Analyzing Logic Programs using "Prop"-ositional Logic Programs and a Magic Wand

Michael Codish*and Bart Demoen†

February 28, 1997

Abstract

This paper illustrates the role of a class of "prop"-ositional logic programs in the analysis of complex properties of logic programs. Analyses are performed by abstracting Prolog programs to corresponding "prop"-ositional logic programs which approximate the original programs and have finite meanings. We focus on a groundness analysis which is equivalent to that obtained by abstract interpretation using the domain *Prop*. The main contribution is in the ease in which a highly efficient implementation of the analysis is obtained. The implementation is bottom-up and provides approximations of a program's *success patterns*. Goal dependent information such as *call patterns* is obtained using a magic-set transformation. A novel compositional approach is applied so that call patterns for arbitrary goals are derived in a precise and efficient way.

1 INTRODUCTION

Groundness analysis is one of the more important analyses for logic programs. The knowledge that a given program variable will always be bound to a ground term is useful for a wide range of applications on its own, such as deriving modes and optimizing unification. In addition, it is useful for improving the precision of many other types of analysis. For example, in sharing analysis we wish to determine if two (or more) program variables will ever be bound to terms which share a common variable. The information that a variable will always be bound to a ground term clearly implies that it never shares with other variables.

In this paper we describe an efficient and simple implementation of a semantic based groundness analysis for logic programs. The analysis we implement is not new. It is known to be one of the more precise static groundness analyses described in the literature. Moreover, it has recently been found to be one of the more efficient analyses of its type [27, 13]. Our main contribution is in the simplicity of our approach and in the ease in which we obtain an efficient and precise analysis.

The framework of abstract interpretation [15] provides the basis for a semantic approach to dataflow analysis. 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 behavior. 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.

The domain *Prop* of propositional formulae is proposed in [30, 14] and further discussed in [31] as a means to describe substitutions and as a basis for groundness analysis defined in terms of abstract interpretation. A similar domain of *dependency formulae* is introduced in [16]. For example, the formula $X \wedge Y$ describes any substitution which binds both X and Y to ground terms. Likewise, $X \leftrightarrow (Y \wedge Z)$ describes those substitutions any instance of which binds X to a ground term if and only if Y and Z are bound to ground terms and $(X \wedge Y) \vee Z$ describes substitutions for which any instance grounds either X and Y, or Z..

*Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, PoB 653 Beer-Sheba 84105, Israel. codish@cs.bgu.ac.il.

†Department of Computer Science, K.U. Leuven, Belgium. bimbart@cs.kuleuven.ac.be

Other analyses involving the domain *Prop* include various applications for suspension analysis described in [11] and simple type analyses described in [13]. A non-trivial application for polymorphic type analysis is described in [9].

In this paper we focus on groundness analysis as captured by abstract interpretation using the domain *Prop*. We apply an approach in which a program is analyzed by applying a concrete semantics to an abstraction of the program itself. This technique is referred to as *abstract compilation* in [24] and originates from an idea presented in [17]. A similar approach is applied in [22] where analyses are defined in terms of regular approximations of programs. Also related are [23] and [12] which describe respectively the bottom-up and top-down evaluation of constraint logic programs on finite domains as a means to obtain analyses for (constraint) logic programs. In contrast to the analyses described in [24] and [17], the abstractions suggested in [22, 23, 12] as well as that proposed here can formally be viewed as abstracting the individual syntactic structures occurring in a program. For example, our abstract programs do not explicitly perform tabulation, least upper bounds or projections which are not present in concrete programs. In fact, our analyses are obtained by evaluating the minimal model of suitable abstract programs and not by executing the abstract program as in the case of [24] and [17]. From a theoretical perspective, abstracting a program's syntax and applying a concrete semantics is equivalent to abstract interpretation, as the domain of a logic program is determined from its syntax. Hence, the abstraction on programs induces an abstraction on the domain of interpretation.

In our case, analyses are guaranteed to terminate as programs are abstracted into Datalog programs which are equipped with an interpretation of the propositional connectives[1]. Such programs are well suited to express and analyze properties in terms of propositional formulae on the set of variables occurring in a program. For example, the following definitions provide an interpretation of $and(X, Y, Z)$ and $or(X, Y, Z)$ which correspond to the disjunctive normal forms of the propositional formulae $(X \wedge Y) \leftrightarrow Z$ and $(X \vee Y) \leftrightarrow Z$ respectively.

```
and(true,true,true).          or(true,true,true).
and(true,false,false).        or(false,true,true).
and(false,true,false).        or(true,false,true).
and(false,false,false).       or(false,false,false).
```

The "prop"-ositional goal $append(X, Y, Z), and(X, Y, Z'), or(Z', Z, true)$ is interpreted as specifying a call of the form $append(t_1, t_2, t_3)$ in which t_1 and t_2 are ground or t_3 is ground.

The analyses described in this paper are bottom-up although it is not difficult to perform also top-down analyses. We show that the meaning $M(P)$ of a logic program P is approximated by the finitely computable meaning $M(P^{\#})$ of a corresponding "prop"-ositional logic program $P^{\#}$. Moreover, the answers for arbitrary goals are approximated by solving (descriptions of) initial goals into $M(P^{\#})$. Bottom-up semantics for logic programs do not capture the set of calls which arise in the computation of an initial goal. However transformation techniques such as the magic-set method (e.g., [1]) can be applied to capture call patterns using bottom-up semantics. These techniques, originally developed to optimize database applications have proven useful in the context of program analysis (see [6] for a list of references). The basic idea is that, for a given program P and goal G, the meaning of the transformed program $magic(P; G)$ reflects both calls and answers for G and P. We apply a novel technique to provide efficient implementation of analyses based on magic-set transformations. The key idea is to partition a transformed program $magic(P; G)$ into two components: a goal-independent module — which is determined by the program P, and a goal dependent module — which is determined from G. The meaning of a magic program is then evaluated by composing the meanings of the individual components using a similar approach to that described in [3, 4]. Consequently, analyses for different goals with the same program avoid recomputing information which concerns only the program dependent component. Moreover, our experiments indicate that compositionality improves the efficiency of the analyses even for one initial goal.

The proposed analysis has been fully implemented in Prolog. Experimental results are consistent with other recent results [27, 13] which indicate that analyses based on propositional formulae are precise and efficient in spite of their potential theoretical complexity. Of particular interest is the high efficiency in which call patterns for arbitrary initial goals are derived. Moreover, the efficiency of our Prolog implementation

[1]Not to be confused with *Propositional Horn Formulae* (see for example [18]) in which all predicates are propositional.

```
reverse(Xs,Ys) ←                reverse(Xs,Ys) ←
   Xs = [],                        iff(Xs),
   Ys = [].                        iff(Ys).
reverse(Xs,Ys) ←                reverse(Xs,Ys) ←
   Xs = [X|Xs1],                   iff(Xs,X,Xs1),
   reverse(Xs1,Ys1),               reverse(Xs1,Ys1),
   A = [X],                        iff(A,X),
   append(Ys1,A,Ys).               append(Ys1,A,Ys).

append(Xs,Ys,Zs) ←              append(Xs,Ys,Zs) ←
   Xs = [],                        iff(Xs),
   Ys = Zs.                        iff(Ys,Zs).
append(Xs,Ys,Zs) ←              append(Xs,Ys,Zs) ←
   Xs = [X|Xs1],                   iff(Xs,X,Xs1),
   Zs = [X|Zs1],                   iff(Zs,X,Zs1),
   append(Xs1,Ys,Zs1).             append(Xs1,Ys,Zs1).

                           iff(true,true,true).    iff(true).
                           iff(false,true,false).
                           iff(false,false,true).  iff(true,true).
                           iff(false,false,false). iff(false,false).
```

Figure 1: Corresponding logic- and "prop"-ositional logic- programs.

is comparable with the C implementations described in [27, 13] which apply advanced techniques in the representation of propositional formulae and in the evaluation of least fixed points. We do note that these implementations are more general than that of our groundness dependency analyzer. However, our approach is extremely simple to implement and is not restricted to the analysis presented in this paper. Recent results illustrate its application to a non-trivial polymorphic type analysis [9].

The rest of this paper is organized as follows. The next section opens with an intuitive example which illustrates the basic idea of our approach. Section 3 sets the notational conventions used throughout and describes several background issues. Section 4 formalizes our groundness analysis and Section 5 describes its implementation and an experimental evaluation indicating good precision and reasonable analysis times for large programs. Section 6 gives a conclusion. The appendix provides the formal justification of selected theorems from Section 4. A preliminary version of this paper appeared in [8].

2 A MOTIVATING EXAMPLE

Figure 1 illustrates the transformation from a Prolog program to a Datalog program. On the left, the classical *naive-reverse* program (in a suitable normal form). On the right, the corresponding "prop"-ositional logic program. The predicates $\mathit{iff}(X, X_1, \ldots, X_n)$ $(n \geq 0)$ specify the propositional formulae $X \leftrightarrow (X_1 \wedge \cdots \wedge X_n)$ with the intended interpretation that X is ground if and only if $\{X_1, \ldots, X_n\}$ are ground. For example, a unification of the form `Xs = [X|Xs1]` in a Prolog program is replaced by a predicate `iff(Xs,X,Xs1)` which specifies that Xs is ground if and only if X and $Xs1$ are ground. Similarly, a unification of the form `Xs = [ ]` in a Prolog program is replaced by a predicate `iff(Xs)` which specifies that Xs is definitely bound to a ground term.

The minimal model of the "prop"-ositional version of *naive-reverse* is illustrated in Figure 2. It is finitely computable and equivalent to the groundness formula obtained in [16] as well as to the abstract interpretation of the original Prolog program over the domain *Prop* [27]. The answers for an arbitrary initial goal, for instance $G = \mathit{reverse}([1,2,3], Y)$, are approximated by solving the corresponding "prop"-ositional goal $G^{\#} = \mathit{reverse}(\mathit{true}, Y)$ with the facts in Figure 2. In this case the answer binds Y to true indicating

```
% reverse(X,Y); X ↔ Y          % append(X,Y,Z); (X ∧ Y) ↔ Z
reverse(true,true).            append(true,true,true).
reverse(false,false).          append(true,false,false).
                               append(false,true,false).
                               append(false,false,false).
```

Figure 2: Concrete semantics for abstract *naive-reverse*.

that the answer of the Prolog query binds Y to a ground term.

$$
\begin{array}{l}
tp \leftarrow \\
\quad my\_clause(H, B),\ call(B), \\
\quad cond\_assert(fact(H)),\ fail. \\
\\
iterate\_tp \leftarrow tp,\ fail. \\
iterate\_tp \leftarrow \\
\quad database\_changed,\ retractall(database\_changed), \\
\quad iterate\_tp. \\
\\
cond\_assert(G) \leftarrow\ in\_database(G),\ !. \\
cond\_assert(G) \leftarrow\ assert(G),\ cond\_assert(database\_changed). \\
\\
in\_database(G) \leftarrow \\
\quad functor(G, N, A),\ functor(B, N, A), \\
\quad call(B),\ variant(B, G),\ !. \\
\\
fact(true).
\end{array}
$$

Figure 3: Simple Prolog interpreter for bottom-up evaluation.

The seven clause Prolog program depicted in Figure 3 illustrates the main principle of our implementation. This simple interpreter can be applied to evaluate the minimal model of any (finitary) logic program. Each clause $h \leftarrow b_1, \ldots, b_n$ in the input program is represented as a fact of the form

$$my\_clause(h,\ (fact(b_1), \ldots, fact(b_n)))$$

which is added to the clauses in Figure 3. Initially, only $fact(true)$ is known. New facts derived by an iteration of the predicate $tp/0$ are asserted to the program if their variants are not already there. The evaluation is initiated by the query `?- iterate_tp` which leaves the non-ground minimal model of the program in the Prolog database.

The rest of this paper formalizes an analysis based on the program abstraction illustrated in Figure 1. The actual implementation is based on the program transformation induced by the interpreter given in Figure 3. Moreover we illustrate how also call patterns for a given program can be derived for arbitrary initial goals in a goal independent way.

3 PRELIMINARIES

Syntax: We assume the standard notation for logic programs as defined in [29]. A tuple of distinct variables is sometimes denoted $\bar{x}$. We often refer to a term of the form $f(\bar{x})$ where f is an n-ary function symbol as a "flat term". We distinguish the syntax of *concrete* and *abstract* programs. Concrete programs involve a first order language with a fixed vocabulary of predicate symbols, function symbols and variables denoted Π, Σ and Var respectively. For abstract programs we assume the fixed vocabulary $\Pi_{prop}, \Sigma_{prop}, Var$ where $\Sigma_{prop} = \{true, false\}$ and $\Pi_{prop} = \Pi \cup \{\, \mathit{iff}/n \mid n \geq 0 \,\}$. For all practical purposes we may assume that $\{\, \mathit{iff}/n \mid n \geq 0 \,\}$ is finite and disjoint from Π. The sets of terms and atoms which may occur in concrete and abstract programs are denoted $Term, Atom$ and $Term_{\mathcal{P}}, Atom_{\mathcal{P}}$ respectively. Elements of $Term_{\mathcal{P}}$ and $Atom_{\mathcal{P}}$ are referred to as *"prop"-ositional terms* and *"prop"-ositional atoms* respectively. Sequences of atoms and sequences of "prop"-ositional atoms are denoted $Atom^*$ and $Atom_{\mathcal{P}}{}^*$ respectively. We write $body :: goals$ to denote the concatenation of ("prop"-ositional) atom sequences $body$ and $goals$.

The set of idempotent substitutions from Var to $Term$ is denoted Sub and defined as usual. Propositional substitutions are truth assignments on Var. These may be viewed as partial mappings from Var to Σ_{prop} (with finite support) and like substitutions are extended to apply to other syntactic objects. The set of propositional substitutions is denoted $Sub_{\mathcal{P}}$. The empty (propositional) substitution is denoted ϵ. Assuming the standard ordering on substitutions, we let $\theta' \leq \theta$ denote that θ is *more general* than θ'. We often view (propositional) substitutions as sets of equations in solved form. The correspondence is well known (see for example [26]). The conjunction of propositional substitutions is defined as their union, if well defined. The conjunction of sets of propositional substitutions is defined in the natural way. A set, Ξ of propositional substitutions is associated with a corresponding propositional formula $prop_{\Xi}$ defined by:

$$prop_{\Xi} = \bigvee_{\xi \in \Xi} \; \bigwedge_{x \in dom(\xi)} x = \xi(x).$$

Note that for any set Ξ of propositional substitutions: $prop_{\Xi} = \bigvee_{\xi \in \Xi} prop_{\{\xi\}}$.

Example. Let Ξ denote the following set of propositional substitutions.

$$\Xi = \left\{ \begin{array}{l} \{x \mapsto true, y \mapsto true\}, \\ \{x \mapsto false, y \mapsto true\}, \\ \{x \mapsto false, y \mapsto false\} \end{array} \right\}.$$

Then

$$\begin{array}{rcl} prop_{\Xi} & \equiv & (x \wedge y) \vee (\neg x \wedge y) \vee (\neg x \wedge \neg y) \\ & \equiv & x \rightarrow y. \end{array}$$

We restrict our attention to definite logic programs which are defined as usual and assume a standard subset of Prolog built-ins (excluding dynamic predicates such as assert and retract). Similarly, "prop"-ositional logic programs consist of clauses involving "prop"-ositional atoms. Without loss of generality we assume that logic programs are in a normal form such that: (1) each of the atoms and built-ins occurring in a program clause is of the form $p(\bar{x})$; and (2) all explicit unifications are of the form $x = y$ or $x = t(\bar{y})$ where x, y are variables and $t(\bar{y})$ is a "flat" term. The use of normal forms of this type is standard practice in many program analysis and transformation tools. See for example [5]. Finally, if P is a logic program and s is a syntactic object then $C \ll_s P$ denotes a clause C from P which is renamed apart from s.

Semantics: Our analyses are formalized and implemented in terms of a standard, immediate consequences operator, T_P, the least fixed point of which provides the minimal model of a program P. This semantics is applied to "prop"-ositional logic programs for which the corresponding least fixed points are guaranteed to be finite. For concrete programs we require a more expressive semantics which enables us to reason also about answer substitutions for arbitrary goals. A natural choice is the s-semantics described in [19] which is defined in terms of a non-ground T_P operator denoted here T_P^V. The s-semantics has proven useful in the context of program analysis (see for example [2, 6]) due to the fact that it constitutes an appropriate bridge between the declarative concept of minimal model and the operational notion of answer substitutions. In the s-semantics the meaning of a program P is a set of facts, often referred to as the *non-ground minimal model* of P. Its ground instances are precisely the minimal model of P; and it determines precisely the answer substitutions for any initial goal to P. Intuitively, this means that for a program P and any goal G, the answers for G with P can be obtained by "solving" G in the (possibly infinite) *non-ground minimal model* of P instead of in P. In the following we show that the standard minimal model of a "prop"-ositional logic program $P^{\#}$ approximates the non-ground minimal model of the corresponding logic program P hence providing an approximation of the answer substitutions for any goal to P. We abusively denote both standard and non-ground minimal models of P by $M(P)$.

The *Prop* domain: In [14] the domain *Prop* of abstract substitutions is formalized as a Galois insertion denoted $(Prop, \gamma, 2^{Sub}, \alpha)$. The domain consists of equivalence classes of propositional formulae constructed using the connectives $\leftrightarrow$, $\wedge$ and $\vee$ ordered by implication (to which the value *false* is added). In [14] the authors show that these are exactly the propositional formulae which are satisfiable by the unit assignment (i.e. which assigns all variables the value *true*.). Further technical details can be found in [14] and in [32]. We review here the definitions required to formalize the concretization function $\gamma : Sub_{\mathcal{P}} \rightarrow 2^{Sub}$ which is applied when justifying our results.

We say that a truth assignment ξ satisfies a propositional formula f denoted $\xi \models f$ if $f\xi$ is a tautology. For groundness analysis, a substitution θ is associated with a corresponding truth assignment $assign_\theta = \lambda x.ground(x\theta)$[2]. The function γ is defined as:

$$\gamma(f) = \left\{ \theta \;\middle|\; \theta' \leq \theta \Rightarrow assign_{\theta'} \models f \right\}.$$

Example. Consider the propositional formula

$$\varphi = (X \wedge Y) \leftrightarrow Z.$$

The following substitutions are in $\gamma(\varphi)$:

$$\begin{aligned}
\theta_1 &= \{ Z \mapsto f(X, Y) \}, \\
\theta_2 &= \{ X \mapsto A, Y \mapsto h(B), Z \mapsto g(A, B) \}, \\
\theta_3 &= \{ X \mapsto a, Y \mapsto b, Z \mapsto f(c, d) \}.
\end{aligned}$$

The substitution

$$\theta_4 = \{ X \mapsto A, Y \mapsto C, Z \mapsto g(A, B) \}$$

is not in $\gamma(\varphi)$ because the truth assignment of its instance

$$\theta_4' = \left\{ \begin{array}{l} X \mapsto a, Y \mapsto C, Z \mapsto g(a, b), \\ A \mapsto a, B \mapsto b \end{array} \right\}$$

does not satisfy φ.

Magic-sets: The magic-set transformation [1] and other related techniques such as Alexander templates [34] originate as an optimization technique in the context of deductive databases. The common principle underlying these techniques is a transformational approach in which an enhanced program $magic(P;G)$ is derived from a given program P and goal G. The minimal model of the derived program is more efficient to compute and contains the information from $M(P)$ which is relevant for the goal G. This same approach has proven useful in the context of program analysis because the minimal model of a transformed program $magic(P;G)$ exhibits also information about the set of *calls* which arise in the computations of G.

In this paper we consider a slightly non-standard magic-set transformation defined as follows: let P be a logic program and $G = g_1, \ldots g_k$ an initial goal. The corresponding magic program is $magic(P;G) = P \cup P_{\mathcal{M}} \cup G_{\mathcal{M}}$ where:

$G_{\mathcal{M}} = \left\{ query\_g_i \leftarrow g_1, \ldots, g_{i-1} \;\middle|\; 1 \leq i \leq k \right\}$; and

[2] The predicate $ground(term)$ is true if and only if $term$ is ground.

$$P_{\mathcal{M}} = \left\{ \; query\_b_i \leftarrow query\_h, b_1, \ldots, b_{i-1} \;\middle|\; \begin{array}{l} h \leftarrow b_1, \ldots, b_n \in P, \\ 1 \leq i \leq n \end{array} \right\}.$$

The program $magic(P;G)$ has the property that if p is a call in a computation of G with P (assuming a left-to-right computation rule), then $query\_p$ is in the non-ground minimal model of $magic(P;G)$.

EXAMPLE. Consider the program P consisting of the single clause

$$p(X) \leftarrow q(X), p(f(X)).$$

and the initial goal $G = \; p(a)$. The corresponding transformed program is

$$magic(P;G) = \left\{ \begin{array}{l} query\_p(a). \\ query\_q(X) \leftarrow query\_p(X). \\ query\_p(f(X)) \leftarrow query\_p(X), q(X). \\ p(X) \leftarrow q(X), p(f(X)). \end{array} \right\}$$

The (non-ground) minimal model of $magic(P;G)$ contains the two facts $query\_p(a)$ and $query\_q(a)$ reflecting the two calls $p(a)$ and $q(a)$ which arise in the computation of G with P. Note that although G has no successful computations the minimal model of the transformed program does contain facts corresponding to possible queries.

The reader familiar with the magic-set technique will observe that we have forfeited the optimizations usually obtained by the magic-set technique. This because we introduce the original program clauses into the transformed program instead of replacing each clause $h \leftarrow b_1, \ldots, b_n \in P$ with a clause of the form $h \leftarrow query\_h, b_1, \ldots, b_n$. Consequently, $M(magic(P;G))$ includes all of the information in $M(P)$ and its evaluation is no more efficient. However, it is this choice which enables us to apply a compositional approach to introduce a goal independent analysis for call patterns as described in Section 4. An additional observation regarding our choice is mentioned in [6] (Example 6.11) where the authors observe that the standard magic-set transformation introduces some degree of inaccuracy. Namely, the transformed program may indicate calls (and hence also answers) which do not occur in any execution of the given goal and program. In contrast, the transformation assumed here has been shown to specify precisely the calls and answers for the given initial goal. This result is referenced to J. Steiner ("personal communication") in [6].

4 "PROP"-OSITIONAL GROUNDNESS ANALYSIS

This section formalizes our groundness analysis which is capable of providing two types of information: *success patterns* and *call patterns*. Success patterns constitute an approximation of a programs non-ground minimal model and can be applied to characterize the answer substitutions of an initial query. The analysis of success patterns is *goal independent*. It does not involve any information about an initial query. Given the result of the analysis a description of the answers for any initial query can be obtained very efficiently. Recall that the answers for a goal G with a program P are fully captured by solving G with the potentially infinite non-ground minimal model of P. Similarly, the answers for G are approximated by solving a suitable abstract goal $G^{\#}$ with the definitely finite non-ground minimal model of a corresponding abstract program $P^{\#}$.

Call patterns describe the set of calls which arise in the computations of a goal G with a program P. These can be applied for example to deriving mode declarations for a given program given some information regarding the modes of the potential initial queries. Contrary to common practice, we illustrate how this analysis can also be performed in a goal independent manner. Call patterns for G with P are derived in our analysis by evaluating the minimal model of the abstraction of a corresponding transformed program $magic(P;G)$. A compositional semantics is applied to partition this task into goal dependent and goal independent components. As a consequence goal dependent analyses become very fast as — surprisingly enough — most of the work is done in the goal independent phase.

The analysis consists of two phases: (1) program abstraction; and (2) bottom-up evaluation. In case call patterns are desired, program abstraction involves an additional program transformation based on the

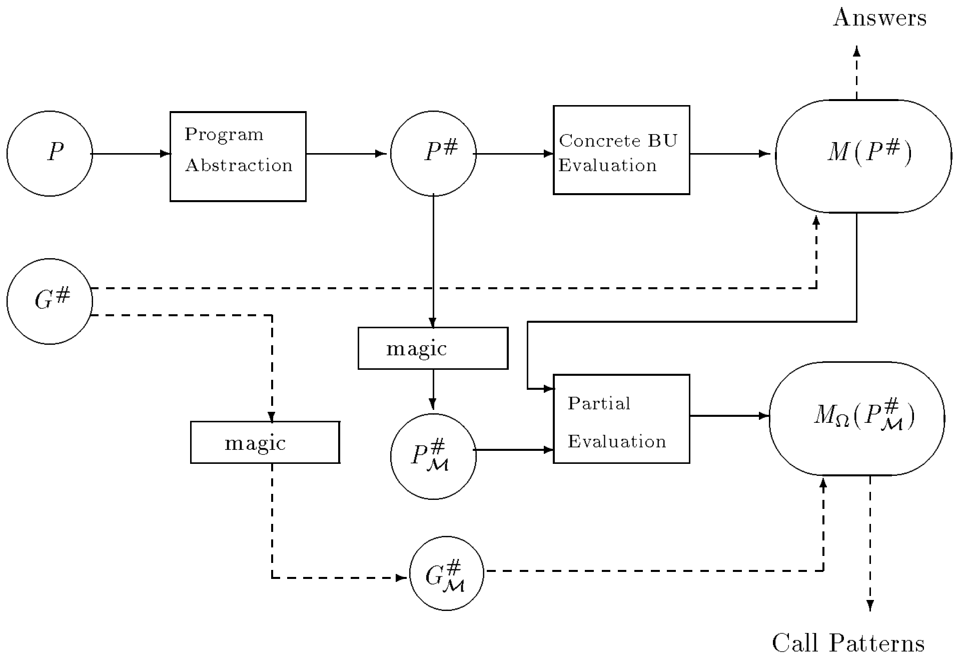

Figure 4: The analyzer and its components

magic-set approach. Figure 4 illustrates the analyzer and its components. The input is a Prolog program P and an optional initial call pattern $G^\#$. The solid lines correspond to the goal independent actions which involve only P. Two alternative types of information are provided: (1) a set $M(P^\#)$ of facts which can be queried to approximate the answers for arbitrary initial goals; and (2) a set $M_\Omega(P^\#_\mathcal{M})$ of binary clauses of the form $query\_b \leftarrow query\_h$, which can be applied to provide the call patterns for arbitrary initial goals. The dotted lines correspond to goal dependent actions which depend on the initial query description $G^\#$. Once the goal independent phase (solid lines) has been completed it can be applied to derive information for arbitrarily many initial call patterns (dotted lines) at almost no additional cost.

The abstraction To abstract a Prolog program P (or goal G) into a corresponding "prop"-ositional logic program $P^\#$ (or goal $G^\#$), first, P (or G) is transformed to a normal form in which all of the atoms are of the form $p(\bar{x})$, all built-ins involve distinct variables and all unifications are explicit. The abstraction is then performed by replacing each of the built-ins in the normal form program by corresponding propositional predicates which are specified by suitable definitions. The abstraction describes the relation that exists between the arguments of a successful call to the built-in predicate. The unification predicate $=/2$ introduced by the normalization process involves a call of the form $X = t$ where $vars(t) = \{X_1, \ldots, X_n\}$. Such a call is transformed to a predicate $\mathit{iff}(X, X_1, \ldots, X_n)$ the meaning of which is logically equivalent to the proposition $X \leftrightarrow (X_1 \wedge \cdots \wedge X_n)$. In particular, the predicate $\mathit{iff}(X)$ is logically equivalent to $X = true$. The abstract program includes definitions for all of those iff/n predicates introduced by the abstraction.

In a similar fashion, Prolog built-in predicates are abstracted. Arithmetic comparisons are straightforward because they require all arguments to be ground. Hence, $X > Y$ is translated to $\mathit{iff}(true, X, Y)$ which is logically equivalent to $X \wedge Y$. Similarly, in an arithmetic assignment X *is* T, both X and T are ground in any successful answer.

The transformation of representative (Prolog) built-ins is summarized in Table 1 where X, Y and Z are distinct variables and t is a term with $vars(t) = \{X_1, \ldots, X_n\}$. The built-in predicate $compound(X)$ is abstracted to the Prolog built-in *true* which for our purposes is also viewed as the propositional formula *true* providing no information about X. This is justified by the fact that theis predicate may succeed for any possibly non-ground compound term. The same holds for the predicate $var(X)$ which is also abstracted to *true*. The built-in $arg(X, Y, Z)$ specifies that Z is the X^{th} argument of term Y. Hence $arg(X, Y, Z)$ is characterized by the propositional formula $X \wedge (Y \rightarrow Z)$ because X must be ground (an integer) and if Y is ground then so is its X^{th} argument Z. Figure 5 illustrates the transformation of a Prolog program defining the *quick-sort* relation. The predicates $iff/1, iff/2, iff/3$ are given in Figure 1.

Negation by failure is treated by means of program transformation. Consider the standard definition of negation by failure by means of the following meta-program:

$$\begin{array}{rl} \backslash\!+(G) \leftarrow & call(G), !, fail. \\ \backslash\!+(G) \leftarrow & true. \end{array}$$

The meta-program induces a transformation which can be formalized by unfolding the negated calls with respect to this standard meta-program. For example, a clause of the form

$$a \leftarrow b, \backslash\!+(c), d$$

is transformed into the pair of clauses

$$\begin{array}{rl} a \leftarrow & b, c, fail. \\ a \leftarrow & b, d. \end{array}$$

In terms of success and call patterns this has the desired effect. The first clause does not influence the program's success set because of the *fail* goal which has no solutions. It does influence the call patterns as expected because c is called after b as in the computation involving $\backslash\!+(c)$. The second clause does influence the success set and d receives bindings from b but not from c just as in the computation involving $\backslash\!+(c)$. In our case we view these clauses declaratively, ignoring the cut and other control related intrinsic details. Hence we obtain an upper approximation of the success and call patterns much the same as when a pure Prolog program is approximated by a corresponding logic program for the purposes of program analysis. In the following we shall assume without loss of generality that programs do not contain negated calls. The formal justification of this assumption is based on the observation that unfolding negation by failure preserves both answers and call patterns for the other predicates. The proof appears in the appendix.

Theorem 4.1 *Let P be a Prolog program with negation by failure and P' the corresponding logic program in which the negated goals have been unfolded as described above. Then, P and P' have the same answers and call patterns (for a definition which ignores cuts).*

We often write goals of the form $p(true, X_2)$ as shorthand for $iff(X_1), p(X_1, X_2)$ which is the abstraction of any goal with a normal form $X_1 = ground\_term, p(X_1, X_2)$.

concrete built-in	abstract built-in	propositional meaning
$X > Y$	$iff(true, X, Y)$	$X \wedge Y$
X *is* Y	$iff(true, X, Y)$	$X \wedge Y$
$integer(X)$	$iff(X)$	X
$compound(X)$	$true$	$true$
$arg(X, Y, Z)$	$iff(X), iff(Y, \_, Z)$	$X \wedge (Y \rightarrow Z)$

Table 1: Abstraction of representative built-ins.

Approximating answers The analysis to approximate answer substitutions is illustrated in the upper half of Figure 4. A Prolog program P is first abstracted to a "prop"-ositional logic program $P^{\#}$ and a bottom-up evaluation is applied to derive the corresponding minimal model $M(P^{\#})$. The approximate answers for a goal pattern $G^{\#}$ are obtained by solving $G^{\#}$ into $M(P^{\#})$. The analysis is formalized by the following theorem, the justification of which is provided in the appendix.

Theorem 4.2 *Let P be a logic program, G an initial goal and let $P^{\#}$, $G^{\#}$ be the corresponding "prop"-ositional logic program and goal. Let X be a variable in G. If all answers of $G^{\#}$ with $M(P^{\#})$ bind X to $true$ then X is ground in any answer of G with P.*

```
qsort(X1,X2) ←
   X3 = [],
   qsort(X1,X2,X3)

partition(X1,X2,X3,X4) ←
   X1 = [],
   X3 = [],
   X4 = [].
partition(X1,X2,X3,X4) ←
   X1 = [X5 | X6],
   X3 = [X5 | X7],
   X5 <= X2,
   partition(X6,X2,X7,X4).
partition(X1,X2,X3,X4) ←
   X1 = [X5 | X6],
   X4 = [X5 | X7],
   X5 > X2,
   partition(X6,X2,X3,X7).

qsort(X1,X2,X3) ←
   X1 = [],
   X3 = X2.
qsort(X1,X2,X3) ←
   X1 = [X4 | X5],
   partition(X5,X4,X6,X7),
   qsort(X6,X2,X8),
   X8 = [X4 | X9],
   qsort(X7,X9,X3).
```

```
qsort(X1,X2) ←
   iff(X3),
   qsort(X1,X2,X3).

partition(X1,X2,X3,X4) ←
   iff(X1),
   iff(X3),
   iff(X4).
partition(X1,X2,X3,X4) ←
   iff(X1,X5,X6),
   iff(X3,X5,X7),
   iff(true,X5,X2),
   partition(X6,X2,X7,X4).
partition(X1,X2,X3,X4) ←
   iff(X1,X5,X6),
   iff(X4,X5,X7),
   iff(true,X5,X2),
   partition(X6,X2,X3,X7).

qsort(X1,X2,X3) ←
   iff(X1),
   iff(X3,X2).
qsort(X1,X2,X3) ←
   iff(X1,X4,X5),
   partition(X5,X4,X6,X7),
   qsort(X6,X2,X8),
   iff(X8,X4,X9),
   qsort(X7,X9,X3).
```

Figure 5: Quick-sort and its abstraction.

```
query_reverse(Xs1,Ys1) ←                query_append(Xs1,Ys,Zs1) ←
   query_reverse(Xs,Ys),                   query_append(Xs,Ys,Zs),
   iff(Xs,X,Xs1).                          iff(Xs,X,Xs1),
query_append(Ys1,A,Ys) ←                   iff(Zs,X,Zs1).
   query_reverse(Xs,Ys),
   iff(Xs,X,Xs1),
   reverse(Xs1,Ys1),
   iff(A,X).
```

Figure 6: The goal independent $P^{\#}_{\mathcal{M}}$ for *naive reverse*.

```
query_reverse(_,_) ← query_reverse(false,_).
query_append(true,true,Ys) ← query_reverse(true,Ys).
query_append(_,false,Ys) ← query_reverse(false,Ys).
query_append(false,_,Ys) ← query_reverse(false,Ys).
query_append(_,Ys,_) ← query_append(false,Ys,false).
```

Figure 7: The goal independent $M_\Omega(P^{\#}_{\mathcal{M}})$ for *naive reverse*.

Consider for example the *naive-reverse* program P in Figure 1 and the minimal model $M(P^{\#})$ depicted in Figure 2. The "prop"-ositional goal $G^{\#} = reverse(true, Y)$ is the abstraction of any goal of the form $G = reverse(list, Y)$ where *list* is ground. The single answer of $G^{\#}$ with $M(P^{\#})$ binds Y to *true* indicating that any answer of G with P binds Y to a ground term.

Approximating calls The analysis to approximate call patterns is illustrated in the lower half of Figure 4. Instead of applying bottom-up evaluation to a magic program in the standard way, we take a compositional approach which has the advantage that the main part of the analysis becomes goal independent. Consequently, the call patterns for arbitrary initial goals are derived very quickly. Moreover, the complexity of analyses is reduced as we apply bottom-up evaluation to smaller program modules. The effect is similar to that obtained using "condensing" as suggested by Jacobs and Langen in [25]. It is also related to the work described in [20] which describes a goal independent analysis for call patterns based on the abstraction of the semantic characterization described in [21]. We illustrate this type of analysis with an example.

Consider again the *naive reverse* program P and its "prop"-ositional counterpart $P^{\#}$ from Figure 1. Recall that the definition of the magic-set transformation:

$$magic(P^{\#}; G^{\#}) = P^{\#} \cup P^{\#}_{\mathcal{M}} \cup G^{\#}_{\mathcal{M}}$$

for any initial call pattern $G^{\#}$. The $P^{\#}_{\mathcal{M}}$ component for *naive reverse* is illustrated in Figure 6. The module $P^{\#}_{\mathcal{M}}$ is an *open* program. The predicates *reverse*/2 and *append*/3 are defined in the module, $P^{\#}$ and the predicates *query_reverse*/2 and *query_append*/3 are only partially defined. Their full definitions depend on the clauses in the module $G^{\#}_{\mathcal{M}}$ which are determined by the specific initial call pattern. We cannot apply standard bottom-up evaluation to $P^{\#}_{\mathcal{M}}$ without providing the clauses in the other modules. However, we can partially evaluate $P^{\#}_{\mathcal{M}}$ by unfolding the clause bodies with the facts in $M(P^{\#})$. The result is a set of binary clauses of the form $query\_b \leftarrow query\_h$ which we denote $M_\Omega(P^{\#}_{\mathcal{M}})$ (Ω denotes that the program is open and that its model is only partially evaluated). Figure 7 illustrates the goal independent partially evaluated clauses for *naive reverse*. The result is similar to applying the compositional bottom-up semantics for logic programs defined in [3, 4]. However, there is a technical difference: we choose not to further unfold the clauses in $M_\Omega(P^{\#}_{\mathcal{M}})$ with each other (i.e. until a fixed point is reached). For example, the first clause in Figure 7 can be used to unfold the second clause obtaining the additional binary clause

```
query_append(true,true,Ys) ← query_reverse(false,_).
```

Such unfolding potentially introduces exponentially many additional binary clauses[3]. Our experimental results indicate that it is far more efficient to perform this extra unfolding when the goal dependent module $G^{\#}_{\mathcal{M}}$ is considered and standard bottom-up evaluation can be applied. Table 2 (below) indicates that the cost of the additional unfolding at this time is marginal. The essential point is to observe that $M_{\Omega}(P^{\#}_{\mathcal{M}})$ constitutes a concise goal independent representation of the call patterns for arbitrary initial goals. For example if $G^{\#} = reverse(true, \_)$, describing a call to $reverse/2$ with a ground first argument, then $G^{\#}_{\mathcal{M}} = query\_reverse(true, \_)$ and

$$M(P^{\#} \cup P^{\#}_{\mathcal{M}} \cup G^{\#}_{\mathcal{M}}) = M(M_{\Omega}(P^{\#}_{\mathcal{M}}) \cup G^{\#}_{\mathcal{M}}) =$$

$$\{\ \mathtt{query\_reverse(true,\_)},\quad \mathtt{query\_append(true,true,\_)}\ \}.$$

Theorem 4.3, the justification of which is provided in the appendix, then enables us to derive the modes

$$\{\ \mathtt{reverse(g,?)},\quad \mathtt{append(g,g,?)}\ \}.$$

Theorem 4.3 *Let $(P; G)$ and $(P^{\#}; G^{\#})$ be corresponding logic- and "prop"-ositional logic- programs and initial goals. Let p/n be a predicate in P and $1 \leq i \leq n$. If $t_i = true$ for every atom of the form $query\_p(t_1, \ldots, t_i, \ldots, t_n)$ in $M(magic(P^{\#}; G^{\#}))$ then the i^{th} argument of any call to p/n in a computation of G with P is ground.*

5 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

Our implementation of the functionality visualized by Figure 4, is based on a series of source-to-source program transformations. The main components of the system are the upper part, which computes from a Prolog program P its abstract semantics $M(P^{\#})$, and the lower part, which computes from P and $M(P^{\#})$ the set of binary clauses $M_{\Omega}(P^{\#}_{\mathcal{M}})$. The abstracting transformation and our version of the magic set transformation are described in Sections 4 and 3 respectively. This section describes the remaining three sub-components: (a) the evaluation of the minimal model $M(P^{\#})$ for a given abstract program $P^{\#}$; (b) the partial evaluation of the clauses in the magic module $P^{\#}_{\mathcal{M}}$ with respect to $M(P^{\#})$ obtaining $M_{\Omega}(P^{\#}_{\mathcal{M}})$; and (c) the further evaluation of $M_{\Omega}(P^{\#}_{\mathcal{M}})$ when a specific module $G^{\#}_{\mathcal{M}}$ is provided. We close this section with an experimental evaluation of our analysis.

Evaluation of $M(P^{\#})$: The technique described here is general and can be applied to evaluate the minimal model of any program P (which has a finite minimal model). In particular we apply this technique to our abstract logic programs. The minimal model of a program P is evaluated by repeated application of the following non-ground immediate consequence operator, starting from the empty set, until no new atoms are derived.

$$T^V_P(I) = \left\{ h\theta \;\middle|\; \begin{array}{l} h \leftarrow b_1, \ldots, b_n \in P, \\ a_1, \ldots, a_n \ \ are\ renamed\ variants\ from\ I, \\ \theta = mgu(\langle b_1, \ldots, b_n \rangle, \langle a_1, \ldots, a_n \rangle) \end{array} \right\}$$

An iteration evaluating $T^V_P(I)$ must consider the unification of all tuples $\langle a_1, \ldots, a_n \rangle$ of atoms in I with the body of (each) clause $h \leftarrow b_1, \ldots, b_n \in P$. The basic idea underlying our implementation is that the facts in I are maintained in the Prolog database and new instances of the head h can be added by "solving" the body in the set of facts I and asserting the corresponding instance of the head.

Our implementation is based on the program transformation induced by the simple interpreter given in Figure 3. This interpreter in itself provides a working implementation. However, by using the induced transformation, evaluation is, on the average, one order of magnitude faster. In our approach, the program P is transformed to a new program P' which when executed leaves the minimal model of P in the Prolog database. The transformation is as follows:

[3]For the *naive reverse* example in Figure 7 we obtain 15 clauses instead of 5, for peep (see Table 3 below) we obtain 1964 clauses instead of 229.

Apart from the clauses defining the predicates $iterate\_tp/0$, $cond\_assert/1$ and $in\_database/1$ from Figure 3, for each clause `h(t̄)` $\leftarrow$ `body` in P, the transformed program P' includes a clause of the form

$$tp \leftarrow body,\ cond\_assert(h(\bar{t})),\ fail.$$

The query, `?- iterate_tp`, if convergent, computes the non-ground minimal model of P. Indeed, the predicate `tp` corresponds to the T_P^V operator and the execution of `iterate_tp` results in repeated applications of this operator. If P is a Datalog program, the execution is guaranteed to terminate.

As an optimization, we have replaced the variance check in Figure 3 by a subsumption check in our implementation. We still take advantage of a concise non-ground representation. On the other hand, we do not maintain the s-semantics. This optimization provides a two-fold speed-up with respect to both the standard ground semantics as well as with respect to the non-ground s-semantics. It is straightforward to redefine the predicate `cond_assert/1` so as to obtain any of these alternative versions of the semantics. Also, as an optimization, the call to `cond_assert` can be specialized since its argument is partially known. This provides for an additional 10% speed-up in the implementation.

In addition, we have experimented with two of the main optimizations described in the literature. These are

1. *semi-naive evaluation* (see for example [35] Chapter 3) which restricts the number of tuples $a_1, \ldots, a_n$ from I which need be considered for a clause $h \leftarrow b_1, \ldots, b_n$ in an iteration evaluating $T_P(I)$. The restriction imposed is that each tuple must contain at least one "new" atom (derived in the last iteration).

   We have experimented with this classic optimization, however it does not prove beneficial due to the fact that we consider a very small underlying Herbrand universe. The overhead involved in distinguishing between "new" and "old" atoms does not pay off in our case.

2. evaluation following *strongly connected components* (see for example [22, 33]) where a topological sort of the strongly connected components in the program's call graph is applied to impose a preferred order on the bottom-up evaluation. This optimization improves drastically the efficiency of our system.

   A directed graph $G = \langle V, E \rangle$ is said to be *strongly connected* if every vertex in V is reachable from every other vertex in V. Given a directed graph G, a subgraph G' of G is said to be a *strongly connected component* of G if G' is a maximal subgraph in G that is strongly connected. The dependency graph of program P is the directed graph $\langle V, E \rangle$ where V is the set of predicates of P and $(p, q) \in E$ if and only if a call to p occurs in the body of a clause for q. The strongly connected components, $\langle P_1, \ldots, P_n \rangle$, of P are viewed as sets of clauses so that $P = \bigcup_{i=1}^{n} P_i$. Moreover, we assume that these components are topologically sorted so that if predicate p has a definition in P_i, then p is not called from P_j, $j < i$.

   As before we transform a program P into a program P' such that execution of P' leaves the minimal model of P in the Prolog database. The program transformation described above is enhanced to ensure that the minimal model of P is evaluated component by component. For each clause `h(t̄)` $\leftarrow$ `body` in P_i, the transformed program P' is defined to include a clause of the form

   $$tp_i \leftarrow body,\ cond\_assert(h(\bar{t})),\ fail.$$

   For every i, P' contains the clauses

   $$\begin{array}{l} iterate\_tp_i \leftarrow tp_i,\ fail. \\ iterate\_tp_i \leftarrow \\ \quad database\_changed,\ retractall(database\_changed), \\ \quad iterate\_tp_i. \end{array}$$

   Finally, there is an overall driver defined by:

$$iterate\_tp :- iterate\_tp_1.$$
$$iterate\_tp :- iterate\_tp_2.$$
$$\ldots\ldots\ldots$$
$$iterate\_tp :- iterate\_tp_n.$$

This optimization provides a speed-up of several orders of magnitude for large programs in which the call graph typically has several layers of strongly connected components. In fact, without this optimization, the analysis of the larger benchmarks we have experimented with is not practically feasible. The full implementation of this component is about 300 lines of Prolog code — including the dependency analysis.

Partial evaluation of $M_\Omega(P^{\#}_{\mathcal{M}})$: The module $P^{\#}_{\mathcal{M}}$ introduced by the magic-set transformation described in Section 3 contains clauses of the form

$$query\_p_i(\bar{t}_i) \leftarrow query\_h(\bar{t}_0), p_1(\bar{t}_1), \ldots, p_{i-1}(\bar{t}_{i-1}).$$

Once $M(P^{\#})$ is computed, a clause of this form can be partially evaluated by solving that part of its body which is defined in $M(P^{\#})$ directly in $M(P^{\#})$. This results in a set of binary clauses of the form

$$query\_p_i(\bar{t}'_i) \leftarrow query\_h(\bar{t}'_0).$$

The resulting program is denoted $M_\Omega(P^{\#}_{\mathcal{M}})$. Our experiments indicate that it is beneficial to eliminate identical clauses introduced by this stage.

Evaluation of $M(M_\Omega(P^{\#}_{\mathcal{M}}) \cup G^{\#}_{\mathcal{M}})$: This is the goal dependent stage. Given a partially evaluated module together with a description of its undefined predicates the remaining evaluation can be performed. The evaluation of $M(M_\Omega(P^{\#}_{\mathcal{M}}) \cup G^{\#}_{\mathcal{M}})$ can be performed in the same manner as any other program (see above). However, the particular binary form of the clauses suggests a simple and more efficient approach which is also based on transforming the given (binary) program to a new program P'.

1. For each predicate p/n in $M_\Omega(P^{\#}_{\mathcal{M}})$ we add to P' a clause of the form

$$tp\_p(\bar{x}) \leftarrow \; in\_database(p(\bar{x})) \; \rightarrow !,\; fail \;;\; assert(p(\bar{x})),\; fail.$$

2. For each clause of the form $p(\bar{t}_1) \leftarrow q(\bar{t}_2)$ in $M_\Omega(P^{\#}_{\mathcal{M}})$ we add to P' a clause of the form

$$tp\_q(\bar{t}_2) \leftarrow \; tp\_p(\bar{t}_1).$$

Execution of the query $G^{\#}$ will now leave the desired result in the Prolog database. This part of the implementation is about 200 lines of straightforward Prolog code.

We use the benchmark programs proposed by Le Charlier *et al.*, a description of which can be found in [27]. The programs range in size from 2 clauses with 5 variables (occurrences) to 227 clauses with 869 variables. Table 2 summarizes the analysis times (in seconds, results on a SPARC station 1). The cost of normalizing and abstracting the source Prolog programs is indicated in the column "trans". This includes the dependency analysis (about 10% of the total transformation time) and the time it takes ProLog by BIM to compile the transformed "prop"-ositional logic program (about 50%). The next two columns indicate the cost of computing $M(P^{\#})$ and $M_\Omega(P^{\#}_{\mathcal{M}})$ which constitute the goal-independent phase of the analysis. The next two columns give the cost of the goal dependent phase. Approximating answers is basically free. For call patterns there is some amount of evaluation on $M_\Omega(P^{\#}_{\mathcal{M}}) \cup G^{\#}_{\mathcal{M}}$. However, the experiments indicate that this is quite limited. The last two columns indicate the total cost of the analysis for a single initial goal. The first of these columns indicates the cost using a compositional approach. The second, gives for comparison the cost in a non-compositional approach. Note that for most of the benchmarks a compositional approach reduces the cost. Moreover, note that in a compositional approach, the cost of the analysis for all subsequent initial goals is negligible.

By keeping the implementation completely in Prolog, we have arrived at a much smaller system than the 6000 lines of C code mentioned in [27]. By adopting a transformational approach we can easily adapt the system to other types of analysis provided these are defined in terms of program abstractions. As an example, a polymorphic type analysis has recently been described in [9]. In addition, the timings for approximating answers are comparable with those obtained by the Toupie system described in [13]. The total results compare reasonably with the top-down analysis described in [27] which is faster on most of the benchmarks. However, our approach is faster if call patterns for several initial goal patterns are required. This due to the fact that once the goal independent phase has been performed the additional effort for specific initial call patterns is negligible. Moreover, for the specific case of analyses based on *Prop*, goal independent analysis does not incur any loss of precision as the domain is condensing [25].

Program	trans	goal indep.		goal dep.		full analysis	
		$M(P^{\#})$	$M(P^{\#}_{\mathcal{M}})$	answ	calls	comp.	non-comp.
append	0.24	0.01	0.04	0.00	0.01	0.30	1.20
qsort	0.33	0.02	0.11	0.00	0.01	0.47	0.66
queens	0.57	0.04	0.22	0.00	0.01	0.84	1.00
pg	0.85	0.07	0.42	0.00	0.03	1.37	1.78
plan	1.18	0.08	0.64	0.00	0.03	1.93	2.62
gabriel	1.68	0.15	0.87	0.00	0.09	2.79	3.83
disj	2.38	0.45	5.39	0.00	0.05	8.27	6.18
cs	2.81	0.64	5.47	0.00	0.10	9.02	6.83
kalah	3.64	0.43	2.65	0.00	0.11	6.83	8.92
press	4.60	0.65	3.45	0.00	0.37	9.07	11.33
read	4.43	0.52	2.99	0.00	0.19	8.13	13.15
peep	4.56	0.77	3.10	0.00	0.07	8.50	8.08

Table 2: Time efficiency of the analyses.

Table 3 provides details concerning the size of the structures created by the analyses and their accuracy. It is interesting to note that the size of the objects created are quite reasonable in spite of the fact that disjunctive normal forms are not the most efficient way to represent propositional formulae. The second column indicates the number of facts in $M(P^{\#})$[4]. The third column gives the maximal number of facts for each program. The fourth column gives the number of clauses in $M_{\Omega}(P^{\#}_{\mathcal{M}})$. While it is difficult to quantify the accuracy of the goal independent analysis (e.g. for $append(x, y, z)$ we get $x \wedge (y \leftrightarrow z)$), we count the number of useful modes derived by the goal dependent phase for given call patterns using the magic-set transformation. The last two columns in Table 3 give the number of modes derived (for standard initial goals) and the total number of arguments for each program. Interestingly enough, we derive basically the same[5] number of modes as in the top-down goal dependent analysis described in [27].

6 CONCLUSION

We have illustrated how program abstraction can be applied to analyze complex properties of programs and obtain useful approximations of program behavior. Although equivalent to the *Prop* analyses proposed in [23] and [12], the contribution of this paper can be measured by the ease in which we obtain a highly efficient implementation. This is due to the fact that analyses involve only standard operations such as concrete unification which are readily available and highly optimized in existing systems. The basic idea developed in this paper is similar to and in fact inspired by that presented in [13]. The main difference is that we apply a transformation into Datalog instead of into the language Toupie. This simplifies the implementation of analyses as well as their justification. In addition, we apply a novel technique so that call patterns for arbitrary initial goals are evaluated in an efficient way. This approach is easily generalized for analyses

[4] Recall that non-ground atoms are sometimes included to achieve a more concise representation.

[5] The small differences are due to the fact that some of the predicates defined in the benchmarks of [27] are predefined built-ins in ProLog by BIM.

Program	$M(P^{\#})$		$M(P^{\#}_{\mathcal{M}})$	modes	
	facts	max	clauses	modes	args
append	3	8	4	1	3
qsort	6	28	12	4	9
queens	11	26	25	7	11
pg	16	120	41	20	31
plan	36	100	80	20	32
gabriel	43	218	102	17	56
disj	59	170	276	38	58
cs	93	1394	294	56	92
kalah	109	1118	238	78	122
press	146	458	317	14	135
read	97	581	272	32	116
peep	105	380	229	34	63

Table 3: Space efficiency and accuracy of the analyses.

which are defined in terms of abstract interpretation by applying the approach described in [7]. We focus on groundness analysis; however, the same approach applies directly for other types of analyses involving program properties which can be expressed as propositional formulae on variables. An additional example, is the polymorphic type analysis, recently described in [9].

We believe that the combination of bottom-up evaluation with techniques based on program transformations and abstract compilation provide a versatile and promising approach for the efficient analysis of logic programs. The advantage of a bottom-up approach is the relative simplicity of the underlying evaluation mechanism. The advantage of a transformational approach is that analyses can become more complex without forfeiting this simplicity. The advantage of abstract compilation is that abstract operations need not be defined and implemented. Moreover, abstract programs can be compiled efficiently hence taking advantage of advancing Prolog technology. Ongoing research focuses on providing a better understanding of the limitations of this approach.

Acknowledgment

This work was partially supported by CEC DGXIII ESPRIT Project "PRINCE" and through DWTC project IT/IF/4. We thank Marc Corsini, Saumya Debray, John Gallagher, Roberto Giacobazzi and Pascal Van Hentenryck for useful comments and suggestions. Pascal Van Hentenryck provided us with his benchmark programs. John Gallagher shared with us his implementation for a magic-set transformation.

References

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

[2] 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, 1993.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. Meo. Contributions to the semantics of open logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, , pages 570–580, ICOT, Japan, 1992. Association for Computing Machinery.

[4] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A compositional semantics for logic programs. *Journal of Theoretical Computer Science*, 122(1–2):3–47, 1994.

[5] M. Bruynooghe. Compile-time garbage collection or how to transform programs in an assignment-free language into code with assignments. In L.G.L.T. Meertens, editor, *IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, pages 113–129. Elsevier (North-Holland), 1986.

[6] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Journal of Theoretical Computer Science*, 124:93–125, 1994.

[7] M. Codish, S. K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proceedings of the Twentieth Annual ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press, 1993.

[8] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 114–129, Massachusetts Institute of Technology, Cambridge, Massachusetts 021-42, 1993. The MIT Press.

[9] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proceedings of the First Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, September 1994.

[10] M. Codish, M. Falaschi, and K. Marriott. Suspension analysis for concurrent logic programs. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 331–345, Paris, France, 1991. The MIT Press.

[11] M. Codish, M. Falaschi, and K. Marriott. Suspension analysis for concurrent logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, May 1994.

[12] P. Codognet and G. Filè. Computations, abstractions and constraints. In *Proceedings Fourth IEEE International Conference on Computer Languages*. IEEE Computer Society Press, 1991.

[13] 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 *Proceedings of the Fifth International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, Talin, August 1993. Springer Verlag.

[14] A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 322–327, Amsterdam, The Netherlands, July 15–18 1991. IEEE Computer Society Press.

[15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[16] P. W. Dart. On derived dependencies and connected databases. *The Journal of Logic Programming*, 11(1 & 2):163–188, July 1991.

[17] S. K. Debray and D. S. Warren. Detection and optimization of functional computations in Prolog. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, volume 225 of Lecture Notes in Computer Science, pages 490–504, London, 1986. Springer-Verlag.

[18] W. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[19] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[20] M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proc. SAC'94*, 1994.

[21] M. Gabbrielli and M. C. Meo. Fixpoint semantics for partial computed answer substitutions and call patterns. In G. Levi and H. Kirchner, editors, *Algebraic and Logic Programming, Third International Conference*, LNCS 632, pages 84–99, Volterra, Italy, September 2–4, 1992. Springer-Verlag.

[22] J. Gallagher and A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margherita Ligure, Italy, 1994. The MIT Press.

[23] R. Giacobazzi, S. K. Debray, and G. Levi. A generalized semantics for constraint logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 581–591, ICOT, Japan, 1992. Association for Computing Machinery.

[24] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(4):349–366, 1992.

[25] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *The Journal of Logic Programming*, 13(2 and 3):291–314, 1992.

[26] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[27] B. Le Charlier and P. Van Hentenryck. Groundness analysis for Prolog: implementation and evaluation of the domain Prop. In *Proceedings Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1993.

[28] G. Levi. Models, unfolding rules and fixpoint semantics. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1649–1665, Seatle, 1988. ALP, IEEE, The MIT Press.

[29] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2^{nd} edition, 1987.

[30] K. Marriott and H. Søndergaard. A tutorial on abstract interpretation of logic programs. Notes distributed at the North American Conference on Logic Programming, 1989.

[31] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems, ACM-LOPLAS*, 2(1–4):181—196, March–December 1993.

[32] K. Marriott, H. Søndergaard, and N. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems, ACM-TOPLAS*, 16(3):607–648, May 1994.

[33] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):501–517, August 1994. A shorter version appeared in VLDB, 1990.

[34] J. Rohmer, R. Lescœr, and J.-M. Kerisit. The Alexander method, a technique for the processing of recusive axioms in deductive databases. *New Generation Computing*, 4, 1986.

[35] J. Ullman. *Principles of Database and Knowledge – Base Systems*, volume 1. Computer Science press, 1988.

A APPENDIX: PROOFS OF SELECTED THEOREMS

In this appendix we first provide the definitions and results needed to justify Theorems 4.2 and 4.3. This is facilitated by introducing a transition system semantics, "lifting" it and showing that the lifted semantics of a "prop"-ositional program corresponds to an abstract interpretation (using *Prop*) of the original program. We then provide the justification of Theorem 4.1. This involves showing that negation by failure can be unfolded while preserving the answers and call patterns of the original program. Consequently, we may assume without loss of generality that programs do not contain negated goals, simply viewing negation as yet another predicate definition.

The following concretization function formalizes a notion of approximation between sets of "prop"-ositional atoms and atoms.

Definition A.1
Define $\bar{\gamma} : 2^{Atom_{\mathcal{P}}} \rightarrow 2^{Atom}$ *by:*

$$\bar{\gamma}(S) = \left\{\ p(\bar{x})\theta \,\middle|\ p/n \in \Pi, \Xi = \{\xi \mid p(\bar{x})\xi \in S\}, \theta \in \gamma(prop_{\Xi})\ \right\}.$$

While our analyses are bottom-up, it is technically more convenient to prove correctness with respect to a top-down transition system semantics which is applied to specify the *non-ground success set* of a program.

Definition A.2 *standard transition system semantics*
Let P *be a logic program. Then,* $\rightarrow_P$ *is the smallest relation on* $State = Atom^* \times Sub$ *such that* $s \rightarrow_P s'$ *if:*

1. $s = \langle p(\bar{x}) :: goals; \theta\rangle$; $s' = \langle \rho(body) :: goals; \theta\rangle$; $p(\bar{y}) \leftarrow body \ll_s P$ *and* $\rho = \{\bar{y} \mapsto \bar{x}\}$[6].
2. $s = \langle x = t :: goals; \theta\rangle$; *and* $s' = \langle goals; mgu(\theta \cup \{x = t\})\rangle$.
3. *additional cases for each built-in.*

The non-ground success set of a program P *is defined as*

$$SS(P) = \{p(\bar{x})\theta \mid \langle p(\bar{x}); \epsilon\rangle \rightarrow_P^* \langle true, \theta\rangle\}.$$

It is well-known that the non-ground success set of any ("prop"-ositional) logic program P is equal to its non-ground minimal model.

Theorem A.3 *[19]* *For any logic program* P, $SS(P) = lfp(T_P^V)$.

The proof of Theorem 4.2 is facilitated by introducing the following "lifted" transition relation for "prop"-ositional logic programs.

Definition A.4 *lifted transition system semantics*
The lifted transition relation for a "prop"-ositional logic program $P^{\#}$ *is the smallest relation* $\Rightarrow_{P^{\#}}$ *on* $State_{\mathcal{P}} = Atom_{\mathcal{P}}^* \times 2^{Sub_{\mathcal{P}}}$ *such that* $s \Rightarrow_{P^{\#}} s'$ *if:*

1. $s = \langle p(\bar{x}) :: goals; \Xi\rangle$; $s' = \langle \rho(body) :: goals; \Xi\rangle$; $p(\bar{y}) \leftarrow body \ll_s P$ *and* $\rho = \{\bar{y} \mapsto \bar{x}\}$[6].
2. $s = \langle iff(\bar{t}) :: goals; \Xi\rangle$; $s' = \langle goals; \Xi \wedge \Xi'\rangle$; *and* $\Xi' = \left\{\ mgu(\bar{t}, \bar{s}) \,\middle|\ iff(\bar{s}) \in P^{\#}\ \right\}$.

The non-ground "prop"-ositional success set of $P^{\#}$ *is defined as:*

$$SS'(P^{\#}) = \left\{\ p(\bar{x})\xi \,\middle|\ \langle p(\bar{x}); \{\epsilon\}\rangle \Rightarrow_{P^{\#}}^* \langle true, \Xi\rangle, \xi \in \Xi\ \right\}.$$

[6] If $\bar{x} = \langle x_1, \ldots, x_n\rangle$ and $\bar{y} = \langle y_1, \ldots, y_n\rangle$ then $\{\bar{y} \mapsto \bar{x}\}$ denotes the substitution $\{y_i \mapsto x_i \mid 1 \leq i \leq n\}$.

The following lemma is a straightforward consequence of Definitions A.2 and A.4.

Lemma A.5 *For any "prop"-ositional logic program P, $SS(P) = SS'(P)$.*

Our motivation for introducing sets of substitutions is related to the fact that we would like a transition involving a "prop"-ositional predicate of the form $\mathit{iff}(X, X_1, \ldots, X_n)$ to mimic a transition involving a built-in of the form $X = t$ with $vars(t) = \{X_1, \ldots, X_n\}$. Since the intended meaning of $\mathit{iff}/n+1$ is specified in terms of a set of ground facts, the concrete transition is mimicked by a transition which considers a set of clauses for such predicates.

The semantics of a logic program and corresponding "prop"-ositional logic program are related by introducing an approximation relation $\propto \ \subseteq State_{\mathcal{P}} \times State$.

Definition A.6 $\langle G^{\#}, \Xi\rangle \propto \langle G, \theta\rangle$ *iff* $G^{\#}$ *is the abstraction of* G *and* $\theta \in \gamma(\Xi)$.

Theorem A.7 *Let P and $P^{\#}$ be corresponding logic- and "prop"-ositional logic- programs and let $t \in State_{\mathcal{P}}$, $s, s' \in State$ such that $t \propto s$. If $\ s \rightarrow_P s' \ $ then $\ \exists t'.\ (t \Rightarrow_{P^{\#}} t') \ \wedge \ (t' \propto s')$.*

Proof. Assume the premise of the theorem, let $s = \langle g :: goal; \theta\rangle$ and let $s \rightarrow_P s'$. There are three cases to consider: (1) g is of the form $p(\bar{x})$; (2) g is of the form $x = term$; and (3) g is a Prolog built-in. Case (1) is straightforward as in both semantics a renaming is applied to the clause body. Consider case (2):

By Definition A.2, $s' = \langle goal; \theta'\rangle$ and $\theta' = mgu(\theta \cup \{x = term\})$. From Definition A.6, t is of the form $\langle \mathit{iff}(\bar{x}) :: goal^{\#}; \Xi\rangle$, where $goal^{\#}$ is the abstraction of $goal$, $\bar{x} = \langle x, x_1, \ldots, x_n\rangle$, $vars(term) = \{x_1, \ldots, x_n\}$ and $\theta \in \gamma(\Xi)$. Let $\Xi' = \{mgu(\bar{x}, \bar{s}) \mid \mathit{iff}(\bar{s}) \in P^{\#}\}$. By Definition A.4, $t \Rightarrow_{P^{\#}} t'$ where $t' = \langle goal^{\#}; \Xi \wedge \Xi'\rangle$. Since $goal^{\#}$ is the abstraction of $goal$, it is sufficient to prove that $\theta' \in \gamma(prop_{\Xi \wedge \Xi'})$ implying that $t' \propto s'$. Let $\varphi = x \ \leftrightarrow \ (x_1 \ \wedge \ \cdots \ \wedge \ x_n)$. By correctness of abstract unification on *Prop* (see for example Lemma 5.14 in [10]), $\theta \in \gamma(prop_{\Xi})$ implies that $\theta' \in \gamma(prop_{\Xi} \ \wedge \ \varphi)$. The theorem follows because $(prop_{\Xi} \ \wedge \ \varphi)$ and $prop_{\Xi \wedge \Xi'}$ are logically equivalent. To see this observe that by definition:

$$prop_{\Xi \wedge \Xi'} = \bigvee_{\xi \in \Xi \wedge \Xi'} prop_{\{\xi\}} \quad = \bigvee_{\substack{\xi \in \Xi \\ \xi' \in \Xi'}} (prop_{\{\xi\}} \ \wedge \ prop_{\{\xi'\}}).$$

Case (3) for Prolog built-ins is similar. $\Box$

An immediate consequence of Theorem A.7 is the following:

Corollary A.8
Let P and $P^{\#}$ be corresponding logic- and "prop"-ositional logic- programs. Then $SS(P) \subseteq \bar{\gamma}(SS'(P^{\#}))$.

Proof. Let $p \in SS(P)$. By Definition A.2, p is of the form $p(\bar{x})\theta$ and $\langle p(\bar{x}); \epsilon\rangle \rightarrow_P^* \langle true; \theta\rangle$. Since $\langle p(\bar{x}); \{\epsilon\}\rangle \propto \langle p(\bar{x}); \epsilon\rangle$, so by finite induction using Theorem A.7, $\langle p(\bar{x}); \{\epsilon\}\rangle \Rightarrow_{P^{\#}}^* \langle true; \Xi\rangle$ and $\theta \in \gamma(prop_{\Xi})$. Let $\Xi' = \{\xi \mid p(\bar{x})\xi \in SS'(P^{\#})\}$. Then by Definition A.4, $\Xi \subseteq \Xi'$ which implies that $\theta \in \gamma(prop_{\Xi'})$ giving by Definition A.1 that $p(\bar{x})\theta \in \bar{\gamma}(SS'(P^{\#}))$. $\Box$

Theorem 4.2 now follows as $SS(P) \subseteq \bar{\gamma}(SS'(P^{\#})) = \bar{\gamma}(SS(P^{\#})) = \bar{\gamma}(\mathit{lfp}(T^{V}_{P^{\#}}))$. It is interesting to note that taking the ground instances of the non-ground minimal model does not introduce imprecision as for any set of "prop"-ositional atoms P, $\bar{\gamma}(P) = \bar{\gamma}(P')$ where P' is the set of all ground instances of the elements in P. Theorem 4.3 follows from the correctness of the magic-set transformation (see for example [6]) and by observing that $magic^{\#}(P; G) = magic(P^{\#}; G^{\#})$.

Before proving Theorem 4.1 we note that the fact that unfolding preserves answer substitutions is by now well-known (see [28]) and hence our proof focuses on the preservation of call patterns.

Theorem A.9 *Let P be a program, $c_0 = \ h \leftarrow \ pre :: p(\bar{t}) :: post \ $ a clause in P containing a distinguished occurrence $p(\bar{t})$ of a call to p/n. Assume that $c_i = \ p(\bar{x}) \leftarrow \ \bar{b}_i \ (i = 1..m)$ are (renamings of) the clauses defining the predicate p/n in P and let $c'_i = \ h \leftarrow \ pre :: \bar{b}_i\theta :: post$ where $\theta = mgu(p(\bar{x}), p(\bar{t}))$. We denote by $P' = (P \setminus \{c\}) \cup \{c'_1, \ldots, c'_m\}$ the program in which the distinguished occurrence of $p(\bar{t})$ in c_0 has been unfolded. Assume without loss of generality that c_i and c_0 are renamed apart $(i = 1..m)$. Then, for any goal G the programs P and P' give rise to the same call patterns (excluding the calls to the predicate p/n itself).*

By call patterns we refer to the set of atoms that are chosen in the derivations of the initial goal assuming a left to right selection rule. The statement of Theorem A.9 contains several subtle points worth mentioning: First, that P' is obtained by unfolding a single occurrence of $p(\bar{t})$. Second, that because c_i is normalized the unfolding affects only $\bar{b}_i$ in the body of c'_i $(i = 1..m)$. Finally, as a left to right selection rule is assumed the pairs of resolution steps involving the clause c_0 (introducing an occurrence of $p(\bar{t})$) and a corresponding subsequent c_i (reducing that occurrence) can be matched up in a nested fashion. The proof of Theorem A.9 is technical and contains no surprises.

Proof. (sketch)
We show that that there is a derivation $\mathcal{D} = \langle G; \epsilon\rangle \rightarrow^*_P \langle a_1, \ldots, a_k; \chi\rangle$ for G with P if and only there is a derivation $\mathcal{D}' = \langle G; \epsilon\rangle \rightarrow^*_{P'} \langle a'_1, \ldots, a'_{k'}; \chi'\rangle$ such that $a_1\chi$ and $a'_1\chi'$ are renamed variants.
($\Longrightarrow$) Let $\mathcal{D} = \langle G\rangle \rightarrow^*_P \langle a_1, \ldots, a_n\rangle$ be a derivation for G with P. We prove the existence of the required derivation $\mathcal{D}'$ for G in P' with a_1 and a'_1 renamed variants. The proof is by induction on the number of applications ℓ of clause c_0 in $\mathcal{D}$. The induction base $(\ell = 0)$ is trivial because in this case we can take $\mathcal{D}' = \mathcal{D}$. For the induction step we assume the theorem for all derivations which apply less than ℓ times the clause c_0 and distinguish three cases depending on the structure of $\mathcal{D}$.
case 1: $\mathcal{D}$ has the following form:

$$\begin{array}{l}\langle G; \epsilon\rangle \rightarrow \cdots \xrightarrow{c_0} \langle pre :: p(\bar{t}) :: post\ \ldots; \varphi\rangle \rightarrow \cdots \rightarrow \\ \qquad \langle p(\bar{t}) :: post\ \ldots; \varphi\psi\rangle \xrightarrow{c_i} \langle \bar{b}_i :: post\ \ldots; \varphi\psi\xi\rangle\end{array}$$

where the steps denoted $\xrightarrow{c_0}$ and $\xrightarrow{c_i}$ are the first application of the clause c_0 and a matching reduction with a clause defining p/n.

In this case the correctness of unfolding for answer substitutions implies that the goal $(pre)\varphi$ has an answer ψ also in P' and we can construct a derivation $\mathcal{D}'$

$$\langle G; \epsilon\rangle \rightarrow \cdots \xrightarrow{c'_i} \langle pre :: \bar{b}_i\theta :: post\ \ldots; \varphi\rangle \rightarrow \cdots \rightarrow \langle \bar{b}_i\theta :: post\ \ldots; \varphi\psi\rangle$$

The proof follows after showing that $\bar{b}_i\theta\varphi\psi$ and $\bar{b}_i\varphi\psi\xi$ are renamed variants. Observe that this case does not rely on the induction hypothesis.
case 2: $\mathcal{D}$ has the following form:

$$\langle G; \epsilon\rangle \rightarrow \cdots \xrightarrow{c_0} \cdots \xrightarrow{c_i} S \rightarrow \cdots \rightarrow \langle a_1, \ldots, a_k; \chi\rangle$$

where the steps denoted $\xrightarrow{c_0}$ and $\xrightarrow{c_i}$ are the first application of the clause c_0 and a matching reduction with a clause defining p/n. By case 1, the state S can be derived also in P'. Now the inductive hypothesis applies to the derivation from S to $\langle a_1, \ldots, a_k; \chi\rangle$ which contains at least one less application of the clause c_0.
case 3: $\mathcal{D}$ has the following form:

$$\begin{array}{l}\langle G; \epsilon\rangle \rightarrow \cdots \xrightarrow{c_0} \langle pre :: p(\bar{t}) :: post\ \ldots; \varphi\rangle \rightarrow \cdots \rightarrow \\ \qquad\qquad \langle a_1\ \ldots\ a_s :: p(\bar{t}) :: post\ \ldots; \varphi\psi\rangle\end{array}$$

where $\xrightarrow{c_0}$ denotes the first application of the clause c_0 however the matching reduction with a clause defining p/n does not occur in the derivation (because $p(\bar{t})$ remains in the resolvant). In this case we may consider the state $S = \langle pre; \varphi\rangle$ which can be obtained in P' using the clause c'_1. The derivation from S to $\langle a_1, \ldots, a_s; \varphi\psi\rangle$ in P contains one less application of c_0 and hence by the induction hypothesis has a corresponding derivation in P'.

($\Longrightarrow$) This direction is easier as each application of a clause c'_i in $\mathcal{D}'$ can be directly mimicked by the application of c_0 followed by c_i in $\mathcal{D}$.
□

The proof of Theorem 4.1 now follows by observing that the definition of negation is non-recursive and hence repeated application of "single occurrence" unfolding steps can eliminate all calls to "$\not\vdash$". We should also note that "$\not\vdash$" is a meta-predicate and that $call(G)$ occurs in its definition. However this does not affect our proofs which are easily adapted to consider reductions of the form $\langle call(G) :: goal; \varphi\rangle \rightarrow \langle G :: goal; \varphi\rangle$.