Using XSB for Abstract Interpretation*

David S. Warren
Steven Dawson
C.R. Ramakrishnan

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
warren@cs.sunysb.edu

Abstract

XSB is a tabling Prolog system. In addition to Prolog's usual SLD, XSB supports SLG resolution, which can be understood as an extension of OLDT resolution to handle negation under the well-founded semantics. SLG terminates for all Horn programs with the bounded term size property, in particular for those with a finite Herbrand Base. This property makes XSB usable as a tool for implementing abstract interpreters. In this paper, we describe our experience with constructing abstract interpreters using XSB. We find that it is straightforward to obtain simple yet practical implementations of analyses using XSB. Our experiments also suggest a number of enhancements to the system that will improve performance as well as the ease of implementing AI in XSB.

1 Introduction

Program analyses based on Abstract Interpretation (AI) interpret the program using a nonstandard semantics, where the concrete domain of values is replaced by an abstract domain of descriptions, and concrete operators by abstract operators that correspond to their nonstandard interpretation. To build an abstract interpreter, we need to describe the abstract operators, an efficient procedure to evaluate programs with these abstract operations, and mechanisms to collect the results of the interpretation. Since considerable work has been done on efficient *concrete* evaluation of declarative programs, it is natural to ask whether these mechanisms can be directly used for efficient *abstract* evaluation of programs. If so, we would be able to construct simple but efficient abstract interpreters by declaratively specifying the abstract operations, and letting the underlying engine take care of the details of actual evaluation.

It turns out that not all declarative systems have the features needed for building such an abstract interpreter. In some systems, the abstract operations may not be naturally expressible in the language itself. For instance, functional languages are deterministic and the nonstandard interpretation of some operations (*e.g.*, if-then-else) are inherently nondeterministic. In other systems, the abstract operations themselves may be naturally expressible, but the computational procedure

*This work was supported in part by NSF grants CCR-9404921, CDA-9303181 and CDA-9504275.

may be unsuitable for AI. For instance, while abstract operations can be easily expressed in Prolog, the evaluation strategy is well-known for its problems with termination.

Tabled evaluation of logic programs (using resolution strategies such as OLDT [5] and SLG [1]) ensures termination, and the tables themselves, which store the results of intermediate computations, provide means for result collection. SLG resolution forms the basis of the evaluation strategy used in the XSB system. Much effort has been put into making the mechanisms used in XSB, including tabling primitives, robust and efficient. Currently, we are studying the suitability of using XSB for AI. We have implemented analyses (*e.g.*, modes), and the preliminary results are indeed encouraging. While the results do indicate the practical applicability of such implementations, we use these results primarily to identify the dominating factors that determine the performance of analyzers implemented in XSB.

In this paper we describe our experience with constructing abstract interpreters using XSB with the help of two example analyses: mode analysis and term-depth abstraction. In Section 2 we present a brief description of the XSB system and its features of interest to our application. We also present some sample analyses, their formulation, and an outline of their implementation in XSB. The implementations themselves, and preliminary experimental results are described in Section 3. The current implementations are written entirely in tabled Prolog, and run on the generic XSB system. The work is still in progress, and we have, doubtless, not extracted maximal performance from this system as yet. Nevertheless, our experience with the implementation suggests a number of additions and modifications that would enhance the power and performance of the system. Some preliminary approaches towards addressing these issues are discussed in Section 4.

2 AI in XSB

At a high level, XSB evaluates programs by recording subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a table. Predicates may be marked as either *tabled* or *nontabled*. A program is evaluated as follows. For nontabled predicates, the subgoal is resolved against program clauses. For tabled predicates, if the subgoal is already present in the table, then it is resolved against the answers recorded in the table; otherwise the subgoal is entered in the table and its answers, computed by resolving the subgoal against program clauses are also entered in the table[1]. The answer entries are associated with the corresponding subgoal entries. For both tabled and nontabled predicates, program clause resolution is carried out using SLD.

XSB provides an efficient fixed point algorithm that terminates for finite domains. This means that the system can be directly used to compute fixed points for Galois-connection based analyses where all approximations are performed *a priori*. Moreover, being a full Prolog system, XSB permits metaprogramming. This facility can be used to perform approximations during the course of analysis – an ability that is necessary for implementing analyses over infinite domains. The system also provides primitives that enable any user program to obtain the entries in the call and answer tables. These primitives are used to collect the results of the interpretation. In addition to these primitives, a suite of aggregation operations have been recently added to the system library. We believe that these operations provide a good abstraction for performing answer collection.

Below, we describe four example analyses and their implementations using XSB. The first two

[1]The presence of a subgoal in the table is tested in the engine by searching for a *variant* of the subgoal in the table. Only unique answers are entered in the table, and duplicates are filtered out using variant checks.

```
ancestor(X,Y) :-
        parent(X,Y).
ancestor(X,Y) :-
        parent(X,Z),
        ancestor(Z,Y).
parent(david,henry).
parent(tessa,david).
...
```

(a)

```
ancestor(X,Y) :-
        parent(X,Y).
ancestor(X,Y) :-
        parent(X,Z),
        ancestor(Z,Y).
parent(g,g).
```

(b)

Figure 1: The **ancestor** program: (a) Concrete program, and (b) Abstract program

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
      append(Xs,Ys,Zs).
```

(a)

```
append(X1,X2,X3) :-
      iff(X1),
      iff(X2,X3).
append(X1,X2,X3) :-
      iff(X1,X,Xs),
      iff(X3,X,Zs),
      append(Xs,X2,Zs).
```

(b)

Figure 2: The **append** program: (a) Concrete program, and (b) Abstract program

analyses use XSB primarily as an efficient engine for computing fixed points. The other two analyses are performed over the concrete domain with "on the fly" approximations. They illustrate how the evaluation mechanism of XSB, when combined with metaprogramming, can yield simple yet practical implementations of non-trivial analyses.

2.1 Mode Inference for Datalog Programs

We begin with the simplest example, that of inferring the modes of a Datalog program. We use the abstract domain $\{g, ng\}$, where g and ng represent ground and nonground terms respectively. Thus all constants are mapped to the same point g. The abstract unification operation is such that two terms t_1 and t_2 unify iff $t_1 = g \wedge t_2 = g$ or $t_1 = ng \wedge t_2 = ng$. Note that the abstract unification can be performed using the (concrete) unification operation provided by the XSB engine. An example program, **ancestor**, appears in Figure 1a, and its abstract version appears in Figure 1b. Tabled evaluation of the abstract program terminates, in contrast to Prolog-style evaluation. Furthermore, the table contains information on all calls made and their returns, which can be interpreted as input and output modes respectively.

2.2 Groundness Analysis based on the Domain 'Prop'

We now extend the previous example for nondatalog programs, using the same domain $\{g, ng\}$. All ground terms are mapped to g, and the rest of the terms are mapped to ng. Abstract unification is extended to accommodate structures, as follows. A term $f(t_1, \ldots t_n)$ unifies with the value g when all t_i unify with g; it unifies with ng when any t_i unifies with ng. Abstract unification is implemented by predicates $\texttt{iff}(t, t_1, \ldots t_n)$. Unification of two general terms can be done by unifying both to g, and upon backtracking, unifying both to ng. The interpretation of builtins can be easily specified, and is not discussed here. This formulation of groundness analysis is equivalent to formulations based on the *Prop*-domain (*e.g.*, [2, 6]): the propositional formulas are represented by their truth tables, and the individual rows of the truth table are computed by backtracking. An example program, **append**, is given in Figure 2a, and the corresponding abstract program is given in Figure 2b. As in the Datalog case, tabled evaluation of the abstract program terminates, and the calls and returns in the table can be interpreted as input and output modes of the predicates.

2.3 Term Depth Abstraction

This example is derived from the *depth-k* abstraction of terms [4]. The abstract domain is the set of all terms of depth k or less, constructed using the function symbols that occur in the program and a countable set of variables. Two abstract terms are considered distinct only if they are not variants of one another. The concretization function maps each abstract term t to the set of concrete terms S, such that $\forall s \in S \quad t \quad \mathit{subsumes} \quad s$. Abstract unification unifies two abstract terms up to depth k. Note that the concrete unification operation cannot be used directly to implement abstract unification, as was done in mode analysis of Datalog programs. Hence, to obtain the abstract program, every unification in the concrete program is replaced by a call to a predicate that performs the abstract unification operation.

Instead of defining a new abstract unification operation, we can implement depth abstraction by retaining the concrete unification operation and modifying the way fixed points are identified by the system. Thus, the analysis may be performed over the concrete domain, while making approximations only when checking to see if a fixed point has been reached. This can be implemented in XSB by abstracting the goals for which answers are computed and, furthermore, placing an approximation of the answers in the return table, as described in Section 3. It should be noted that this implements a restricted form of *widening*, where the next iteration value is widened independent of the value of the previous iterations.

We implement this analysis by tabling every user defined predicate. We use the depth-k abstraction for approximating both calls and returns. Clearly, this makes the number of distinct calls and returns finite. Since the interpretation of built-ins is also finite, only a finite amount of computation is performed between any two table accesses. Hence, a tabled evaluation with the call and answer approximation always terminates. In the next section, this analysis is used to illustrate the trade-offs between different implementation strategies.

2.4 Groundness with Term Depth Abstraction

The abstract domain used here is the set of terms of depth k or less, constructed with the set of function symbols that occur in the program, a special 0-ary symbol γ, and a countable set of variables. Note that this is an extension of the domain used for term depth abstraction. The concretization function is an extension of the one used for term depth abstraction such that γ represents the set of all ground concrete terms. Note that, unlike in the earlier example, the concrete unification operation cannot be used to perform abstract unification. Hence, in an implementation of this analysis, we define an abstract unification operation that modifies the concrete operation such that a term t unifies with γ, with the variables in t acquiring γ as their substitution. We use the abstraction function to approximate both calls and returns, thereby modifying the computation of fixed points. This analysis is equivalent to the groundness analysis formulated using the constraint-

```
interp((Goal1,Goal2)) :- !,
        interp(Goal1),
        interp(Goal2).
interp((Goal1;Goal2)) :- !,
        interp(Goal1);
        interp(Goal2).
interp(Goal) :-
        is_builtin(Goal), !,
        call_builtin(Goal).
interp(Goal) :-
        interp_pred(Goal).
```

```
:- table interp_pred/1.
interp_pred(Goal) :-
        copy_abstract(Goal,Goal1),
        clause(Goal2,Body),
        call_builtin(Goal1 = Goal2),
        interp(Body),
        copy_abstract(Goal2,Goal3),
        call_builtin(Goal3 = Goal).
```

Figure 3: Core of abstract meta-interpreter in XSB

based framework of [3]. Implementation of this example shows how XSB can be used for analyses that define their own abstract operations in addition to modifying the fixed point computation.

3 Experience

We have thus far experimented with implementations of simple abstract interpreters in XSB at three levels, which we refer to as meta-interpretation, partial compilation, and direct execution. Meta-interpretation is the highest level of implementation, where an unmodified object program is evaluated by an abstract meta-interpreter. Partial compilation and direct execution constitute, in effect, degrees of partial evaluation of the meta-interpreter applied to object programs. The object program is first transformed and then either interpreted at a lower level (partial compilation) or compiled and executed (direct execution). Thus, these three levels of implementation represent tradeoffs between preprocessing time and evaluation time, both of which are important factors in the efficiency of abstract interpretation. Below we briefly describe the three levels of AI implementation in XSB mentioned above. We then present preliminary performance figures for two analyses (groundness and term-depth abstraction) that shed light on the preprocessing/evaluation time tradeoffs as well as provide promising evidence of XSB's potential as an AI system.

3.1 Implementation

Meta-interpretation The (somewhat idealized) core of an abstract meta-interpreter in XSB is shown in Figure 3. The basic structure of the interpreter will be familiar to writers of Prolog meta-interpreters. Predicate `interp/1` performs abstract interpretation of a goal according to the definitions of `call_builtin/1` and `copy_abstract/2`. Calls to built-in predicates are evaluated abstractly by `call_builtin/1`. Calls to predicates defined in the object program are evaluated by the tabled predicate `interp_pred/1`, which performs call and return abstraction via `copy_abstract/2`. Recall that it is this call and return abstraction, together with the tabling of `interp_pred/1`, that ensures termination of the meta-interpreter. The abstract calls and answers for predicates in the object program can be obtained as interpretation proceeds or after interpretation by using tabling primitives.

For term-depth abstraction, the built-in operations need not be abstracted (`call/1` can be used in place of `call_builtin/1`). Instead, `copy_abstract/2` is defined to copy and truncate a

```
partition([],_,[],[]).
partition([F|T],P,[F|S],B) :-
   F <= P,
   partition(T,P,S,B).
partition([F|T],P,S,[F|B]) :-
   F > P,
   partition(T,P,S,B).
```

(a)

```
partition(X1,_,X3,X4) :-
   abstract_=(X1,[]), abstract_=(X3,[]),
   abstract_=(X4,[]).
partition(X1,P,X3,B) :-
   abstract_=(X1,[F|T]), abstract_=(X3,[F|S]),
   abstract_<=(F,P),
   abstract_call(partition(T,P,S,B)).
partition(X1,P,S,X4) :-
   abstract_=(X1,[F|T]), abstract_=(X4,[F|B]),
   abstract_>(F,P),
   abstract_call(partition(T,P,S,B)).
```

(b)

Figure 4: AI via partial compilation: (a) original program and (b) abstract metaprogram

given term at a particular depth (by replacing terms below that depth with "new" variables). For groundness analysis, abstract versions of built-in predicates (in particular, unification) are defined, and `copy_abstract/1` is defined to produce an abstract copy of a given (concrete or abstract) term.

Partial compilation Partial compilation seeks to reduce the overhead of meta-interpretation by embedding calls to abstract operations within the object program itself. In addition, calls to predicates defined in the object program are replaced by calls to `abstract_call/1` (essentially a tabled abstraction of `call/1`), similar to `interp_pred/1` from the meta-interpreter:

```
:- table abstract_call/1.
abstract_call(Goal) :-
   copy_abstract(Goal,Goal1),
   call(Goal1),
   copy_abstract(Goal1,Goal2),
   abstract_=(Goal,Goal2).
```

A simple application of this transformation is illustrated in Figure 4. As with the original program in meta-interpretation, the transformed program is asserted, although it could also be compiled and executed (at much expense in preprocessing time). Abstract interpretation is then initiated by one or more queries to `abstract_call/1`.

Direct execution The transformation used in partial compilation can be taken a step further to yield a program that can be compiled and executed directly, avoiding the overhead of `call/1`. Calls to `abstract_call/1` are replaced by calls to a tabled abstract version of the predicate to be called, where the abstract predicate performs the operations that would have been carried out by `abstract_call/1`. An abstract predicate is defined for each predicate in the object program, and has the form:

```
:- table abstract_p/n.
abstract_p(X1,...,Xn) :-
   copy_abstract(t(X1,...,Xn),t(Y1,...,Yn)),
   p(Y1,...,Yn),
   copy_abstract(t(Y1,...,Yn),t(Z1,...,Zn)),
   abstract_=(t(X1,...,Xn),t(Z1,...,Zn)).
```

where t can be any function symbol, as it is used simply to ensure that shared variables among the arguments remain shared (modulo abstraction) in the copy. The result of applying this transformation to the predicate in Figure 4(a) is given in Figure 5. The abstract execution is initiated by queries to the abstract predicates.

```
partition(X1,_,X3,X4) :-
   abstract_=(X1,[]), abstract_=(X3,[]),
   abstract_=(X4,[]).
partition(X1,P,X3,B) :-
   abstract_=(X1,[F|T]), abstract_=(X3,[F|S]),
   abstract_<=(F,P),
   abstract_partition(T,P,S,B).
partition(X1,P,S,X4) :-
   abstract_=(X1,[F|T]), abstract_=(X4,[F|B]),
   abstract_>(F,P),
   abstract_partition(T,P,S,B).

:- table abstract_partition/4.
abstract_partition(X1,X2,X3,X4) :-
   copy_abstract(t(X1,X2,X3,X4),t(Y1,Y2,Y3,Y4)),
   partition(Y1,Y2,Y3,Y4),
   copy_abstract(t(Y1,Y2,Y3,Y4),t(Z1,Z2,Z3,Z4)),
   abstract_=(t(X1,X2,X3,X4),t(Z1,Z2,Z3,Z4)).
```

Figure 5: AI through direct execution

3.2 Performance evaluation

Preliminary experiments with the above three approaches to AI in XSB have helped to identify and clarify issues in using XSB for abstract interpretation. The results are encouraging that, using XSB, we can obtain simple and high-level, yet efficient, implementations of analyses. Below we present the results of our experiments[2] for the three simple abstract interpretations described in Section 2.

Term-depth abstraction Performance figures for term-depth abstraction on two simple programs are given in Table 1. The first program is a parser over lists of constants, evaluated to depth 100. The second is a parser over tree structures, evaluated to depth 4. As expected, preprocessing time (Prep) is lowest for meta-interpretation and is slightly higher for partial compilation due to transformation. For direct execution, preprocessing time is substantially higher, since the transformed object program must be compiled. Somewhat unexpected is the consistency of evaluation time (Eval) across the three methods. This uniformity suggests that the overhead normally associated with interpretation is being masked by greater overheads, the possibilities being tabling and/or term copying. Indeed, the evaluation of both programs involves a good deal of copying of large structures, both explicitly (through **copy_abstract/2**) and implicitly (through tabling). To help assess the overhead due to copying, the term-depth abstraction version of **copy_abstract/2**

[2]All reported figures were obtained using XSB version 1.4.2 on a 486 DX2-50 PC running Linux. For the programs reported on here, this machine configuration yields performance roughly comparable to that of a Sun SPARCstation 2.

Program	Meta-interpretation		Partial compilation		Direct execution	
	Prep time	Eval time	Prep time	Eval time	Prep time	Eval time
List parser	0.12	2.54	0.14	2.54	0.87	2.48
Tree parser	0.19	11.00	0.25	10.20	1.55	10.10

Table 1: Times (seconds) for term-depth abstraction

Program	Meta-interpretation		Partial compilation		Direct execution	
	Prep time	Eval time	Prep time	Eval time	Prep time	Eval time
List parser	0.12	0.43	0.14	0.38	0.87	0.38
Tree parser	0.19	3.29	0.25	2.38	1.55	2.38

Table 2: Times (seconds) for term-depth abstraction (fast copy)

(originally written in Prolog), was rewritten in C. The result was a reduction in evaluation time by a factor of six for the list parser, and a factor of three to four for the tree parser (see Table 2). In the case of the tree parser, the reduction was sufficient to begin to reveal the overhead of meta-interpretation.

Groundness with term-depth abstraction Table 3 shows results for depth 1 (predicate argument-level) groundness analysis for a set of benchmarks from [2]. As before, columns "Prep" and "Eval" show program preprocessing and evaluation time, respectively, while column "Total" shows total analysis time. To indicate the practicality of this implementation, we also show (in column "% Comp") the total analysis time as the percentage of base compilation time. Thus, for partial compilation, groundness analysis would add approximately 15-45% to the overall compilation time for these examples. The improvement of partial compilation over meta-interpretation indicates, not surprisingly, that the overhead of term copying (done in Prolog) is not as great when the depth of terms is kept low. No measurements were made for the direct execution approach, since compiling the abstract program for analysis guarantees a more than doubling of overall compilation time.

Program	Meta-interpretation				Partial compilation			
	Prep	Eval	Total	% Comp	Prep	Eval	Total	% Comp
CS	0.57	0.28	0.85	19.7	0.61	0.12	0.73	16.9
Peep	1.32	0.75	2.07	20.1	1.35	0.19	1.54	15.0
QSort	0.05	0.05	0.10	32.3	0.05	0.02	0.07	22.3
Read	1.07	2.76	3.83	59.8	1.10	1.70	2.80	43.4

Table 3: Times (seconds) for groundness with term depth abstraction

Groundness using Prop domain Results for the groundness analysis benchmark programs using the Prop domain (see Section 2) are given in Table 4. The analysis times are comparable to those for term-depth groundness analysis, although preprocessing times are higher due to flattening of all terms into `iff` calls (see Figure 2).

Program	Prep	Eval	Total	% Comp
CS	0.78	0.27	1.05	24.3
Peep	2.13	0.29	2.42	23.5
QSort	0.08	0.01	0.09	29.0
Read	1.70	1.64	3.34	52.1

Table 4: Times (seconds) for Prop domain (partial compilation)

3.3 Discussion

Given the reduction in evaluation time for groundness analysis (with term depth abstraction) by a simple partial evaluation, it is natural to ask whether more sophisticated partial evaluation might yield even greater performance improvements. At the same time, preprocessing is the dominant component in overall analysis time for both types of groundness analysis tested. Furthermore, the results of term-depth abstraction, where evaluation time dominates, indicate that direct execution (full compilation) may gain little in evaluation time and cost much in terms of preprocessing time. And, as noted earlier, the additional preprocessing time required by direct execution[3] would certainly increase the cost of groundness analysis to more than 100% of overall compilation time. Thus, the benefits of partial evaluation to evaluation time must be weighed against the costs of preprocessing. Of the three levels of implementation tested, partial compilation appears to offer the best balance by reducing evaluation time with little increase in preprocessing time.

4 Issues

There are many ways to improve the usability and performance of XSB for AI. Below, we outline several such ways indicated by our preliminary experiments. The issues of answer collection and bottom-up evaluation are general issues that, when addressed, would improve not only analyses but applications such as deductive databases as well. The first issue of call and return abstraction, however, appears to be specific to applications such as AI that perform approximate computations.

Call and Return Abstraction Observe that an abstraction of a term may be less, or differently instantiated than the original term. Hence, abstraction cannot be performed *in place*. Due to this, all the approaches for implementing analysis that need call and return abstraction involve a substantial amount of copying. Every call is copied when it is entered in the call table, and another copy is made when the call is abstracted. Similarly, return abstraction and tabling lead to two copies for every return. When the calls are abstracted *after* tabling, two calls that have the same abstraction may be considered distinct by the tabling mechanism, and the resulting computations are not shared. On the other hand, if the calls are abstracted *before* tabling, then the copy for abstraction is done on *every call*. These inefficiencies can be avoided if the abstraction is done *while* tabling. We are planning to provide hooks to the tabling primitives, so that the abstraction to be performed while tabling can be specified by the user.

For return abstraction, note that the unification of the abstracted return with the original call

[3]It should be pointed out that in XSB it is possible to assert (and execute via `call/1`) programs with tabled predicates like the one in Figure 5. However, doing so involves a transformation more costly than that of the simpler partial compilation, and yet would likely not lead to better evaluation times.

```
partition(X1,_,X3,X4) :-
   abstract_=(X1,[]), abstract_=(X3,[]),
   abstract_=(X4,[]).
partition(X1,P,X3,B) :-
   abstract_=(X1,[F|T]), abstract_=(X3,[F|S]),
   abstract_<=(F,P),
   subsumes(abstract_call(partition(T,P,S,B))).
partition(X1,P,S,X4) :-
   abstract_=(X1,[F|T]), abstract_=(X4,[F|B]),
   abstract_>(F,P),
   subsumes(abstract_call(partition(T,P,S,B))).
```
(a)

```
:- hilog subsumesPO.

subsumes(Call) :-
   bagPO(Call, subsumesPO).

subsumesPO(X,Y) :-
   % X is subsumed by Y
   subsumes_chk(Y, X).
```
(b)

Figure 6: Aggregates for Answer Collection: (a) abstract metaprogram, and (b) aggregate

will always succeed, and is performed only to transfer variable bindings. We are investigating mechanisms that capture the correspondence between variables in the original and the abstracted calls, so that the return unification can be entirely avoided. Recall that the return abstraction we have explored so far is a restriction of widening. More sophisticated widening operations need (1) the knowledge of other returns already present in the table, and (2) a mechanism to modify any or all of the returns in the table. Although this can be done using existing low level system primitives, we do not yet have a high level abstraction of widening. The operations to perform aggregation over sets have exactly the same requirements, and we believe widening can be implemented along the same lines as the aggregation operations that have been recently added to the system library.

Answer Collection The examples discussed in this paper are formulated such that the answers are maintained as a disjunction of substitutions. Some analyses are formulated so as to retain only the least upper bound of answers, while others maintain only the answers that are 'maximal' with respect to some partial order. This is done primarily for efficiency, in order to eliminate redundant information from the representation. When the answers for a call are viewed as a set, finding the least upper bound or the maximal elements can be considered as aggregates over this set. Recently, a collection of generic aggregation operations, implemented using the tabling primitives, have been added to the system library. Using these operations, answer collection can be implemented very simply and at a high level, as illustrated by the following example.

Consider an analysis in which we want to maintain only the most general answers. This analysis can be implemented using the partial compilation method (see example in Figure 4) as follows. Note that **abstract_call/1** returns a set of answers. We can pick the subsuming answers by using an aggregation operation **subsumes/1**. The abstract metaprogram in Figure 4b is now modified to the program given in Figure 6a. The operation **subsumes/1** can be defined using a generic aggregation operation, **bagPO/2**, provided by the system. The predicate **bagPO(Call, PO)** returns only the those answers to **Call** that are maximal with respect to the partial order **PO**. The partial orders themselves are defined using higher-order (HiLog) predicates. For example, the definition of **subsumes/1** is given in Figure 6b.

Even in analyses that maintain the entire set of answers (such as those in the paper), aggregation operations can be used to simplify the collection of final results. Efficiency of these operations depends on the tabling primitives provided by the system, as well as the scheduling strategy used to return answers to tabled predicates. We are investigating breadth-first scheduling strategies and their impact on aggregation. Such strategies can also be used to improve the efficiency of bottom-up evaluation of programs in XSB.

Bottom-up Analysis Top-down analyses benefit from XSB's goal-directed evaluation strategy. This benefit is clear from the implementation of mode analysis in XSB: the tabling mechanisms enable the computation of input and output modes in one analysis pass without requiring transformations such as Magic (*e.g.*, [2]). With a full implementation of call subsumption, bottom-up analyses need not pay the price for goal-directedness. While *forward* subsumption can be implemented in the engine at a relatively low cost, the cost of implementing *backward* subsumption is unclear. However, bottom-up computations may be performed efficiently by the following strategy that uses forward subsumption. On the first call to a tabled predicate, we fill the table by simulating an open call. Forward subsumption can now be used to return the answers to any specific call. The impact of this strategy will be clear only after the implementation of forward subsumption in the engine is complete.

5 Conclusions

Our experiments indicate that it is straightforward to implement simple abstract interpreters in XSB. Furthermore, such implementations perform well enough to be practical. However, it should be stressed that these experiments only just begin to explore the use of XSB for AI. For instance, we have focused our efforts on identifying potential bottlenecks in evaluation time, and little attention has been paid to making preprocessing efficient. We believe that there is considerable scope for improvement in the abstraction mechanisms provided by the system as well as in the overall system performance, which should make XSB even more attractive for AI.

References

[1] W. Chen and D. S. Warren. Query evaluation under the well-founded semantics. In *ACM Symposium on Priciples of Database Systems*. ACM Press, 1993.

[2] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a Magic wand. In *International Logic Programming Symposium*, pages 114–129. MIT Press, 1993.

[3] C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. A symbolic constraint solving framework for analysis of logic programs. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1995.

[4] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.

[5] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. MIT Press, 1986.

[6] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain Prop. *Journal of Logic Programming*, 23(3):237–278, 1995.