Appears in **International Journal on Software Tools for Technology Transfer** Dec. 1998.

Semantics-Based Program Analysis for Logic-Based Languages using XSB

Michael Codish[1], **Bart Demoen**[2], **and Konstantinos Sagonas**[2]

[1] Department of Mathematics & Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel
[2] Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Received November 2, 1997

Abstract. This article describes a simple and efficient way of using a logic programming language with builtin tabulation for general purpose semantics-based program analysis. The simplicity of the method is based on a clear separation of abstraction and control: conceptually, a concrete program is executed over an abstract domain and the tabulation mechanism avoids recomputation, ensures termination and collects the results of the analysis. The efficiency derives from the fact that an abstract interpreter induces an abstract compiler which maps input programs to corresponding abstract programs which are (compiled and) run in XSB [43]. The design of new analyses is an easy and fast process, because XSB is a general purpose logic programming system which supports efficient fix-point computations through tabling implemented at the abstract machine level and comes off the shelf: in fact, due to its optimized control and superior tabling performance, our approach using XSB competitively compares with most of the existing special purpose abstract interpretation tools for logic programs. We demonstrate that our approach to using XSB for abstract interpretation supports the usual techniques and optimizations of other frameworks in a straightforward and flexible way.

1 Introduction

We propose and fully describe the use of a general purpose logic programming system which supports tabled evaluation as the foundation for a generic abstract interpretation system for the analysis of logic programming languages. The theory of *abstract interpretation* [21] lies at the heart of most of the major advances of the last decade in the area of program analysis in general and logic program analysis in particular. In this approach, a program analysis is viewed as a non-standard, abstract semantics defined over a domain of data descriptions. An abstract semantics is constructed by replacing operations in a suitable concrete semantics with corresponding abstract operations defined on data descriptions. Program analyses are defined by providing finitely computable abstract interpretations which preserve interesting aspects of program behaviour. Formal justification of program analyses is reduced to proving conditions on the relation between data and data descriptions and on the elementary operations defined on the data descriptions.

Evolution of program analysis tools In the context of logic-based languages, early attempts to apply abstract interpretation were mainly ad hoc approaches to program analysis aiming to derive specific information, usually about modes, types or variable dependencies [39, 24, 10, 25, 23]. Recent years have witnessed immense efforts devoted to the design and implementation of tools to aid in the semantics-based analysis of logic programs. The brief survey presented here is by no means complete; however the state of the art of these tools is continuously advancing. In addition to some ad hoc techniques for specific program analyses (such as for example those described in [17, 29, 9]), several generic frameworks have also emerged. In particular, we mention the bottom-up frameworks described in [37, 4, 15] as well as the top-down approach described in [8]. The latter has led to the implementation of GAIA [35] and PLAI [12, 40] which have evolved into analysis systems with good performance incorporating various optimizations. These systems are "interpreter-based" in the sense that a given (concrete) program is interpreted over a non-standard (abstract) domain of execution. Relatively recently, in [33], the authors describe an abstract machine AMAI for the abstract interpretation of logic programs. The proposed abstract machine is presented as a starting point for the design of efficient abstract interpretation

Correspondence to: Kostis.Sagonas@cs.kuleuven.ac.be

frameworks just as the WAM [46] is a good starting point for obtaining efficient implementations of logic programming languages with backtracking.

Requirements for efficient program analyses Apart from the choice of semantics and domain specific considerations, two issues are central for an efficient implementation of program analyses:

1. the control of the analysis (e.g. the algorithm applied to compute a fix-point) and
2. the storage and retrieval of intermediate results during the evaluation process.

Systems implemented in Prolog (e.g. those in [17, 33, 40]) usually rely on the dynamic (or recorded) database which provides a convenient way of manipulating dynamically created terms but is inefficient and incurs a severe time penalty. The GAIA system [35], implemented in C, has gone further in this area, by applying advanced hashing techniques for storing the results of the analysis. As for control, both bottom-up and top-down systems alike, use sophisticated fix-point algorithms in an attempt to mimic a semi-naive control strategy to avoid repeated subcomputations.

A high-level description of our approach In this article we advocate the use of a logic programming system with builtin tabulation, and XSB in particular, as a generic program analysis tool for logic-based languages. XSB[1] is a general purpose logic programming system based on SLG resolution [13] which supports a highly efficient tabling mechanism based on *tries* [41] and whose control strategy is particularly well-suited for abstract interpretation, as first noted in [47]. We demonstrate how a wide range of analysis techniques found in a variety of different semantics-based analyzers can be implemented in XSB with minimal effort: indeed, the underlying abstract machine of XSB supports the main ingredients needed to implement efficient program analyses. In [22] the authors make an important first step, illustrating the practicality of XSB's use for program analyses. The authors construct a simple mode analysis based on groundness dependencies using the abstract domain Pos. In their approach a given logic program is transformed to a corresponding tabled logic program whose concrete execution provides the analysis information about the original logic program. In principle the approach is similar to that described in [17] except that the tabulation mechanism of XSB is used to optimize the implementation strategy.

We present a general methodology for using XSB as a vehicle for implementing abstract interpretations for logic programs and as such also formally reconstructs the work described in [22]. The basic idea is to specify an abstract interpretation as an interpreter written in XSB. In this way the analysis of a given logic program is obtained as its interpretation (using the XSB interpreter). Efficient program analyzers are derived by program specialization — the XSB interpreter induces a corresponding program analyzer which compiles logic programs to corresponding XSB programs which when run give the desired analysis information. In this way a simple four line XSB interpreter (cf. Section 2.1) provides exactly the same functionality as that described in [22] for analyses based on the Pos domain [38]. The main difference is that all of the low-level implementation details are hidden from the user and provided by the efficient tabulation mechanism and control strategy of the underlying XSB system.

The contribution of this article is more than a mere exercise in aesthetics to provide a concise reconstruction of the work described in [22]; we show how XSB can be applied to support more advanced program analyzers for logic programs: in particular we show how the framework of abstract compilation in the context of XSB can be adapted to support the derivation of information at program points, most general call analyses, how it can include abstract operations such as abstract unification and least upper bound, and how an extension of the four line interpreter can be used for fix-point based program analyses that go beyond applications of abstract compilation. The simplicity of our interpreter-based approach facilitates the application of XSB to support these more advanced program analyzers for logic programs. The simple tabled interpreters we present reflect clean and elegant logical specifications for a variety of program analysis techniques and optimizations commonly supported by other logic program analysis tools. Each such specification is a working XSB interpreter which induces a corresponding program transformation that provides an efficient implementation of the corresponding program analysis technique.

Figure 1 shows schematically the different uses of a tabled interpreter in the context of this article: the double arrows indicate input to, or output from a program while the single arrows mean 'runs in'. It can be seen that the tabled interpreter can run directly in XSB together with the domain-specific operations and with the concrete program (and possibly a query) as input; the information derived by the analysis is collected in the tables of XSB. Another more efficient use of the tabled interpreter is to specialize it (either automatically by a specializer or by hand) thereby producing an induced transformation which transforms the concrete program into an abstract program: this abstract program runs directly with the domain in XSB and leaves the same analysis result in the table.

The orthogonality of the approach is exhibited by the fact that starting with a different tabled interpreter leads to a different analysis strategy, while a different (abstract) domain leads to the analysis of a different property of the program.

[1] available at `http://www.cs.sunysb.edu/~sbprolog/xsb-page.html`.

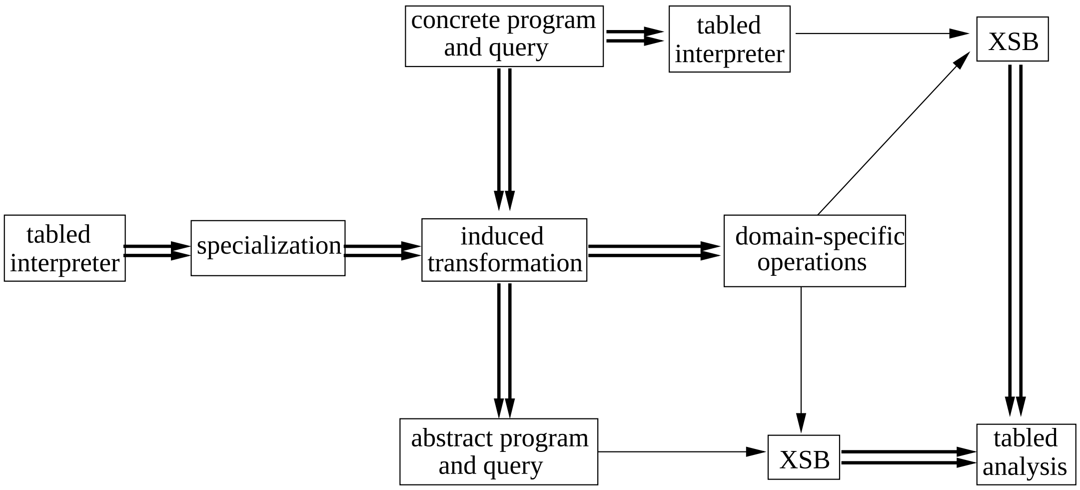

Fig. 1. Different uses of a tabled interpreter.

It is therefore our view that the combination of these techniques provide a powerful, efficient and versatile general purpose analysis tool. When provided with the definitions of suitable abstract operations (e.g. abstract unification) and with a specification of the type of analysis to be performed (e.g. goal-dependent) the system generates an analyzer which compiles logic programs to corresponding analysis programs which when run in XSB provide the desired program analysis. The simplicity of the described approach allows a strong separation of abstraction from control. This is due to the fact that all of the elements related to obtaining an efficient and terminating evaluation strategy are provided by the underlying XSB system.

Organization of this article The remainder of the article provides an overview of our methodology and its implementation in XSB. In the next section, we describe a simple analysis based on program abstraction with concrete evaluation, illustrating the general idea applied throughout this article: each analysis concept is portrayed as a simple tabled interpreter. In Section 3 we describe a simple type analysis based on [18], and a freeness and sharing analysis based on [9]. These examples serve as vehicles to illustrate how abstract domain-dependent operations can be introduced into the interpreters of the previous section and make a step towards a generic program analysis tool. In Section 4 we illustrate an alternative strategy for collecting answers during program analysis based on an abstract least upper bound operation and present results of using such collecting strategies for a sharing analysis based on [19]. Section 5 describes how our framework can be used for analyses that do not use the variables in the abstracted terms to collect the results of the analysis (the abstract substitutions). Section 6 compares the use of XSB for abstract interpretation with a special purpose abstract interpreter written in a high-performance commercial Prolog system without tabling. Finally, Section 7 presents our conclusion: nowadays, a logic programming language with tabling in general — such as XSB in particular — supports the main ingredients for efficient program analysis, eliminates (some of) the necessity for special-purpose tools for abstract interpretation, and therefore can serve as a common platform for both design of and experimentation with program analyses.

2 The First Step: Abstract Compilation and Concrete Tabulation

This section illustrates our basic approach and also reconstructs the work of [22]. We focus here on the implementation of program analyses using a technique which is often referred to as *abstract compilation* in the literature and (in the context of logic programming) goes back to the work described in [24, 31]. The underlying idea is that instead of applying an abstract semantics to a (concrete) program, as in abstract interpretation, a concrete semantics is applied to an abstraction of the program itself. Consequently, the implementation of a program analysis boils down to computing the concrete semantics of (abstract) programs. In applications of this type, it is up to the abstraction function to guarantee that the concrete semantics of an abstract program is finite. This approach has been applied for a variety of domains.

As an an example consider the groundness dependency analysis described in [17] where the concrete semantics is the s-semantics described in [27, 6]. The s-semantics is defined (and computed) in terms of a bottom-up operator which is very similar to the standard T_P operator but defined over a domain of terms with variables. The main advantage of this approach is that the resulting non-ground "minimal s-model" (which strictly speaking is not a model) captures both declarative as well as operational aspects of program behaviour. In particular, the minimal model of the program is obtained as the ground instances of the s-model; and solving a query in the s-model gives the same answers as solving it with the original program. Two types of analyses are considered:

1. *goal-independent* analyses — based on the s-semantics of an abstract program — which approximates the answers for arbitrary queries; and
2. *goal-dependent* analyses — based on the s-semantics of the magic-set transformation of a given abstract program and initial goal — which approximates the set of calls and answers in all computations of the initial goal with the program.

In the remainder of this section we describe simple XSB interpreters which can be used for goal-independent and goal-dependent concrete evaluations. In the first case the interpreter will compute the non-ground success set of a given program (if it is finite), and in the second, the set of calls and answers for a given program and initial goal (if they are finite).

2.1 Concrete evaluation

Figure 2 depicts the core of the XSB interpreter for both goal-dependent and goal-independent concrete evaluation. The interpreter assumes that each (abstracted) program clause

H^α :- $G_1^\alpha, \ldots, G_n^\alpha$

is represented as a fact of the form:

`program_clause`$(H^\alpha, [G_1^\alpha, \ldots, G_n^\alpha])$.

A query to the predicate `fact/1` with some (abstracted) goal argument triggers a fix-point computation that records information about the calls and success patterns which arise in the computations for that goal.

The `table fact/1` declaration instructs the evaluation strategy to record calls and answers for the predicate `fact/1` in the XSB tables, and to alter the control by substituting answer resolution for resolution against the program clauses as follows: Whenever a new call to `fact/1` is encountered, program clause resolution is used to derive answers for the call. Repeated occurrences of the same (up to renaming of variables) call suspend the computation and consume already derived answer substitutions from the appropriate call table. In this way, previously computed answers are reused during the course of the evaluation. Moreover, XSB's evaluation strategy guarantees that answer substitutions are returned exactly once to each call that waits to consume them; in this way, it essentially mimicks execution of a semi-naive bottom-up evaluation strategy (see [13, 42, 43] for a description of XSB's evaluation strategy).

The interpreter of Fig. 2 is applied to a program by adding one of the two entry-level predicates given in Fig. 3 for goal-dependent or goal-independent evaluation respectively. For goal-dependent evaluation, a query of the form `?- goal_dep`$(Goal^\alpha)$ insures that the abstracted initial goal $Goal^\alpha$ is executed against the interpreter in all possible ways (i.e. in the context of a failure driven loop). For goal-independent evaluation, there is no initial goal. In this case we know (see [27]) that the non-ground success set of the program is obtained by finding all solutions for the program's predicates in their most general form (i.e., where the arguments are distinct variables). A query of the form `?- goal_indep` achieves precisely this objective assuming that the predicate `is_pred/1` generates the most general form for each predicate defined in the given program. The definition for this predicate is given in Fig. 4. The predicate `same_skel/2` unifies its second argument with the most general version of the atom in its first argument. Tabling is used in the definition of `is_pred/1` to eliminate duplicate answers.

Figure 2 depicts the basic evaluation strategy applied in all our analyzers which are portrayed as simple enhancements and extensions of this basic interpreter. The simplicity of the code hides the complexity of the underlying fix-point computations and is a main factor in the elegance of our approach. Although this is an executable XSB specification, it is not intended to provide an implementation. Instead, analyses are implemented as program transformations, induced by the corresponding interpreters. While this approach is standard (and applied for example also in [17]), we note that in our case due to the stronger separation between control and abstraction, the specialization of the interpreters is particularly straightforward and often even trivial as illustrated below and in Section 3.5.

2.2 The domain Pos

One of the better known applications of abstract compilation is the analysis of groundness dependencies (mode analysis) using the abstract domain Pos (see e.g. [38, 20]), also called Prop in earlier works. Pos gets its name from the fact that the state of a program is described as a *positive* Boolean formula on the program's variables. Namely, a formula which obtains a positive value (i.e., `true`) under the unit assignment (i.e., the assignment of `true` to all of its variables). For example, the formula X expresses that the program variable X is (definitely) bound to a ground term, $X \leftrightarrow Y$ expresses that X is bound to a ground term if and only if Y is, so an eventual binding of X to a ground term will be propagated to Y and vice versa. Groundness analysis corresponds to the use of the so called *term norm* and abstracts a unification such as $X = [Y|Z]$ by the Boolean formula $X \leftrightarrow Y \wedge Z$. With abstractions such as these, Pos is able to optimally characterise both pure groundness, i.e. whether a variable is instantiated to a ground term during program execution, and the dependencies between the groundness of different variables, providing in this sense a clear example of relational analysis. The interested reader can find

```
:- table fact/1.
fact(Head) :- program_clause(Head,Body), prove(Body).
prove([]).
prove([G|Gs]) :- fact(G), prove(Gs).
```

Fig. 2. An interpreter for concrete evaluation.

```
goal_dep(Goal) :- fact(Goal), fail.   goal_indep :- is_pred(P), fact(P), fail.
goal_dep(_).                          goal_indep.
```

Fig. 3. Predicates for goal-dependent and goal-independent evaluation.

```
:- table is_pred/1.
is_pred(PredSkel) :- program_clause(Head,_), same_skel(Head,PredSkel).
same_skel(P1,P2) :- functor(P1,N,A), functor(P2,N,A).
```

Fig. 4. Supporting predicates for the goal-independent interpreters.

additional details on groundness analyses using the Pos domain in [17, 22, 44] where bottom-up evaluation with magic-sets and top-down evaluation with tabling are considered. We also note that, as shown recently in [30], the Pos domain can be interpreted as an *implicational* domain and can thus be used to capture any directional information for a *downward closed property*. In particular, it can capture dependencies between variables or argument positions, such as type dependencies [16], finiteness analysis [5], and quite recently binding-time analysis [11]. We note that the last of these analyses was implemented using the approach described in this article.

2.3 Example 1: Goal-dependent analysis with Pos

Figure 5 illustrates a Prolog program which rotates the elements of a list (on the left) and its corresponding abstraction (on the right) for groundness dependencies. The predicate *iff*(X, X_1, X_2) specifies the propositional formula $X \leftrightarrow X_1 \wedge X_2$ with the intended interpretation that X is ground if and only if X_1 and X_2 are. Consequently, a unification of the form `X = [Y|Z]` in the concrete program is replaced by a literal `iff(X,Y,Z)`. Similarly, a unification of the form `X = []` in the concrete program can simply be replaced by `iff(X)` which specifies that `X` is definitely bound to a ground term. For the purpose of this article it is sufficient to understand that the problem of analyzing the concrete program (on the left part of Fig. 5) has been reduced to the problem of applying concrete evaluation to the abstract program (on the right). In our example analysis, this means that mode information for the original program can be obtained by direct execution of the abstracted program. For issues concerning the correctness of this approach, refer to [38, 20, 17].

Using a suitable representation of the abstracted program from Fig. 5, the goal-dependent analysis for the initial query: `?- goal_dep(rotate(true,_))` results in leaving in the XSB tables the call/answer pairs shown in Fig. 6.

Call	Answer
`rotate(true,_)`	`rotate(true,true)`
`append(_,_,true)`	`append(true,true,true)`
`append(true,true,_)`	`append(true,true,true)`

Fig. 6. Groundness dependencies recorded in the XSB tables.

Rather than executing the groundness dependency analysis using the interpreter of Fig. 2 we can apply a simple program transformation which is induced by the interpreter. Note that this transformation is quite straightforward and more or less the standard way of removing the meta-interpretation overhead. In particular, note that the transformation does not depend on the results of the analysis. In the case of concrete tabulation and for the `rotate` example, the result of this transformation amounts to adding to the abstract program on the right part of Fig. 5 the program fragment shown in Fig. 7.

```
goal_dep(Goal) :- call(Goal), fail.
goal_dep(_).
:- table rotate/2, append/3.
```

Fig. 7. Addendum to the abstract `rotate` program of Fig. 5.

The resulting program can then be run directly in XSB, and when this is done, it terminates and leaves the results of the corresponding goal-dependent program analysis in the XSB tables. Note that for this type of analysis, the program transformation induced by the interpreter of Fig. 2 is trivial and basically involves running the original

Concrete rotate	Abstract rotate & Auxiliary predicates

<pre>rotate(Xs,Ys) :-
    append(As,Bs,Xs),
    append(Bs,As,Ys).
 	
append(Xs,Ys,Zs) :-
    Xs = [],
    Ys = Zs.
append(Xs,Ys,Zs) :-
    Xs = [X|Xs1],
    Zs = [X|Zs1],
    append(Xs1,Ys,Zs1).</pre>
	<pre>rotate(Xs,Ys) :- iff(true).
	    append(As,Bs,Xs),
	    append(Bs,As,Ys). iff(true,true).
	                             iff(false,false).
	append(Xs,Ys,Zs) :-
	    iff(Xs), iff(true,true,true).
	    iff(Ys,Zs). iff(false,true,false).
	append(Xs,Ys,Zs) :- iff(false,false,true).
	    iff(Xs,X,Xs1), iff(false,false,false).
	    iff(Zs,X,Zs1),
	    append(Xs1,Ys,Zs1).</pre>

Fig. 5. Corresponding concrete and abstract program.

logic program under a tabled execution strategy and in the context of a failure driven loop (so that all answers are computed).

2.4 Example 2: Goal-independent analysis with Pos

The goal-independent analysis using concrete evaluation and an abstraction for the Pos domain is equally simple. It can be achieved using the interpreter of Fig. 2 and the corresponding entry-level predicate from Fig. 3. An initial query of the form `?- goal_indep` triggers the computation of the (abstract) program's non-ground minimal model that remains in the XSB tables. More specifically, the analysis of a program P is obtained by querying the abstract program P^α for all the solutions of its set of most general queries to each predicate p/n in the alphabet of P; i.e., queries of the form $p(X_1,\ldots,X_n)$.

The cost of an analysis in our approach using XSB is related to the number of different call patterns encountered and tabled in the computations. A variation on the interpreter of Fig. 2 is given in Fig. 8 and implements what is known as the *most general calls* transformation; i.e., it enforces that each predicate is called only in its most general form hence reducing the size of the tables. In the interpreter, this is achieved by introducing a "fresh skeleton" as an argument in the call to the predicate `fact/1` from the recursive clause for `prove/1`.

In conventional analysis systems this type of evaluation often results in considerable speedups in analysis times. This is largely due to the lack of efficient tabling mechanisms in these systems. In our XSB implementation of Pos there is no clear gain in efficiency as indicated by the time and space performance shown in Table 1 later in this section.

The decision whether to apply most general call evaluation is left to the user of the system. For some domains it may lower the cost of analyses, yet result in a tradeoff incurring a loss of precision. Moreover, it prohibits the generation of information about call patterns. For a deeper discussion on these issues, see [14] where a similar technique is applied.

A goal-independent analysis using the interpreter of Fig. 8 and a suitable representation of the abstracted program from Fig. 5 results in the following groundness dependencies:

$$\begin{array}{rcl} \texttt{rotate}(X,Y) & \leftarrow & X \leftrightarrow Y \\ \texttt{append}(X,Y,Z) & \leftarrow & (X \wedge Y) \leftrightarrow Z \end{array}$$

which can be extracted from the XSB tables and exhibit the expected groundness dependencies for the two predicates.

2.5 An enhancement: collecting information on program points

The analysis of logic programs is often required to associate call patterns to program points: this is what is done in systems like PLAI, GAIA, and AMAI. This information enables a compiler to produce specialized versions of predicates according to their call patterns. The analysis for program points is obtained in our system using a simple program transformation. Consider the clause for `rotate/2` in Fig. 5. We can record the calls and answers to the individual calls to `append/3` by introducing a call to a "placebo" predicate `point/2` as in the following:

```
:- table point/2.
point(_,_).

rotate(X,Y) :-
     point(rotate:1,[A,B,X]), append(A,B,X),
     point(rotate:2,[B,A,Y]), append(B,A,Y),
     point(rotate:3,[X,Y]).
```

Predicate `point/2` has no effect on the computation as can be seen from its definition. However, tabling it facilitates the recording of information of call patterns in (possibly selected) program points. For example, using the adapted abstract `rotate` program we can now associate the call pattern $append(\_,\_,true)$ to the first program point (identified as `rotate:1`) of the clause for `rotate/2` and $append(true,true,\_)$ to the second program point (`rotate:2`). This, for instance, allows the specialization of the second call to `append/2` to a version which uses matching for its first argument instead of unification, and which has less general indexing code.

```
goal_indep :- is_pred(PredSkel), fact(PredSkel), fail.
goal_indep.

:- table fact/1.

fact(Head) :-
    program_clause(Head,Body), prove(Body).

prove([]).
prove([G|Gs]) :-
    same_skel(G,F), fact(F), F = G, prove(Gs).
```

Fig. 8. An interpreter for goal-independent concrete evaluation.

Collecting information on program points through this simple program transformation has the advantage that the interpreter of Fig. 2 requires no changes but for the addition of predicate `point/2`. The overhead for deriving program point information, depends on the number of such points and on the program analyzed. It is straightforward to provide the user of the analysis system with the flexibility to indicate specific program points for which information is required.

2.6 Performance on Pos

A note on the benchmarking environment: Throughout this article we use a set of by now "standard" benchmarks for the abstract interpretation of logic programs taken from [44] (excluding the smaller programs). The size of these programs ranges from 50 lines (pg) to ≈450 lines (read_o) and they can be considered as medium-sized logic programs when developing applications in a modular way. For all benchmarks in this article XSB version 1.6.1 was used on a SPARCstation 20 (100 MHz) under Solaris 2.5. Also, throughout this article all times are measured in seconds while space is measured in bytes for the total of subgoal and answer tables (as obtained by XSB's `statistics/0` builtin predicate). The times given are the total of the analysis and the collection time; i.e., the time to perform the corresponding analysis and extract the results from the XSB tables by backtracking through them. The measurements in all tables do not include the preprocessing time: the time required for program abstraction and compilation of the abstracted program.[2] Our experience with the approach advocated in this article is that times for program abstraction — especially for programs of this size — are usually too small to measure accurately and are clearly not a bottleneck. Also, times for abstract compilation are more or less independent of the domain used: they range from 0.2 to 1.3 seconds for these benchmarks. On the other hand, admittedly the compilation times sometimes occupy a significant percentage of the total cost of the analyses. This is partly due to the speed of the XSB compiler (for this reason [22] proposes the use of Prolog's `assert/1` for the abstracted program) and partly due to the efficiency of the analysis (cf. the numbers in Table 1). This effect of abstract compilation is noticeable in simple analyses, like e.g. those based on the Pos domain, but becomes almost negligible for analyses with "heavier" abstract domains (see e.g. the domain of abstract equation systems or those in [32, 9]) that are able to capture more information than just groundness dependencies that is arguably more useful for an optimizing compiler.

For all goal-dependent analyses in this article, the "natural" call to the main predicate of the benchmark program was used; for example in the Pos analysis of the read_o program the call used was $read\_tokens(true, \_)$ indicating that the top-level predicate of this benchmark will be called with a ground first argument and a possibly non-ground (or free) second argument.

Table 1 shows the time and space performance for the Pos analyses described above. The columns labeled GI, GI^{mgc}, GD, and GD^{pp} contain times for goal-independent analyses (with and without the most general calls transformation) and for goal-dependent analyses (with and without information for all program points), respectively. The last three columns of the table indicate the XSB table space requirements for the two goal-independent analyses, as well as their ratio.

The Pos domain is known to result in efficient analyses which derive modes with a high degree of precision [20, 44]. Our times compare well with times in the literature (see for example also [44, 17, 22, 3]). For this simple domain, GI^{mgc} does not have any relevant effect on time, but as expected, improves the space requirements of the analysis. When times are not too small to measure (like in e.g. the goal-dependent analysis of gabriel), we note an overhead of up to 100% when collecting information at *all* program points (GD vs. GD^{pp}). In actual analyses this functionality may be provided at only specific points of interest.

3 The Second Step: Abstract Compilation and Abstract Tabulation

This section shows how we can extend the concrete evaluation scheme of Fig. 8 by introducing two abstract operations: *abstract unification* and *abstract normalization*.

[2] With only few exceptions, this seems to be standard practice in the abstract interpretation literature; however, this is a factor that one needs to take into account in an optimizing compiler.

Table 1. Time and Space performance for Pos.

Program	Time				Space		
	GI	GI^{mgc}	GD	GD^{pp}	GI	GI^{mgc}	GI/GI^{mgc}
cs_o	0.21	0.40	0.00	0.00	42516	10384	4.09
disj_o	0.12	0.16	0.00	0.00	22572	6736	3.35
gabriel	0.02	0.02	0.00	0.01	18432	5372	3.43
kalah_o	0.07	0.08	0.00	0.01	43020	13792	3.12
peep	0.59	0.27	0.11	0.17	86092	13888	6.20
pg	0.02	0.01	0.00	0.00	11568	2704	4.28
plan	0.01	0.00	0.00	0.00	12708	3396	3.74
press1	0.14	0.11	0.06	0.10	55148	15140	3.64
read_o	0.08	0.07	0.04	0.08	44616	12376	3.60

The basic evaluation strategy remains the same except that concrete unification is replaced by abstract unification. Moreover, in this case we are tabling abstract atoms which have a domain-dependent syntactic representation. As we want to rely on the efficient underlying tabling mechanism to identify abstract atoms which are variants, and as this mechanism uses syntactic variance, we must ensure a canonical form representation of abstract atoms in the tables so that terms that are variants in the given abstract domain are also syntactic variants, and vice-versa (e.g. if the abstract domain is based on sets, one can order their syntactic representation).

Figure 9 illustrates the general form of an interpreter for analysis over an abstract domain. This interpreter, given suitable domain-dependent abstract unification algorithms, performs goal-independent analyses implicitly implementing the transformation for most general calls (as, for simplicity, do all our interpreters henceforth; but we note that e.g., goal-dependent interpreters for all analyses can also be obtained). The interpreter of Fig. 9 extends the one of Fig. 8 by three calls:

`abstract_unify(`F, G`)` (in `prove/1`) — applies abstract unification to a fact F which is looked up and returned from the XSB tables and a goal G from a program clause;

`abstract_normalize(`$Head, NormHead$`)` (in `fact/1`) — a deterministic predicate that ensures that the information is always recorded in a canonical form in the XSB tables for reasons explained above. Doing so, usually also facilitates the *abstract unification* operation in the body of `prove/1`; and

`same_skel(`$NormHead, Head$`)` (in `fact/1`) — makes a copy of the skeleton of the input argument to `fact/1` and allows indexing on the program clauses by backtracking only through the clauses of the same predicate. Note that this call is not strictly necessary, but is an obvious optimization: the code without the `same_skel/2` call has the same functionality but is inefficient since it then unnecessarily expands the search space and relies on the `abstract_normalize/2` call to disregard (by failing) those `Head`s that are not of the same predicate as the `NormHead` (the argument of `fact/1`).

To support a program analysis, the interpreter of Fig. 9 should be supplied with domain-dependent definitions for *abstract unification* and *abstract normalization*. Observe that by providing the trivial definitions:

```
abstract_unify(X,Y) :- X = Y.
abstract_normalize(X,Y) :- X = Y.
```

and unfolding the calls to these predicates the interpreter of Fig. 9 reduces to the functionality of the concrete interpreter of Fig. 8 (as explained the `same_skel/2` call is an optimization influencing only the performance, not the semantics of the interpreter). We describe two domains for program analysis using the abstract interpreter of Fig. 9.

3.1 Example 3: A domain for type dependencies

The first analysis we present in this category is a simple type dependency analysis for logic programs based on set-based (ACI) unification [2, 36], described in [18]. This domain characterizes the dependencies between the types of the terms that can be bound to a program's variables and generalizes the notion of *directional types* [1, 7]. The analysis is performed in two stages.

First, a given program P is abstracted to a corresponding *set logic program* P^τ in which each of the concrete terms from P is replaced by a (possibly non-ground) type expression. A type expression is a *flat set* (where nested sets are viewed as flattened, i.e., $\{a, \{a\}, b\} \equiv \{a, b\}$) of function symbols (and variables) which occur on specific designated paths in the tree representation of a concrete term.

As an example, consider a domain dealing with lists and binary trees. In this case the designated paths are defined to descend on the right branch of a `'.'/2` node and down the second and third branches of a `tree/3` node. Assuming the following mapping of constructors to type descriptions:

`'.'/2` $\rightarrow$ *cons*
`[]/0` $\rightarrow$ *nil*
`tree/3` $\rightarrow$ *tree*
`void/0` $\rightarrow$ *void*

```
goal_indep :- is_pred(PredSkel), fact(PredSkel), fail.
goal_indep.

:- table fact/1.

fact(NormHead) :-
    same_skel(NormHead,Head),
    program_clause(Head,Body), prove(Body),
    abstract_normalize(Head,NormHead).

prove([]).
prove([G|Gs]) :-
    same_skel(G,F), fact(F), abstract_unify(F,G), prove(Gs).
```

Fig. 9. A goal-independent interpreter that uses abstract domain-dependent operations.

the abstraction of the term `[1,2,3]` is $\{cons, nil\}$, the abstraction of `[1,2,3|X]` is $\{cons, X\}$, the abstraction of `tree(1,tree(2,void,void),tree(3,void,void))` is $\{tree, void\}$ and finally the abstraction of the "ill-typed" term `[1,2,3|tree(void,1,void)]` (in which the list is terminated by a tree instead of by a nil constructor) is $\{cons, tree, void\}$. Figure 10 illustrates a concrete program specifying the "inorder" relation on a binary tree (left) and its set-based type abstraction (right).

The second phase of the analysis applies an evaluation strategy which is similar to the concrete strategy with tabulation, except that standard unification is replaced by a set-based abstract unification implemented as ACI-unification. Namely, unification in the presence of an equality theory which is Associative, Commutative and Idempotent. This turns out to provide a correct and optimal type unification for the given domain. The specific details of the type analysis are beyond the scope of this article and the interested reader is referred to [18].

The result of a goal-independent type dependency analysis for the `inorder` example is illustrated in Fig. 11. As an example of how this result captures goal-independent information about the possible directed types for the given program, consider the following examples of input/output behaviours that can be derived from it:

1. an abstract query of the form:

   `inorder(`$\{tree, void\}$`,`$\{X\}$`)`

   is answered by $X = \{cons, nil\}$ indicating that a successful query to the concrete program in which the first argument of this predicate is a non-void tree, succeeds with the second argument bound to a non-empty nil-terminated list.
2. an abstract query of the form:

   `append(`$\{cons, nil\}$`,`$\{cons, nil\}$`,`$\{X\}$`)`

   is answered by $X = \{cons, nil\}$ indicating that a successful query to the concrete program in which the first two arguments of this predicate are non-empty nil-terminated lists, succeeds with the third argument bound to a nil-terminated list.
3. an abstract query of the form:

   `append(`$\{A\}$`,`$\{B\}$`,`$\{cons, nil\}$`)`

   is answered either by $A = \{nil\}, B = \{cons, nil\}$ (using the first abstract fact for append); or by $A = \{cons, nil\}, B = \{nil\}$ (using the second abstract fact for append); or by $A = \{cons, nil\}, B = \{cons, nil\}$ (using the second abstract fact for append). This indicates that a successful query of the concrete program with a non-empty nil-terminated list on the third argument of append, succeeds with the first two arguments bound to nil-terminated lists (at most one of which is empty). Observe that the set-based unification of $\{cons, nil\}$ with $\{cons, T\}$ succeeds either with $T = \{nil\}$ or with $T = \{cons, nil\}$.

```
inorder({void}, {nil}).
inorder({tree,void}, {cons,nil}).
append({nil}, {T}, {T}).
append({cons,nil}, {T}, {cons,T}).
```

Fig. 11. Goal-independent type dependencies for the `inorder` example.

The set-based unification for type expressions and the normal form representation for atoms containing type expressions were adapted to our framework from the implementation described in [18].

The example described in Fig. 10 deals with *monomorphic* type analysis. A possible variation of this analysis maps concrete function symbols to abstract type descriptors. For example, *cons* and *nil* might both be mapped to the descriptor *list* and numbers might all be mapped to a descriptor *num*. The technique has also been extended to consider *parametric polymorphism* [18]. As an example of the way the polymorphic analysis works, Fig. 12 illustrates the goal-dependent directed type annotations for a `quicksort` program which are derived using this analysis. The results shown are for a goal-dependent analysis and the example assumes that the `qsort/2` predicate is called with its first argument bound to a list of numbers. For example, the inferred type declaration for `qsort/2` indicates that the first argument is bound to a list of numbers in all calls to the predicate and that both arguments are bound to lists of numbers in all returns from

Concrete inorder	Abstract inorder

`inorder(void,[]).`
`inorder(tree(X,L,R),List) :-`
`    inorder(L,Ls), inorder(R,Rs),`
`    append(Ls,[X|Rs],List).`
 	
`append([],Ys,Ys).`
`append([X|Xs],Ys,[X|Zs]) :-`
`    append(Xs,Ys,Zs).`
	`inorder({void},{nil}).`
	`inorder({tree,L,R},{List}) :-`
	`    inorder({L},{Ls}), inorder({R},{Rs}),`
	`    append({Ls},{cons,Rs},{List}).`
	
	`append({nil},{Ys},{Ys}).`
	`append({cons,Xs},{Ys},{cons,Zs}) :-`
	`    append({Xs},{Ys},{Zs}).`

Fig. 10. Corresponding concrete and abstract programs for simple types.

type $qsort(\mathbf{list}(\mathbf{num}), \_) \longrightarrow qsort(\mathbf{list}(\mathbf{num}), \mathbf{list}(\mathbf{num}))$.

```
qsort([],[]).
qsort([X|Xs],Ys) :-
    partition(Xs,X,Ls,Bs),
    qsort(Ls,SLs), qsort(Bs,SBs),
    append(SLs,[X|SBs],Ys).
```

type $partition(\mathbf{list}(\mathbf{num}), \mathbf{num}, \_, \_) \longrightarrow partition(\mathbf{list}(\mathbf{num}), \mathbf{num}, \mathbf{list}(\mathbf{num}), \mathbf{list}(\mathbf{num}))$.

```
partition([],A,[],[]).
partition([X|Xs],N,[X|Ls],Bs) :-
    X =< N, partition(Xs,N,Ls,Bs).
partition([X|Xs],N,Ls,[X|Bs]) :-
    X > N, partition(Xs,N,Ls,Bs).
```

type $append(\mathbf{list}(\mathbf{num}), \mathbf{list}(\mathbf{num}), \_) \longrightarrow append(\mathbf{list}(\mathbf{num}), \mathbf{list}(\mathbf{num}), \mathbf{list}(\mathbf{num}))$.

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

Fig. 12. Goal-dependent directed types for a `quicksort` program.

the predicate. Experimental results of using this analysis in XSB are presented in Section 3.4.

3.2 Example 4: A domain for freeness and sharing

The second analysis using (the transformation induced by) the interpreter that incorporates abstract domain operations is the freeness and sharing analysis based on a pre-interpretation with the three constants `g,i,f` as described in [9]: the analysis is performed in a goal-independent way. Since this domain is used for the derivation of definite freeness and sharing, abstract answers are sets of atoms and must be maintained as such. Consequently, the number of different call and answer patterns is very high. This results in a high number of large tabling operations. Also abstract unification and normalization are computationally demanding. We shall come back to this analysis in Section 6 where we directly compare the implementation of this domain in XSB and in BIMprolog.

3.3 An optimization: tabling the abstract operations

In [26], the authors propose a technique termed 'Caching of Operations': remembering previous calls/answers to the abstract operations. The speedup obtained by this technique depends on the domain: for Pos it does not make sense, but for domains where the abstract operations are the most time-consuming ingredient, as for example for the type domain of this section, the improvement in time can be substantial. This optimization is straightforward to implement in XSB since it simply amounts to adding in the source program tabling declarations of the form:

```
:- table abstract_unify/2.
:- table abstract_normalize/2.
```

This illustrates the advantage of using a flexible general purpose logic programming language in which tabling can be easily switched on and off allowing experimentation on the right operations to table.

We also note that the *clause prefix dependency* optimization mentioned in [26], which amounts to reconsidering only clauses with goals for which a new answer has been derived, follows naturally from XSB's strategy for performing answer resolution.

3.4 Performance of the type analyses

Table 2 shows the timings of performing monomorphic directed type analyses based on the corresponding program transformations induced by the interpreter of Fig. 9 and the goal-dependent version of that interpreter.

Table 2. Times for Monomorphic Type Analysis.

Program	GI_{tt}	GI_t	GI	GD_{tt}
cs_o	0.19	0.37	10.12	0.03
disj_o	0.09	0.20	0.79	0.01
gabriel	0.05	0.05	0.09	0.12
kalah_o	0.15	0.22	0.74	0.16
peep	0.35	0.69	2.49	0.35
pg	0.10	0.17	0.83	0.09
plan	0.03	0.02	0.05	0.02
press1	1.67	4.44	18.98	0.99
read_o	0.51	0.92	4.68	0.05

The columns labeled with GI contain times for goal-independent analyses, while the GD label indicates goal-dependent analysis. The subscript tt means that abstract unification and normalization were both tabled, while the subscript t means that only abstract unification was tabled. Times for performing type analysis using XSB compare well with those published in the literature. Furthermore, the time savings obtained just by tabling the abstract operations in this abstract domain are impressive; from one up to almost two orders of magnitude difference for all non-trivial benchmarks of this set. As mentioned, these speedups are obtained with minimal effort in our setting showing clearly that XSB can offer much more to general purpose program analysis than just providing control and efficient storage mechanisms.

Although time savings due to tabling the abstract operations are shown for goal-independent analyses only, similar results were obtained for goal-dependent analyses in this domain. Naturally, tabling the abstract operations has an associated cost in space. This space trade-off is shown in Table 3 for the goal-independent analyses: the figures reported are in bytes and indicate the total memory needed for XSB's subgoal and answer tables.

Table 3. Space Trade-off for Tabling the Abstract Operations.

Program	GI_{tt}	GI_t	GI
cs_o	114028	24460	12744
disj_o	49236	19776	8616
gabriel	37072	13880	6944
kalah_o	100980	31100	18252
peep	171928	21128	14048
pg	53124	14324	5064
plan	21764	9400	4416
press1	473036	117788	21260
read_o	179532	42064	20524

3.5 The program transformation exemplified on the type analysis

Before finishing this section, we make explicit what we mean by the phrase: 'the program transformation induced by the interpreter'. Consider the monomorphic type dependency analysis of the `inorder` program of Fig. 10 and assume the mapping in the beginning of Section 3.1. A straightforward goal-independent analysis in our framework would be to supply to the interpreter of Fig. 9 the definitions of predicates `abstract_unify/2` and `abstract_normalize/2` and the abstracted version of the `inorder` program in the form shown in Fig. 13. The resulting program is directly executable in XSB. Assuming correct definitions of abstract operations, a query `?- goal_indep` will leave in the XSB tables the calls and answers shown in Fig. 14. However, one would ideally want to avoid the cost of the meta-interpretation and the calls to supporting builtins like `same_skel/2` or `is_pred/1`. As it turns out this is (almost) straightforward. For example, the code in Fig. 15 has the same functionality as the interpreter of Fig. 9 when specialized for the abstracted program clauses of the `inorder` program. The only "trick" is that an auxiliary predicate is used to perform the most general calls transformation that the interpreter achieves through the use of the `same_skel/2` call. Note that in the following, all calls to `append/3` pass through the Prolog version of append which in turn calls the tabled version (`append_1/3`) with fresh variables so that all recorded information to the table are for the most general call. Also notice the natural placement of the abstract operations in the code. It should be clear that this transformation, on which our implementations are based, was easily automated and is linear in the size of the program to be analyzed. The query `?- goal_indep` to this program when run under XSB will leave in the XSB tables the calls and answers shown in Fig. 16.

4 The Third Step: Alternative Collecting Strategies

The abstract domains underlying the program analyses described in the previous sections are based on a power set construction collecting a *set* of concrete or abstract atoms (or substitutions). For program analyses, working with sets can become costly and often a *single* abstract atom which represents (or approximates) the elements of the set can be used instead. For domains of this type, the *abstract least upper bound operation* is not set union and must be specified. In addition, analyses using abstract domains which have infinite ascending chains and require *widening* [21] for termination or analyses in which syntactic variance is replaced by forwards or backwards subsumption can be handled similarly. For example, in [17], the authors show that collecting (abstract) atoms which are not subsumed by those collected so far can reduce considerably the number of collected atoms as well as the analysis times. This type of collecting strategy is often referred to as *forward subsumption*. Similarly, *backwards subsumption* refers to the case when an atom is deleted from the set of collected atoms because it is subsumed by a new atom. Implementing an evaluation strategy based on an abstract least upper bound operation combines (and subsumes) the functionality of both forward and backwards subsumption.

As mentioned, working with sets of abstract substitutions is costly; the associated performance penalty is especially noticeable in systems which rely on the Prolog dynamic database to represent each result of the analysis and do not employ dedicated tabling/data structures for storing the analysis results. Also, control becomes harder when multiple abstract substitutions need to be returned to each call. For these reasons, most abstract interpretation systems for logic programs have as their default (and

```
program_clause(inorder({void},{nil}), []).
program_clause(inorder({tree,T1,T2},{T}),
               [inorder({T1},{T3}), inorder({T2},{T4}), append({T3},{cons,T4},{T})]).
program_clause(append({nil},{T},{T}), []).
program_clause(append({cons,T1},{T2},{cons,T3}), [append({T1},{T2},{T3})]).
```

Fig. 13. Program clauses of an abstracted `inorder` program.

Call	Answers
`fact(inorder(_,_))`	`fact(inorder({void},{nil}))` `fact(inorder({tree,void},{cons,nil}))`
`fact(append(_,_,_))`	`fact(append({nil},{T},{T}))` `fact(append({cons,nil},{T},{cons,T}))`

Fig. 14. Calls and answers left in the XSB tables when performing the analysis using the interpreter of Fig. 9.

```
goal_indep :- inorder(_,_), fail.
goal_indep :- append(_,_,_), fail.
goal_indep.

:- table append_1/3.

append_1(X1,X2,X3) :-
    {nil} = A1, {T} = A2, {T} = A3,
    abstract_normalize([A1,A2,A3],[X1,X2,X3]).
append_1(X1,X2,X3) :-
    {cons,T1} = A1, {T2} = A2, {cons,T3} = A3,
    append({T1},{T2},{T3}),
    abstract_normalize([A1,A2,A3],[X1,X2,X3]).

append(TS1,TS2,TS3) :-
    append_1(S1,S2,S3),
    abstract_unify([TS1,TS2,TS3], [S1,S2,S3]).

:- table inorder_1/2.

inorder_1(X1,X2) :-
    {void} = A1, {nil} = A2,
    abstract_normalize([A1,A2],[X1,X2]).
inorder_1(X1,X2) :-
    {tree,T1,T2} = A1, {T} = A2,
    inorder({T1},{T3}),
    inorder({T2},{T4}),
    append({T3},{cons,T4},{T}),
    abstract_normalize([A1,A2],[X1,X2]).

inorder(TS1,TS2) :-
    inorder_1(S1,S2),
    abstract_unify([TS1,TS2], [S1,S2]).
```

Fig. 15. Tabled logic program with the same functionality as the interpreter of Fig. 9 used for the program clauses of Fig. 13.

sometimes only) policy to maintain the analysis result by a *single* abstract atom and upon the derivation of a new abstract substitution to update the prior approximation by taking its domain-dependent lub with the previous.

XSB's efficient storage and control mechanism for fixpoint computations enable us to relax this restriction. In fact, in the current XSB version (1.6.1), calls do not take advantage of the fact that at a given point there will only be one answer for them to consume. Instead the general scheme of returning to them a set of answers by backtracking is used. As this imposes an overhead which is easily avoidable by a small change in the underlying abstract machine [42], we expect the performance of our lub-based program analyses to improve in future versions of XSB which will optimize single answer scheduling. In any case, the user of our framework has the flexibility to decide on how the analysis information will be collected and easily experiment with the trade-offs between using a lub-based answer collection strategy or not (see also [45] for a discussion and an extensive experimental evaluation of the trade-offs that are involved in this choice). Adapting our interpreter to consider an abstract lub operation is straightforward and the resulting interpreter is shown in Fig. 17.

The abstract interpreter of Fig. 17 is almost identical to that of Fig. 9. The only difference is in the definition of `fact/1` where the call to `abstract_normalize/2` is replaced by a call to `abstract_lub/2`. The mechanism is as follows: whenever a new `Head` (answer) pattern is obtained by proving the `Body` of a program clause, the abstract least upper bound of the new and the existing answer for the same call is computed. If the result of this abstract lub operation, `Lub`, is different than the current answer, the existing answer is deleted and the `Lub` is inserted, effectively replacing the current answer. This way, every call has at most one answer in its table. The XSB code for performing the lub operation in goal-independent analyses where all calls are in their most general form is given as Fig. 18. The call `get_answer(ExistingAnswer,Ref)` uses the XSB builtins `get_call/3` and `get_returns/3` which are accessible from the Prolog level to retrieve the existing answer of the call (which is to be lubbed with the new one) from the XSB tables (`get_returns/3` fails if no answers are present). The extra argument `Ref` is a reference identifying the existing answer which — if needed — is later used by the XSB builtin predicate `delete_return/1` to remove the old answer from the table. The predicate `abstract_lub/3`

Call	Answers
`inorder_1(_,_)`	`inorder_1(`$\{void\}$`,`$\{nil\}$`)` `inorder_1(`$\{tree, void\}$`,`$\{cons, nil\}$`)`
`append_1(_,_,_)`	`append_1(`$\{nil\}$`,`$\{T\}$`,`$\{T\}$`)` `append_1(`$\{cons, nil\}$`,`$\{T\}$`,`$\{cons, T\}$`)`

Fig. 16. Calls and answers left in the XSB tables when performing the analysis using the tabled logic program of Fig. 15.

```
goal_indep :- is_pred(PredSkel), fact(PredSkel), fail.
goal_indep.

:- table fact/1.

fact(LubbedHead) :-
    same_skel(LubbedHead,Head),
    program_clause(Head,Body), prove(Body),
    abstract_lub(Head,LubbedHead).

prove([]).
prove([G|Gs]) :-
    same_skel(G,F), fact(F), abstract_unify(F,G), prove(Gs).
```

Fig. 17. A goal-independent abstract interpreter with lub.

```
abstract_lub(NewAnswer, Lub) :-
    abstract_normalize(NewAnswer, NormNewAnswer),
    same_skel(NewAnswer, ExistingAnswer),     /* Creates the appropriate call */
    ( get_answer(ExistingAnswer, Ref) ->
        abstract_lub(NormNewAnswer, AnswerCopy, Lub),
        ( variant(ExistingAnswer, Lub) -> true ; delete_return(Ref) )
    ; Lub = NormNewAnswer     /* no answer in table yet */
    ).

get_answer(Answer,Ref) :- get_call(Answer,C,R), get_returns(C,R,Ref).
```

Fig. 18. Code for lubbing answers for analyses with most general calls.

is user-supplied and specifies the abstract least upper bound operation for the given domain. We also assume that it returns its result (`Lub`) in a canonical (i.e., normalized) form.

We have applied the (transformation induced by the) interpreter of Fig. 17 to the domain of *sharing* analysis. A Prolog implementation (within a bottom-up framework) of the analysis using ACI-unification is described in [19]. The analysis produces similar results with the sharing analysis of Jacobs and Langen [32] implemented in PLAI. Times for goal-independent sharing analysis using this domain are shown in Table 4 (XSB version 1.6.1 which was used for the benchmarks was crashing on press1).

5 The Final Step: Beyond Abstract Compilation

So far all of the described interpreters perform what is usually understood as abstract interpretation based on abstract compilation: they first perform a program abstraction (usually to non-ground terms) and then use the variables in these abstracted terms to collect information about the abstract substitutions. Many analyses of logic-based languages go beyond that; they use abstract domains in which abstract terms (and substitutions) do not get more and more instantiated, but rather change their values from one term to the other during abstract execution. In addition to this, an abstract substitution might, apart from being a function from variables to elements of the abstract domain, also contain some information about relations of these variables. A typical example of such an abstract domain is the Sharing domain of Jacobs and Langen [32].

Our framework is easily extended to support general abstract interpretation as well. An interpreter that provides this functionality is shown in Fig. 19. Its main change is in the representation of terms that are recorded in the tables. Instead of using the variables of the abstract calls to keep track of the abstract substitutions that are collected during execution, terms can be thought as represented by *abstract pairs* of the form

$$\texttt{ap}(Skel, AbsSubst)$$

where:

Skel is an abstract term skeleton; i.e., it identifies the predicate symbol and contains a description of the variables in the term, and

Table 4. Performance of Goal-Independent Sharing Analysis in XSB.

cs_o	disj_o	gabriel	kalah_o	peep	pg	plan	press1	read_o
0.27	0.33	0.16	0.27	0.62	0.07	0.23	†	1.09

```
goal_indep :-
    is_pred(Skel), abstract_epsilon(Skel,Epsilon), fact(Skel,Epsilon,_), fail.
goal_indep.

:- table fact/3.

fact(Call,CallSubst,AnswerSubst) :-
    same_skel(Call,Head),
    program_clause(Head,Body),
    abstract_epsilon((Head:-Body),Epsilon),
    abstract_unify(ap(Call,CallSubst), ap(Head,Epsilon), BodyEntrySubst),
    prove(Body,BodyEntrySubst,BodyExitSubst),
    abstract_unify(ap(Head,BodyExitSubst), ap(Call,CallSubst), CallExitSubst),
    abstract_normalize(CallExitSubst,AnswerSubst).

prove([],In,Out) :- Out = In.
prove([G|Gs],In,Out) :-
    same_skel(G,F),
    abstract_epsilon(F,Epsilon),
    abstract_unify(ap(G,In), ap(F,Epsilon), CallSubst), /* project */
    fact(F,CallSubst,AnswerSubst),
    abstract_unify(ap(F,AnswerSubst), ap(G,In), Mid),   /* extend */
    prove(Gs,Mid,Out).
```

Fig. 19. A goal-independent interpreter which explicitly keeps abstract substitutions.

AbsSubst contains the representation of the abstract substitution for the variables in the skeleton.

In this representation, for each term, the abstract domain needs to provide the empty substitution whose form might depend on the variables (possibly in a ground representation) in the skeleton of the term. This is the purpose of the user-supplied predicate `abstract_epsilon/2`: given a abstract term in its first argument, it unifies its second argument with the representation of the empty abstract substitution for the variables in the term. This domain-dependent representation of the empty abstract substitution also imposes a minimal change in the definition of the `goal_indep/0` entry predicate: most general calls for predicates in the program to be analyzed have their abstract substitution explicitly constructed by calling `abstract_epsilon/2`. A similar change occurs in `prove/3`.

For a given abstract term skeleton we need to keep track of the call and answer patterns that are encountered during abstract execution. To do so, the `fact/1` and `prove/1` predicates of our previous interpreters are extended by two arguments that keep track of the abstract substitutions on entry and exit. Consider the information recorded in the tables at the end of the execution of a query `?- goal_indep` against the interpreter of Fig. 19. `fact/3` is the only tabled predicate: its first two arguments are input arguments and when viewed as an abstract pair they identify the abstract calls that are encountered during the fix-point computation; similarly the first and third argument combination keeps track of the (normalized forms of the) abstract answers that the analysis has produced.

The only remaining change of this interpreter that requires explanation is the use of the *abstract unification* operation. Abstract unification plays an important role in this setting. On unifying two abstract terms represented as abstract pairs, it produces an abstract substitution that is *projected* to the variables of the second abstract pair. An instance of the use of this functionality can be seen in the "wrapping" of the call to `fact/3` in the recursive clause of `prove/3` by abstract unification operations: The first unification projects the `In` abstract substitution to the set of variables of the `F` skeleton that is the call, while the second unification extends `In` by the variables in the `AnswerSubst` used in `G` to produce the `Mid` answer substitution for the recursive call to `prove/3`.

6 One Last Experiment

This article is not intended to provide a performance comparison between existing abstract interpretation implementations (e.g., PLAI, GAIA, AMAI) and our proposed framework for general program analysis. It is also not intended to promote the specific types of analysis chosen as examples. Instead, it is intended to fully describe our methodology of using a general purpose logic programming system with tabling for semantics-based

Table 5. XSB - BIMprolog comparison for **gif**.

Program	*XSB*			*BIM*		
	Time	Table Ops	Abs%	Time	Table Ops	Abs%
gabriel	4.55	346	95	2.63	571	54
kalah_o	3.30	251	99	10.20	362	90
peep	205.10	7568	96	509.00	21543	45
pg	0.41	111	85	0.58	138	67
plan	0.19	77	90	0.25	111	48
press1	1223.50	386769	93	1549.50	499748	28
qsort	0.09	28	66	0.07	39	42
queens	0.09	21	77	0.11	32	36
read_o	85.80	5918	96	85.80	10452	83

program analyses of logic-based languages and demonstrate the benefits of doing so. The timings in Tables 1, 2, and 4 indicate clearly that for a variety of both simple and more complicated domains, XSB results in analyses with very reasonable performances which compare well with figures quoted in the literature [14, 17, 22, 35]. Also, as reported in [22, 43], XSB scales well for fix-point computations over large datasets, indicating that analysis of large programs will not suffer due to inefficiency of the underlying control strategy. On the other hand, the analysis of large programs might suffer from inefficiencies due to the particular abstract domain.

Table 5 compares the goal-independent **gif** analysis using an XSB implementation with a special purpose abstract interpreter in a commercial Prolog system. The Prolog implementation was that developed for the needs of [9] on top of BIMprolog (release 4.1.0).[3] For control it uses a naive bottom-up operator with an evaluation order based on strongly connected components (as described in [17]). Both implementations derive exactly the same information for all programs.

This experiment serves two purposes:

1. it provides insight regarding the efficiency of analyses in XSB; and
2. it serves as a basis to anticipate the expected effect of combining a strong Prolog implementation with an advanced tabling mechanism.

Admittedly, analysis times for the **gif** domain are not always satisfactory; and there are much faster ways of deriving freeness and sharing information for logic programs. However, it is precisely for this reason that the **gif** domain is particularly suitable for the purposes of our experiment: Its abstract operations are complex and computationally expensive and the generated analyses are heavily demanding on storage/table access facilities.

In Table 5, besides the analysis Time (in seconds), the columns for *XSB* and for *BIM* indicate the more interesting measurements of:

Table Ops the number of tabling operations, i.e., the number of attempts to insert answers in the XSB tables or to conditionally assert them in BIM, and

Abs% the percentage of time that was spent in the abstract operations (which were not tabled in this experiment) — it is reasonable to consider the rest of the time to be spent in tabling operations.

Because the *BIM* implementation is based on a naive evaluation strategy, the Table Ops number for *BIM* is always higher than for *XSB*. Although the basic execution speed of XSB is about a factor of 4 slower than BIM[4], *XSB* is from faster (press1) to considerably faster (kalah_o, peep) on the large benchmarks. There are two reasons for that: XSB does less tabling operations and more importantly, the tabling operations themselves are much more efficient due to the superior data structures that XSB uses for tabling [41]. Most striking is the result for the press1 program, where XSB spends *13 times less* in tabling operations than BIM, although it does only some 20% less tabling operations (tabling in BIM is implemented with dynamic database predicates as in other Prolog-based abstract interpretation frameworks). It should be noted that for a fixed program (i.e., dissallowing the possibility of program transformations), *XSB* performs the minimal number of answer check/insert operations possible (as in an optimal semi-naive implementation of T_P): every combination of derived answers is used in every clause body exactly once. This is apparent from the implementation of SLG resolution in XSB, but we have also checked empirically that this is the case.

From the column *XSB* one can deduce the expected performance of a system which combines the strengths of both *BIM* and *XSB*; i.e. of a system that would integrate a fast Prolog engine (such as that of BIMprolog) with an implementation of an efficient tabling mechanism (such as XSB's). The expected performance is obtained by assuming that the time *XSB* spends in abstract operations — i.e. in the part of the analysis executed by the Prolog engine — is reduced by a factor of 3: this would result in a speedup of the analysis by *XSB* by a factor close to 3 for the larger benchmarks (if the very small benchmarks are removed, it will be for all our benchmarks close to 3), because *XSB* spends most of its time in the abstract operations. The speedup compared to *BIM* of such a combined system, is often even higher.

[3] Currently called MasterProLog.

[4] XSB is an emulator; BIMprolog generates native code.

It is realistic to assume that such a combination of strengths can be realized in the near future. The experiment indicates that the combination of a high performance logic programming system with tabulation is potentially far better than any abstract interpretation framework which can be handcrafted in a Prolog system without a built-in tabling mechanism.

7 Discussion

This article advocates the use of a logic programming system with tabling in general — and XSB in particular — as a tool for implementing program analyses. The main advantages of our approach are its *simplicity* and *elegance* which are best illustrated by our almost trivial little meta-interpreters. We stress again that the interpreters mainly serve as pedagogical vehicles to develop the step-by-step introduction of ingredients and usual optimizations of static analysis of logic programs; however, as shown in Section 3.5, the corresponding transformation induced by these interpreters is obtained in a straightforward way. We have shown that our approach easily supports a wide range of techniques applied in a variety of different semantics-based program analyzers (program points, most-general-calls, tabling of abstract operations, enhanced local control strategies, keeping abstract substitutions explicitly). Notice that all these techniques are orthogonal in our framework. This flexibility derives from the fact that the central elements related to an efficient and terminating evaluation strategy are already provided by the underlying XSB system. As expected from XSB's optimal control and superior tabling performance, and confirmed in the experimental part of this article, simplicity and elegance are obtained without compromising *efficiency* and thus having to resort to a low-level language such as C for specifying the abstract operations.

Most of the analyses considered in this article have a particular structure involving a combination of program abstraction and abstract interpretation. However, our approach allows for analyses without program abstraction: see [34] for a recent such experiment. We plan to integrate the techniques illustrated in this article to provide a flexible and general purpose tool for program analysis. The user of the system will be able to specify an abstract domain (and its operations), importing for example specifications originally designed within Prolog-based systems such as PLAI and AMAI, as well as to tune into the analysis options most appropriate for the given task.

Apart from simplicity and good performance, there are other reasons to use a general purpose logic programming system with tabulation as a tool for program analysis. For example, improvements to XSB on the level of control and tabling will make our analyses even more efficient. We believe that these improvements are more likely to happen in XSB, which is a general purpose system with a lot of effort currently being invested in it, than in special purpose tools used only for program analysis which are usually developed by small teams of researchers. As a specific examples of improvements currently underway, we mention the alternative scheduling strategies that have already been developed [28] (but have not yet been flexibly integrated in the system). It is known that different scheduling strategies can have different performance characteristics, with none of them being uniformly better that the others. When integrated at the predicate level, they will offer a selection of different control strategies far more diverse than that of any other program analysis system. Another useful improvement for analyses which are *lub*-based are enhancements to the tabling mechanism for single answers.

In summary, we hold that the use of a logic programming language with tabulation for abstract interpretation offers new chances to the field of program analysis, because analyses are significantly easier to design and modify and, due to the efficiency of fix-point computations in XSB and the underlying data structures, can be at least as efficient as in other frameworks. At the same time, this result also indicates that there is a need for more and faster Prolog systems with tabulation.

Acknowledgements. Sincere thanks to Gerda Janssens for valuable discussions; to Vitaly Lagoon for helping with the type and sharing analyses; and to Prasad Rao of the XSB group for his prompt handling of an XSB anomaly (i.e., bug) our analyzers were confronted with. This work was conducted in part under the auspices of the EC Collaborative project ISC-IL-90 PARFORCE while the first and the third author were visiting K.U. Leuven and Ben-Gurion University respectively. The third author is supported by the Fund for Scientific Research (FWO Flanders, Belgium) and by a K.U. Leuven junior scientist fellowship.

References

1. A. Aiken and T. K. Lakshman. Directional Type Checking of Logic Programs. In B. Le Charlier, editor, *Proceedings of the First International Symposium on Static Analysis*, number 864 in LNCS, pages 43–60, Namur, Belgium, Sept. 1994. Springer-Verlag.
2. F. Baader and J. H. Siekmann. Unification Theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *The Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 41–125. Oxford University Press, Oxford, UK, 1994.
3. R. Bagnara. A Reactive Implementation of *Pos* using ROBDDs. In H. Kuchen and S. D. Swierstra, editors, *Proceedings of the 8th International Symposium on PLILP*, number 1140 in LNCS, pages 107–121, Aachen, Germany, Sept. 1996. Springer-Verlag.
4. R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, Jan. 1993.

5. P. A. Bigot, S. K. Debray, and K. Marriott. Understanding Finiteness Analysis using Abstract Interpretation. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 735–749, Washington, USA, Oct. 1992. The MIT Press.
6. A. Bossi, M. Gabrielli, G. Levi, and M. Martelli. The S-Semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19/20:149–197, May/July 1994.
7. J. Boye and J. Małuszyński. Directional Types and the Annotation Method. *Journal of Logic Programming*, 33(3):179–220, Dec. 1997.
8. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, Feb. 1991.
9. M. Bruynooghe, B. Demoen, D. Boulanger, M. Denecker, and A. Mulkers. A Freeness and Sharing Analysis of Logic Programs Based on a Pre-interpretation. In R. Cousot and D. A. Schmidt, editors, *Proceedings of the Third International Symposium on Static Analysis*, number 1145 in LNCS, pages 128–142, Aachen, Germany, Sept. 1996. Springer-Verlag.
10. M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, Sept. 1987. IEEE, Computer Society Press.
11. M. Bruynooghe, M. Leuschel, and K. Sagonas. A Polyvariant Binding Time Analysis for Off-line Partial Deduction. In C. Hankin, editor, *Programming Languages and Systems: Proceedings of the 7th European Symposium on Programming*, number 1381 in LNCS, pages 27–41, Lisbon, Portugal, Mar./Apr. 1998. Springer-Verlag.
12. F. Bueno, M. García de la Banda, and M. Hermenegildo. The PLAI Abstract Interpretation System. Technical report, CLIP2/94.0, Computer Science Dept., Technical U. of Madrid (UPM), Feb. 1994.
13. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, Jan. 1996.
14. M. Codish, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Exploiting Goal Independence in the Analysis of Logic Programs. *Journal of Logic Programming*, 32(3):247–262, Sept. 1997.
15. M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. *Theoretical Computer Science*, 124(1):93–125, Feb. 1994.
16. M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In B. Le Charlier, editor, *Proceedings of the First International Symposium on Static Analysis*, number 864 in LNCS, pages 281–297, Namur, Belgium, Sept. 1994. Springer-Verlag.
17. M. Codish and B. Demoen. Analysing Logic Programs using "Prop"-ositional Logic Programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, Dec. 1995.
18. M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Computer Science Press, June 1996. Extended version to appear in *Theoretical Computer Science*.
19. M. Codish, V. Lagoon, and F. Bueno. An Algebraic Approach to Sharing Analysis of Logic Programs. In P. Van Henteryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis*, number 1302 in LNCS, pages 68–82, Paris, France, Sept. 1997. Springer-Verlag. Full version at `http://www.cs.bgu.ac.il/~mcodish/papers.html`.
20. A. Cortesi, G. Filé, and W. Winsborough. Optimal Groundness Analysis using Propositional Logic. *Journal of Logic Programming*, 27(2):137–167, May 1996.
21. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, Jan. 1977. ACM.
22. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–126, Philadelphia, Pennsylvania, May 1996. ACM Press.
23. S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, July 1989.
24. S. K. Debray and D. S. Warren. Detection and Optimization of Functional Computations in Prolog. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in LNCS, pages 490–504, London, July 1986. Springer-Verlag.
25. S. K. Debray and D. S. Warren. Automatic Mode Inference for Logic Programs. *Journal of Logic Programming*, 5(3):207–229, Sept. 1988.
26. V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and their Experimental Evaluation. *Software Practice and Experience*, 23(4):419–459, Apr. 1993.
27. M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
28. J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First Strategies: Improving Tabled Logic Programs through Alternative Scheduling. *Journal of Functional and Logic Programming*, 1998(3), Apr. 1998.
29. J. P. Gallagher and D. A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margarita, Italy, June 1994. The MIT Press.
30. R. Giacobazzi and F. Scozzari. Intuitionistic Implication in Abstract Interpretation. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of Ninth International Symposium on Programming Languages, Implementations, Logics and Programs PLILP'97*, number 1292 in LNCS, pages 175–189. Springer-Verlag, Sept. 1997.
31. M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–366, Aug. 1992.

32. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, July 1992.
33. G. Janssens, M. Bruynooghe, and V. Dumortier. A Blueprint for an Abstract Machine for Abstract Interpretation of (Constraint) Logic Programs. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 336–350, Portland, Oregon, Dec. 1995. The MIT Press.
34. G. Janssens and K. Sagonas. On the Use of Tabling for Abstract Interpretation: An Experiment with Abstract Equation Systems. In *Proceedings of TAPD-98: Tabulation in Parsing and Deduction*, pages 118–126, Paris, France, Apr. 1998.
35. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, Jan. 1994.
36. P. Lincoln and J. Christian. Adventures in Associative-Commutative Unification. *Journal of Symbolic Computation*, 8(1/2):217–240, July/Aug. 1989. Also appears in *Unification*, edited by Claude Kirchner (Academic Press, 1990), pages 393–416.
37. K. Marriott and H. Søndergaard. Bottom-Up Dataflow Analysis of Normal Logic Programs. *Journal of Logic Programming*, 13(2 & 3):181–204, July 1992.
38. K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, Mar.–Dec. 1993.
39. C. Mellish. Abstract Interpretation of Prolog Programs. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–474, London, July 1986. Springer-Verlag.
40. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency using Abstract Interpretation. *Journal of Logic Programming*, 13(2 & 3):315–347, July 1992.
41. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 687–711, Tokyo, Japan, June 1995. The MIT Press. Extended version to appear in Journal of Logic Programming.
42. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3), May 1998.
43. K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM Press.
44. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the Domain Prop. *Journal of Logic Programming*, 23(3):237–278, June 1995.
45. P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The Impact of Granularity in Abstract Interpretation of Prolog. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis (WSA'93)*, number 724 in LNCS, pages 1–14, Padova, Italy, Sept. 1993. Springer-Verlag.
46. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.
47. D. S. Warren. Memoing for Logic Programs. *Communications of the ACM*, 35(3):93–111, Mar. 1992.