these include Generalized Horn Programs [7], Extended Deductive Databases [59], and IMEX Negation [31] — and each of these can be evaluated using XSB. In addition meta-interpreters have been written in XSB for Head-cycle-free disjunctive logic programs [6], for Defeasible Logic, using the transformation of [5], for the program update method of [36], and for abduction over $WFSX_P$ [4]. Two features of tabling as presented in Section 2 make construction of these meta-interpreters possible. First, since each of these formalisms can be transformed into the well-founded semantics, the soundness, completeness, and termination properties of tabling extend to all of these formalisms and implicitly underly the meta-interpreters. Also, since tabling is combined with Prolog evaluation, Prolog techniques, such as the cut and the use of *call*/1, can be used to program the meta-interpreters.

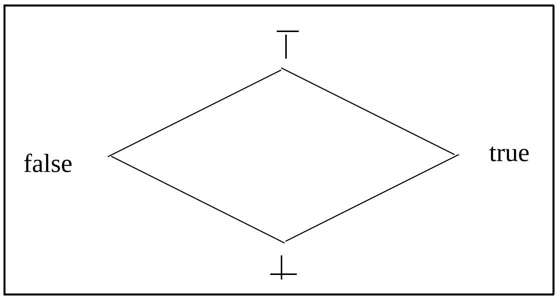
3.2. Implementing Generalized Annotated Programs with Tabled Aggregation

An interpretation I in the well-founded semantics can be seen as a mapping from the set of ground atoms to a truth ordering consisting of the elements $false$, *undefined*, and *true*, where an atom A is mapped to *true* if $A \in I$, to $false$ if *not* $A \in I$, and to *undefined* if neither A nor *not* A is an element of I. In general, however, an interpretation could also be defined as a mapping to some other ordering or lattice — such as Belnap's four-valued lattice, whose values are $false$, *bottom* (i.e. undefined), *true*, and *top* (i.e. inconsistent). *Generalized Annotated Programs (GAPs)* [32] realize this notion by allowing atoms to be mapped into arbitrary lattices of truth values. Using the Belnap or similar lattices as the domain of truth values, GAPs may express inconsistent or uncertain information. In addition, GAPs introduce the underlying lattice of truth values within the language of a program by explicitly annotating atoms with their truth values. These features make GAPs a powerful extension of definite programs. As shown in [32], GAPs can encode bilattace-based logic programs [24], van Emden's Quantitative Deduction [56], as well as several forms of temporal reasoning. In addition, GAPs can express types of probabilistic reasoning and as shown below, can encode certain uses of recursive aggregation.

Because of their explicit use of annotations, GAPs cannot be directly embedded into normal programs, and so cannot be directly computed with SLG as described in Section 2.1. However, by making a small change to the SLG Answer Return operation to allow a type of tabled aggregation, SLG can be extended to evaluate queries to GAPs. As a result, a simple meta-interpreter for GAPs can be defined using XSB.

3.2.1. A Review of Generalized Annotated Programs

Example 19. We introduce Generalized Annotated Programs by showing how they can be used to reason in the presence of inconsistency in a logic program. For instance, the left side of Figure 19 represents Belnap's four valued lattice of truth values, in which the truth value $\top$ is used to represent inconsistent information. In P_{Belnap}, atoms are annotated with explicit truth values according to the Belnap lattice. The truth value of the atom p is $true$, while that of the atom q is explicitly declared to be $false$. The truth value of the atom r is the least upper bound of that of p and q, or $\top$.



$p{:}_{belnap}\ true.$
$q{:}_{belnap} false.$
$r{:}_{belnap} X$:- $q{:}_{belnap} X.$
$r{:}_{belnap} X$:- $p{:}_{belnap} X.$

Figure 8. Belnap Lattice and Annotated Program Defined over the Belnap Lattice

We begin by reviewing those aspects of GAPs as presented in [32] that are relevant to developing a tabled implementation, and afterwards describe the meta-interpreter for GAPs.

Definition 20 [32]. Let $\mathcal{T}$ be a complete lattice of truth values, whose ordering is denoted by $\leq_{\mathcal{T}}$; whose least upper bound operation is denoted by $\sqcup_{\mathcal{T}}$, whose bottom element is denoted by $\perp$, and whose top element by $\top$. Also, for each $i \geq 1$, let $\mathcal{G}_i$ denote a family of computable continuous functions of type $\mathcal{T}^i \rightarrow \mathcal{T}$. Finally, let $\mathcal{L}$ consist of a countable set of predicate symbols, function symbols and variables.

An *annotation* is either an element of $\mathcal{T}$, an *annotation variable*, or a *complex annotation term*. Annotation terms are built from annotation variables and elements of $\mathcal{G}$ in the standard manner. If A is an atom of $\mathcal{L}$, and α an annotation, then $A : \alpha$ is an annotated atom. An annotated atom $A : \alpha$ is *ground* if A is ground, *constant-annotated* if $\alpha \in \mathcal{T}$, *variable-annotated* if α is an annotation variable, and *term-annotated* if α is a complex annotation term. $A : \alpha$ is ground annotated if α, considered as a term, is ground.

If $A : \rho$ is an annotated atom, and $B_1 : \mu_1, ..., B_n : \mu_n$ are constant- or variable- annotated atoms, then

$$A : \rho \text{ :- } B_1 : \mu_1, ..., B_n : \mu_n$$

is an *annotated rule*. A GAP (over $\mathcal{L}$ and $\mathcal{T}$) is a set of annotated rules

Example 21. We further illustrate Definition 20 through the familiar example of computing a shortest path. Written as a GAP, a shortest path program has the form:

$$shortest\_path(X,Y) :_{min} D_1 + D_2 \text{ :- } \quad shortest\_path(X,Z) :_{min} D_1, \; edge(Z,Y,D_2).$$

$$shortest\_path(X,Y) :_{min} D \text{ :- } \quad edge(Z,Y,D).$$

in which the lattice of annotations, min, is explicitly denoted for clarity. In this case, the elements of min are the non-negative integers along with ∞, in which ∞ is the bottom element, 0 is the top element, and $X \leq_{min} Y$ iff Y is less than or equal to X in the usual ordering of $[0, \infty]$. The shortest path for a given pair of nodes, a and b is simply the greatest annotation μ, over the min ordering such that $sp(a, b) : \mu$ is true.

For our purposes, the semantics of a GAP P may be represented by the so-called *Restricted interpretation (R-interpretation)* of P which maps the ground atoms in the Herbrand Base of P to elements of $\mathcal{T}$. Formally,

Definition 22 (R-Satisfiability of Annotated Literals) [32]. Let P be a GAP over a language $\mathcal{L}$ and complete lattice $\mathcal{T}$. Then the *Herbrand Base* $\mathcal{H}_P$ of P consists of all ground atoms of $\mathcal{L}$. A R-interpretation I is a mapping from $\mathcal{H}_P$ into $\mathcal{T}$.

Let I be an R interpretation, $A : \mu$ a constant-annotated atom, and $\neg$ a function from $\mathcal{T}$ to $\mathcal{T}$. Then

- $I \models A : \mu$ iff $\mu \leq_{\mathcal{T}} I(A)$;
- $I \models \neg A : \mu$ iff $\neg\mu \leq_{\mathcal{T}} I(A)$.
- $I \models F_1, F_2$ iff $I \models F_1$ and $I \models F_2$.
- $I \models F_1$:- F_2 iff $I \models F_1$ or $I \not\models F_2$.
- $I \models (\forall x)F$ iff $I \models F(x/t)$ for all ground terms t. Here x is a variable in $\mathcal{L}$ or an annotation variable, and t must be of the same sort as x. $F(x/t)$ denotes the replacement of all free occurrences of x in F by t.
- $I \models (\exists x)F$ iff $I \models F(x/t)$ for some ground term t. As above, x is a variable in $\mathcal{L}$ or an annotation variable, and t must be of the same sort as x. $F(x/t)$ denotes the replacement of all free occurrences of x in F by t.
- If F is not a closed formula, then $I \models F$ iff $I \models (\forall)F$, where $(\forall)F$ denotes the universal closure of F.

Satisfiability of formulas of R-literals is defined in the usual manner of the predicate calculus. Note that the rules for satisfiability of $\neg A : \mu$ indicate that this term can always be replaced by $A : \neg\mu$. Accordingly, Definition 20, which constructed rule bodies out of conjunctions of annotated atoms did not restrict the generality of GAP rules [5].

Let I_1 and I_2 be R-interpretations of a program P. Then $I_1 \leq^I I_2$ iff for all $A \in \mathcal{H}_P$, $I_1(A) \leq_{\mathcal{T}} I_2(A)$. Note that by Definition 22, if I is an R-interpretation, then for any atom A, $I \models A : \bot$. Thus a unique $\leq^I$-least R-interpretation on P can be defined as the R-interpretation that maps every atom in $\mathcal{H}_P$ to the bottom element in $\mathcal{T}$.

The following operator, analogous to the T_P operator of definite programs, maps R-interpretations to R-interpretations.

Definition 23. [32] Let P be a GAP over a language $\mathcal{L}$ and complete lattice $\mathcal{T}$, I be an R-interpretation, and A a ground atom in $\mathcal{H}_P$. Then $R_P(I)(A) = \sqcup\{\mu | A : \mu$:- *Body* is a ground instance of a clause in P, and $I \models$ *Body*$\}$.

It can easily be seen that R_P is monotonic over $\leq^I$, and as a result, if $I_\bot$ is the least R-interpretation, R_P will have a least fixed point $lfp(R_P(I_\bot))$. It then is straightforward to show that $lfp(R_P(I_\bot))$ will be a $\leq^I$-least model of P.

Example 24. Suppose the program P_{sp} consists of the relation *shortest_path/2* along with the relation

$$\begin{array}{l} edge(a,b,2). \\ edge(b,c,2). \\ edge(a,c,5). \end{array}$$

Furthermore, let $I_\bot$ denote the R-interpretation that maps all elements of $\mathcal{H}_P$ to $\bot$. Then

$$R_P(I_\bot)(shortest\_path(a,c)) = \infty,$$

$$R_P^1(I_\bot)(shortest\_path(a,c)) = 5,$$

and

$$R_P^3(I_\bot)(shortest\_path(a,c)) = lfp(R_P(I\bot))(shortest\_path(a,c)) = 4.$$

Also, by Definition 22,

$$lfp(R_P(I_\bot))(shortest\_path(a,c)) \models n$$

for all $n \geq 4$.

R_P is not necessarily continuous, however, as can be seen from the following example in which a GAP is used to represent Zeno's paradox.

[5] A semantics for GAPs with default negation, but without negation as a function on lattices, was presented in [52].

Example 25. Let $real$ be the lattice induced by points on the real interval $[0, 1]$. Consider the program:

$$achilles :_{real} \frac{1+D}{2} \text{ :- } achilles_{real} : D.$$
$$achilles :_{real} 0.$$

$$finishes :_{real} 1 \text{ :- } achilles :_{real} 1.$$

Clearly, $lfp(R^P(I_\perp))(finishes) \models 1$, but $R^P \uparrow^\omega (I_\perp)(finishes) \not\models 1$.

3.2.2. A Method for Tabled Evaluation of GAPs

In order to evaluate GAPs, the Answer Return operation of SLG must be modified to use tabled aggregation to derive least upper bounds for the truth values of sets of answers. In order to implement this new operation, a program transformation is derived to separate the process of unifying the atom portion of annotated atoms from that of checking their annotations. A meta-interpreter using the extended Answer Return operation is then defined.

Extending SLG with Tabled Aggregation Annotated subgoals can be seen as instances of the predicate `:/2` so that concepts such as SLG trees and SLG answer resolution can be directly applied to GAPs. The first step, then to formulating tabled resolution for GAPs is to integrate the least upper bound operation of Definition 23 with the Answer Return operation of Definition 6. In order to do this, we define the notion of an *atom reductant*:

Definition 26. Let Set be a set of annotated atoms such that a mgu, θ, exists for the set $\{A_i |\ A_i : \mu_i \in Set\}$, and for any $A_i : \mu_i \in Set$ denote $A_i\theta$ by A. Next, let $\mu = \sqcup\{inst(\mu_i') |\ A_i : \mu_i \in Set\}$, where $inst(\mu_i) = \top$ if μ is a variable annotation and $inst(\mu_i) = \mu_i$ otherwise. Then $A : \mu$ is the *atom reductant* of Set.

Example 27. Atom reductants are used to extend the evaluation of GAPs to situations in which non-ground answers are present. For instance; the atom reductant of the set

$$\{p(a, Y) :_{belnap} true, p(X, X) :_{belnap} false\}$$

over the Belnap lattice is $p(a, a) :_{belnap} \top$. This reflects the property that, given a model M such that

$$M \models \forall Y\ p(a, Y) :_{belnap} true \text{ and } M \models \forall X\ p(X, X) :_{belnap} false$$

it can easily be seen that $M \models p(a, a) :_{belnap} \top$. Likewise, Definition 26 captures the fact that if $M \models \forall X\ A : X$ for some atom A, then $M \models A : \top$

We occasionally abuse terminology slightly by referring to the answer reductant of a set of unconditional answers. Next, we specify a simplifying assumption that will be made on the form of annotations.

Definition 28. Let μ_1, μ_2 be annotations. Then $\mu_1 \leq_{\mathcal{T}} \mu_2$ is *immediately evaluable* iff μ_1 is either a variable or a ground annotation and μ_2 is either a variable or a ground annotation.

When attempting to return an answer $A_2 : \mu_2$ to a subgoal $A_1 : \mu_1$, if $\mu_1 \leq_{\mathcal{T}} \mu_2$ is immediately evaluable, then it can be determined immediately whether $A_1 : \mu_1$ implies $A_2 : \mu_2$. Thus restricting GAP evaluations to those that contain only immediately evaluable inequality checks avoids the necessity of a constraint handler. Constraint handlers for GAP inequalities that are not immediately evaluable have been presented in the literature (e.g. [32,35]), and the design of these constraint handlers is orthogonal to the use of tabling for GAP evaluation. For simplicity of presentation, the definition of GAP Answer Return presented below causes floundering when an evaluation encounters an inequality that is not immediately evaluable.

- GAP Answer Return Let N be a node Ans_Templ :- $S, Goals$ whose selected subgoal, S is positive
  - ∗ If $S = A_s : \mu_s$ is an annotated atom, Let $A : \mu$ be the atom reductant of a subset of answers in the tree for S in $\mathcal{F}_n$.
    1. if $\mu_s \leq_{\mathcal{T}} \mu$, is not immediately evaluable, create a child *flounder* for N.
    2. Otherwise, if $inst(\mu_s) \leq_{\mathcal{T}} inst(\mu)$, and the mgu of A and A_s is ρ. Assume that N does not have a child $N_{child} = (Ans\_Templ$:- $S, Goals)\rho$. Then create N_{child} as a child of N.
  - ∗ If S is not an annotated atom, let $A = AT$:- be an answer for S in $\mathcal{F}$, If A has not been used for resolution against N, then produce a child N' that is the SLG answer resolvent of N and A.

Use of atom reductants by GAP answer resolution ensures that all possible atom, annotation pairs will be returned to the appropriate selected literals (cf. Lemma A.1). Note that since negation in GAPs is simply a function on annotations, a SLG$_{GAP}$ evaluation need consist of operations for definite programs. New Subgoal, Program Clause Resolution, GAP Answer Return, and Completion. An SLG$_{GAP}$ evaluation is *floundered* if its final forest contains a node *flounder*.

The following theorem state conditions on the soundness and completeness of SLG$_{GAP}$. Its proof is contained in the Appendix.

Theorem 29. Let P be a GAP over a language $\mathcal{L}$ and complete lattice $\mathcal{T}$, and $\mathcal{F}_n$ the final forest of a non-floundering SLG$_{GAP}$ evaluation of a query $Q : \mu$ to

P. Then $lfp(R_P) \models Q' : \mu'$ iff $Q' : \mu'$ is in the ground instantiation of an atom reductant of a set of answers in the tree for $Q : \mu$ in $\mathcal{F}_n$.

The following theorem states a sufficient condition for termination of an SLG_{GAP} evaluation. A GAP, P, is termed *function-free* if P contains no functions in its underlying atoms, and if all annotated atoms in P have either variable or constant annotations.

Theorem 30. Let P be a GAP program over a function-free language $\mathcal{L}$ and finite lattice $\mathcal{T}$, in which all predicates are declared to be tabled. Then the SLG_{GAP} evaluation of any query to P_N will reach a final forest after a finite number of SLG_{GAP} operations.

Proof: Clearly, a function-free GAP can contain only a finite number of annotated atoms, and there will only be a finite number of GAP ANSWER RETURN operations that can be performed. Furthermore, evaluation of each C_i in Definition 31 will terminate. The rest of the proof of termination then resembles the proof of termination for SLG. $\square$

Theorem 30 can be strengthened in a straightforward manner by defining concepts such as bounded term-size over GAPs.

Normalized form of Rules in a GAP The design of a meta-interpreter for tabled evaluation of GAPs can be simplified by defining its operations on programs whose rules have been *normalized*. Normalization alters the syntax of a rule so that the work of resolving annotated atoms against annotated subgoals can be separated from that of evaluating the annotations themselves. For instance given the domain *real* of Example 25, the normalized form of

$$p(X,Y) :_{real} \frac{D}{2} \text{ :- } r(X,Z) : 2, s(Z,Y) : D.$$

is

$$p(X,Y) : V_0 \text{ :- } r(X,Z) : V_1, eval(V_1,2), s(Z,Y) : V_2, eval(V_2,D), eval(V_0,D/2).$$

The normal form replaces all annotations with variables, and then feeds the variables into calls to the predicate *eval/2*. The predicate $eval(Ann_{ret}, Ann_{call})$ acts somewhat like a constraint processor, attempting to evaluate Ann_{ret} and Ann_{call} and to check that $Ann_{call} \leq_{\mathcal{T}} Ann_{ret}$. Before specifying the semantics of *eval/2*, we define the normalized form of a rule.

Definition 31. Let

$$R = A : \mu_0 \text{ :- } B_1 : \mu_1, ..., B_n : \mu_n$$

be an annotated rule. The normalized form of R consists of two rules:

$$R'_1 = A : \bot$$

$$R'_2 = A : V_0 \text{ :- } B_1 : V_1, eval(V_1, \mu_1), ..., B_n : V_n, eval(V_n, \mu_n), eval(V_0, \mu_0)$$

In R'_2, each V_i is a variable such that for $0 \leq i \leq n$, $V_i \neq V_j$, and V_i does not occur in R.

As mentioned in the previous section, the assumption that evaluations are non-floundered greatly simplifies the meta-interpreter, since in order to handle floundering a constraint handler is needed. The semantics for $eval/2$ is:

$$eval(Ans, Call) = \begin{cases} \text{if } Call \text{ is a variable:} & \text{succeed unifying } Call \text{ and } Ans \\ \text{if } Ans \text{ is a variable:} & \text{succeed} \\ \text{if } Ans \text{ and } Call & \\ \ \text{are ground:} & \text{evaluate } Ans \text{ and} \\ & \text{succeed if } Call \leq_{\mathcal{T}} Ans \end{cases}$$

Correctness of this semantics for $eval/2$ requires a rather technical statement, but is show in Lemma A.2.

3.2.3. A Meta-Interpreter for GAPs

Figure 9 shows the program of Example 21 as written for a simple GAP meta-interpreter. The predicate *shortest_path* is represented in XSB almost exactly as in Example 21 with the exception that evaluation of function symbols over $\mathcal{T}$ is converted to Prolog-style syntax through a source-code transformation. In addition, predicates are needed to define relevant operations for each lattice, and Figure 9 shows those for the lattices in Examples 19 and 21. The bottom element of the lattice *Type* is declared by a fact $bottom(+Type, -Fact)$; the least upper bound of two elements of the lattice *Type* by the predicate $lub(+Type, +Elt1, +Elt2, -Lub)$; and the partial order relation of lattice *Type* by the predicate $gte(+Type, +Elt1, +Elt2)$. Finally, for the lattice *belnap* a predicate defines the negation of truth values.

The meta-interpreter itself handles calls to annotated atoms by a call to *maintain_lub/3* which performs the explicit call of $eval/2$ and also calls *maintain_lub1/4*. The latter predicate implements a restriction of GAP ANSWER RETURN which assumes that each annotated answer is range restricted in its atom portion[6]. `maintain_lub1/4` is available as an XSB library predicate and is closely related to the XSB library predicate `bagReduce/4` [49]. Intuitively, `maintain_lub1/4` maintains a table for each call to an annotated atom along with (ground) answers for that call. In addition, *maintain_lub1/4* keeps the join of all truth values for a particular ground answer through destructive assignment. From an implementational point of view, `maintain_lub/4` *aggregates* all of the truth values for a ground atom into their least upper bound in a manner similar

[6] A meta-interpreter for GAPs, including source code to `maintain_lub1/4`, will be included as a library for Version 1.9 of XSB. The meta-interpreter is also currently available at `http://www.cs.sunysb.edu/ tswift`

```
/* Transformed Shortest Path Program */
shortest_path(X,Y):[min,D]  :-
   shortest_path(X,Z):[min,D1],edge(Z,Y,D2),D is D1 + D2.
shortest_path(X,Y):[min,D]  :- edge(Z,Y,D).

/* Definitions for the lattice min */
bottom(min,infinity).

lub(min,A,B,Min):- min(A,B,Min).

gte(min,A,B):- min(A,B,A).

min(One,infinity,One):-!.
min(infinity,Two,Two):-!.
min(One,Two,Min):- One > Two -> Min = Two ; Min = One.

/* Definitions for the lattice belnap */
bottom(belnap,bottom).

lub(belnap,true,false,top):- !.
lub(belnap,false,true,top):- !.
lub(belnap,A,B,A):- gte(belnap,A,B),!.
lub(belnap,A,B,B):- gte(belnap,B,A).

gte(belnap,top,_):-!.
gte(belnap,_,bottom):-!.
gte(belnap,X,X).

negate(true,false):-!.
negate(false,true):-!.
negate(X,X):-!.
```

Figure 9. Shortest-Path Program and Lattice Definitions

```
/* Meta interpreter for GAPs */
demo_gap(true):-!.
demo_gap(',' (Term1,Term2)):-!,
   demo_gap(Term1),
   demo_gap(Term2).
demo_gap((Atom:[Lattice,Annot]):-!,
   maintain_lub(Lattice,Atom,Annot).
demo_gap(gap_not(Atom:[Lattice,Annot])):-!,
   neg_maintain_lub(Lattice,Atom,Annot).
demo_gap(Term):- call(Term).

maintain_lub(Lattice,Atom,Annot):-
   bottom(Lattice,Bot),
   maintain_lub1(Lattice,Atom,Annot1,Bot),   /* Library Predicate */
   eval(Lattice,Annot1,Annot).

neg_maintain_lub(Lattice,Atom,Annot):-
   bottom(Lattice,Bot),
   maintain_lub1(Lattice,Atom,Annot1,Bot),   /* Library Predicate */
   negate(Lattice,Annot1,Annot2),
   eval(Lattice,Annot2,Annot).

eval(_Type,Val1,_Val2):- var(Val1),!.
eval(_Type,Val1,Val2):- var(Val2),Val1 = Val2,!.
eval(Type,Val1,Val2):- eval_1(Type,Val1,Val2).
        /* Evaluate Val1 and succeed if greater than Val2 */
```

Figure 10. GAP Meta-Interpreter

to the way a set of database facts is aggregated into their sum, count, minimum or maximum.

Tabling is thus used to implement GAP programs in two ways: to ensure termination for programs such as the shortest path program of Example 21, and to maintain working least upper bounds throughout an evaluation via `maintain_lub/2`. These actions are hidden from programmers so that to implement GAPs, a user must only program the relevant aspects of the lattice over which computation is to be performed: the bottom element, the partial order, the least upper bound operation and a negation operation (if desired). GAPs can be particularly efficient when a local evaluation is used as a scheduling strategy (Section 2.2). Recall that local evaluation returns an answer out of an independent SCC only when all subgoals in the SCC have been completely evaluated.

For GAPs, this means that an answer $A : \mu$ is returned to a node outside an independent SCC only when it can be determined that the annotation μ will not be superceded by a greater annotation, leading to gains in efficiency which are sometimes asymptotic [7].

The resulting meta-interpreter under local scheduling is powerful enough to be used as a component that can handle paraconsistent or quantitative information within large-scale applications. As an example, GAPs have been used to combine probabilistic information from mined association rules (using a GAP-like subset of Hybrid Probabilistic Programs [19]) in processing over 20 million records of a large database. Similar forms of tabled aggregation have also been used to implement Preference Logic Programs [29] whose uses are discussed in Section 4.2,

We note that other evaluation methods for GAPs are known besides SLG_{GAP}. The constrained-query evaluation method [32,35] handles what are called here non-floundering evaluations, but both are based on SLD-style evaluation, so that termination properties along the lines of Theorem 30 have not been shown for these methods. [1], however, introduces a tabling method for handling GAPs, although its method for determining least upper bounds of sets of atoms differs from the use of atom reductants in Definition 26. The significance of the implementation of GAPs described in this section lies its use of general-purpose tabling routines, along with its scalability.

4. Applications of Non-Monotonic Reasoning Implemented in XSB

4.1. Psychiatric Diagnosis and the Well-Founded Semantics with Explicit Negation

It is sometimes thought that non-stratified negation is not of use for practical programs, and is only of theoretical importance. However, the ability to represent psychiatric knowledge depends directly on non-stratified negation. Knowledge about the classification of psychiatric illnesses is codified in various ways, for instance by a system called *ICD-10* sponsored by the World Health Organization, and by *DSM-IV* [21] sponsored by the American Psychiatric Association. Figure 11 provides an example of one psychiatric diagnosis, Dementia of the Alzheimer's Type and is taken verbatim from [21], pg. 142-143. A patient can be diagnosed with Alzheimer's Dementia if conditions 1-6 are met. These conditions may in turn depend on other conditions: for instance, condition 1 depends on both 1(a) and 1(b) being true. Conditions 1a and 1(b)i-1(b)iv depend on no other conditions and may be termed *leaf criteria.* Because of its logical structure,

[7] [25] provides an example where local evaluation of a query to a shortest-path program that resembles the program in Example 21 has better asymptotic complexity over non-local evaluation.

DSM-IV can be modeled as a propositional program in which various diagnoses or criteria can depend either positively or negatively on other diagnoses or criteria. One example of a negative dependence is criterion 6, which is called an *exclusion criterion.*

To see the use of non-stratified negation In codifying psychiatric diagnoses, consider that the exclusion criterion 6 indicates that a diagnosis of Alzheimer's Dementia should be made only if all other diagnoses can be ruled out. Alzheimer's Dementia was thus specified by the committee on cognitive disorders as a sort of "catch-all" to be used when other specific cognitive disorders cannot be proven. However, other classes of disorders, perhaps created by different committees, also have "catch-all" diagnoses — for instance the diagnosis of Adjustment Disorder ([21], pg. 626) contains an exclusion criterion almost identical to criterion 6 of Alzheimer's Dementia. If, as far as a physician can tell, a patient fulfills all criteria for Alzheimer's Dementia besides criterion 6, as well as all criteria for Adjustment Disorder (besides the criterion that the disturbance is not better accounted for by another disorder), the physician will essentially be faced with the situation:

The patient has an Adjustment Disorder if he does not have Alzheimer's Dementia, and has Alzheimer's Dementia if the patient does not have an Adjustment Disorder.

Modeling DSM-IV thus requires the ability to handle non-stratified negation, either through the well-founded semantics or other non-monotonic formalisms, such as stable models.

The representation of psychiatric knowledge must also distinguish between default and explicit knowledge. For instance, if a patient arrived at an emergency room presenting symptoms consistent with Schizophrenia, many physicians would not order a brain scan even though the symptoms could also indicate certain brain tumors. Usually a patient would be given a brain scan only if s/he did not respond to medication. Thus, a physician may distinguish between the initial default assumption that the patient did not have a brain tumor and the explicit (positive or negative) knowledge of a brain tumor based on test results. Default assumptions are revisable — if the patient does not improve the assumptions are re-examined through further testing or observation — but explicit knowledge is not.

These two features of psychiatric reasoning can be represented using $WFSX_P$ as presented in Section 3.1. A given leaf criterion may be taken to be true; explicitly false if a patient tests negatively; or default false if a patient does not exhibit a symptom. In this way, determining whether a diagnosis is satisfied or excluded, given knowledge of a particular set of criteria can be seen as one of query processing over $WFSX_P$.

Such a model underlies Diagnostica 1.0, a diagnostic program for DSM-IV developed using XSB by Medicine Rules, Inc (`http://medicinerules.com`)

Diagnostic Criteria for Dementia of the Alzheimer's Type

1. The development of multiple cognitive defects manifested by both
   (a) memory impairment (impaired ability to learn new information or to recall previously learned information)
   (b) one (or more) of the following cognitive disturbances
      i. aphasia (language disturbance)
      ii. apraxia (impaired ability to carry out motor activities despite intact motor function)
      iii. agnosia (failure to recognize or identify objects despite intact sensory functions)
      iv. disturbance in executive functioning (i.e., planning, organizing, sequencing, abstracting)
2. The cognitive deficits in criteria A1 and A2 each cause significant impairment in social or occupational functioning and represent a significant decline from a previous level of functioning.
3. The course is characterized by gradual onset and continuing cognitive decline.
4. The cognitive deficits in Criteria A1 and A2 are not due to any of the following
   (a) other central nervous system conditions that cause progressive deficits in memory and cognition (e.g. cerebrovascular disease, Parkinson's disease, Huntington's disease, subdural hematoma, normal-pressure hydrocephalus, brain tumor)
   (b) systemic conditions that are known to cause dementia (g. hypothyroidism, vitamin B_{12} or folic acid deficiency, niacin deficiency, hypercalcemia, neurosyphilis, HIV infection
   (c) substance-induced conditions
5. The deficits do not occur exclusively during the course of a delirium
6. The disturbance is not better accounted for by another Axis I disorder (e.g., Major Depressive Disorder, Schizophrenia)

Figure 11. DSM-IV Diagnostic Criteria for Adjustment Disorder.

which is currently under beta release. Users see a tree-like interface to DSM-IV, and assert or exclude leaf criteria. As they do so, they may check whether various diagnoses have become satisfied or excluded, by making calls to satisfaction routines, implemented in XSB. At an architectural level, DSM-IV is designed to run on a personal computer, and uses Microsoft Visual Basic as its interface. Visual Basic communicates with Microsoft Access as a permanent store, and, through a DLL interface, with XSB.

It should be mentioned, however, that adequate representation of the reasoning of psychiatric diagnosis is under investigation, but does requires a more sophisticated logic than $WFSX_P$. Work is currently underway to implement abduction over $WFSX_P$ [4] in Diagnostica so that abduced literals can represent information about which the physician is uncertain, but is unwilling to take as default false. Work is also underway to extend the model of reasoning in Diagnostica to include reasoning about the occurrence of symptoms over time and the ability to represent the strength of symptoms.

4.2. Using Preference Logic for Standardization

Surprisingly, non-monotonic reasoning also can be useful in the commercially significant problem of extracting information from poorly structured textual data. As an example of such data, a customer name may be entered into a database with or without a middle name, with or without a title, or with any number of other variations. Other textual data elements, such as addresses, suffer from similar problems. Clearly, there is a cost to poorly structured data: in the case of customer names and addresses, customers may be contacted several times, or profiling of customers may be affected. The process of extracting pertinent information, such as names or addresses, from poorly structured fields in databases and then transforming this information into a standard form was termed *data standardization* in [54]. As an example of name and address standardization, a corporate database might contain the following (misspelled) textual data within a single textual field:

```
TO THE ORDR OF XSB INC. STE 113 NASSAU HALL SO. LOOP RD STONYBROOK NY
```

For many purposes, the data should in fact contain the relevant pieces of information in different fields: i.e. the company name in one field, the city in another field. Furthermore, information within each field should be in a canonical form. After standardization, then, the above data might have the following form:

```
Entity:
        Name:  XSB
        Title:  INC
Address:
        Room:  113
        Building:  NASSAU HALL
        Street:  S LOOP RD
        Town:  STONY BROOK
```

```
Town:  NY
Zip:  11794
```

Prolog is a highly suitable implementation language for standardization. Flexible parsers are easy to create using Definite Clause Grammars (DCGs), and Prolog can also easily implement other standardizer rules, such as the rule to change `AVENUE` to `AVE` in the example above. Indeed, the use of Prolog to standardize data for several different organizations has been documented in [54, 45].

These Prolog standardizers have been commercially successful, at least on a small scale, but this success has required the standardizers to handle a large number of different forms of data in several different languages. As a result the standardizers are quite large: the Prolog standardizer for names and addresses reported in [54,45] has about 100,000 lines of code. While most of this code consists of a knowledge base of facts along with automatically generated parse tables, parts of the Prolog standardizers are complicated enough that they are difficult to maintain. In particular the name and address standardizer consists of a LL-parser written using Prolog DCGs that tentatively parses organization or personal names, street addresses, and so on. Like most LL-parsers written using Prolog, the parser used in the name address standardizer has a limited look-ahead, and does not use global information in constructing its parse. Because of this, the parsing phase of standardization is followed by a post-processing phase that consists of rules to fix common mistakes in the LL-parser. For instance if the city name was mistakenly included as part of a street address, the post-processor would move the city name from the street address field to the city field, and perhaps clean the street address field.

Recently, the name-address parser was rewritten both to use tabling in the parse and to use a form of Preference Logic [29,13] to simplify the post-processing step. As described in [15], the use of tabled DCGs allow the LL-parser to be replaced by a simpler LR-parser. Just as importantly, Preference Logic allows a declarative way to disambiguate ambiguous parses.

From the point of view of standardization, Preference Logic Programs can be understood as an extension of normal programs that allow declarations that specify that certain answers are preferred to other answers. Preference Logic thus bears some resemblance to GAPs and can be implemented using a form of tabled aggregation. For instance, a rule that only returns preferred addresses can be written as follows:

```
preferred_address(Address):-
        bagPO(scan_address,Address,prefer_address).
```

`bagPO/3` was introduced in Section 2.3 and performs a type of tabled aggregation similar to the routine `maintain_lub/4` of Section 3.2.3. The predicate *bagPO(scan_address,Addr,prefer_address)* succeeds only if `scan_address(Addr)` is true, and if there is no *Addr1* different from *Addr* such that user-written rule *prefer_address(Addr1,Addr)* is true [8]. Simply put, by using the rules *preferred_address/1* and *prefer_address/2* the parser can generate all parses and return only the preferred ones. The inefficiency of this approach can be surmounted by adding preference rules that return only preferred sub-parses, thus performing pruning. In this example, preference rules may be added to process city-state-zip-code tuples, or preferred street addresses.

The goal of standardization is to correctly classify elements in a textual string and to translate each of these elements into a standardized form. The "correctness" of a standardizer is gauged by how well it performs that task compared to a knowledgeable human being. Because large commercial databases typically contain a percentage of data of very low quality it is rare to be able to standardize data correctly in all cases. Realistically, if by manual checking of random data, a standardizer is found to standardize above 90% of data records correctly, it is usually deemed successful. As reported in [15], the use of tabled DCGs and Preference Logic does not significantly affect the percentage of data records correctly standardized, but it greatly reduces the code size for the parsing and post-processing steps of the name and address standardizer. However, the standardizer based on LR-parsing and Preference Logic is currently about twice as slow as the Prolog standardizer. For most commercial purposes, though, the savings accrued from the ease of maintenance of smaller and more declarative routines is expected to outweigh the reduction in speed.

4.3. The XMC Model Checker

The XMC model checker, implemented using XSB, relies on non-stratified negation to verify properties of concurrent systems that contain a finite number of states. XMC has been described in the literature [43,14], so that only an overview, focusing on its non-monotonic aspects, is provided here. Typically model checkers consist of a process calculus to represent states of processes and their communication methods, along with a temporal logic to reason about sequences of possible system states, given an initial state description. As pointed out in [43], finite-state model checking can be seen as a problem in computing fixed points of formulas over process states. For instance, determining whether a particular formula ϕ (perhaps an error condition) is true for a given state reachable from the initial state can be decided by a least fixed point operation that determines the transitive closure of states reachable from the initial state, and checks whether ϕ is true on any of these states. Similarly, determining whether a formula (say freedom from deadlock) is true on all states reachable from an initial state can be modeled using a greatest fixed point operation.

[8] `bagPO/3` ensures that a local-style evaluation (Section 2.2) is performed when finding preferred answers.

XMC uses as its temporal logic the *modal-mu calculus* into which most other temporal logics used for verification are themselves embeddable (see e.g., [23,11]). Briefly, the modal-mu calculus includes the usual connectives of propositional logic formulas along with the labeled modal constructors $< a >$ and $[a]$. In addition, there is an explicit least fixed point operator, μ, and greatest fixed point operator ν defined so that for a given formula ϕ, $\nu\phi = \neg\mu\neg\phi$. This equality is used by XMC to compute the greatest fixed point over finite structures. If a formula $\nu\phi$ is to be obtained, a positive formula ϕ' is constructed such that ϕ' is true if and only if ϕ is false. The least fixed point of ϕ' is then computed, and its negation taken using XSB's `tnot/1` predicate.

In this manner the modal-mu calculus can be modeled as a normal program, as can most process calculi. In [43] it was shown that, on finite transition structures, a restriction of the modal-mu calculus to *alternation-free* formulas [23] [9] can be evaluated without encountering loops through negation when represented by a normal program. In fact, such programs are left-to-right dynamically stratified (Section 2.2). While alternation-free formulas are a powerful class for verification, there exist important properties, such as fairness, that cannot be expressed using this restriction. In [38] it is shown that when unrestricted modal-mu calculus formulas are evaluated using normal programs they give rise to non-stratified negation. Recently, the XMC model checker has been extended to unrestricted modal-mu calculus formulas. The XSB engine first produces a residual program which, as noted in Section 2.1 consists of rules whose bodies contain only those literals undefined under the well-founded semantics. These undefined literals depend on loops through negation that contain an even number of negative literals (intuitively, this is because of the double negation in defining ν as $\neg\mu\neg$). As a result, any partial stable model of the relevant program can be extended to a stable model of the full program. This fact is utilized by sending the residual program to a stable model generator (Smodels [41]) to finish evaluation of the query by breaking the negative dependencies.

The process of developing XMC was similar to the process of constructing a meta-interpreter described in Section 3, and it is illustrative to trace the steps in its development. First came the discovery of an embedding of the modal-mu calculus into Datalog with negation [51]. Next, an interpreter was written for the modal-mu calculus and for a process calculus that resembles CCS [40]. Finally, the interpreter was tested on benchmark programs and its performance improved using specialization and other optimizations. This method has led to a highly declarative model checking engine whose core consists of about 200 lines of code. Despite its simplicity the performance of the interpreter is comparable with that of four leading model checkers on a variety of standard benchmark programs as

[9] Simply put, a modal-mu calculus formula is alternation-free if it is decomposable into subformulas each of which can be evaluated using a sequence of μ operators *or* a sequence of ν operators.

well as on practical programs such as the GNU UUCP protocol. For alternation-fee formulas XMC is competitive in terms of time and space with the other model checkers. For general formulas XMC usually outperforms the only other model checker that computes general modal-mu calculus formulas.

Several points can be drawn from the experience with XMC that relate both to its performance and to the use of NMR techniques in practical applications.

The importance of flexible forms of negation The delay minimality of XSB's search strategy avoids delay and simplification for alternation-free modal-mu calculus formulas, and is one factor leading to the good performance of XMC. The performance of general modal-mu calculus formulas depends on efficient construction of the residual program, as well as on efficient derivation of the preferred stable model by Smodels. Such derivation also highlights the benefits that can be gained by good interfaces between NMR systems.

The importance of goal orientation In the terminology of the model checking community, XMC is a *local* model checker, which means that it need not necessarily traverse the entire transition graph to determine an answer. From a logic programming point of view, a local model checker is simply a goal-oriented model checker. For alternation-free model checking, XMC's goal-orientation is a direct result of XSB's implementation of SLG. For general queries, the goal-orientation of XSB is again important since the residual program that is sent to Smodels consists only of those clauses that are relevant to the initial model checking query (as mentioned above, because all loos through negation are even, it is unnecessary to compute the stable model for the entire program in this case, see [38] for details).

The ability to combine tabled and non-tabled evaluations The XMC interpreter is optimized using a technique called *Clause Resolution Automata* [17] which allows particular clauses to be tabled rather than predicates as a whole. Using this technique, tabling need only be used to avoid infinite loops, to provide well-founded negation, and to avoid recomputation. By this use of selective tabling, the amount of space required by XMC is greatly reduced as is the time needed to evaluate queries.

Reliance on an efficient implementation of tabling The performance of XMC to the model checking community has been surprising partly because XMC does not store dynamic information about states in a data structure called *Binary Decision Diagrams (BDDs)*, which have been thought to be necessary for efficient finite-state model checkers. Instead, XMC stores dynamic information in tables as tries. In [44] it was shown that the use of tries allows a great deal of space sharing for model checking state information. Furthermore, the general memory efficiency of batched evaluation (as shown in [26]) also appears to be a factor in the good space performance of XMC.

5. Discussion

This paper has shown how different non-monotonic approaches can be implemented using tabled logic programs. In the case of WFSX and several other formalisms, there exists an embedding, of polynomial complexity, into normal programs under the well-founded semantics. When such an embedding is known, a meta-interpreter can be constructed directly using XSB and its library predicates. Even when an embedding into normal programs under the well-founded semantics is not known to exist, as in the case of GAPs and Preference Logic, the use of tabled aggregation may allow such programs to be evaluated using tabling. In principle, implementing a meta-interpreter using tabled aggregation may require using low-level XSB built-ins. However, in the case of Preference Logic and the GAP meta-interpreter of Section 3.2, XSB library predicates suffice for implementation. As Section 4 demonstrates, this methodology can lead to sophisticated interpreters that are highly relevant to research and commercial applications. The WFSX interpreter is used to for a psychiatric diagnosis system, while a Preference Logic extension to parsing is used to standardize poorly structured textual data. Furthermore, in conjunction with stable model generators, interpreters for the modal-mu calculus and for process calculi have been used to construct highly efficient model-checkers. These experiences show that non-monotonic extensions, if amenable to efficient implementation, can be invaluable in developing novel applications of logic programming.

Acknowledgements

The author would like to thank Luís M. Pereira, David S. Warren, and V.S. Subrahmanian along with the anonymous reviewers their comments on this paper. This research was supported in part by NSF grant CCR-9702681. Of the research surveyed in this article, the inclusion of WFSX in XSB was supported in part by NSF grant CCR-9404921, and the model checking application in part by NSF grant ESS-9705998. Finally, the XSB system has been implemented by a number of developers in SUNY Stony Brook, Universidade Nova de Lisboa and Katolieke Universiteit Leuven. Credits for XSB can be found in the XSB manual at `http://www.cs.sunysb.edu/~sbprolog`.

References

[1] S. Adah and V. S. Subrahmanian. Amalgamating knowledge bases, II: Algorithms, data structures and query processing. *Journal of Logic Programming*, 1996.

[2] J. Alferes, C. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14(1):93–147, 1995.

[3] J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*, volume 1111 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin Germany, 1996.

[4] J. Alferes, L. M. Pereira, and T. Swift. Well-founded abduction via tabling dual programs. Technical report, SUNY Stony Brook, 1999. In Preparation.

[5] G. Antoniou, D. Billington, and M. Maher. Normal forms for defeasible logic. In *International Conference on Logic Programming*. MIT Press, 1998.

[6] R. Ben-Eliahu and R. Dechter. Propositional semantics for disjunctive logic programs. In *Joint International Conference/Symposium on Logic Programming*, pages 813–127. MIT Press, 1992.

[7] H. Blair and V. S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[8] R. Bol and L. Degerstedt. Tabulated resolution for well-founded semantics. *Journal of Logic Programming*, 38(1):31–55, February 1998.

[9] D. Boulanger. Fine-grained goal-directed declarative analysis of logic programs. *Proceedings of the International Workshop on Verification, Model Checking and Abstract Interpretation*, 1997. Available through http://www.dsi.unive.it/˜bossi/VMCAI.html.

[10] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[11] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, pages 36–73, 1993.

[12] M. Codish, B. Demoen, and K. Sagonas. Semantics-Based Program Analysis for Logic-Based Languages using XSB. *Springer International Journal of Software Tools for Technology Transfer*, 2(1):29–45, Nov. 1998.

[13] C. Crowner, K. Govindarajan, B. Jayaraman, and S. Mantha. Preference logic grammars. *Computer Languages*, 1999. To apear.

[14] B. Cui, Y. Dong, X. Du, K. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. Smolka, and D. S. Warren. Logic programming and model checking. In *PLILP 98*, pages 1–20. Springer-Verlag, 1998.

[15] B. Cui, T. Swift, and D. S. Warren. Using tabled logic programs and preference logic for data standardization. Available at http://www.cs.sunysb.edu/˜tswift, 1998.

[16] C. Damásio. *Paraconsistent Extended Logic Programming with Constraints*. PhD thesis, Univ. Nova de Lisboa, 1996.

[17] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and T. Swift. Optimizing clause resolution: Beyond unification factoring. In *International Logic Programming Symposium*, pages 194–208. MIT Press, 1995.

[18] 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, May 1996.

[19] A. Dekhtyar and V. S. Subrahmanian. Hybrid probabilistic programs. In *International Conference on Logic Programming*, pages 391–407. MIT Press, 1997.

[20] K. Devlin. *Fundamentals of Contemporary Set Theory*. Springer-Verlag, Berlin Germany, 1980.

[21] *Diagnostic and Statistical Manual of Mental Disorders*. American Psychiatric Association, Washington,DC, 4th edition, 1994. Prepared by the Task Force on DSM-IV and other committees and work groups of the American Psychiatric Association.

[22] Diagnostica. Available from Medicine Rules Inc: http://medicinerules.com.

[23] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Conference on Logic in Computer Science*, pages 267–278. IEEE, 1986.

[24] M. Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programming*, 11:91–116, 1991.

[25] J. Freire, T. Swift, and D. S. Warren. Treating I/O seriously: Resolution reconsidered for disk. In *International Conference on Logic Programming*, pages 198–212, 1997.

[26] J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*, 1998(3), 1998.

[27] M. Gelfond and V. Lifshitz. The stable model semantics for logic programming. In *Joint International Conference/Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[28] M. Gelfond and V. Lifshitz. Logic programs with classical negation. In *International Conference on Logic Programming*, pages 579–597. MIT Press, 1990.

[29] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference logic programming. In *International Conference on Logic Programming*, pages 731–746. MIT Press, 1995.

[30] ISO working group JTC1/SC22. Prolog international standard. Technical report, International Standards Organization, 1995.

[31] C. Jonker. *Constraints and Negations in Logic Programs.* PhD thesis, Utrecht University, 1994.

[32] M. Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. Logic Programming*, 12(4):335–368, 1992.

[33] E. Lamma, F. Riguzzi, and L. M. Pereira. Strategies in learning with extended logic programs. Technical report, Universita di Bologna, 1998.

[34] R. Larson, D. S. Warren, J. Freire, and K. Sagonas. *Syntactica, Symantica.* MIT Press, Cambridge Ma., 1995.

[35] S. Leach and J. Lu. Computing annotated logic programs. In *International Conference on Logic Programming*, pages 257–271. MIT Press, 1994.

[36] J. Leite and L. M. Pereira. Iterated logic programming updates. In *International Conference on Logic Programming*, pages 265–278. MIT Press, 1998.

[37] R. Li and L. M. Pereira. Representing and reasoning about actions with abductive logic programming. *Annals of Mathematics and Artificial Intelligence*, 21:245–303, 1997.

[38] X. Liu, C. R. Ramakrishnan, and S. Smolka. Fully local and efficient evaluation of alternating fixed points. In *TACAS 98: Tools and Algorithms for Construction and Analysis of Systems*, pages 5–19. Springer-Verlag, 1998.

[39] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin Germany, 1984.

[40] R. Milner. *Communication and Concurrency.* Prentice-Hall, New York, 1989.

[41] I. Niemelä and P. Simons. Smodels — An implementation of the stable model and well-founded semantics for normal LP. In *International Workshop on Logic Programming and Non-Monotonic Reasoning*, pages 420–429. Springer-Verlag, 1997.

[42] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM Principles of Database Systems*, pages 11–21. ACM Press, 1989.

[43] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings on the Conference on Automated Verification*, pages 143–154, 1997.

[44] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–55, January 1999.

[45] I. V. Ramakrishnan, A. Roychoudhury, and T. Swift. A standardization tool for data warehousing. In *Practical Applications of Prolog*, 1997.

[46] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM TOPLAS*, 20(3):586 – 635, May 1998.

[47] K. Sagonas, T. Swift, and D. S. Warren. An abstract machine for computing the well-

founded semantics. Extended version of article in Joint International Conference and Symposium on Logic Programming, 1996, pp. 274-289, MIT Press.
[48] K. Sagonas, T. Swift, and D. S. Warren. The limits of fixed-order computation. *Theoretical Computer Science*, 1999. To apperar. Preliminary version apeared in International Workshop on Logic and Databases, LNCS.
[49] K. Sagonas, T. Swift, D. S. Warren, J. Freire, and P. Rao. *The XSB Programmer's Manual: version 1.9*, 1998.
[50] M. Schroeder. *Autonomous, Model-based Diagnosis Agents*. Kluwer Academic Press, 1998.
[51] S. Smolka, O. Sokolsky, and S. Zhang. Model checking in the modal μ-calculus. In *IEEE Symposium on Logic in Computer Science*, 1994.
[52] V. S. Subrahmanian. Amalgamating knowledge bases. *ACM Transactions on Database Systems*, 19(2):291–331, 1994.
[53] T. Swift. A new formulation of tabled resolution with delay. Submitted for Publication. Available at http://www.cs.sunysb.edu/˜tswift.
[54] T. Swift, C. Henderson, R. Holberger, J. Murphey, and E. Neham. CCTIS: an expert transaction processing system. In *Sixth Conference on Industrial Applications of Artificial Intelligence*, pages 131–140, 1994.
[55] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. Springer-Verlag, 1986.
[56] M. van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 4:37–53, 1986.
[57] A. van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
[58] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.
[59] G. Wagner. Reasoning with inconsistency in extended deductive databases. In *International Workshop on Logic Programming and Non-Monotonic Reasoning*, pages 300–315. MIT Press, 1994.

Appendix

A. Supplimentary Material for Section 3.2

Lemma A.1. Let $\mathcal{A}$ be a set of annotated atoms over a language $\mathcal{L}$ and complete lattice $\mathcal{T}$ of truth values, and I a minimal model of $\mathcal{A}$. Then, for any ground annotated atom $A : \mu$:

- $I \models A : \mu$ iff A is in the ground instantiation of A', where $A' : \mu$ is an atomic reductant of $\mathcal{A}$.

Proof: Assume that $I \models A : \mu$. Let S the set of annotated atoms $A_i : \nu_i$ such that A is in the ground instantiation of each A_i. S must be non-empty, otherwise I would not be a minimal model of $\mathcal{A}$. Furthermore $\mu = \sqcup \nu_i$. To see this, note that $\sqcup \nu_i$ must exist since $\mathcal{T}$ is a complete lattice. In addition, $\mu \geq \sqcup \nu_i$, since I is a model for each of the elemets of $\mathcal{A}$; while if $\mu > \sqcup \nu_i$, I is not minimal. Thus A is in the ground instantiation of $A' : \mu$ where $A' : \mu$ is an atomic reductant of $\mathcal{A}$.

Next, suppose $A' : \mu$ is a reductant of $\mathcal{A}$, and suppose that $A : \mu$ is in the ground instantiation of $A' : \mu'$. Let $Set \subseteq \mathcal{A}$ consist of those atoms in $\mathcal{A}$ that was used to produce $A' : \mu'$, and $A_i : \mu_i$ be the atoms in Set. Then $I \models A' : \mu_i$, and by Definition 23, μ' is the least element of $\mathcal{T}$ such that $\mu_i \leq \mu'$ for all $i \in Set$. □

Theorem 29. Let P be a GAP over a language $\mathcal{L}$ and complete lattice $\mathcal{T}$, and $\mathcal{F}_n$ the final forest of a non-floundering SLG$_{GAP}$ evaluation of a query $Q : \mu$ to P. Then $lfp(R_P) \models Q' : \mu'$ iff $Q' : \mu'$ is in the ground instantiation of an atom reductant of a set of answers in the tree for $Q : \mu$ in $\mathcal{F}_n$.

Proof: (Sketch) The forward implication can be shown by induction on the forests of $\mathcal{F}_n$. Because SLG is sound for definite programs, only the operation GAP Answer Return need be considered. From Lemma A.1 it can be seen that this operation is sound, also.

For the backward implication, induction must be done on iterations of the R^P operator. Again, because SLG is complete for definite programs, only the operation GAP Answer Return need be considered. From Lemma A.1 it can be seen that this operation is complete also. □

Lemma A.2. Let $B_1 : \mu_1$ and $B_2 : \mu_2$ be annotated atoms over a language $\mathcal{L}$ and complete lattice $\mathcal{T}$ such that μ_1 is either a variable annotation or a ground annotation, and μ_2 is either a variable annotation or a ground annotation. Let I_1 be an interpreation formed by taking all ground instances of $B_1 : \mu_1$ over $\mathcal{L}$ and $\mathcal{T}$. Then $I_1 \models (B_2 : \mu_2)\theta$ iff $B_1 : \mu_1$ and $B_2 : \mu_2$ are unifiable with substitution θ and $eval(\mu_1, \mu_2)\theta$ succeeds.

Proof: We sketch the forward implication. If μ_1 and μ_2 are constant, the proof is straightforward. If μ_2 is a variable, then $\mu_2\theta$ will either be a variable or a constant. If $\mu_2\theta$ is a variable, then, by Definition 22, $I_1 \models B_2 : \top$. μ_1 will then either be $\top$ or a variable, and in either case $eval(\mu_1, \mu_2)$ will succeed. If $\mu_2\theta$ is a constant, then the case is straightforward. □