XSB: Extending Prolog with Tabled Logic Programming

TERRANCE SWIFT
CENTRIA, Faculdade de Ciências e Tecnologia
Univ. Nova de Lisboa, 2825-516 Caparica, Portugal
(*e-mail:* `tswift@cs.sunysb.edu`)

DAVID S. WARREN
Computer Science Department, SUNY Stony Brook
Stony Brook, New York, USA
(*e-mail:* `warren@cs.sunysb.edu`)

submitted 4th October 2009; revised 28th February 2010; accepted 22nd November 2010

Abstract

The paradigm of Tabled Logic Programming (TLP) is now supported by a number of Prolog systems, including XSB, YAP Prolog, B-Prolog, Mercury, ALS, and Ciao. The reasons for this are partly theoretical: tabling ensures termination and optimal known complexity for queries to a large class of programs. However the overriding reasons are practical. TLP allows sophisticated programs to be written concisely and efficiently, especially when mechanisms such as tabled negation and call and answer subsumption are supported. As a result TLP has now been used in a variety of applications from program analysis to querying over the semantic web. This paper provides a survey of TLP and its applications as implemented in XSB Prolog, along with discussion of how XSB supports tabling with dynamically changing code, and in a multi-threaded environment [1].

KEYWORDS: Prolog, Tabling, Implementation, Non-monotonic Reasoning

1 Introduction

Since its inception, a primary goal of XSB has been to expand the areas in which Prolog is used, by making Prolog more powerful, more efficient, and more declarative. In 1993 when XSB was first released, it supported this goal by including both tabled resolution for definite programs, which provided it with deductive database-style features of such systems as Coral (Ramakrishnan et al. 1992) and LDL (Chimenti et al. 1990). At the time, while XSB was faster than those systems, it was basically suitable only for research by its developers. Since then, XSB has become a widely used multi-threaded Prolog that is compliant with most standards. During this development, XSB's research focus has continued to be centered on tabling.

[1] To appear in Theory and Practice of Logic Programming (TPLP)

At one level, the idea behind tabling is simple; subgoals encountered in a query evaluation are maintained in a table, along with answers to these subgoals. If a subgoal is re-encountered, the evaluation reuses information from the table rather than re-performing resolution against program clauses. For instance using tabling, a Prolog predicate for transitive closure over a graph:

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

could just as easily be written as

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

(and, as discussed below, there are good reasons for performing this rewrite).

This simple idea has profound consequences. First, tabling ensures termination of programs with the *bounded term-size property* – those programs where the sizes of subgoals and answers produced during an evaluation are less than some fixed number. This makes it much easier to reason about termination than in basic Prolog. Second, tabling can be extended to evaluate programs with negation according to the Well-Founded Semantics (WFS) (van Gelder et al. 1991). Third, for queries to wide classes of programs, such as datalog programs with negation, tabling (perhaps combined with compiler transformations) can achieve the optimal complexity for query evaluation. And finally, tabling integrates closely with Prolog, so that Prolog's familiar programming environment can be used, and no other language is required to build complete systems.

These properties have led to the emerging paradigm of Tabled Logic Programming (TLP). The properties of termination and optimal complexity have made TLP useful to explore state spaces in applications from program analysis to process and temporal logics. The termination properties together with WFS have supported a variety of extensions for non-monotonic constructs such as annotations, preferences, explicit negation, and abduction; and have led to the integration of Prolog with Answer Set Programming (ASP) through XSB's XASP package. Together these properties have fostered combinations of TLP with more declarative and less procedural knowledge representation approaches.

This paper discusses how XSB supports TLP, and how TLP supports applications when combined with other features of Prolog such as dynamic code and constraints. Section 2 describes the tabling features of XSB (including reclaiming table space) through a series of examples including tabling for definite programs, tabled negation, call and answer subsumption, and tabled constraints. Section 3 then describes XSB's approach to dynamic code, including its integration with TLP via incremental tabling. Section 4 discusses XSB's multi-threading. Finally Section 5 discusses two applications that we consider particularly innovative and significant.

However before proceeding, we briefly consider XSB purely as a Prolog system. Each version of XSB runs on Linux, Mac OS and Windows (compiled with either MSVC or Cygwin); for Linux and Mac OS 64-bit compilation is supported in addition to 32-bit. With a few exceptions, XSB supports the core Prolog standard

(ISO-IEC 13211-1), the core revision working draft (ISO/IEC DTR 13211-1:2006) and, as discussed in Section 4, the multi-threading working draft (ISO/IEC DTR 13211-5:2007). XSB also supports constraint logic programming through attributed variables, the interface to which is nearly identical to that of SWI and YAP Prolog. As a result, constraint libraries are ported to XSB in a routine manner, and XSB supports Constraint Handling Rules, CLP(R) and CLP(FD).

2 Tabling By Example

We use a series of examples to introduce various aspects of programming with tabling. XSB's tabling is based on SLG resolution (Chen and Warren 1996), with extensions for call and answer subsumption as described below. This presentation uses a forest-of-trees model (Swift 1999a) focusing on programming aspects and system features needed for tabling support. Accordingly, The presentation is informal; the full formalism of tabling and its algorithms can be found in the references.

2.1 Definite Programs

From a theoretical perspective SLD, the resolution method underlying Prolog, is complete in that there is an SLD proof for every correct answer for a query Q to a program P. However, the search for an SLD proof may not terminate, even when P is a datalog program. For example, consider `?- reach(1,Y)` to the program P_{Lrec}:

```
reach(X,Y):- reach(X,Z),edge(Z,Y).            edge(1,2).
reach(X,Y):- edge(X,Y).                       edge(2,3).
```

An SLD search tree for this query provides proofs for both of the correct answer substitutions, $Y = 2$ and $Y = 3$. However the SLD tree is infinite, and, when Prolog's search strategy is used, both answers lie after an infinite branch. I.e., Prolog will go into an infinite loop before deriving the first answer. Indeed, since the tree is infinite, no complete search will ever terminate.

Example 2.1 Figure 1 shows how the XSB declaration `:- table reach/1` affects the above program and query. A tabled evaluation is represented as an SLG forest in which each tabled subgoal S is represented by a unique tree with root S :- S, which represents resolutions of program clauses and answers to prove S. (The head term conveniently collects the bindings of subcomputations.) The reach predicate in P_{Lrec} is left-recursive and gives rise to a single tabled subgoal, `reach(1,Y)`, and correspondingly to a forest with a single tree. In Figure 1 each non-root node of the form $K.N$ where $N = (S$:- $Goals)\theta$ is a clause in which the bindings to a subgoal S are maintained in $S\theta$, the goals remaining to prove S are in $Goals\theta$, and the order of creation of N within the tabled evaluation is represented by a number, K. The tabled evaluation of Figure 1 at first resembles that of SLD: a program clause resolves against the root node to create node 1. However, rather than (fruitlessly) re-applying the program clause, the computation *suspends* node 1, since its selected literal has been seen before, and uses another program clause to create node 2. The selected literal for node 2, `edge(1,Y)`, has no table declaration so that it is resolved as with SLD and does not create a new tree. Program clause resolution thus creates the first answer in node 3: a node with an empty body represents an answer. The computation then resumes node 1 resolving the *answer* against the selected literal `reach(1,X)`, and continues to derive the second answer for the query and then to return the answer to node 1. At that point node 6 is created and fails; no more resolutions are applicable for `reach(1,Y)` and it is determined to be *completely evaluated*.

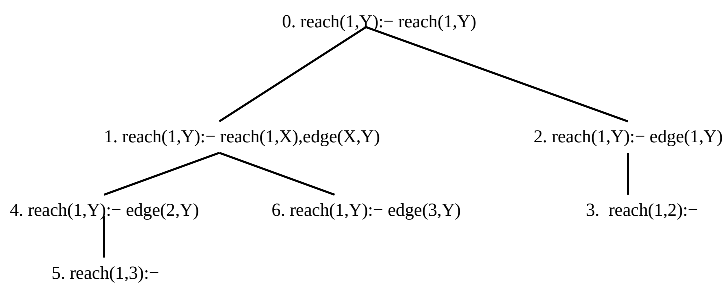

Fig. 1. Tabling tree for the query *reach(1,Y)* to P_{Lrec}

While simple, Example 2.1 illustrates several points. First, the evaluation keeps track of each tabled subgoal S that it encounters. Later if S is selected again, resolution will use answers rather than program clauses; if no answers are available, the computation will *suspend* at that point and the evaluation will backtrack to try to derive answers using some other computation path. Once more answers have been derived, the evaluation *resumes* the suspended computation. Similarly, once the computation has backtracked through all answers available for S in the current state, the computation path will suspend, and resume after further answers are found. Thus a tabled evaluation is a fixed point computation for a set of interdependent subgoals. The second point is that by keeping a table of subgoals and their answers, tabling can factor out redundant subcomputations – such as the repeated SLD resolution of the selected subgoal `reach(1,Y)`. And third, the evaluation mixes goals to tabled and non-tabled predicates; by default predicates use SLD resolution, and only use SLG if a `table/1` declaration has been made.

At the same time, because Example 2.1 has only a single tabled subgoal, it does not illustrate other important features of tabling. Consider for instance, the right recursive form of `reach/1` shown in the program P_{Rrec} in Figure 2, which also shows a tabled evaluation of the query `?- reach(1,Y)`. There are three separate trees in Figure 2. At an implementation level, a tabled subgoal together with its answers is maintained by a unique *table*, so XSB maintains three separate tables for this evaluation. Note that the tree for `reach(1,Y)` depends on `reach(2,Y)` in node 3, and on `reach(3,Y)` in node 11. Also, note that in Figure 2 the label *complete* is associated with the tree for `reach(2,Y)`. If a subgoal is completed, many of its computational resources can be reclaimed, as will be described in the next section. The notion of subgoal dependency can be made precise. In a given forest, a non-completed subgoal S_1 directly depends on a non-completed subgoal S_2

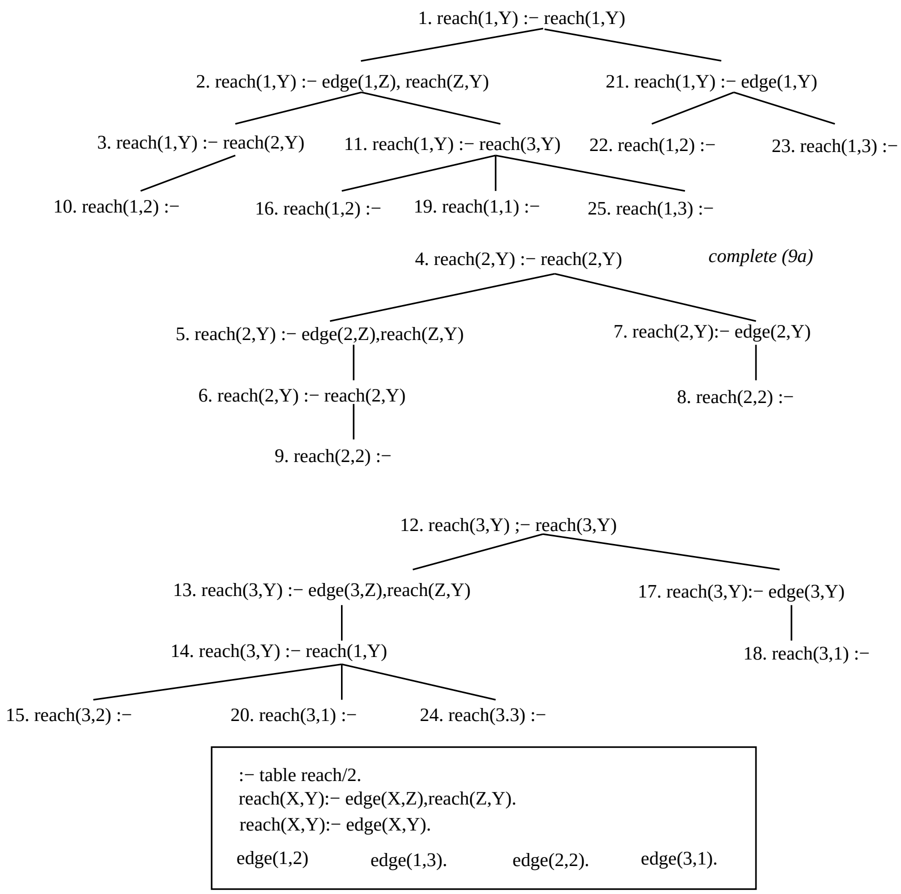

Fig. 2. A program P_{Rrec} and SLG forest for evaluation of `?- reach(1,Y)`

if S_2 is the selected atom of a node in the tree for S_1; the definition of dependency is just the transitive closure of direct dependency. The direct dependency relation for an SLG forest $\mathcal{F}$ gives rise to a *Subgoal Dependency Graph (SDG($\mathcal{F}$))*. Since the SDG is a directed graph, a (maximal) set of mutually dependent goals is a strongly connected component, or (maximal) SCC, and an independent SCC $\mathcal{S}$ is a maximal SCC such that no subgoal in $\mathcal{S}$ depends on any subgoal outside of $\mathcal{S}$ (cf. (Marques and Swift 2008)). Note that since the SDG depends on a forest, it changes as the forest changes, adding dependencies as new literals are selected and deleting them as subgoals are completed. In Figure 2, there is one independent SCC consisting of `reach(1,Y)` and `reach(3,Y)`; however in the earlier forest that consisted of nodes 1-8, `reach(2,Y)` is a trivial independent SCC. The importance of independent SCCs is that their subgoals can be efficiently determined to be completely evaluated and marked as *completed* before the tabled evaluation as a whole is finished.

Scheduling Strategies for Tabling As noted, tabled evaluation has new operations for creating a new tree, for resolving an answer against a tabled subgoal, and for completing a mutually dependent set of subgoals. The order in which these operations are applied within an evaluation is determined by a *scheduling strategy*. By default XSB uses the scheduling strategy of *Local evaluation*, which was introduced in (Freire et al. 1998) and formalized in (Marques and Swift 2008). The key idea behind Local evaluation is that all operations are performed only in a maximal independent SCC. An alternate scheduling strategy is *Batched evaluation*, whose key idea is to return an answer for a subgoal S to the first node that called S as soon as the answer is derived.

Local and Batched evaluation differ in that Batched evaluation eagerly returns answers while Local evaluation may not return any answers out of an SCC until that SCC is completely evaluated. In general Local evaluation uses less stack space and is more efficient for answer subsumption (Section 2.5). Batched evaluation may find first answers faster.

In addition to the decision of whether to used Batched or Local evaluation, we mention two other principles for programming efficiently using tabling. First, left recursion is usually faster for computing single-source reachability goals than other forms of recursion, such as right recursion, as left recursion creates only a single table, and requires fewer operations. Second, tabling should be used sparingly: for many predicates tabling will add no benefit although the table will take up space and time to accumulate it. In certain cases, tabling can actually increase the complexity of a query. For instance in XSB and all other tabled Prologs, the query: `?- append([a,b,c],[d,e,f],Z)` to the tabled version of the normal append predicate will be quadratic in the size of the query, as the goals `append([a,b,c],[d,e,f],Z)`, `append([b,c],[d,e,f],Z)`, etc. will be copied into the tables.

Example 2.2 (Analyzing a Process Logic) The analysis of process logics in the style of Petri Nets will illustrate various types of tabling evaluations. Reachability is a central problem for Petri Net analysis, to which problems such as liveness, deadlock-freedom, and the existence of home states can be reduced [2]. Elementary Petri Nets (EPNs) (cf. (Rozenberg and Engelfriet 1998)) are particularly simple to analyze using tabling. Consider the EPN of Figure 3, which depicts a simple producer-consumer system, with circles representing places and rectangles representing transitions. An EPN allows a place to contain at most 1 token; thus a

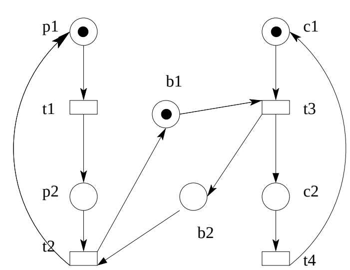

Fig. 3. A Simple Producer-Consumer Net

[2] All programs can be obtained via `http://xsb.cvs.sourceforge.net/xsb/mttests/benches`.

finite EPN has only a finite number of configurations so that determining reachability of an EPN is decidable. Our encoding represents the configuration of an EPN by an ordered list of its marked places: thus the configuration in Figure 3 is represented as the list `[b1,c1,p1]`. Next, a transition T is represented by a list of places with input arcs to T ($\bullet T$) and output arcs from T ($T\bullet$). Predicate `trans/3` in Figure 4 represents each transition of Figure 3 as a fact, using XSB's trie indexing (see Section 3) to obtain full indexing on list elements. Figure 4 also shows code for determining reachability in an EPN; for instance solutions to the goal `reachable([b1,c1,p1],X)` are configurations reachable from the EPN in Figure 3. For a transition T to have concession (be able to fire) in a configuration C of an EPN, every place in $\bullet T$ must be marked, and no place in $T\bullet$ can be marked. These conditions are checked by `hasTransition/2` in Figure 4 which finds sets of transitions that might have concession. The recursion (in `get_trans_for_conf_1/3`) allows indexed calls to transitions to be made based on each place in the input configuration. Each set of possible transitions is then filtered to include only those transitions that actually have concession using operations on ordered sets (via `check_concession/2`). `hasTransition/2` succeeds when the first of these transitions is applied; further transitions are applied upon backtracking.

The predicate `reachable/2` is a left-recursive reachability definition based on `hasTransition/2`. Tabling `reachable/2` is useful in two ways: it prevents looping when a given configuration is reachable from itself; and it filters out redundant paths to a reachable configuration. With the left recursive form of `reachable/2` a typical call, such as `reachable([b1,c1,p1],X)` with first argument bound and second free, requires a single tabled subgoal, and has as answers all configurations reachable from `[b1,c1,p1]`. XSB's use of tries to represent tabled subgoals and their answers allows efficient checking of answers and efficient use of memory, since the trie data structure factors out common list prefixes (cf. Section 2.3). Using this program, nets with millions of transitions can be fully traversed in under a minute.

2.2 Tabled Negation

The following example illustrates evaluation of WFS using tabled negation in XSB.

Example 2.3 Figure 5 shows the normal program P_{neg} where `tnot/1` is XSB's predicate for tabled negation. The atom `p(c)` is true in the well-founded model of P_{neg} and the schematic ground instantiation of P_{neg} in Figure 6 illustrates why this is so. First, `p(c)` is true because `p(a)` is false. All except 2 of the 8 ground instances of clauses for `p(a)` are false because their first literal, a call to `t/3` is false; the remaining two:

```
p(a) :- t(a,a,b), tnot p(a), tnot p(b).
p(a) :- t(a,b,a), tnot p(b), tnot p(a).
```

are false because `p(b)` is true, so that `tnot p(b)` is false.

However in a tabled evaluation of `p(a)` that uses Prolog's literal selection strategy, the literal `tnot p(a)` is selected while evaluating the clause

```
p(a) :- t(a,b,a), tnot p(a), tnot p(b).
```

```
:- table reachable/2.
reachable(InConf,NewConf):-
   reachable(InConf,Conf),
   hasTransition(Conf,NewConf).
reachable(InConf,NewConf):-
    hasTransition(InConf,NewConf).

hasTransition(Conf,NewConf):-
   get_trans_for_conf(Conf,AllTrans),
   member(Trans,AllTrans),
   apply_trans_to_conf(Trans,Conf,NewConf).

get_trans_for_conf(Conf,Flattrans):-
   get_trans_for_conf_1(Conf,Conf,Trans),
   flatten(Trans,Flattrans).

get_trans_for_conf_1([],_Conf,[]).
get_trans_for_conf_1([H|T],Conf,[Trans1|RT]):-
   findall(trans([H|In],Out,Tran),trans([H|In],Out,Tran),Trans),
   check_concession(Trans,Conf,Trans1),
   get_trans_for_conf_1(T,Conf,RT).

check_concession([],_,[]).
check_concession([trans(In,Out,Name)|T],Input,[trans(In,Out,Name)|T1]):-
   ord_subset(In,Input),
   ord_disjoint(Out,Input),!,
   check_concession(T,Input,T1).
check_concession([_Trans|T],Input,T1):-
   check_concession(T,Input,T1).

apply_trans_to_conf(trans(In,Out_Name),Conf,NewConf):-
   ord_subtract(Conf,In,Diff),
   flatten([Out|Diff],Temp),
   sort(Temp,NewConf).

% Prolog representation of the Producer-Consumer Net
:- dynamic trans/2.
:- index(trans/2,trie).
trans([p1],[p2],t1).          trans([b2,p2],[p1,b1],t2).
trans([b1,c1],[b2,c2],t3).    trans([c2],[c1],t4).
```

Fig. 4. TLP Program for Analyzing Reachability of Elementary Petri Nets

leading to a loop through negation. At this point, it might be tempting to try a different search strategy, but it turns out that no deterministic search strategy can evaluate WFS top-down without encountering loops through negation. The approach of SLG resolution is to *delay* the evaluation of a literal involved in such a loop and then to *simplify* that literal later if it is determined to be true or false.

Figure 7 illustrates SLG resolution for this query and program. Within the nodes of Figure 7, the new symbol `|` separates the unresolved goals to the right from the delayed goals to the left. In the evaluation state where nodes 1 through 10 have been

```
:- table p/1.
p(b).
p(c) :- tnot p(a).
p(X) :- t(X,Y,Z), tnot p(Y), tnot p(Z).

t(a,a,b).      t(a,b,a).
```

Fig. 5. A program, P_{neg}

```
p(b).
p(c):- tnot p(a).
p(a) :- t(a,a,a), tnot p(a), tnot p(a).
p(a) :- t(a,a,b), tnot p(a), tnot p(b).
:
p(a) :- t(a,b,a), tnot p(b), tnot p(a).
:
p(b) :- t(b,b,b), tnot p(b), tnot p(b).

t(a,a,b).       t(a,b,a).
```

Fig. 6. The ground instantiation of P_{neg}

created, `p(b)` has been completed, and `p(a)` and `p(c)` are in the same SCC. There are no more clauses or answers to resolve, but `p(a)` is involved in a loop through negation in node 5, and nodes 2 and 10 involve `p(a)` and `p(c)` in a negative loop [3].

In situations such as this, where all resolution has been performed for nodes in an SCC and where there are multiple literals that can be delayed, an arbitrary one is chosen to be delayed first. So the evaluation delays the selected literal of node 2 to generate node 12 producing a *conditional answer* – an answer with a non-empty delay list. Next `tnot p(a)` in node 5 is delayed, failing that computation path, and `tnot p(c)` in node 10 is delayed to produce node 15 and failing the final computation path for `p(a)`. At this stage the computation of the SCC $\{p(a), p(c)\}$ is *completely evaluated* meaning that there are no more operations applicable for goal literals. Since `p(a)` is completely evaluated with no answers, conditional or otherwise, the evaluation determines it to be false and a *simplification* operation can be applied to the conditional answer of node 12, leading to the unconditional answer in node 17.

Example 2.3 illustrates several aspects of tabled negation in XSB. First, the SLG operations of delay and simplification are used to evaluate according to the WFS. These operations are implemented at the engine level, along with a mechanism for determining when tabled literals may be involved in a negative loop. Although delay and simplification do not affect the complexity of SLG, it is inefficient to perform them unnecessarily. XSB has been implemented so that it is *delay minimal* for a large class of programs (Sagonas et al. 2000). Second, the complexity of WFS

[3] In this example, we ignore the effects of early completion which would complete `p(b)` immediately upon creation of node 8, obviating the need to create node 9.

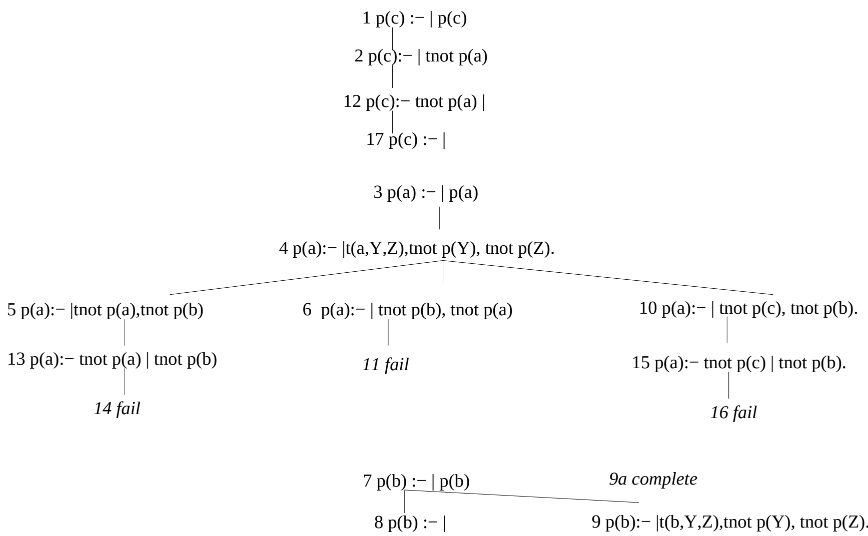

Fig. 7. SLG Evaluation of query `?- p(c)` to P_{neg}

is reflected in the cost of operations in the example. While a definite program P can be evaluated with abstract complexity $size(P)$, the size of P, normal programs under WFS have abstract complexity $atoms(P) \times size(P)$, the number of atoms in P times the size of P. This complexity is reflected in an evaluation when XSB checks to see whether loops of dependency are positive or negative: a check that in the worst case might need to be done once per subgoal in an SCC (Swift et al. 2009). Practically, these checks are usually performed only once or twice per SCC, even when the SCCs are large and non-stratified. The result is that WFS evaluation usually scales linearly with the size of a program: in fact it is difficult to construct an example that scales with complexity $atoms(P) \times size(P)$.

P_{neg} has a 2-valued well-founded model, but in WFS the truth value of atoms can be undefined. From our tabling perspective, this means that some conditional answers may have delayed literals that are never simplified away, such as those in the program $P_{nonstrat}$:

```
:- table p/1, q/1.
p(1) :- tnot q(1).       q(1):- tnot p(1).
```

The query `?- p(X)` will succeed writing `X = 1 undefined` beneath the command-line prompt. A call to `get_residual(p(X),D)` allows the conditional answer to be examined, returning `X = 1, D = [tnot(q(1))]`. This highlights another feature of SLG: it can be seen as a program transformation. Given a query Q to a program P, XSB traverses that part of P that is relevant to proving Q, and creates its reduction with respect to the well-founded model of P. The tables produced by query evaluation thus create a *residual program* that can be meta-interpreted using `get_residual/2` or sent to a stable models solver through XSB's XASP package.

XASP (XSB's Answer Set Programming package) provides several ways in which information can be sent from XSB to an ASP solver. Version 3.3 of XSB uses smodels (Simons et al. 2002), which is included in XSB's distribution, as its ASP solver. The simplest way to use XASP is to construct a partial stable model or 2-valued layered stable model (Pereira and Pinto 2009) from the residual program. For the preceding program, the query `?- pstable_model(q(1),Model,any)` will bind `Model` to each of the partial stable models for the residual program of `q(1)` – in this case first to `[p(1)]` and then to `[q(1)]`; `?- pstable_model(q(1),Model,restrict)` will restrict the models returned to those in which `q(1)` is true. Similarly the predicate `in_all_stable_models(Lit)` will succeed if `Lit` is true in all stable models, and there is at least one stable model. Using these routines, XSB can be used as a query-oriented ASP grounder with both advantages and disadvantages compared to grounders like `lparse` and `gringo`. On the one hand, cardinality or weight constraints, which are often used in ASP (Niemelä 1999), cannot be exploited if the residual program is sent directly to smodels (although XASP has commands to send such constraints separately to smodels). Additionally, XSB may not be as fast as a grounder like `gringo` if a fully grounded program is desired. On the other hand, XASP is superior for grounding programs that contain recursive data structures such as lists, for programs where variables are instantiated over large domains, and for programs and queries where only partial grounding is required.

2.3 Implementation Aspects

So far tabling has been presented almost entirely through the forest of trees model, which is sufficient for understanding many operational aspects. However, there are implementation aspects of XSB that are useful in order to mix tabling with full Prolog, to understand and control the space required by a tabled evaluation, and to write programs that efficiently use XSB's tabling subsystem.

Mixing Tabling and Prolog SLD and SLG evaluation can be intermixed arbitrarily only if a program does not contain side-effects or cuts. A programmer should take care when using a side effect for, say, I/O in a tabled predicate: such a side-effect will be executed only the first time a given subgoal is called, and not subsequently when the table is used. The behavior of cuts with tabled predicates requires explanation. Version 3.3 of XSB throws an exception if a computation attempts to cut over a choice point for an incomplete table: that is, a choice point that represents the root of an SLG tree or an internal node with a selected tabled literal. There is a semantic reason for this. Suppose a subgoal S is called in two different places in the computation, place 1 with a cut and place 2 without. If the cut for place 1 removed a choice point of the above type, it could prohibit the derivation of answers for place 2, and so give rise to incompleteness. On the other hand, XSB allows cuts over SLD choice points as well as cuts over choice points for completed tables.

Implementing a Mechanism to Suspend and Resume While details of XSB's tabling engine, the SLG-WAM (Sagonas and Swift 1998), are beyond the scope of this paper, we discuss a few aspects that are relevant to its practical use. First, the SLG-

WAM implements the ability to suspend and resume a computation by maintaining multiple computation states within its environment stack, heap, choice point stack and trail. Whenever a binding is made to a trailed variable, that binding is added to the trail frame itself, so that the SLG-WAM maintains a *forward trail*. Suspending and resuming are thus handled by backtracking to unbind variables, and using the forward trail to rebind the variables in a resumed path. At various points in a tabled evaluation, XSB freezes stack space so that the memory for suspended computation paths is retained; stack space is reclaimed upon completion of subgoals. Thus, at a general level, the forest of trees model maps to an implementation as follows: XSB associates each tabled subgoal with its answers in a table. Each non-completed tree, minus its answers, is maintained in XSB's stacks, and its space is reclaimed upon completion. Heap space is also reclaimed by XSB's heap garbage collector, which accounts for the multiple computation paths maintained by the frozen stacks (Demoen and Sagonas 2001; Castro and Costa 2001).

Differences in the mechanism for suspending and resuming form the main architectural differences in tabling engines. YAP Prolog also implements the SLG-WAM: its implementation is currently limited to definite programs, but YAP Prolog also makes some important optimizations to the SLG-WAM to improve speed (Rocha 2001). Ciao Prolog implements a different strategy, called CHAT (Demoen and Sagonas 1999), which suspends by copying (part of) a computation path from the WAM stacks to a separate area of memory, and resumes by copying the computation path back into the WAM stacks. CHAT can thus be thought of as an approach based on copying rather than one based on sharing as with the SLG-WAM (performance analysis in (Castro et al. 2002) found the sharing approach to be superior for many tabled programs). B-Prolog uses a still different approach, called linear tabling (Zhou and Sato 2003), which *rederives* a suspended computation path rather than saving the suspended path in trail frames or in a separate CHAT area.

Reclaiming Table Space Tables factor subcomputations at the price of taking up space, so that a practical system for tabled Prolog must provide a means to reclaim the space that tables use. XSB provides a number of predicates that abolish table space safely, and that support different modes of tabling. Perhaps *query-level tabling* is the most common mode, where tables ensure termination or a particular complexity for a user query. A second mode is *amortizing tabling* where tables reduce the cost of multiple top-level queries by tabling repeated subcomputations, even if these subcomputation are not repeated within the same query. A third mode is *user-controlled tabling* where an application that uses tabling heavily decides itself when a table is no longer needed and abolishes it, perhaps deeply within a top-level query. We offer a brief summary of the approaches taken to support these modes.

Query-level tabling is perhaps the simplest use to support: if the command-line interpreter (or a similar controller) calls the predicate `abolish_all_tables/0` at the end of a top-level query, all tables are abolished and their space immediately reclaimed. Furthermore, semantic problems with reclaiming space for incomplete tables are avoided. Nonetheless, XSB does not perform this by default, since there are many reasons for maintaining tables between user queries[4]. For amortizing tabling, the table space for certain predicates or groups of predicates often needs to be reclaimed at once, possibly at the command-line, while tables for other predicates should persist. To support this, XSB provides the predicate `abolish_table_pred/1`, which abolishes tables for a given predicate, and `abolish_table_module/1`, which abolishes tables for all predicates in a given module. User-level tabling sometimes requires a finer level of control, provided by the predicate `abolish_table_call/1` which abolishes a single table.

In order to ensure the safety of user-controlled tabling, the system must prevent the situation in which an evaluation abolishes a table, reclaims its space, and then backtracks to a state that accesses the released structures. To avoid this, XSB does not reclaim space for an abolished table until there are no choice points pointing into that table. Instead, a *table garbage collector* periodically reclaims space for abolished predicates and calls. Any call made before the abolish will be able to use answers from the abolished table.

The existence of residual programs further complicates space reclamation. In the program $P_{nonstrat}$ of Section 2.2, suppose that the table for `p(1)` were abolished but not that for `q(1)`. If the residual program were traversed for meta-interpretation or some other reason, the conditional answer `q(1):- p(1)|` would point into the abolished table for `p(1)`, To handle situations like this, when a table T is abolished, XSB ensures that other tables with conditional clauses that depend on T or its answers are also (transitively) abolished. Of course, some applications may use the well-founded semantics but have no desire to examine a residual program. These applications can set the Prolog flag `table_gc_action` to override this behavior, so that abolishing T does not cause other tables to be abolished.

Efficiently Accessing Tables A data structure for tabling needs to support three main operations: checking and/or inserting a new subgoal in a table, checking and/or inserting a new answer in a table, and backtracking through answers (Ramakrishnan et al. 1999). A simple trie data structure is well suited to support all of these operations. Figure 8 depicts a set of terms along with a schematic trie representing these terms. The trie is built from a prefix ordering of each term and thus in this case factors out the common prefix of the first two terms. Such factoring has several advantages. First, it can save space when sets of terms have common prefixes. Second, checking and inserting a term into a trie can be done in one root to leaf pass, checking as long as the trie has a symbol for the corresponding position of a term and inserting once a position with no match is reached. Third, backtracking through a trie can be efficient, as common prefixes do not need to be untrailed and rebound. And fourth, in XSB each node of a trie is indexed, so full term indexing is achieved.

In XSB, subgoals for a tabled predicate are kept in a subgoal trie, and answers for each subgoal are kept in that subgoal's answer trie. The overall structure is as if all tables for a given predicate had been factored according to their subgoals. These

[4] XSB's command-line interpreter automatically reclaims space for any incomplete tables at the end of a query. Setting the Prolog flag `query_level_tabling` ensures that all other tables are abolished as well.

`rt(a,f(a,b),a).   rt(a,f(a,X),Y).   rt(b,V,d).`

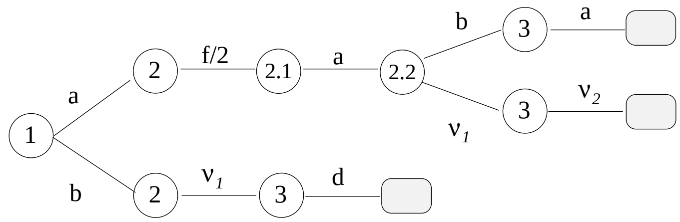

Fig. 8. A set of terms and a schematic trie

tables support an optimization called *substitution factoring* which allows the answer trie to contain only bindings. For instance for a tabled subgoal `p(a,X,f(1,Y))` the answer trie would contain only the bindings for `X` and `Y` and would not contain any constants or structure symbols from the subgoal. XSB also supports a *completed table optimization* in which the trie nodes themselves are SLG-WAM instructions. Thus backtracking through answers from a completed table amounts to a direct execution of virtual machine instructions: no meta-interpretation of the trie is needed.

To summarize, a programmer can make use of XSB's table access by writing programs where subgoals and answers can make use of the left-to-right factoring provided by tries: such programs will also backtrack through completed answers very quickly. On the other hand, substitution factoring ensures that when a subgoal has a large structure, the structure needs to be traversed and stored only once for the subgoal; subsequent answers pay no cost for the subgoal.

2.4 Call Subsumption

The preceding discussion of tabling was intentionally vague about exactly how to determine whether a subgoal or answer is contained in a given forest. Given a selected atom S and forest $\mathcal{F}$, S can reuse the computation of a tree S_{tab} :- S_{tab} in $\mathcal{F}$ as long as S_{tab} is at least as general as S – for our purposes, as long as S_{tab} subsumes S [5]. Most implementations of tabling only reuse tabled information if S_{tab} is a *variant* of S, but create a new tree otherwise: e.g. a subgoal `p(a,X)` will reuse the variant subgoal `p(a,Y)`, but not the subsuming subgoal `p(X,Y)`. In fact, the evaluations in the preceding examples had no properly subsuming subgoals, so their descriptions could ignore this distinction.

The distinction between these two approaches, call variance and call subsumption, can radically affect the behavior of tabled evaluations. Call subsumption can be especially useful when an entire model of a target (sub-)program is desired – as is the case in applications such as program analysis, RDF inferencing or when combining rules with ontologies. This is because a fixed point can be initiated with a set of open goals (goals without any bindings in their arguments); as evaluation proceeds,

[5] A term T_1 subsumes a term T_2 if there is an mgu of T_1 and T_2 whose domain consists only of variables of T_1. If in addition the range of the mgu consists only of variables in T_2, T_1 and T_2 are variants.

tabled subgoals can reuse the answers from the original set of subsuming subgoals. For similar reasons, call subsumption can be useful for deductive database-style queries that are generated from a declarative framework as call subsumption is more "forgiving" when a program is not optimally written: for this reason Flora-2 (see Section 5) makes use of call subsumption for certain generated queries.

When used in XSB, call subsumption allows a subgoal S to use the answers from a subgoal $S_{subsuming}$ as long as there is a table for $S_{subsuming}$, regardless of whether $S_{subsuming}$ is completed or not. When $S_{subsuming}$ is computed along with several subsumed non-completed subgoals, XSB's engine takes care to ensure that answers are returned efficiently to the subsumed subgoals during the iterations required to completely evaluate $S_{subsuming}$ (Johnson et al. 1999). In Version 3.3 call subsumption has been extended to WFS, and supports answers with attributed variables (to implement logical constraints), although subgoals must be free of attributed variables (Swift 2009). Either call subsumption or call variance can be made the default for XSB, and a programmer can combine them at a predicate level using the declaration: `:- table p/n as <subsumptive/variant>`. Call subsumption can provide substantial efficiency gains for many programs; as a worst case, call subsumption in XSB imposes an overhead of about 25% for tabled evaluations that have no subsuming tabled subgoals and so cannot make use of call subsumption.

Example 2.4 (Querying Ontologies) An example of using call subsumption can be found in querying the OWL wine ontology (`http://www.w3.org/TR/owl-guide`). OWL ontologies can be translated from RDF or HTML format into disjunctive datalog by the KAON-2 system (Motik 2006). When translated by KAON-2, the wine ontology produces a definite program with about 1000 clauses. A small fragment of the translated code has the form:

```
pinotblanc(X) :- q24(X).
pinotblanc(X) :- pinotblanc(Y),kaon2equal(X, Y).
pinotblanc(X) :- wine(X),madefromgrape(X, Y),ot____nom21(Y).
madefromgrape(Y, X) :- madeintowine(X, Y).
madefromgrape(X, X) :- riesling(X),kaon2namedobjects(X).
madefromgrape(X, X) :- wine(X),kaon2namedobjects(X).
% 18 other clauses

wine(X) :- q14(X).                  q24(X) :- pinotblanc(X).
wine(X) :- texaswine(X).            q24(X) :- muscadet(X).
% 24 other clauses                  q24(X) :- q24(Y),kaon2equal(X,Y).
wine(X) :- q24(X).
% 31 other other clauses
```

The generated program is highly recursive: for instance, `pinotblanc(yellowTail)` depends on `pinotblanc(X)` which depends on `wine(X)` and on `wine(yellowTail)`. In fact, nearly every concept depends on nearly every other concept (more or less) due to `wine(X)` atoms occurring in body literals, and mixed instantiations are often present in loops, due to the propagation of bindings in rules.

Tabling can be used to evaluate a query to this ontology in XSB by first using the

`:- auto_table` declaration in the translated file. When XSB automatically tables, it chooses enough tables to break all loops in the predicate dependency graph of a file or module. Finding the minimal number of tables to break all loops is an NP-complete problem, so that XSB uses a simple greedy algorithm.

When XSB evaluates the query `pinotblanc(yellowTail)` using call variance, it runs out of memory on a laptop machine. When using call subsumption to evaluate the query, XSB's time is comparable with the ontoBroker system and is much faster than some ASP systems (Liang et al. 2009). Of course, WFS is not powerful enough to evaluate all ontologies – however for many translations XSB can create a residual program that is passed to an ASP system using XASP.

2.5 Answer Subsumption: Lattices, Partial Orders and Aggregation

As with calls, there is a choice of whether to use answer variance or answer subsumption: i.e., if a subgoal has answers `p(a,Y)` and `p(a,b)`, does the evaluation use only the subsuming `p(a,Y)` for resolution with tabled subgoals, or does it use both answers? XSB and all other tabling Prologs use answer variance by default, as answer subsumption between simple Prolog terms seems of limited use for most programs. However, if we generalize the notion of subsumption from the lattice of terms to an arbitrary ordering $\mathcal{O}$, answer subsumption becomes quite useful (Swift and Warren 2010). Consider first a case where $\mathcal{O}$ is a partial order. Such a case may prove useful, for instance, in an inheritance hierarchy, where a query about the relations of an object returns only the most specific answers. In this case, specific answers can be seen as the subsuming answers according to the partial order of the inheritance hierarchy.

In addition to being a partial order, $\mathcal{O}$ may also be a lattice [6]: answer subsumption over lattices is extremely useful for implementing paraconsistent and quantitative reasoning.

Example 2.5 (Quantitative Degrees of Belief) Consider a model of quantitative degrees of belief (van Emden 1986). An annotated atom $A : [E_T, E_F]$ is an atom A together with the annotation

- E_T, a number between 0 and 1 indicating evidence that A is true
- E_F, a number between 0 and 1 indicating evidence that A is false.

In this model, $[E_T, E_F] \geq [E'_T, E'_F]$ if $E_T \geq E'_T$ and $E_F \geq E'_F$, leading directly to a definition of a join operator:

$$[E_T, E_F] \vee [E'_T, E'_F] = [max(E_T, E'_T), max(E_F, E'_F)]$$

A ground atom $A : [E_T, E_F]$ is true in an interpretation I of a program P if there are rules $A : [E^k_T, E^k_I F] : -Body^k$ in P such that each $Body^k$ is true in I and $\bigvee_{\forall k}[E^k_T, E^k_F] \geq [E_T, E_F]$ in I.

Writing programs over such a lattice is easy in XSB. First, the tabling declaration

[6] In Version 3.3 of XSB only an upper semi-lattice need be defined.

indicates that this lattice is to be used in a particular argument of a tabled predicate. The declaration:

```
:- table pred(_,_,qdb/3-[0,0]).
qdb([T1,F1],[T2,F2],[T3,F3]):-                    % join
    (T1 > T2 -> T3 = T1 ; T3 = T2),
    (F1 > F2 -> F3 = F1 ; F3 = F2).
```

indicates that the third argument of `pred/3` is to be maintained via the lattice with join operation `qdb/3` and identity (bottom) `[0,0]`. `pred/3` is translated into

```
pred(X,Y,Z) :- bagReduce(Z1,pred_clause(X,Y,Z1),Z,qdb(_,_,_),[0,0]).
```

where `bagReduce/5` is a tabled predicate that performs the answer subsumption. Each clause of `pred/3` is translated into a clause with head `pred_clause/3` where `pred_clause/3` is non-tabled. When the `bagReduce` literal is called, a check is made for a variant table. If such a table is not found, the subgoal creates a new table and calls clauses of `pred_clause/3` with a variable in the annotated position. When a potential answer, $pred(X_{ans}, Y_{ans}, Z_{ans})$ is derived, `bagReduce/5` determines whether some answer exists whose first two arguments are variants of X_{ans} and Y_{ans}, with a third argument Z_{table}. If so, it takes the join Z_{join} of Z_{ans} and Z_{table}. If Z_{join} is greater than Z_{table}, the old answer is *deleted* from the table and the new answer $pred(X_{ans}, Y_{ans}, Z_{join})$ is added. The table for `bagReduce/5` makes use of trie indexing for tables: the atom part is earlier in the trie than the annotation part, which helps to make the variant check of the atom efficient.

Version 3.3 of XSB allows only positive recursion answer subsumption – uses of negation must be stratified. Non-stratified programs over partial orders, can be modeled with preferences. However, as shown in (Swift 1999b) this method is sufficient to implement various frameworks such as GAPs (Kifer and Subrahmanian 1992), and Residuated Programs (Damásio and Pereira 2001). In addition, it is easy to see that typical aggregate functions, such as `sum/3`, `count/3`, `min/3`, `max/3` etc. are simple extensions of answer subsumption.

When answer subsumption is used, subsumed answers may be derived before subsuming answers, so it is efficient to restrict those places where non-maximal answers are used for resolution. Local evaluation is ideal for this, as it restricts all operations to a maximal SCC. This property implies that no non-maximal answer will be used outside of the SCC in which it was derived. For this reason, the use of Local evaluation can be critical for efficient answer subsumption: (Freire et al. 1998) provides an example where answer subsumption is used to find the shortest paths in a graph G. When Local evaluation is used the time is proportional to the number of edges in G, but when Batched evaluation is used, the time is proportional to the number of *paths* in G, which is exponential in the number of edges of G.

2.6 Tabling with Constraints

XSB offers a simple integration of TLP and Constraint Logic Programming (CLP) as follows. XSB implements CLP by using attributed variables, as do many other

Prologs. When an attributed (constrained) variable V_A is part of a tabled subgoal or a derived answer, V_A is copied into the table along with its attributes; later when V_A is copied out of a table, its attributes are also copied out and associated with V_A. For instance the query `?- p1(X)` to the CLP(R) program

```
:- table p1/1.
p1(X):- {X < 9}.
```

will return the answer { `X < 9.0000`}. In Version 3.3 of XSB, attributed variables are supported even when they occur in literals that are delayed, so that variables in a residual program may be constrained. Since entailment of constraints seen as a relation is a partial order, answer subsumption can be supported for constrained variables using the methods of the previous section.

Many CLP applications will not benefit from tabling, particularly if Prolog interacts with a constraint processor mainly to generate a set of equations to be solved. However for situations that require traversal through a state space where states are associated with constraints, tabling can be useful. Tabled constraints have been used to analyze security protocols (Sarna-Starosta 2005), and for abstract interpretation (cf. (Codognet and Filé 1992) which pre-dates XSB) and when grammar rules involve constraints (cf. (Shieber 1992)), tabled constraints can provide the benefits to parsing. The following example shows how tabled constraints can be used to analyze a process logic.

Example 2.6 (Constraint-Based Nets) A variety of formalisms extend Place Transition Nets to add conditions that must be evaluated for a transition to fire and to add effects that must occur upon its firing, creating applications that can be termed workflow nets. A constraint net follows this model: a condition is the entailment of a formula in a given constraint domain, and the effect is the propagation of new constraints to variables associated with given places in the net. Using such an approach, constraint-based reasoning can be incorporated into workflow logics. The top-level change to add constraints to the running example affects the application of a transition to a configuration, in `apply_trans_to_conf/3` of Figure 4:

```
apply_trans_to_conf(trans(In,Entailment,Out),Conf,NewConf):-
   unify_for_entailment(In,Conf,MidConf),
   entailed(Entailment),
   call_new_constraints(Out,OutPlaces),
   flatsort([OutPlaces|MidConf],NewConf).
```

First, variables in the transition are unified with those of the configuration to produce a new constraint store. If the formula `Entailment` is entailed by the constraint store, new constraints from the transition are placed on the output variables via calling the constraints in the list `Out`. Note that this extension is not specific to a given constraint domain.

3 Dynamic Code and Indexing

In XSB, clauses for dynamic predicates are directly compiled with a simplified compilation strategy, and dynamic predicates may be tabled. While there are a variety of uses for dynamic code, perhaps the most common use is for large and changing knowledge bases. Dynamic code in XSB supports a wider variety of indexing strategies than does static code. The performance of executing dynamic facts is comparable to executing compiled facts with the same indexing, and with better indexing strategies, dynamic code can be arbitrarily faster than the corresponding static code.

In XSB, large files of dynamic code can be efficiently loaded via `load_dyn/1` and its variants. `load_dyn(File)` acts like a compiler in recognizing directives, but treats all clauses in `File` as dynamic; it re-consults `File` by reading clauses from `File` and asserting them. `load_dync/1` is a variation of `load_dyn/1` that can be used if all clauses in a file are in (an extended) canonical form – where no operators are used except for lists and comma-lists. `load_dync/1` is extremely fast: as a general measure, it can read, compile and load files of binary facts at a rate of about 300,000 facts per second on current hardware.

Indexing of Dynamic Code Indexes for dynamic code are built using

- `Trie Indexing` for which a trie is maintained to represent the entire predicate. For instance, `:-index(p/5,trie)` specifies trie-indexing for `p/5`

or as combinations of

- `Main-functor Indexing` for which a hash table is maintained for values of the main functor symbol of the indicated argument. For example `:-index(p/5,3)` indexes the main functor symbol for the third argument of `p/5`.
- `Star Indexing` for which a hash table is maintained for (up to) the first five symbols of the indicated argument. Thus for example, star-indexing can distinguish the term `[p(a)]` from the term `[p(b)]`. A declaration such as `:-index(p/5,*(3))` star-indexes the third argument of `p/5`.

Trie indexing is a special form of all-argument indexing, where asserted facts are maintained in a trie (cf. Section 2.3). As with tabled answers and subgoals, the trie is built from a preorder traversal of a fact, indexing at every position. Clause ordering is not maintained for trie indexed facts, and trie indexing cannot be combined with any other indexing for a given predicate. However asserting and retracting to trie indexed code is about 3 times faster than asserting or retracting to regular dynamic code.

Main functor and star indexing may be combined into multiple joint indexes. For example `:-index(p/5,*(1)+3)` asks for a joint index to be built (for future asserts to `p/5`) for the first and third arguments, so that if a call is ground on both its first and third arguments, it will index the indicated symbols together. Joint indexes may use up to three arguments. Using this joint index in a multiple index, the declaration `:-index(p/5,[*(1)+2,*(1)])` causes two indexes to be built. When calling `p/5`, the indexes are tried in left-to-right order. For this example, if the first and second arguments of a call are bound, the index `*(1)+2` is used. If argument 1 is bound but not argument 2, then `*(1)` will be used.

Incremental Table Maintenance By default in XSB, tables are created when tabled goals are called and are used until they are abolished. But if a tabled predicate depends on a dynamic predicate and the dynamic predicate changes, the table becomes out of date. This is known as the *view maintenance* problem in databases and as the *truth maintenance* problem in artificial intelligence. XSB provides support for *incremental tabling*, so that when changes are made to dynamic predicates, dependent tables are automatically updated to contain the corrected values [7]. Incremental tabling is declared as: `:- table p/2 as incremental`. To make use of incremental tabling, any dynamic predicate, such as `q/2`, whose change should trigger incremental table maintenance is declared as: `:- use_incremental_dynamic q/2`. In Version 3.3 of XSB, incremental updates to a table can be triggered in different ways. In order to update a table based on a single change to a dynamic predicate, calls such as `incr_assert(q(a,5))` or `incr_retract(q(a,5))` can be used. For bulk changes to dynamic predicates, calls to `assert/1` or `retract/1` are made, which will not trigger updates to tables. At the end of the bulk "transaction" a call such as `incr_table_update` triggers the appropriate updates. Finally, calls such as `incr_assert_inval/1` *invalidate* tables that depend on a dynamic predicate, for those cases where incremental updates are deemed to be inefficient (e.g. a clause for a dynamic tabled predicate is retracted).

A Deductive Spreadsheet system (Ramakrishnan et al. 2007) has been built by programming an MS Excel plug-in in XSB to support recursive set expressions in spreadsheet cells. When a spreadsheet user updates the contents of a cell, the engine must update the values of all cells that depend on the updated cell. This is an ideal application for incremental table maintenance since the values of cells often depend on the values of a few other cells, so most cells are not affected by an update to some particular cell. The implementation uses incremental tabling to determine exactly the affected cells and then updates only them. Without incremental tabling, the plug-in was limited to spreadsheets with a few hundred cells; with incremental tabling the plug-in became practical and could recompute tables for very large spreadsheets almost instantaneously.

4 Multi-threading

Multi-threading, the ability to concurrently perform multiple computations, allows Prolog to be used as a server that handles multiple requests or as an agent that handles multiple types of input from its environment. The further ability to coordinate these computations provides support for various types of parallel and distributed processing of a single query. When multi-threading is combined with tabling and specialized indexing, Prolog acquires functionality similar to that of a deductive database, so that it can support applications from semantic web reasoning systems to interactive GUI control.

XSB has been multi-threaded since version 3.0 and supports a draft standard

[7] Currently incremental tabling is implemented only for call variance and for stratified programs.

for multi-threaded Prologs (ISO/IEC DTR 13211-5:2007 2007). The predicates in this standard, many of which originated in SWI-Prolog (Wielemaker 2003), provide facilities for various operations. Predicates for creating, joining and exiting threads and for handling mutexes provide a high-level interface to Prolog threads under a Posix-style semantics. Coordination among threads is handled by message queues, which are used to pass Prolog terms among threads. These message queues may be *public*, with multiple readers and writers; or *thread-specific*, associated with a thread that is the queue's only reader. Among other uses, thread-specific message queues form the basis for thread signaling, which allows one thread to send a goal to another thread. Threads check for signals frequently, so that signaling becomes a powerful mechanism for fine-grained, interrupt-based coordination.

Version 3.3 of XSB may be configured either as single- or as multi-threaded. On Linux and Windows, Prolog evaluation is about 5-15% slower for the multi-threaded engine than for the single-threaded engine; however for Mac OS X the multi-threaded engine is about 5-10% faster. The interface used to call C from XSB supports both single- and multi-threading, so nearly all of XSB's libraries and packages work under both engines. In addition, both engines can be embedded in C code. When multi-threading is used, any C thread can query any XSB thread that is not in the midst of a query.

All of these features form the basis of XSB's multi-threaded tabling engine (Marques 2007), which allows a thread to use private tables to support an independent query, along with shared tables for subgoals that may be used repeatedly by different threads. The simplest execution model is that of private tables, where each thread keeps its own copy of tabled information. Private tables offer several advantages:

- Private tables use sequential tabling algorithms so that they naturally support all tabling features including tabled negation, tabled constraints, and call and answer subsumption.
- Private tables generally require no synchronization among threads above the level of memory allocation.
- Private tables are suitable to ensure query completeness or to support a particular semantics. Tables are automatically reclaimed when the thread that computed them exits. This reclamation includes not only subgoal and answer tries, but the delay lists and supporting structures used to compute WFS.

Shared tables are also important:

- If different threads require the same tables, memory usage for shared tables will be significantly lower than for private tables.
- Shared tables allow the decomposition of a program, so that a set of threads can together compute a set of tables, partially supporting Table-Parallelism (Freire et al. 1995).

Execution Models for Shared Tables By default when tables are shared in Version 3.3, a model called *Concurrent Local evaluation* is used, which relies on Local evaluation and dynamically partitions tables among threads. The idea behind Concurrent Local evaluation is that when a thread T encounters a (shared) tabled subgoal S that has not been encountered by any thread, T evaluates S. Other threads are allowed to use the table for S only after T has completed S. Concurrency control for tables mainly arises when more than one thread evaluates different tabled subgoals in the same SCC at the same time. In this case, a deadlock will occur, which the engine detects and resolves, so that a single thread assumes computation of all tabled subgoals in the SCC (Marques and Swift 2008; Marques et al. 2010). For example, in Figure 2 such a situation would occur if a thread T_1 called `reach(1,Y)` and another called `reach(3,Y)` before it was called by T_1.

Because it is a type of Local evaluation, Concurrent Local evaluation does not allow a consuming node to use answers produced by a subgoal outside of its SCC until the table for the answers is completed – a restriction that prevents producer-consumer models of parallelism. This limitation is overcome by *Concurrent Batched evaluation* which allows several threads to compute (inter-)dependent tabled subgoals in parallel. As with Concurrent Local evaluation, each subgoal can be computed by only one thread. However, a given thread may consume answers as they are produced by another thread. The implementation of Concurrent Batched evaluation extends the implementation of sequential Batched evaluation. In sequential Batched evaluation, when the engine backtracks to the oldest subgoal in an SCC, it schedules the return of unconsumed answers for each consuming node in the SCC, and then proceeds to return the answers via backtracking. In the multi-threaded context, if different threads compute different SCCs, they can work independently, and can consume answers from other threads as they become available. However threads that run out of work will suspend until a single thread institutes a fixpoint check, after which the threads re-awaken. Thus Concurrent Batched evaluation allows parallel computation of subgoals, but has a sequential fixpoint check that synchronizes multiple threads when they compute the same SCC.

Implementation Status In Version 3.3 of XSB, private tables support all tabling features. Concurrent Local evaluation supports most features, but does not yet support call subsumption. Both private tables and shared tables under Concurrent Local evaluation have been heavily tested. Concurrent Batched evaluation should be considered experimental and is currently restricted to left-to-right dynamically stratified programs.

5 Sample Applications of XSB

Numerous applications have been written using XSB: for space reasons we restrict our discussion to two major applications.

XSB, Inc's Ontology-Driven Classification and Extraction Several large applications in XSB have been developed by the company XSB, Inc.[8] Two important ones are the Ontology-Directed Classifier (ODC) and the Ontology-Directed Extractor (ODE). The ODC uses a modified Bayesian classification algorithm to classify item descriptions to categories in a taxonomy. It is in use quarterly by

[8] XSB, Inc. (http://www.xsb.com) is a privately held company that pursues applications of XSB and other technologies to information retrieval and management. XSB, Inc. has also helped support the development of XSB and the related packages InterProlog and XJ.

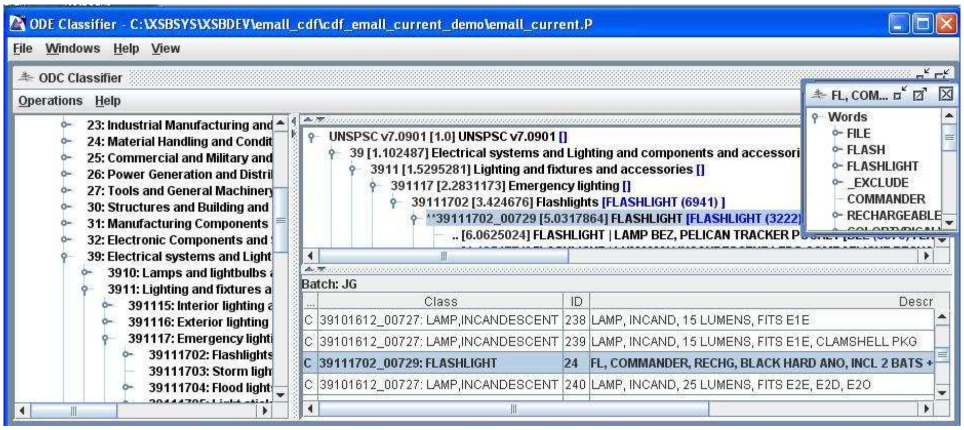

Fig. 9. A Screenshot of XSB Inc.'s Ontology-Directed Classifier

the U.S. Department of Defense to classify over 80 million part descriptions with respect to an extension of the UNSPSC taxonomy that contains over 60,000 categories. The ODE extracts attribute-value pairs from those classified descriptions to build structured, queriable knowledge about parts and their attributes.

Both depend critically on XSB. First, the ontologies for both systems are represented using XSB's CDF ontology management package. CDF has two facets: as a research system it supports the experimental interaction of ontology axioms and rules as a hybrid MKNF knowledge base (cf.(Gomes et al. 2010)). However, for commercial use, it can simply support the representation of classes with inheritance and typing, objects belonging to those classes, and relationships among these components. It is very efficient for such ontologies, scaling to gigabyte-sized in-memory ontologies and to even larger ontologies when a relational database is used as a backing store. CDF makes heavy use of tabling (including tabled negation) along with the joint, multiple, and star indexing discussed in Section 3. Second, the applications perform a significant amount of text processing using an XSB super-tokenizer module. This module supports the declaration of complex rewriting rules for token lists: tens of thousands of these rules implement abbreviations and token corrections in ODC and complex pattern-matching rules in ODE. The super-tokenizer uses tabled grammars and trie-based indexing in fundamental ways, as well as negation to implement preferred rewritings. Third, both applications have complex training interfaces which allow a knowledge expert to add extraction or classification rules, experiment with how extraction or classification works on sample data, and add information to improve the processes. These interfaces are built with XJ (`www.xsb.com/xj.aspx`), an open-source package that allows XSB to construct complex graphical user interfaces through the Java Swing library. XJ itself uses the open-source InterProlog interface (Calejo 2004) to communicate efficiently between Java and XSB.

Figure 9 shows a screenshot of the ODC training GUI driven by XSB using XJ. The left panel displays the taxonomy that is the target of the classification, here the standard UNSPSC taxonomy, extended with further categories from the

Federal Cataloging System. The lower right panel shows item descriptions that are automatically classified to the taxonomy categories. The upper right panel shows classification weights and thus an "explanation" of how the selected description was classified as it was. The optional floating window in the upper right shows the words used in the classification after abbreviation expansion and other rewriting. A knowledge expert uses this interface to explore how the classifier assigns descriptions and to modify it by adding abbreviations, training items, and other tuning options.

Flora-2 and Silk Flora-2 (Yang et al. 2003) (`flora.sourceforge.org`) is a programming system supporting Frame Logic (F-Logic) (Kifer et al. 1995) HiLog and Transaction logic, all of which are implemented in XSB. Flora-2 is a higher-level language than Prolog in the sense that it may represent knowledge more concisely than Prolog, although it offers less procedural control.

Example 5.1 (Flora-2) Figure 10 shows a fragment of a publications knowledge base written in Flora-2. This example was used in (Kifer et al. 1995) to explain various features of F-logic; here we use it to briefly give a flavor of Flora-2. First, note that Flora-2 has a different syntax from Prolog, although each of the statements is a well-defined term and unification can be performed on these terms. In addition, the fragment is divided into a schema and its objects. The subclass relation is indicated by `::/2` and class membership for an object by `:/2`. A class or object is associated with a set of its attributes through brackets (`[]`). Within a schema, class attributes are indicated by `=>>/2`, and by `=>/2` if the attributes are functional. Inheritance for these attributes is monotonic, that is each subclass inherits any concrete attributes of its classes and super-classes and attributes may not be over-ridden. Other predicates provide for inheritable attributes that may be over-ridden.

In addition to the features shown above inheritance and attribute predicates can be also defined in terms of rules. When deriving the answer to a query of a Flora-2 knowledge base, resolution is performed as in Prolog, but the derivation also makes use of inherited attributes and these attributes can be based on other rules as can be the inheritance hierarchy itself. Thus a Flora-2 knowledge base has the advantages of an inheritance-based system for knowledge representation. The price it pays for this is the need to traverse a potentially large portion of an inheritance hierarchy when answering a query. Tabling is a natural mechanism to factor out subcomputations involving the inheritance hierarchies of objects, and Flora-2 makes heavy use of tabling. Flora-2 also relies on tabled negation under WFS for non-monotonic inheritance. The intuition behind this is that an object non-monotonically inherits an attribute if that attribute is *not* over-ridden by some other inherited attribute. Hierarchies with a well-defined "over-rides" relation are stratified, but inheritance may be undefined in WFS. For instance, answers to the query `?- nixon[policy *-> X]` will be undefined in the well-known "Nixon Diamond" example:

```
republican[policy *-> nonpacifist].     quaker[policy *-> pacifist].
nixon:republican.                       nixon:quaker.
```

Flora-2 programs are compiled into XSB using a sophisticated series of transformations. These transformations decide what (XSB) predicates need to be tabled, *Schema:*

```
conference_paper :: paper.
journal_paper :: paper.
paper[authors =>> person, title => string].
journal_paper[in_vol => volume].
conf_p[at_conf => conference_procs].
journal_vol[of => journal, volume => integer, number => integer,
            year => integer].
journal[name => string, publisher => string, editors =>> person].
conference_procs[of_conf => conf_series,year => integer,editors =>> person].
conference_series[name => string].
publisher[name => string].
person[name => string, affiliation(integer) => institution].
institution[name => string, address => string].
```

Objects:

```
o_j1 : journal p[title -> 'Records, Relations, Sets, Entities, and Things',
                 authors ->> {o_mes}, in vol -> o_i11].
o_di : conference_paper[title -> 'DIAM II and Levels of Abstraction',
                        authors ->> {o_mes, o_eba}, at_conf -> o_v76].
o_i11 : journal_vol[of -> o_is, number -> 1, volume -> 1, year -> 1975].
o_is : journal[name -> 'Information Systems', editors ->> {o_mj}].
o_v76 : conference_procs[of -> vldb, year -> 1976,
                         editors ->> {o_pcl, o_ejn}].
o_vldb : conference_series[name -> 'Very Large Databases'].
o_mes : person[name -> 'Michael E. Senko'].
o_mj : person[name -> 'Matthias Jarke', affiliation(1976) -> o_rwt].
o_rwt : institution[name -> 'RWTH Aachen'].
```

Fig. 10. A Publications Object Base and its Schema in Flora-2

and also determine situations in which space can be reclaimed making Flora-2 an example of *user-controlled tabling* as discussed in Section 2.3. In many programs, a hierarchy may be repeatedly traversed using calls in different modes, so that the current experimental version of Flora-2 makes use of call subsumption. In addition, the Flora-2 compiler makes heavy use of XSB's trie-indexed dynamic facts (Section 3) to represent object code. While it is a logic programming language, a Flora-2 program is substantially different from a Prolog program. Accordingly Flora-2 used Prolog to implement its own command-line interpreter, debugger and module system rather than using those of XSB.

The advantages of Flora-2 and XSB have given rise to its use in the ambitious Digital Aristotle project (`www.projecthalo.com`) described as "a reasoning system capable of answering novel questions and solving advanced problems in a broad range of scientific disciplines and related human affairs." Digital Aristotle is based on an extension of Flora-2 called Silk (Grosof 2009) that contains further features of defeasible reasoning and belief logic (Wan et al. 2009), and which is implemented using the techniques of the previous sections.

6 Discussion

The various features discussed in this paper significantly expand the types of programming that can be done in Prolog. Tabling for definite programs in itself allows sophisticated recursions to be coded simply and efficiently; furthermore, these recursions can be combined with CLP as shown in Section 2.6. The additions of tabled negation and answer subsumption support a number of extensions such as preferences and annotations; and well-founded residual programs form a basis for combining Prolog and ASP. The use of call subsumption, incremental tabling, and flexible indexing techniques for dynamic code supports extensions of logic programs to deductive, object-oriented, and semantic web databases – this is particularly true when multi-threading is also exploited.

Robust implementation of these extensions have led to a profusion of research and commercial applications, some of which we cite here. Applications include those in program verification (Ramakrishna et al. 1997; Du et al. 2000; Mukund et al. 2000; Ramakrishnan et al. 2000; Kalantari and Ternovska 2002; Pemmasani et al. 2002; Pokorny and Ramakrishnan 2004; Sarna-Starosta 2005), in program analysis (Dawson et al. 1996; Boulanger 1997; Codish et al. 1998; Janssens and Sagonas 1998; Saha and Ramakrishnan 2005); in natural language analysis and data standardization (Larson et al. 1995; Ramakrishnan et al. 1997; Rocio and Lopes 1998; Cui and Swift 2002; Davulcu et al. 2002), in agent implementations (Alferes et al. 2000; Letia et al. 2001; Kagal and Finin 2004; Lattner et al. 2005; Lattner et al. 2005; Santana and Pereira 2006) and in semantic web applications (Peterson et al. 1998; Davulcu et al. 2000; Li et al. 2002; Tangmunarunkit et al. 2003; Swift and Warren 2003; Swift 2004; Zou et al. 2004; Bhansali and Grosof 2005; Drabent et al. 2007), in diagnosis (Castro and Pereira 2004; Alferes et al. 2004; Barata et al. 2007), in medical informatics (Gartner et al. 2000; Muller et al. 2004), in machine learning (Lamma et al. 2000; Papaterpos et al. 2001) and in software engineering (Pereira and Viegas 2007; Shankar et al. 2006; Oquendo 2004; Ramakrishnan et al. 2007). Many other commercial applications have been developed by XSB, Inc., Medical Decision Logics, Inc (`www.mdlogix.com`), Ontology Works (`www.ontologyworks.com`) and other companies.

All of these applications demonstrate that TLP is a vibrant field of research, involving numerous Prologs including XSB.

Acknowledgements

Dozens of people have contributed to the development XSB. Among those who have made contributions over a sustained period of time are (in alphabetical order) Luis de Castro, Baoqiu Cui, Steve Dawson, Juliana Freire, Ernie Johnson, Michael Kifer, Rui F. Marques, C.R. Ramakrishnsn, I.V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas and Diptikalyan Saha. And we especially thank our user community who have helped us find and fix so many bugs over the years.

References

Alferes, J. J., Azevedo, F., Barahona, P., Damásio, C., and Swift, T. 2004. Logic programming techniques for solving circuit diagnosis. In *Artificial Intelligence Applications and Innovations*. 155–166.

Alferes, J. J., Leite, J. A., Pereira, L. M., and Quaresma, P. 2000. Planning as abductive updating. In *AISB Symposium on AI Planning and Intelligent Agents*. 1–8.

Barata, J., Ribeiro, L., and Onori, M. 2007. Diagnosis on evolvable production systems. In *IEEE Intl. Symp. on Industrial Electronics*.

Bhansali, S. and Grosof, B. 2005. Extending the SweetDeal approach for e-procurement using SweetRules and RuleML. In *RULE-ML*. 113–129.

Boulanger, D. 1997. Fine-grained goal-directed declarative analysis of logic programs. *International Workshop on Verification, Model Checking and Abstract Interpretation*.

Calejo, M. 2004. Interprolog: Towards a declrative embedding of logic programming in Java. In *JELIA*. 714–717.

Castro, J. and Pereira, L. M. 2004. Abductive validation of a power-grid diagnoser. In *IEA/AIE*. 838–847.

Castro, L. and Costa, V. S. 2001. Understanding memory management in prolog systems. In *Intl. Conf. on Logic Prog.* 11–26.

Castro, L., Swift, T., and Warren, D. 2002. Suspending and resuming computations in engines for SLG evaluation. In *Practical Applications of Declarative Languages*. 332–346.

Chen, W. and Warren, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM 43,* 1, 20–74.

Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S., Tsur, S., and Zaniolo, C. 1990. The LDL system prototype. *IEEE Data and Knowledge Engineering 2*, 76–89.

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

Codognet, P. and Filé, G. 1992. Computations, abstractions and constraints in logic programs. In *ICCL*. 155–164.

Cui, B. and Swift, T. 2002. Preference logic grammars: Fixed-point semantics and application to data standardization. *Artificial Intelligence 138*, 117–147.

Damásio, C. V. and Pereira, L. M. 2001. Monotonic and residuated logic programs. In *ECSQARU*. 748–759.

Davulcu, H., Jones, J., Pokorny, L., Rued, C., Swift, T., Vidrevich, T., and Warren, D. 2002. Ensuring consistency in self-reported data: A case study. In *Seventh International Conference on Information Quality*. 155–166.

Davulcu, H., Yang, G., Kifer, M., and Ramakrishnan, I. 2000. Design and implementation of the physical layer in webbases: The XRover experience. In *Computational Logic*. 1094–1105.

Dawson, S., Ramakrishnan, C. R., and Warren, D. S. 1996. Practical program analysis using general purpose logic programming systems — a case study. In *ACM PLDI*. 117–126.

Demoen, B. and Sagonas, K. 1999. CHAT: the Copy-Hybrid Approach to Tabling. In *Practical Applications of Declarative Languages*. 106–121.

Demoen, B. and Sagonas, K. 2001. Heap memory management in prolog with tabling: Principles and practice. *Journal of Functional and Logic Prog. 2001,* 9.

Drabent, W., Henriksson, J., and Maluszyński, J. 2007. Hybrid reasoning with rules and constraints under well-founded semantics. In *Web Reasoning and Rules*. 348–357.

Du, X., Ramakrishnan, C. R., and Smolka, S. A. 2000. Tabled resolution + constraints: A recipe for model checking real-time systems. In *Real-Time Systems Symp.* 175–184.

Freire, J., Hu, R., Swift, T., and Warren, D. S. 1995. Parallelizing tabled evaluation. In *Prog. Langs: Implementations, Logics and Progs.* 115–132.

Freire, J., Swift, T., and Warren, D. S. 1998. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Prog. 1998,* 3, 243–268.

Gartner, J., Swift, T., Tien, A., Pereira, L. M., and Damásio, C. 2000. Psychiatric diagnosis from the viewpoint of computational logic. In *Computational Logic*. 1362–1376.

Gomes, S., Alferes, J., and Swift, T. 2010. Implementing query answering for hybrid MKNF knowledge bases. In *Practical Applications of Declarative Languages*. 25–39.

Grosof, B. 2009. SILK: Semantic rules take the next big step in power. http://silk.semwebcentral.org.

Janssens, G. and Sagonas, K. 1998. On the use of tabling for abstract interpretation: An experiment with abstract equation systems. In *Tabling in Parsing and Deduction*. 118–126.

Johnson, E., Ramakrishnan, C., Ramakrishnan, I., and Rao, P. 1999. A space-efficient engine for subsumption based tabled evaluation of logic programs. In *Symp. on Funct. and Log. Prog.* 284–300.

Kagal, L. and Finin, T. 2004. Modeling communicative behavior using permissions and obligations. In *International Workshop on Agent Communication*. 120–133.

Kalantari, L. and Ternovska, E. 2002. A model checker for verifying ConGolog programs. In *Eighteenth National Conference on Artificial intelligence*. 953–954.

Kifer, M., Lausen, G., and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM 42*, 741–843.

Kifer, M. and Subrahmanian, V. S. 1992. Theory of generalized annotated logic programming and its applications. *Journal of Logic Prog. 12,* 4, 335–368.

Lamma, E., Riguzzi, F., and Pereira, L. M. 2000. Strategies in combined learning via logic programs. *Machine Learning 38,* 1-2, 63–87.

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

Lattner, A., Gehrke, J., Timm, I., and Herzog, O. 2005. A knowledge-based approach to behavior decision in intelligent vehicles. In *IEEE Intelligent Vehicles Symp*. 466–471.

Lattner, A., Timm, I., Lorenz, M., and Herzog, O. 2005. Knowledge-based risk assessment for intelligent vehicles. In *IEEE Conference on Integration of Knowledge Intensive Multi-Agent Systems*. 191–196.

Letia, I., Craciun, F., and Kpe, Z. 2001. Norms for DLP agents working in a warehouse scenario. In *IEA/AIE*. 728–733.

Li, J., Pease, A., and Barbee, C. 2002. Experimenting with ACSC semantic search. Tech. rep., Teknowlege Corporation.

Liang, S., Fodor, P., Wan, H., and Kifer, M. 2009. OpenRuleBench: An analysis of the performance of rule engines. In *WWW: Semantic Data Track*. 601–608.

Marques, R. 2007. Concurrent tabling: Algorithms and implementation. Ph.D. thesis, Universidade Nova de Lisboa.

Marques, R. and Swift, T. 2008. Concurrent and local evaluation of normal programs. In *Intl. Conf. on Logic Prog.* 206–222.

Marques, R., Swift, T., and Cunha, J. 2010. A simple and efficient implementation of concurrent local tabling. In *Practical Applications of Declarative Languages*. 264–278.

Motik, B. 2006. Reasoning in description logics using resolution and deductive databases. Ph.D. thesis.

Mukund, M., Ramakrishnan, C. R., Ramakrishnan, I. V., and Verma, R. 2000. Symbolic bisimulation using tabled constraint logic programming. In *Tabling in Parsing and Deduction*. 166–180.

Muller, R., Greiner, U., and Rahm, E. 2004. Agentwork: a workflow system supporting rule-based workflow adaptation. *IEEE Data and Knowledge Engineering 51,* 2, 223–256.

Niemelä, I. 1999. Logic programs with stable model semanticsas a constraint logic programming paradigm. *Annals of Mathematics and Artificial Intelligence 25*, 241–273.

Oquendo, F. 2004. Formally describing dynamic software architectures with π-ADL. *World Scientific and Engineering Transactions on Systems 3,* 8, 673–679.

Papaterpos, C., Georgantis, N., and Papatheodorou, T. 2001. An ontology for modeling ill-structured domains in intelligent educational systems. In *IEEE International Conference on Advanced Learning Technologies*. 41–42.

Pemmasani, G., Ramakrishnan, C. R., and Ramakrishnan, I. V. 2002. Efficient model checking of real time systems using tabled logic programming and constraints. In *Intl. Conf. on Logic Prog.* 100–114.

Pereira, L. M. and Pinto, A. M. 2009. Layered models top-down querying of normal logic programs. In *Practical Applications of Declarative Languages*. 254–268.

Pereira, L. M. and Viegas, R. D. 2007. Architectural design via declarative programming. In *International Conference on Enterprise Information Systems*. 363–369.

Peterson, B., Andersen, W., and Engel, J. 1998. Knowledge Bus: Generating application-focused databases from large ontologies. In *KRDB-98*.

Pokorny, L. and Ramakrishnan, C. 2004. Modeling and verification of distributed autonomous agents using logic programming. In *Second International Workshop on Declarative Agent Languages and Technologies*. 148–165.

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

Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Dong, Y., Du, X., Roychoudhury, A., and Venkatakrishnan, V. 2000. XMC: A logic-programming-based verification toolset. In *Proceedings on the Conf. on Automated Verification*. 576–590.

Ramakrishnan, C., Ramakrishnan, I., and Warren, D. S. 2007. XcelLog: A deductive spreadsheet system. *Knowledge Engineering Review 22,* 3, 269–279.

Ramakrishnan, I. V., Rao, P., Sagonas, K., Swift, T., and Warren, D. S. 1999. Efficient access mechanisms for tabled logic programs. *Journal of Logic Prog. 38,* 1, 31–55.

Ramakrishnan, I. V., Roychoudhury, A., and Swift, T. 1997. A rule-based data standardizer for enterprise data bases. In *Practical Applications of Prolog*. 255–270.

Ramakrishnan, R., Srivastava, D., and Sudarshan, S. 1992. CORAL: Control, relations, and logic. In *VLDB*. VLDB End., 238–249.

Rocha, R. 2001. On applying or-parallelism and tabling to logic programs. Ph.D. thesis, Universidade do Porto.

Rocio, V. and Lopes, J. 1998. Partial parsing, deduction and tabling. In *Tabling in Parsing and Deduction*.

Rozenberg, G. and Engelfriet, J. 1998. Elemenary net systems. In *Lectures on Petri Nets I: Basic Models*. Springer LNCS 1491, 12–121.

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

Sagonas, K., Swift, T., and Warren, D. S. 2000. The limits of fixed-order computation. *Theoretical Computer Science 254,* 1-2, 465–499.

Saha, D. and Ramakrishnan, C. 2005. Incemental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Decl. Prog.* 117–128.

Santana, P. and Pereira, L. M. 2006. Emergence of cooperation through mutual preference revision. In *IEA/AIE*. 81–90.

Sarna-Starosta, B. 2005. Constraint-based analysis of security protocols. Ph.D. thesis, SUNY Stony Brook.

Shankar, C., Talwar, V., Iyer, S., Chen, Y., Milojicic, D., and Campbell, R. 2006. Specification-enhanced policies for automated management of changes in it systems. In *20th Large Installation Systems Administration Conference*. 101–115.

Shieber, S. 1992. *Constraint-based Grammar Formailsms*. MIT Press.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.

Swift, T. 1999a. A new formulation of tabled resolution with delay. In *Progress in Art. Intel.* 163–177.

Swift, T. 1999b. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 201–240.

Swift, T. 2004. Deduction in ontologies via answer set programming. In *Intl. Conf. on Logic Prog. and Non-Monotonic Reasoning*. 275–289.

Swift, T. 2009. An engine for efficiently computing (sub-)models. In *Intl. Conf. on Logic Prog.* 514–518.

Swift, T., Pinto, A., and Pereira, L. M. 2009. Incremental answer completion. In *Intl. Conf. on Logic Prog.* 519–524.

Swift, T. and Warren, D. 2010. Tabling with answer subsumption: Implementation, applications and performance. Available at `http://www.cs.sunysb.edu/~tswift`.

Swift, T. and Warren, D. S. 2003. *Coherent Description Framework*. Available via xsb.sourceforge.net.

Tangmunarunkit, H., Decker, S., and Kesselman, C. 2003. Ontology-based resource matching in the grid — the grid meets the semantic web. In *Intl. Semantic Web Conf.* 706–721.

van Emden, M. 1986. Quantitative deduction and its fixpoint theory. *Journal of Logic Prog. 4*, 37–53.

van Gelder, A., Ross, K., and Schlipf, J. 1991. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM 38,* 3, 620–650.

Wan, H., Grossof, B., Kifer, M., Fodor, P., and Liang, S. 2009. Logic programming with defaults and argumentation theories. In *Intl. Conf. on Logic Prog.* 432–448.

Wielemaker, J. 2003. Native preemptive threads in SWI-Prolog. In *Practical Applications of Declarative Languages*. 331–345.

Yang, G., Kifer, M., and Zhao, C. 2003. FLORA-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. In *ODBASE-2003*. 671–688.

Zhou, N. and Sato, T. 2003. Efficient fixpoint computation in linear tabling. In *Principles and Practice of Decl. Prog.* 275–283.

Zou, Y., Finin, T., and Chen, H. 2004. F-OWL: An inference engine for the semantic web. In *Formal Approaches to Agent-Based Systems*. 238–248.