# Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study*

Steven Dawson† dawson@csl.sri.com
C.R. Ramakrishnan cram@cs.sunysb.edu
David S. Warren warren@cs.sunysb.edu

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

**Abstract**

Many analysis problems can be cast in the form of evaluating minimal models of a logic program. Although such formulations are appealing due to their simplicity and declarativeness, they have not been widely used in practice because, either existing logic programming systems do not guarantee completeness, or those that do have been viewed as too inefficient for integration into a compiler. The objective of this paper is to re-examine this issue in the context of recent advances in implementation technologies of logic programming systems.

We find that such declarative formulations can indeed be used in practical systems, when combined with the appropriate tool for evaluation. We use *existing* formulations of analysis problems — groundness analysis of logic programs, and strictness analysis of functional programs — in this case study, and the XSB system, a table-based logic programming system, as the evaluation tool of choice. We give experimental evidence that the resultant groundness and strictness analysis systems are practical in terms of both time and space. In terms of implementation effort, the analyzers took less than 2 man-weeks (in total), to develop, optimize and evaluate. The analyzer itself consists of about 100 lines of tabled Prolog code and the entire system, including the components to read and preprocess input programs and to collect the analysis results, consists of about 500 lines of code.

## 1 Introduction

Modern compilers perform a variety of analyses to gather information about the input program for use in program optimization. For instance, in imperative languages, inter- as well as intra- procedural dataflow analysis techniques collect information about variable usage, which is then used to optimize storage allocation (see [1]); abstract interpretation techniques (*e.g.*, [12]) detect loop invariants, optimize array bounds checking and promote code hoisting. In functional programming, a large number of analyses have been recognized as useful: strictness analysis [26], binding time analysis [20] and update analysis [19], to name just a few. Compiler writers for logic languages have long realized the importance of information such as groundness, freeness and types for performing routine as well as sophisticated optimizations [13, 41]. Many of these analyses can be cast in the form of evaluating the minimal model of a (constraint) logic program (see *e.g.*, [39, 9]). Although such formulations are appealing due to their simplicity and declarativeness, they have seldom been used in practice. The main reason for this gap is that, until very recently, there have been few logic programming tools that offered the two fundamental ingredients needed to make such formulations usable in a practical compiler: completeness (the ability to find the minimal model whenever one exists) and efficiency. Lacking this, such formulations have been deemed impractical.

Recent developments in the area of logic program implementation have made available a few systems that guarantee completeness while offering good performance. Efficient systems such as Coral [29], which are based on complete (bottom-up) evaluation strategies, have emerged from the deductive database community. The performance of these systems has been considerably improved upon by the XSB system [36]. XSB combines the benefits of the WAM technology [2] — a well-optimized engine developed primarily for Prolog program execution — with the completeness guaranteed by the use of extension tables. Although resolution methods such as OLDT [38], and SLG [7] (on which XSB is based) have been known for some time, it is only now that the technology is mature enough to offer an implementation with good raw performance [35]. Availability of such systems forces one to re-evaluate the question: can one obtain *practical* implementations of program analysis from a logical formulation with *minimal* programming effort?

We find that it is indeed possible to use declarative formulations in practical systems, when combined judiciously with the appropriate tools of evaluation. In this paper, we describe the implementation of logical formulations of two existing analysis techniques: groundness analysis of logic programs using the *Prop*-domain [23] and strictness analysis of functional programs based on demand propagation [37]. We use the XSB system, currently the fastest system available that guarantees completeness, as the evaluation tool.

*This work was supported in part by NSF grants CCR-9404921, CDA-9303181, CDA-9504275 and INT-9314412.

†Current Address: SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025-3493.

Our experiments, results of which are presented here, provide evidence of the practicality of the resulting groundness and strictness analysis systems: the total times for analysis (including preprocessing, evaluation and result collection) are significantly less than the compilation time. Moreover, the groundness analyzer is competitive with the fastest known implementation of the same analysis constructed using special-purpose analysis tools. It should, however, be noted that to obtain efficient analyzers, the logical rules were coded so as to take advantage of the underlying evaluation mechanism. The techniques used to encode the rules, although relatively straightforward, are noteworthy since they often result in significant performance gains.

In both groundness and strictness analyses, we find that preprocessing dominates the cost of analysis. Most of the published literature on program analysis ignores preprocessing costs, but our experiments suggest that the evaluation times are small enough that reducing the preprocessing times is a problem of significant practical importance. Apart from analysis time, the analyzers have very low space requirements — an important metric that impacts the practicality of the system. In terms of programming effort, the analyzers took less than 2 man-weeks to build. The entire system consists of about 500 lines of tabled Prolog code, with the code for the analysis phase accounting for less than 100 lines. It is very encouraging to find that an easily verifiable logical formulation is no longer confined to being just a prototype but can be directly used to build a practical system.

As of the time of this writing, we have but preliminary experimental evidence about the practicality of performing dataflow analysis of imperative programs using a general purpose logic programming system. The shape analysis for programs with destructive updates [33] has been implemented in XSB very recently, and exhibits good performance. However, the practicality of this implementation (in terms of how it compares to special-purpose dataflow analyzers) remains to be established. Nevertheless, the figures in [31] indicating the relative performance of dataflow analysis implemented using Coral with respect to that in C, and the relative performance of shape analysis in Coral and XSB lead us to believe that the practicality results will carry over to dataflow analysis of imperative programs as well. We discuss this issue at greater length in Section 7.

All analyses modeled in this paper are performed over finite abstract domains. Analysis over infinite domains will require support for on-the-fly approximation operations such as *widening* [12]. Moreover, the system used for this case study, XSB, currently handles term-equality constraints only. The mechanisms needed to make the implementations of analyses based on infinite domains and/or other constraint domains as straightforward and efficient as those reported in this paper are discussed in Section 6.

The rest of the paper is organized as follows. In the next section we discuss the advantages of logic-based formulation of program analysis and provide a brief overview of tabled evaluation and its implementation. The formulation of groundness and strictness analyses forms the topic of Section 3. In Section 4 we describe the implementation and performance of the analyses. In Section 5 we discuss the implementation of analyses using non-enumerative representations, to illustrate the power of combining tabling with meta-programming. Optimizations and extensions to the system are described in Section 6. We compare this work with current literature in Section 7. Concluding remarks appear in Section 8.

## 2 Background

Analysis methods are usually described in terms of semantic equations whose fix point represents the program property under consideration. These semantic equations can be viewed as a (constraint) logic program, where each equation is translated into a Horn clause. Many program analysis methods have been explicitly formulated as logic programs. For instance, a formulation of flow analysis problems as a logic program is presented in [39, pages 984–987]. Not surprisingly, there have been many proposals to formulate logic program analysis in this form (see, *e.g.*, [9]); some carry this approach a step further, to the level of implementation (*e.g.*, [8]). The main advantage of this approach is that it separates the concepts of formulation from the implementation details, yielding the following benefits:

- The logic program is a straightforward representation of the abstract semantic equations. Hence, once the soundness of the analysis is established (at the level of the semantic equations), and given a sound and complete implementation to evaluate the logic program, the overall correctness of the analysis is readily established.
- The interface between the formulation and its implementation is a well-established language. This allows the implementor to choose *any* complete logic programming system as the evaluation engine.
- Active research in the logic programming area to enhance the power and performance of these systems immediately benefits the implementation of analyses.

There are two strategies for complete evaluation of logic programs that are available in existing systems: *bottom-up evaluation* (used, *e.g.*, in Coral), and *top-down tabled evaluation* (used, *e.g.*, in XSB). Below, we briefly review the mechanism of tabled evaluation, the evaluation method used in this case study. For an exposition of bottom-up analysis methods see [39].

**Tabled Evaluation** At a high level, a top-down, tabled evaluation engine 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.

**The XSB System** The tabled evaluation engine of XSB system provides an efficient fix-point algorithm that terminates for finite domains. This means that the system can be directly used to compute fix points for Galois-connection

[1]In XSB, the presence of a subgoal in the table is tested in the engine by searching for a *variant* of the subgoal in the table. Two terms $t_1$ and $t_2$ are variants of each other if they are identical up to variable naming. Only unique answers are entered in the table, and duplicates are filtered out using variant checks.

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 — a feature that is necessary for implementing infinite domain analyses.

The XSB system permits dynamic compilation, mainly in the form of `assert`. The primitives for dynamic compilation in XSB are, in general, faster than the corresponding primitives provided by other Prolog systems such as SICStus and Quintus Prolog. Dynamic compilation is an important feature affecting the practicality of analyzers built using XSB, since it results in much lower preprocessing overheads compared to full compilation, while adding little to the evaluation time. The preprocessing times, which dominate the total analysis times, are low enough that analyzers built using dynamic compilation are significantly faster than their fully compiled counterparts.

## 3 Formulation of Finite-domain Analyses

In this section, we describe the formulation of two simple finite domain analyses — groundness analysis of logic programs based on the *Prop*-domain [23] and strictness analysis of functional programs based on demand propagation [37].

### 3.1 Groundness Analysis using the *Prop*-domain

The *Prop*-domain [23] is a simple, yet effective abstract domain to compute groundness properties of logic programs. The substitutions in the concrete program are represented in the *Prop*-domain by boolean formulae over the variables in the substitution, with the connectives $\wedge$, $\vee$ and $\Leftrightarrow$. For instance, a concrete substitution $\{X \leftarrow t\}$ is mapped to the formula $X \Leftrightarrow (Y_1 \wedge Y_2 \wedge \cdots Y_k)$, where $\{Y_1, Y_2, \ldots, Y_k\}$ is the set of variables in $t$. Substitutions over multiple variables and sets of substitutions are represented by conjunction and disjunction of formulae representing individual substitutions. This domain has been used in several previous implementations (*e.g.,* [8, 40]) and has been shown to yield accurate results for both offline and online analyses.

Following [8], we represent the boolean formulae by their truth tables. For instance, $X_1 \Leftrightarrow X_2 \wedge X_3$ is represented by a predicate `iff(X1, X2, X3)` whose success set is {`(true, true, true)`, `(false, false, true)`, `(false, true, false)`, `(false, false, false)`}. Disjunction and conjunction are simply the union and join of the success sets. This leads to a simple transformation, given in Figure 1, which maps a given logic program $P$ to an abstract program $P'$ that computes the groundness properties of the predicates in $P$. The transformation is such that the *output* groundness of a predicate $p$ in $P$ is represented by the success set of the corresponding predicate $\mathtt{gp}_p$ in $P'$. Each variable $X$ in the source program is associated with a unique variable $\tau_X$ in the target program.

An example program, append, is given in Figure 2a and the corresponding abstract program is given in Figure 2b. Based on the definition of `iff`, it is easy to verify that the success set of `gp_ap(X,Y,Z)` is {`(true, true, true)`, `(true, false, false)`, `(false, true, false)`, `(false, false, false)` }, the truth table for the formula $X \wedge Y \Leftrightarrow Z$ which represents the output groundness of ap.

**Input and Output Groundness** The translation rules in Figure 1 can be used directly to obtain a program $P'$ whose minimal model computes the output groundness properties of a given program $P$. In order to obtain the *input* groundness, we can define a second transformation based on the Magic Sets transformation [3, 34] (such as the one given in [8]) such that the minimal model of the resultant program $P''$ represents the input groundness of the given program $P$. However, table-driven methods, such as OLDT and SLG, record all the *subgoals* (calls) encountered during evaluation, as well as their *answers* (returns). If an implementation of these resolution methods selects the literals in left-to-right order, the calls of $P'$ capture the *input* groundness of $P$. Since the calls are anyway recorded, we do not have to pay an additional price for obtaining input modes. This property is exploited by engines designed for top-down abstract interpretation, such as GAIA [22], and can also be readily exploited by evaluating $P'$ using the XSB system.

### 3.2 Strictness Analysis based on demand propagation

A function $f$ is said to be strict in its $i$-th argument iff evaluation of $f(e_1, \ldots, e_i, \ldots, e_n)$ fails to terminate whenever evaluation of $e_i$ is nonterminating. It follows that $f$ is strict in its $i$-th argument iff $e_i$ needs to be evaluated in every terminating application $f(e_1, \ldots, e_i, \ldots, e_n)$ of $f$. Since expressions in a lazy functional program are evaluated only if necessary, the process of evaluation can be considered as a form of demand flow. When an application of $f$ needs to be evaluated, we say that there is a *demand* on the output of $f$; if $f$ is strict in its $i$-th argument, then $f$ transforms a demand on its output to a demand on its $i$-th argument. We encode the strictness equations from [37], considering the following demand extents: normal form demand denoted by $e$, head normal form demand denoted by $d$ and null demand denoted by $n$.[2]

For each function $f$ in the input program, we derive a predicate $\mathtt{sp}_f$ that models the propagation of demand by an application of $f$. For a function $f(x)$ we derive a predicate $\mathtt{sp}_f\mathtt{(D,X)}$ such that the substitutions of `X` represent the demands on the variable $x$ whenever the demand on the output of an application of $f$ is `D`. For instance, the strictness of the list building operator ':' may be defined using a predicate `sp_cons(D, X, Y)` such that `sp_cons(`$e$`, X, Y)` succeeds only with `X` $= e$ and `Y` $= e$; `sp_cons(`$d$`, X, Y)` and `sp_cons(`$n$`, X, Y)` succeed for any values of `X` and `Y`.

For each user-defined function $f$, we derive a clause defining $\mathtt{sp}_f$ corresponding to each equation defining $f$, based on the demand propagation properties of the two basic constructs: *function composition* (application) that is used to define expressions, and *pattern matching* that guides the selection of the appropriate equation. Let $f(g(x))$ be a function application on the rhs of some equation. The strictness property of this application is given by the conjunction $\mathtt{sp}_f\mathtt{(D,\ D1)}$, $\mathtt{sp}_g\mathtt{(D1,\ X)}$, where `D` represents the demand on $f(g(x))$, `X` represents the demand on $x$ and `D1` represents the demand placed on $g(x)$ by the application of $f$. The demands on the variables of a rhs expression are transferred to the demands on the arguments of the lhs by interpreting the pattern matching operation. For instance, consider a position on the lhs with a pattern $x : xs$. If the demand flow is such that the evaluation of the rhs expression places an e-demand on both $x$ and $xs$, then we can conclude that the evaluation of this equation places an e-demand on $x : xs$.

[2]It should be noted that the strictness analysis of [37] used here generalizes Mycroft's strictness analysis [26] to non-flat domains.

$$
\begin{array}{rcl}
\mathcal{P}\ [\![p(t_1,t_2,\ldots,t_n) \text{ :- } c]\!] & \rightarrow & \mathtt{gp}_p(\mathtt{X1},\mathtt{X2},\ldots,\mathtt{Xn}) : - \\
 & & \mathcal{E}\ [\![t_1]\!]\ \mathtt{X1},\ \mathcal{E}\ [\![t_2]\!]\ \mathtt{X2},\ldots,\ \mathcal{E}\ [\![t_n]\!]\ \mathtt{Xn},\ \mathcal{L}\ [\![c]\!]\ \mathtt{D}. \\
\mathcal{E}\ [\![t]\!]\ \alpha & \rightarrow & \mathtt{iff}(\alpha,\ \alpha_1,\ \alpha_2,\ldots,\alpha_k) \\
 & & \qquad\qquad \text{where}\{\alpha_1,\alpha_2,\ldots,\alpha_k\} = \mathit{Vars}(t) \\
\mathcal{L}\ [\![l_1, l_2]\!] & \rightarrow & \mathcal{L}\ [\![l_1]\!]\ ,\ \mathcal{L}\ [\![l_2]\!] \\
\mathcal{L}\ [\![q(t_1,t_2,\ldots,t_k)]\!] & \rightarrow & \text{let} \\
 & & \{\alpha_1,\alpha_2,\ldots,\alpha_k\} \leftarrow \mathit{GetNewVariables}() \\
 & & \text{in} \\
 & & \mathcal{E}\ [\![t_1]\!]\ \alpha_1,\ \mathcal{E}\ [\![t_2]\!]\ \alpha_2,\ldots,\ \mathcal{E}\ [\![t_k]\!]\ \alpha_k, \\
 & & \mathtt{gp}_q(\alpha_1,\alpha_2,\ldots,\alpha_k) \\
\mathcal{L}\ [\![X = t]\!] & \rightarrow & \mathcal{E}\ [\![t]\!]\ \tau_X
\end{array}
$$

Figure 1: Formulation of Groundness Analysis

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

(a)

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

(b)

Figure 2: The ap program (a), and its abstraction gp_ap (b).

The effect of matching an input expression with the pattern $x : xs$ can be described by using a predicate `pm_cons(D, X, Xs)` such that `pm_cons(D, e, e)` succeeds only with $\mathtt{D} = e$, while `pm_cons(D, X, Xs)` succeeds with $\mathtt{D} = d$ whenever either `X` or `Xs` is *not* bound to $e$.

The construction of strictness predicates from an input functional program is given in Figure 3. As in Figure 1, each variable $x$ in the source program is associated with a unique variable $\tau_x$ in the target program. In addition to the Horn clauses generated by the rules in Figure 3, for each user-defined function symbol $f$ we derive one clause '$\mathtt{sp}_f$`(n, X1, X2, ..., Xn).`' to handle the propagation of $n$-demand that arises due to non-strictness of functions.

The strictness predicates thus derived for the example program in Figure 4a is given in Figure 4b. The query `sp_ap(e, X, Y)` has only one solution, with `X = e` and `Y = e`, indicating that the function *ap* is *ee*-strict in both its arguments. On the other hand, `sp_ap(d, X, Y)` has two solutions, $\{\mathtt{X} = \mathtt{e}, \mathtt{Y} = \mathtt{d}\}$ and $\{\mathtt{X} = \mathtt{d}, \mathtt{Y} = \mathtt{n}\}$, indicating that *ap* is *dd*-strict in the first argument, but not in the second argument.

**Efficiency Issues** In the formulation of strictness, we have treated the output program as a definite logic program, and correctness was based on the minimal-model semantics. In essence, the order of the literals in the rhs of clauses, as well as the order of clauses themselves, is irrelevant to the soundness of the analysis. However, note in Figure 3 that, apart from the fact that $\mathcal{E}$ generates sp predicates while $\mathcal{P}$ generates pm predicates, $\mathcal{E}$ and $\mathcal{P}$ differ only in the order of the literals in the output conjunction. This follows from the observation that while the demand flows top-down through the rhs expressions (*i.e.*, from an expression to its components), evaluation extents flow bottom-up through the pattern in the lhs. The order of literals encodes this flow information, and significantly improves the *efficiency* of the resultant program by reducing backtracking[3].

## 4 Implementation and Experimental Results

Once the Horn clauses have been generated according to the analysis formulation of the previous section, they can be evaluated directly to produce the results of the analysis by any logic programming system that guarantees completeness. The straightforward approach is to compile the clauses just as any other logic program. However, for practical analysis, we must consider all of the costs involved, including the time required to prepare the rules for evaluation. We assess the performance of the analysis implementations using the following metrics corresponding to the different phases of analysis.

**Preprocessing time** The total time required to prepare the source program for analysis, including the time required to transform the program into the logical rules to be evaluated and the time to "compile" the logical rules for evaluation (*e.g.*, full compilation into WAM code, or asserting the rules as dynamic code).

**Analysis time** The total time required to evaluate the logical rules to yield their minimal model.

**Collection time** The time required to extract the results of the analysis.

[3]In a goal-oriented set-at-a-time evaluation (as in bottom-up evaluation with Magic Sets), the reduction in backtracking corresponds to reduction in the size of the joins.

$$\begin{array}{rcl}
\mathcal{R}\ [\![f(t_1,t_2,\ldots,t_n)=e]\!] & \rightarrow & \mathtt{sp}_f(\mathtt{D},\mathtt{X1},\ldots,\mathtt{Xn}) :- \\
 & & \quad \mathcal{E}\ [\![e]\!]\ \mathtt{D},\ \mathcal{P}\ [\![t_1]\!]\ \mathtt{X1},\ \mathcal{P}\ [\![t_2]\!]\ \mathtt{X2}, \ldots,\ \mathcal{P}\ [\![t_n]\!]\ \mathtt{Xn}. \\
\mathcal{E}\ [\![f(e_1,e_2,\ldots,e_l)]\!]\ \alpha & \rightarrow & \text{let} \\
 & & \quad \{\alpha_1,\alpha_2,\ldots,\alpha_l\} \leftarrow \mathit{GetNewVariables}() \\
 & & \text{in} \\
 & & \quad \mathtt{sp}_f(\alpha,\alpha_1,\alpha_2,\ldots,\alpha_l), \\
 & & \quad \mathcal{E}\ [\![e_1]\!]\ \alpha_1,\ \mathcal{E}\ [\![e_2]\!]\ \alpha_2, \ldots,\ \mathcal{E}\ [\![e_l]\!]\ \alpha_l \\
\mathcal{E}\ [\![x]\!]\ \alpha & \rightarrow & \tau_x = \alpha \\
\mathcal{P}\ [\![c(t_1,t_2,\ldots,t_k)]\!]\ \alpha & \rightarrow & \text{let} \\
 & & \quad \{\alpha_1,\alpha_2,\ldots,\alpha_k\} \leftarrow \mathit{GetNewVariables}() \\
 & & \text{in} \\
 & & \quad \mathcal{P}\ [\![t_1]\!]\ \alpha_1,\ \mathcal{P}\ [\![t_2]\!]\ \alpha_2, \ldots,\ \mathcal{P}\ [\![t_k]\!]\ \alpha_k, \\
 & & \quad \mathtt{pm}_c(\alpha,\alpha_1,\alpha_2,\ldots,\alpha_k) \\
\mathcal{P}\ [\![x]\!]\ \alpha & \rightarrow & \tau_x = \alpha
\end{array}$$

Figure 3: Formulation of Strictness Analysis

$$\begin{array}{rcl}
ap(nil, ys) & = & ys \\
ap(x : xs, ys) & = & x : ap(xs, ys)
\end{array}$$

(a)

```
sp_ap(D,X1,X2):-
        Tys = D,
        pm_nil(X1), Tys = X2.
sp_ap(D,X1,X2):-
        sp_cons(D,D1,D2), Tx = D1,
        sp_ap(D2,Txs,Tys),
        pm_cons(X1, Tx, Txs), Tys = X2.
sp_ap(n,X1,X2).
```

(b)

Figure 4: The *append* program: (a) Concrete program, and (b) Abstract program

The total of the three times listed above is the overall analysis time, and this will be the primary indicator of the practicality of our analysis implementations.

When all of the above costs are taken into account, it is not clear that full compilation of the rules for evaluation is the best approach. An alternative is to compile the rules dynamically (assert) and interpret them (through `call/1` in Prolog). Although it may not be obvious at first, the latter method turns out to be the better choice. As will be shown in Sections 4.1 and 4.2, preprocessing time is generally much greater than analysis (evaluation) time, so that keeping preprocessing time as low as possible is critical to good overall analysis times. When we consider that the logical rules generated for logic program analysis resemble the original logic program, it becomes apparent that compiling those rules for fastest evaluation may itself take as much time (or even more, when initial transformation time is included) as compiling the original program itself. By loading the analysis rules as dynamic code, preprocessing time is reduced substantially, at some cost in evaluation time due to the overhead of interpretation (`call/1`). However, even using this interpretation approach, the evaluation times we observe are generally low compared to preprocessing time.

The last performance metric, collection time, is independent of the evaluation approach. For either full compilation or interpretation the calls occurring during evaluation and the computed returns are stored in a table using the same tabling mechanisms. When the analysis phase is completed, there may be multiple calls and/or answers that must be combined into unique analysis results. For example, in the *Prop* formulation of groundness analysis, if the return table for some predicate $p/3$ contained two answers `p(true,false,true)` and `p(true,true,false)`, they would be combined into the single result `p(true,false,false)`.

At first glance, our use of an enumerative representation of boolean formulae may seem to be inefficient. Many implementations [10, 40] use Bryant's Decision Diagrams (BDDs) [6] to represent boolean formulae compactly. However, experimental results show that our analysis times are very competitive. This effect is due to the underlying engine which computes fix points incrementally: computing in one iteration using the change in results (delta-sets, in deductive database terms) from the previous iteration. The apparently inefficient representation we use actually allows for efficient computation of the delta-sets.

## 4.1 Performance of Groundness Analysis

Table 1 shows performance measurements for *Prop*-based groundness analysis in XSB on a set of benchmarks from [40][4]. The column labeled "Compile time increase" shows the ratio of total analysis time to total compilation time (with no analysis) in XSB, and indicates the increase in

[4]Measurements were taken on a Sun SPARCStation 10/30, with 64M memory, running SunOS version 4.1.3 and XSB version 1.4.2. The machine configuration was chosen to match that used by [40].

| Program | Program size (lines) | Time (sec.) | | | | Compile time increase (%) | Table space (bytes) |
|---|---|---|---|---|---|---|---|
| | | Preproc. | Analysis | Collection | Total | | |
| CS | 182 | 0.31 | 0.11 | 0.15 | 0.57 | 22.1 | 8056 |
| Disj | 172 | 0.27 | 0.03 | 0.10 | 0.40 | 26.9 | 5768 |
| Gabriel | 122 | 0.20 | 0.05 | 0.11 | 0.36 | 43.6 | 6912 |
| Kalah | 278 | 0.48 | 0.06 | 0.23 | 0.77 | 37.4 | 10580 |
| Peep | 369 | 0.84 | 0.16 | 0.09 | 1.09 | 23.4 | 5800 |
| PG | 53 | 0.10 | 0.01 | 0.02 | 0.13 | 31.0 | 2332 |
| Plan | 84 | 0.14 | 0.01 | 0.03 | 0.18 | 30.8 | 2888 |
| Press1 | 349 | 0.62 | 0.38 | 0.82 | 1.82 | 59.5 | 29400 |
| Press2 | 351 | 0.60 | 0.41 | 0.83 | 1.84 | 60.7 | 29400 |
| QSort | 21 | 0.04 | 0.00 | 0.01 | 0.05 | 33.3 | 916 |
| Queens | 33 | 0.04 | 0.00 | 0.01 | 0.05 | 27.8 | 976 |
| Read | 443 | 0.72 | 0.60 | 0.70 | 2.02 | 64.4 | 26528 |

Table 1: Performance of *Prop*-based groundness analysis in XSB

| System | CS | Disj | Gab. | Kalah | Peep | PG | Plan | Press1 | Press2 | QSort | Queens | Read |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| XSB | 0.57 | 0.40 | 0.36 | 0.77 | 1.09 | 0.13 | 0.18 | 1.82 | 1.84 | 0.05 | 0.05 | 2.02 |
| GAIA | 1.34 | 1.01 | 0.47 | 0.93 | 1.16 | 0.16 | 0.12 | 5.96 | 6.03 | 0.05 | 0.04 | 1.66 |

Table 2: Comparison of XSB and GAIA for *Prop*-based groundness analysis

compilation time that could be expected if groundness analysis were included as a phase of compilation. In all cases, total analysis time is less than compilation time, indicating that simple, high-level analysis implementations can indeed be practical. The table space used during analysis is given in the last column, and affirms the practicality of the analysis.

Table 2 compares the total analysis time for XSB with analysis times reported for GAIA [40] on the same benchmarks. It should be noted that the times given for XSB include the time needed to extract all results from the internal representation (the tables), while those for GAIA do not. The results obtained on the two systems are identical, since they implement the same analysis. It is indeed encouraging to note that our high-level implementation in a general purpose system not only performs well enough to be practical, but compares very well with a fast, highly optimized C-based system designed specifically for abstract interpretation.

## 4.2 Performance of Strictness Analysis

Table 3 reports the performance of strictness analysis in XSB[5]. The programs were taken from the benchmarks for EQUALS [21], and include those translated from the benchmarks in [16]. The preprocessing times include the time to compute and output the logic rules for strictness from EQUALS programs and to read and load these rules into the XSB system. Thus the total time represents the increase in compilation time due to strictness analysis. The analyzer processes about 200 to 350 source lines per second. The total time to perform the analysis is about 5% of time taken by *ghc*, the Glasgow Haskell Compiler, to compile (without analysis or optimizations) an equivalent program written in Haskell. The speed of the analysis and its space behavior provide strong evidence of its practicality.

Observe from the table that the preprocessing times once again dominate the total analysis times, except for the program *pcprove*. The analysis times are higher for *pcprove* mainly due to a characteristic of the formulation (see Figure 3): deep nesting of function applications result in excessively long Horn clauses. This, in conjunction with the enumerative definition of base functions, leads to deep backtracking. However, that the occurrence of demand variables is highly localized means that tabling intermediate results (thereby eliminating the existentially quantified demand variables) will reduce backtracking, thereby potentially improving analysis times. Such an optimization, called ***supplementary magic sets*** [4] is performed in deductive databases, and XSB offers an analogous (compile-time) optimization called ***supplementary tabling***. However, the effectiveness of this optimization in reducing analysis time remains to be established.

## 5 Beyond Enumerative Analyses

The underlying representation used in the implementations of groundness and strictness analyses is enumerative. Although enumerative representations may be efficient for small domains, performance may significantly degrade when the domain sizes increase. The results of [10] show that even when a particularly efficient enumerative representation, such as a BDD [6] is used, analysis times increase with increases in domain size. In contrast, analyses based on non-enumerative representations such as symbolic constraints show little reduction in performance due to increase in the size of the underlying domain [27]. In this section, we describe how such an analysis can be implemented efficiently in XSB.

The abstract domain is the set of all terms of depth $k$ or less, constructed using the function symbols that occur in the program, a special 0-ary symbol $\gamma$ and a countable set of variables. The symbol $\gamma$ is used to represent the set of all ground terms. The concretization function maps each abstract term $t$ to the set of concrete terms $S$, such that for each $s \in S$, either (i) $t$ is a variable, or (ii) $t = \gamma$ and $s$ is ground, or (iii) the roots of $t$ and $s$ are identical and each subterm of $s$ is a concretization of the corresponding subterm in $t$. The abstract term can be viewed as a constraint, with the symbol $\gamma$ representing a membership constraint

[5]Measurements were taken on a Sun SPARC LX with 64M memory, running SunOS version 5.4 and XSB version 1.4.2.

| Program | Program size (lines) | Times (sec.) | | | | Table space (bytes) |
|---|---|---|---|---|---|---|
| | | Preproc. | Analysis | Collection | Total | |
| *eu* | 67 | 0.12 | 0.03 | 0.01 | 0.16 | 2852 |
| *event* | 384 | 0.67 | 0.63 | 0.08 | 1.38 | 22056 |
| *fft* | 343 | 0.63 | 0.19 | 0.06 | 0.88 | 15780 |
| *listcompr* | 241 | 0.75 | 0.07 | 0.02 | 0.84 | 4688 |
| *mergesort* | 65 | 0.11 | 0.02 | 0.01 | 0.14 | 2332 |
| *nq* | 90 | 0.20 | 0.12 | 0.02 | 0.34 | 8912 |
| *odprove* | 160 | 0.39 | 0.17 | 0.02 | 0.58 | 3776 |
| *pcprove* | 595 | 1.01 | 1.60 | 0.10 | 2.71 | 25972 |
| *quicksort* | 70 | 0.10 | 0.03 | 0.01 | 0.14 | 2660 |
| *strassen* | 93 | 0.09 | 0.08 | 0.01 | 0.18 | 2760 |

Table 3: Performance of Strictness Analysis in XSB

| Program | Time (sec.) | | | | % Compile Time | Table space (bytes) |
|---|---|---|---|---|---|---|
| | Preproc. | Analysis | Collection | Total | | |
| `CS` | 0.16 | 0.03 | 0.07 | 0.26 | 16 | 12988 |
| `Disj` | 0.14 | 0.03 | 0.06 | 0.23 | 23 | 9552 |
| `Kalah` | 0.24 | 0.05 | 0.11 | 0.40 | 29 | 17068 |
| `Peep` | 0.44 | 0.08 | 0.05 | 0.57 | 18 | 12784 |
| `PG` | 0.05 | 0.01 | 0.02 | 0.08 | 29 | 4136 |
| `Plan` | 0.08 | 0.01 | 0.02 | 0.11 | 29 | 5324 |
| `QSort` | 0.02 | 0.01 | 0.02 | 0.05 | 56 | 1684 |
| `Queens` | 0.03 | 0.00 | 0.01 | 0.04 | 33 | 1740 |
| `Read` | 0.36 | 0.25 | 0.43 | 1.04 | 50 | 52508 |

Table 4: Performance of groundness analysis with term depth abstraction in XSB

(constraining a term to the set of all ground terms), and the other symbols representing equality constraints.

Abstract unification unifies two abstract terms up to depth $k$, with $\gamma$ unifying with any ground term. Note that abstract unification is different from the unification operation provided by the underlying engine; hence we have to implement abstract unification at a higher level. Table 4 shows performance measurements for groundness analysis with term depth abstraction in XSB[6] on the set of benchmarks used in Section 4.1.

## 6 Discussion

### 6.1 Analyses over Infinite Domains

It should be noted that the abstract domains used in all analyses described thus far are finite. Below, we describe strategies to handle infinite domains, including those with infinite ascending chains.

**Widening** Consider the evaluation of the fix point of a functional $F$, over a domain $D$. An iterative technique to find fix points computes a sequence of iterates $X^0, X^1, \ldots$ such that $X^0 = \bot_D$ and $X^{i+1} = F(X^i)$. The iteration process terminates when $X^i = X^{i+1}$. If the domain $D$ has infinite ascending chains, iteration may not terminate even when $F$ is monotonic. For analysis over such domains, termination is ensured by using "on-the-fly" approximation operations such as *widening* [12]. Intuitively, we use a widening operator to accelerate the convergence of fix-point iteration sequences by extrapolating the iterates. The new iteration sequence is such that $X^{i+1} = X^i \nabla F(X^i)$, where $\nabla$ is the widening operator. See [11] for a formal definition of widening operation and the requisite properties that ensure termination of the iteration sequence.

[6]Measurements were taken on a Sun SPARCStation 20 with 64M memory, running SunOS version 5.3 and XSB version 1.4.2.

In the context of tabled evaluation, widening operations require (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 accomplished using existing low level system primitives, a high level abstraction of widening will enable infinite domain analyses to be implemented as directly as finite domain analyses. Operations for aggregation over sets, provided by many deductive database systems, have exactly the same requirements, and we believe that widening can be implemented along the same lines.

**Constraints** In a logic programming system, constraint operations can be implemented using the programming primitives provided by the system. Thus, the programmability of the system not only provides a way to control the efficiency of the resulting analyses (see Section 3), but also permits implementation of a larger class of analyses. A case in point is the groundness analysis using a constraint-based representation, described in Section 5.

It should be noted that the methods described in this paper are not restricted to analyses where the size of constraints are fixed *a priori*. For instance, consider Hindley-Milner type analysis [18, 24] for functional programs, where the type of functions in the input program is formulated as the solution to *type equations*, which are equations over the domain of equality constraints. The type equations are nonrecursive, since any recursion in the input program is eliminated using an explicit fix-point operator. Note that tabled evaluation is not needed to solve the nonrecursive type equations. Indeed, the only requirement is that occur-check be performed by the unification operation. In case the unification operation provided by the underlying engine does not meet this condition, unification with occur-check can be easily programmed at a higher level. In fact, the abstract unification operation of the depth-bounded constraint analysis of Section 5 performs occur-check. Thus, a straightforward implementation of the logical formulation is not limited to only finite-domain analyses. However, whether such an implementation of the Hindley-Milner type analysis can be practical remains to be seen.

### 6.2 Impact of Engine Optimizations

**Answer Collection** Using generic aggregation operations, such as those typically provided by deductive database systems, answer collection can be implemented very simply and at a high level. 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 the impact of breadth-first scheduling strategies on aggregation. It should be noted that such strategies are being sought mainly to improve the efficiency of the system for database programs [15]. This illustrates one of the advantages in formulating analysis problems as logic programs: the analyzer can benefit directly from optimizations to the evaluation techniques motivated by the application of the underlying system to problems in other areas.

**Bottom-up Analysis** Top-down analyses benefit from a goal-directed evaluation strategy. This benefit is clear from the implementation: the tabling mechanisms enable the computation of input and output modes in one analysis pass without requiring transformations such as Magic (*e.g.*, [8]). However, such goal orientation may lead to inefficiencies in bottom-up analyses, especially when variant checks are used for tabling. For instance, while evaluating logic programs derived for bottom-up analysis, open calls (*i.e.*, calls of the form $p(X_1, X_2)$, where $X_1$ and $X_2$ are variables) are eventually made. Moreover, many particular calls, (*i.e.*, of the form $p(t_1, t_2)$ where $t_1$ or $t_2$ are non-variables) may be generated. For each predicate in the program, two cases naturally arise: (1) the open call is encountered before any particular call *(forward subsumption)*, and (2) some particular calls are encountered before the open call *(backward subsumption)*. In a tabling system based on variant checks, answers for particular calls are recomputed instead of reusing the answers for the open call.

Recently, we completed the implementation of an engine that exploits forward subsumption at a relatively low cost [30]. At present, it is unclear whether backward subsumption can be exploited as effectively. Nevertheless, bottom-up computations can be performed efficiently by using a simple strategy that uses forward subsumption as follows. On the first call to a tabled predicate, we generate an open call, since we know that open calls will eventually be made. Forward subsumption can now be used to return the answers to any specific call. The effectiveness of this strategy is currently under evaluation.

## 7 Related Work

As mentioned in Section 2, a number of analyses for logic programs have been formulated in terms of rules. Of direct relevance are the works where such a formulation has been used in an implementation, as in [14, 8]. The use of extension tables to compute fix points, and the approach of analyzing the program by executing an abstract program were presented in [14]. However, the technology was not mature enough at that time to study the practicality of the approach. Our formulation of the *Prop*-domain analysis is based on the formulation in [8], where magic-set transformation was used to obtain call patterns. In contrast, we obtain both call and answer patterns by simply executing the abstract program on a (complete) top-down engine. Our implementation establishes that analyses derived in this manner are practical, and paves way for further investigation of factors affecting practicality, such as seeking the right balance between compilation and interpretation to reduce total analysis time.

In [17], analyses are formulated in terms of solving Set Constraints, and implemented using a special-purpose constraint solver. In [10], groundness and type analysis of logic programs have been formulated as a constraint solving problem, and implemented using *Toupie*, a finite domain constraint solver. In the area of logic programming, many analyses for logic programs have been implemented based on the framework in [5]; generic tools for abstract interpretation, such as GAIA [22] and PLAI [25] have been implemented and have evolved into well-optimized systems. In contrast, in this paper, we investigated the use of a *general purpose logic programming system* for solving analysis problems (irrespective of the application language) and its practicality.

In [31, 32], dataflow properties for imperative programs have been formulated as database facts. The *demand analysis* problem is formulated as a query solved over such a set of facts. Results in [31] suggest that a general purpose system (Coral) takes about 6 times longer to evaluate the queries, compared to a special purpose demand algorithm implemented in C. In [35] it is reported that XSB is roughly a order of magnitude faster than Coral. In particular, for the sample demand analysis program given in [32] we find that it is indeed the case, leading us to believe that XSB can be used to construct practical dataflow analyzers[7]. Further investigation is clearly needed.

## 8 Conclusion

Many program analysis problems can be cast in the form of evaluating minimal models of a logic program. The results of the paper strongly suggest that practical analyzers can be built from such declarative formulations with minimum effort using general purpose logic programming systems. Furthermore, we find that these systems offer sufficient expressive power to formulate many common analyses. Nevertheless, as with any general purpose system, carefully exploiting the capabilities of the system is crucial to attain good performance. There is reason to believe that the results of this paper will carry beyond functional and logic program analyses. Such extensions are the subject of ongoing research.

[7]It must be noted that the results presented in this paper were taken using XSB v1.4.2, which offers significant performance improvement over earlier releases due to optimized tabling primitives (see [28]).

## References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison Wesley, 1988.

[2] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, Mass., 1991.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and other strange ways to implement logic programs. In *ACM Symposium on Principles of Database Systems*, pages 1–15. ACM Press, 1986.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. In *ACM Symposium on Principles of Database Systems*, pages 269–283. ACM Press, 1987.

[5] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10:91–124, 1991.

[6] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

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

[8] 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.

[9] P. Codognet and G. Filé. Computations, abstractions and constraints. In *International Conference on Computer Languages*, pages 155–164. IEEE Press, 1992.

[10] M-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. In *International Symposium on Programming Language Implementation and Logic Programming*, number 714 in Lecture Notes in Computer Science, pages 75–91. Springer Verlag, 1993.

[11] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM Press, 1978.

[13] S. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, July 1989.

[14] S. Debray and D.S. Warren. Automatic mode inference for Prolog programs. In *Proceedings of the Third Symposium on Logic Programming*, pages 78–88, 1986.

[15] J. Freire, T. Swift, and D.S. Warren. Taking I/O seriously: Resolution reconsidered for disk. Technical report, Department of Computer Science, SUNY, Stony Brook, 1996.

[16] P.H. Hartel and K.G. Langendoen. Benchmarking implementations of lazy functional languages. In *Symposium on Functional Programming Languages and Computer Architecture*, pages 341–349. ACM Press, 1993.

[17] N. Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[18] J. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[19] P. Hudak. A semantic model for reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

[20] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[21] O. Kaser, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. EQUALS — a parallel implementation of a lazy language. *Journal of Functional Programming*, To appear.

[22] 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, January 1994.

[23] K. Marriot and H. Sondergaard. Notes for a tutorial on abstract interpretation of logic programs (unpublished). In *North American Conference on Logic Programming*, 1989.

[24] R. Milner. A theory of type polymorphism in programming. *Journal of Computer System Sciences*, 17:348–375, 1978.

[25] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13:315–347, 1992.

[26] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[27] 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*, pages 12–23. ACM Press, 1995.

[28] I.V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D.S. Warren. Efficient tabling mechanisms for logic programs. In *International Conference on Logic Programming*, pages 697–711. MIT Press, 1995.

[29] R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan. The Coral user's manual. Technical report, Computer Sciences Department, Univerity of Wisconsin, Madison, 1993.

[30] P. Rao, C.R. Ramakrishnan, and I.V. Ramakrishnan. A thread in time saves tabling time. Technical report, Department of Computer Science, SUNY, Stony Brook, 1996.

[31] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases.* Kluwer Academic, 1994.

[32] T. Reps. Shape analysis as a generalized path problem. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 1–11. ACM Press, 1995.

[33] T. Reps, M. Sagiv, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *ACM Symposium on Principles of Programming Languages.* ACM Press, 1996.

[34] R. Rohmer, R. Lescoeur, and J.-M. Kersit. The Alexander method, a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4(3):273–285, 1986.

[35] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD Symposium on Management of Data.* ACM Press, 1994.

[36] K. Sagonas, T. Swift, and D.S. Warren. The XSB programmer's manual, Version 1.4.2. Technical report, Department of Computer Science, SUNY, Stony Brook, 1995.

[37] R.C. Sekar and I.V. Ramakrishnan. Fast strictness analysis based on demand propagation. *ACM Transactions on Programming Languages and Systems*, 17(6), November 1995.

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

[39] J.D. Ullman. *Principles of Database and Knowledge-base Systems, Volume II.* Computer Science Press, 1989.

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

[41] P. Van Roy, B. Demoen, and Y. D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In *Theory and Practice of Software Development*, pages 111–125, March 1987.