Abstract Interpretation of Logic Programs Using Magic Transformations *

Saumya Debray
Department of Computer Science,
University of Arizona-Tucson, AZ 85721, U.S.A.

Raghu Ramakrishnan
Department of Computer Sciences,
University of Wisconsin-Madison, WI 53706, U.S.A.

Abstract

In dataflow analysis of logic programs, information must be propagated according to the control strategy of the language under consideration. However, for languages with top-down control flow, naive top-down dataflow analyses may have problems guaranteeing completeness and/or termination. What is required in the dataflow analysis is a bottom-up fixpoint computation, guided by the (possibly top-down) control strategy of the language. This paper describes the application of the *Magic Templates* algorithm, originally devised as a technique for efficient bottom-up computation of logic programs, to dataflow analysis of logic programs. It turns out that a direct application of the Magic Templates algorithm can result in an undesirable loss in precision, because connections between "calling patterns" and the corresponding "success patterns" may be lost. We show how the original Magic Templates algorithm can be modified to avoid this problem, and prove that the resulting analysis algorithm is at least as precise as any other abstract interpretation that uses the same abstract domain and abstract operations.

*The work of S. Debray was supported in part by NSF grant CCR-8702939. The work of R. Ramakrishnan was supported in part by an IBM Faculty Development Award, a David and Lucile Packard Foundation Fellowship in Science and Engineering and NSF grant IRI-8804319.

1 Introduction

Abstract Interpretation has been proposed as a methodology for static analysis of programs [9]. Several frameworks based on abstract interpretation have been proposed for analysing Horn Clause logic programs [4, 11, 17, 26, 28, 35]. The semantics of a Horn program is typically given as the least fixpoint of a continuous function over an appropriate domain (e.g. the lattice of Herbrand interpretations [38], or the cpo of substitution sequences [12, 16]); these proposals for abstract interpretation are formulated, analogously, in terms of fixpoints of continuous functions over "simplified" domains, called *abstract domains*. The computation of a least fixpoint is naturally modelled as the evaluation of the limit of a Kleene chain, which is most naturally performed bottom-up. However, because traditional approaches to abstract interpretation of logic programs usually proceed in a top-down manner, ensuring completeness (i.e. that all computational paths have been explored) and termination is somewhat awkward, involving techniques such as memoization that are extraneous to the operational behavior of the language under consideration. In simple terms, this means that while the computation of an ordinary Prolog program over an abstract domain can be described simply by replacing "concrete domain" operations by corresponding operations over the abstract domain, this transformed program cannot be evaluated by an ordinary Prolog interpreter and still be guaranteed to terminate.

This paper shows that abstract interpretation of languages with a top-down execution strategy need not itself be top-down. We present a novel approach based on rewriting strategies originally developed for evaluating queries in deductive databases [2, 3, 33]. These evaluation strategies rewrite a given program in such a way that the fixpoint evaluation of the rewritten program is efficient, in that unnecessary facts are not generated. The rewriting essentially modifies the rules in the original program by adding literals that act as "filters", preventing the generation of irrelevant facts. Further, new rules defining the predicates in these literals are added to the program. These predicates in effect compute the set of goals that are invoked a top-down (Prolog-style) evaluation of the original program. When the fixpoint of the rewritten program is evaluated over an abstract domain, the facts represent the calling patterns for the predicate together with the possible success patterns for each such calling pattern. Thus, we obtain an elegant abstract interpretation technique based on a fixpoint evaluation of the transformed program over the abstract domain. This technique results in analyses that are at least as precise as the analogous top-down ones.

In effect, the program transformation phase of our approach captures at compile-time the binding propagation aspects of a top-down control strategy, and allows us to understand this aspect of an evaluation in terms of the least model of a logic program (the rewritten program). Many details of the control strategy – for example, the exact order in which different rules are explored – are thus abstracted away. The bottom-up fixpoint evaluation of the rewritten program also allows a clean separation between two often intertwined issues, namely termination and completeness.

The principal technical contributions of this paper are as follows:

1. The application of bottom-up fixpoint computation techniques to dataflow analysis of top-down languages is described. The resulting analysis is—in our opinion, at least—cleaner and easier to implement than an analysis that uses a top-down control strategy augmented with features such as memoization.

2. The precision of abstract interpretations is characterized. We show that the bottom-up analysis is at least as precise as any corresponding top-down abstract interpretation using the same abstract domain and abstract functions. Further, it terminates at least whenever the top-down

version does.

The rest of the paper is organized as follows. We present notation and basic definitions in Section 2. We describe the notion of binding propagation, formalized as *sideways information passing graphs*, in Section 3. Section 4 contains an overview of the Magic Templates rewriting algorithm, and the bottom-up evaluation approach. Sections 3 and 4, included to keep this paper self-contained, review material from [3, 33] and can be skipped without loss of continuity by the reader who is familiar with that work. We give an overview of abstract interpretation of logic programs, carefully distinguishing the various components, in Section 5. Section 6 brings together concepts introduced in earlier sections and describes how the Magic Templates rewriting followed by bottom-up evaluation can be used for abstract interpretation of logic programs. We introduce a further program transformation in order to make explicit various operations whose choice determines the domain of computation, and to maintain precision in computations over abstract domains. We characterize the precision of our analysis in Section 7, and present some examples to illustrate our approach in Section 8. We conclude with a discussion of related work in Section 9.

2 Preliminaries

The language considered in this paper is essentially that of Horn logic. Such a language has a countably infinite set of variables and countable sets of function and predicate symbols, these sets being mutually disjoint. It is assumed, without loss of generality, that with each function symbol f and each predicate symbol p, is associated a unique natural number n, referred to as the *arity* of the symbol; f and p are then said to be n-ary symbols (written f/n and p/n respectively). A 0-ary function symbol is referred to as a constant. A *term* in a first order language is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms. We shall find it convenient to consider a vector, or tuple, of terms to be a term. Thus, a vector of n terms, $t_1, \ldots, t_n$, is a term, denoted $\langle t_1, \ldots, t_n \rangle$. When the individual elements comprising a tuple of terms are not significant, the tuple is sometimes denoted simply by the use of an overbar, e.g., $\bar{t}$.

A *substitution* is an idempotent mapping from the set of variables of the language under consideration to the set of terms that is the identity mapping at all but finitely many points. The *domain* of a substitution θ, written $dom(\theta)$, is the set of variables x such that $\theta(x) \neq x$. A substitution σ is *more general* than a substitution θ if there is a substitution φ such that $\theta = \varphi \circ \sigma$. Substitutions are denoted by lower case Greek letters $\theta, \sigma, \phi, \ldots$, while sets of substitutions are denoted by upper case Greek letters $\Theta, \Phi, \ldots$. The application of a substitution to a term can be expressed by defining a predicate *app_subst*, such that given a substitution θ and a term t_1, $app\_subst(\theta, t_1, t_2)$ if and only if $\theta(t_1) = t_2$. Two terms t_1 and t_2 are said to be *unifiable* if there is a substitution σ such that $\sigma(t_1) = \sigma(t_2)$; σ is said to be a *unifier* of t_1 and t_2. If two terms have a unifier, they have a most general unifier that is unique upto renaming of variables. Operationally, logic programming languages typically have the notion of unifying two terms in the context of a "current substitution", representing the substitution obtained in the process of solving the given query upto that point in the computation. This can be expressed by defining a predicate *unify*, such that $unify(\theta, t_1, t_2, \sigma)$ is true if and only if $\theta(t_1)$ and $\theta(t_2)$ are unifiable with most general unifier ψ, and $\sigma = \psi \circ \theta$; here, θ represents the "current substitution". The primitive operations *unify* and *app_subst* are fundamental to most logic programming languages.

A *fact*, or *atom*, is of the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. We adopt the convention that an atom is to be constructed by applying an n-ary predicate symbol to a single term $\langle t_1, \ldots, t_n \rangle$. Since each predicate symbol in a program is assumed to have a unique arity, it is hoped that this sloppiness will not cause undue confusion.

A *clause* is the disjunction of a finite number of literals, and is said to be *Horn* if it has at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause.* The positive literal in a definite clause is its *head*, and the remaining literals, if any, constitute its *body.* A predicate definition consists of a set of definite clauses, whose heads all have the same predicate symbol; a goal is a set of negative literals. We consider a logic program to be a pair $\langle P, Q\rangle$ where P is a set of predicate definitions and Q is the *input*, which consists of a query, or goal, and possibly a set of facts for "database relations" appearing in the program. We follow the convention in deductive database literature of separating the set of rules with non-empty bodies (the set P) from the set of facts, or unit clauses, which appear in Q and are called the *database.* P is referred to as the *program*, or the set of rules. The motivation is that the rewriting algorithms to be discussed are applied only to the program, and not to the database. This is important in the database context since the set of facts can be very large. However, the distinction is artificial, and we may choose to consider (a subset of) facts to be rules if we wish. The meaning of a logic program is the conjunction of the meanings of its clauses, with the free variables of each clause implicitly universally quantified.

Following the syntax of Edinburgh Prolog, definite clauses (rules) are written as

$$p :- q_1, \ldots, q_n.$$

read declaratively as "q_1 *and* ... *and* q_n *implies* p". Names of variables begin with upper case letters, while names of non-variable (i.e. function and predicate) symbols begin with lower case letters. In addition, the following notation is used for lists: the empty list is written [ ], and a list with head H and tail L is written $[H|L]$.

We will use *derivation trees* in several proofs:

Definition 2.1 Given a program P and input Q, derivation trees in $\langle P, Q\rangle$ are defined as follows:

- Every fact h in Q is a derivation tree for itself, consisting of a single node with label h.
- Let r be a rule: $h :- b_1, \ldots, b_k$ in P, let $d_i,\ i = 1 \ldots k$ be atoms with derivation trees t_i, and let θ be the mgu of $(b_1, \ldots, b_k)$ and $(d_1, \ldots, d_k)$. Then, the following is a derivation tree for $\theta(h)$: The root is a node labeled $\theta(h)$, and each $t_i,\ i = 1 \ldots n$, is a child of the root. Each arc from the root to a child has the label r.

■

3 An Overview of the Magic Templates Evaluation Strategy

Consider the following program:

1. $sg(\langle X, Y\rangle) :- flat(\langle X, Y\rangle).$
2. $sg(\langle X, Y\rangle) :- up(\langle X, Z1\rangle), sg(\langle Z1, Z2\rangle), down(\langle Z2, Y\rangle).$

$?- sg(\langle john, X\rangle)$

This is the "same-generations" program, well-known in the deductive database community. We have used $\langle\cdots\rangle$ to emphasize that each predicate has a single argument, which is a tuple.

Given the query, the natural way to use the second rule is to solve the predicates in the indicated order, using bindings from each predicate to solve the next predicate; this is what Prolog does. It is desirable to achieve the same binding propagation in a bottom-up evaluation of this program, and this can be achieved by first rewriting the program. We present a generalization of the Generalized Magic Sets rewriting algorithm [2, 3], called the *Magic Templates* algorithm [33], to keep this paper self-contained. The idea is to introduce a set of auxiliary clauses that compute, intuitively, subgoals generated in the top-down execution. The rules in the original program are then modified by attaching additional literals that act as filters and prevent the rule from generating irrelevant tuples. We now present a simplified version of the Magic Templates algorithm, tailored to the case that each rule is always evaluated left-to-right, as in Prolog. The rewriting algorithm can actually be parametrized in terms of a *sideways-information-passing strategy*, or *sip*, that specifies a different (perhaps partial) ordering for body literals. Further, it is possible to choose a different order for different patterns of bound/free arguments (or "adornments") of the head predicate. Our results are orthogonal to these refinements of the basic algorithm, and we have therefore chosen to address only the case of evaluation methods that always proceed left-to-right. It is straightforward to adapt our results to the general case by following the same lines as in [3, 33] to deal with adornments and sips.

Definition 3.1 The Magic Templates Algorithm

1. Create a new unary predicate $magic\_p$ for each p in P.
2. For each rule in P, add the *modified version* of the rule to P^{mg}. If rule r has head, say, $p(\bar{t})$, the modified version is obtained by adding the literal $magic\_p(\bar{t})$ to the body (at the leftmost position).
3. For each rule r in P,and for each literal $q_i(\bar{t}_i)$ in its body, add a *magic rule* to P^{mg}. The head is $magic\_q_i(\bar{t}_i)$. The body consists of all literals to the left of the literal $q_i(\bar{t}_i)$ in the *modified version* of r.
4. Create a *seed* fact $magic\_q(\langle\bar{c}\rangle)$ from the query.

■

Example 3.1 The Magic Templates algorithm rewrites the same-generation program into the following set of rules:

$magic\_sg(\langle john, U\rangle).$ `/* Seed from the query rule */`
$magic\_sg(\langle Z1, Z2\rangle) :-$
  $magic\_sg(\langle X, Y\rangle), up(\langle X, Z1\rangle).$ `/* From rule 2, 2nd body literal */`
$sg(\langle X, Y\rangle) :-$
  $magic\_sg(\langle X, Y\rangle),\ flat(\langle X, Y\rangle).$ `/* Modified rule 1 */`
$sg(\langle X, Y\rangle) :-$
  $magic\_sg(\langle X, Y\rangle),\ up(\langle X, Z1\rangle),\ sg(\langle Z1, Z2\rangle),\ down(\langle Z2, Y\rangle).$
  `/* Modified rule 2 */`

□

4 Abstract Interpretation of Logic Programs

4.1 Basic Ideas

Let the *program points* of a clause for a given sip be the points between the literals in the clause ordered according to the sip. The execution of a logic program with respect to some set of queries S can be summarized by describing, at each program point, the set of substitutions, or variable bindings, that may be encountered when execution reaches that point, over all possible executions of the program starting from queries in S. Such a description of the behavior of a program is referred to as its *collecting semantics*. The domain of the collecting semantics of the language, also referred to as the *concrete domain* D_{conc}, is thus the powerset of the set of substitutions; it forms a complete lattice under set inclusion. In general, such sets of bindings may be arbitrarily large, making the static inference of most interesting program properties undecidable in general. Since static analyses are expected to always terminate, it is necessary to approximate the collecting semantics. This is done by defining an *abstract domain* $\langle D_{abs}, \sqsubseteq \rangle$ whose structure reflects that of the concrete domain $\langle D_{conc}, \subseteq \rangle$. The relationship between the two domains is given by two functions $\alpha: D_{conc} \rightarrow D_{abs}$ and $\gamma: D_{abs} \rightarrow D_{conc}$, known as the *abstraction* and *concretization* functions respectively. To ensure that D_{abs} and D_{conc} have similar structure, the abstraction and concretization functions are required to be monotone and satisfy the following adjointness requirement:

$$x \subseteq \gamma(\alpha(x)) \text{ for all } x \in D_{conc}; \text{ and } \alpha(\gamma(x)) = x \text{ for all } x \in D_{abs}.$$

In the context of logic programs, the concrete domain D_{conc} is the powerset of the set of substitutions. Equivalently, the concrete domain may be taken to consist of sets of tuples of terms. For each clause, consider the tuple $\bar{V}$ of the variables occurring in that clause, and a tuple of terms $\bar{V}' = \theta(\bar{V})$, where θ is a substitution: in one direction, $\bar{V}'$ can be obtained as $\theta(\bar{V})$ given the substitution θ, and in the other direction the substitution θ can be obtained, given the tuple $\bar{V}'$, as the most general unifier of $\bar{V}$ and $\bar{V}'$. Similarly, a set of substitutions at a point within a clause can be represented equivalently by a set of tuples of terms. As mentioned earlier, tuples of terms are themselves taken to be terms, which means that the concrete domain can equivalently be considered to consist of sets of terms, as we do in the discussion that follows. Analogously, we consider abstract domain elements to represent sets of terms.

In addition to this, for each primitive operation f of the language, defined on the concrete domain, there is a corresponding operation *abs_f* defined on the abstract domain that "mimics" the execution of f. Soundness requirements specify that such an operation *abs_f* should always capture everything that can actually happen at runtime when the "concrete operator" f is executed, though in general *abs_f* may be conservative: in other words, for any element x of the concrete domain,

$$f(x) \subseteq \gamma(abs\_f(\alpha(x))). \qquad (*)$$

In the context of logic programming, two primitive operations of interest are *unify* and *app_subst*, discussed in Section 2, "lifted" to the collecting semantics. For example, whereas *app_subst* ordinarily applies a single substitution to a term to yield another term, in the collecting semantics it must deal with sets of substitutions. A set of substitutions, "applied" to a term, yields a set of terms, so it is more natural to have *app_subst* apply a set of substitutions in the collecting semantics to a set of terms, yielding a set of terms. Thus, given a set of substitutions Θ and sets of terms T_1 and T_2,

$$app\_subst(\Theta, T_1, T_2) \quad \Leftrightarrow \quad T_2 = \{t_2 \mid \exists \theta \in \Theta, t_1 \in T_1 : app\_subst(\theta, t_1, t_2)\}.$$

Similarly, whereas *unify* ordinarily performs unification in the context of a single substitution and returns a single substitution, the corresponding operation lifted to the collecting semantics performs unification in the context of a set of substitutions and two sets of terms to yield a set of substitutions. For any set of substitutions Θ and sets of terms T_1 and T_2,

$$\mathit{unify}(\Theta, T_1, T_2, \Phi) \quad \Leftrightarrow \quad \Phi = \{\varphi \mid \exists t_1 \in T_1, t_2 \in T_2, \theta \in \Theta : \mathit{unify}(\theta, t_1, t_2, \varphi)\}.$$

Recall that the set of terms is assumed to contain tuples of terms as well, which means that *abs_unify* is able to describe the effects of unifying two tuples of terms, e.g., in the context of a call to a predicate or the return from one.

In semantic descriptions of logic programming languages that take their operational behavior into account, the meaning of a predicate in a program is typically given as a function from substitutions to sets, or sequences, of substitutions [12, 16]. Equivalently, by applying such substitutions to the arguments appearing in calls, the meaning of a predicate p can be characterized as a mapping F_p from terms to sets, or sequences, of terms. For any given abstract interpretation, this can be abstracted to obtain the meaning of a predicate p as a partial function μ_p that maps descriptions of tuples of terms to sets of descriptions of tuples of terms:

$$\mu_p : D_{abs} \rightarrow \mathcal{P}(D_{abs}).$$

The idea is that descriptions appearing in $dom(\mu_p)$, the domain of μ_p, are "calling patterns", and indicate how the predicate p may be called; and for each such calling pattern $\bar{a}$, the set $\mu_p(\bar{a})$ gives its "success patterns", i.e. describes how a call described by $\bar{a}$ may succeed. Thus, for any given abstract interpretation, μ_p provides a (presumably safe) approximation to the collecting semantics specified by F_p.

The class of abstract interpretations we consider are those that infer calling and success patterns. *This is assumed in the remainder of the paper, and references to "any abstract interpretation" or "all abstract interpretations" are understood to be relative to this class.* Thus, for example, the development of Marriott and Søndergaard [23], which computes descriptions of goals that succeed and those that finitely fail, is not included; one can also imagine other developments that make inferences about sets of goals that do or do not terminate, or – in the context of concurrent languages – that deadlock (e.g. see [7]): these would be beyond the development in this paper. However, the ideas in this paper can be reformulated to deal with such frameworks as well, and we expect that theorems that are very similar to ours can also be proved for them.

The function F_p giving the meaning of a predicate computing over the concrete domain is typically given as the least fixpoint of a continuous functional [12, 16]. This fixpoint is computed by approximating F_p from below, i.e. beginning at $\perp$ and getting better and better approximations to F_p. The process of improving such approximations can be made explicit using a metalanguage operation *merge*. Conceptually, *merge* is a functional that takes a function F'_p, and a pair of tuples of terms $\langle \bar{t}, \bar{t}' \rangle$ indicating that a call $p(\bar{t})$ may succeed with its arguments bound to $\bar{t}'$, and updates F'_p to produce a new function F''_p that additionally expresses the fact that the call $p(\bar{t})$ can also succeed with its arguments bound to $\bar{t}'$. This functional is abstracted in various ways for static analysis. For example, some researchers collect the results obtained from different execution paths in a set [14, 23], while others combine such results into a single piece of information that is essentially a "worst case" approximation to them [5, 13]. Such abstractions can be described by an "abstract merge" operation, denoted by *abs_merge*: in the case where results from different paths are collected in a set, *abs_merge* is set union; in the case where results are combined to yield a worst case approximation, *abs_merge* is the least upper bound operation in the abstract domain. We return to this point in Section 6.

Abstract interpretations of logic programs can thus be characterized in terms of the following parameters:

1. an abstract domain $\langle D_{abs}, \sqsubseteq \rangle$;
2. abstraction and concretization functions α and γ; and
3. functions *abs_unify*, *abs_app_subst* and *abs_merge* that simulate the primitive operations *unify*, *app_subst* and *merge*, respectively, over the abstract domain.

Further, when the technique is used to analyze a program, it provides a summary of a family of executions in which the literals in the body of each rule are solved in a given order (which is necessary for us to have well-defined program points). Thus, there is another parameter to an abstract interpretation of a program: a choice of sips for the rules of the program. In this paper, we will assume that a left-to-right sip is always chosen, for simplicity. It is straightforward to apply our method with any choice of sips.

4.2 Sound Abstract Interpretations

An abstract interpretation is intended to predict, or to summarize, the expected run-time behaviour of the program. In order to be sound, we require that all run-time possibilities are included in the summary. That is, an abstract interpretation must necessarily predict any calling and success pattern that can arise at runtime from the execution of the program on the given input. Recall that we assume that the meaning of a program is given by the set of input-output functions μ_p for the predicates p in the program. Let the sets of calls for a predicate p in a program given by μ_p be

$$calls(\mu_p) \quad = \quad \bigcup \{\gamma(\bar{a}) \mid \bar{a} \in dom(\mu_p)\}$$

and let the set of successes for a predicate p for a call described by $\bar{a}$, be

$$succs(\mu_p(\bar{a})) \quad = \quad \bigcup \{\gamma(\bar{a}') \mid \bar{a}' \in \mu_p(\bar{a})\}.$$

This soundness criterion can be stated as follows: Given a program P and a set of sips for its clauses, and a query Q, let $\mathcal{S}$ be an evaluation strategy that evaluates P according to these sips. Then

1. if there is any computation of P according to $\mathcal{S}$, for the query Q, such that $p(\bar{t})$ is a goal that has to evaluated at some point in the computation, then $p(\bar{t}) \in calls(\mu_p)$; and
2. if the goal $p(\bar{t})$ can succeed with its arguments bound to the tuple $\bar{t}'$, then there is some $\bar{a} \in dom(\mu_p)$ such that $\bar{t} \in \gamma(\bar{a})$ and $\bar{t}' \in succs(\mu_p(\bar{a}))$.

Any abstract interpretation must infer at least the information in *calls* and *succs*, since otherwise goals and facts in an actual computation may not be covered by the analysis, and soundness is lost. The abstract interpretation may in fact infer additional entries, and each such entry reduces the precision of the analysis. (Loss of precision also arises from entries that are too "coarse" in their approximation of the sets of calling and success patterns.)

4.3 The Scope of this Paper

Aspects of an abstract interpretation such as soundness, termination and precision of analysis are implicit in the abstract domain D_{abs} and the various functions on it enumerated above. For example, soundness is implicit in the requirement that *abs_app_subst* and *abs_unify* satisfy condition $(*)$; termination is usually addressed by restrictions on the structure of D_{abs}; and the precision of an analysis depends partly on the abstract domain D_{abs}, which specifies what is expressible in the analysis; partly on *abs_unify*, which specifies how precisely unification is to be modelled; and partly on the operation *abs_merge*, which specifies how much information is lost in merging the results of analysis over different execution paths. Such issues are not addressed explicitly in this paper. These issues are implicit in the choice of the abstract domain and the abstract functions defined on it for abstract interpretation. Intuitively, these choices represent a trade-off in precision versus the cost of static analysis. Our main result will be to show that for any given choice of abstract domain and abstract operations on it, the abstract interpretation can be carried out according to a bottom-up strategy in such a way that, informally:

1. It is at least as precise as any corresponding top-down abstract interpretation using the same abstract domain and abstract functions; and
2. It terminates at least whenever the top-down version does.

5 A Rewriting Approach to Abstract Interpretation

5.1 Explicated Programs: Making Primitive Operations Explicit

Consider the Magic Templates algorithm. For every predicate in the original program, the set of facts computed by the fixpoint evaluation of the rewritten program must also be computed by any evaluation of the original program that proceeds left-to-right (and only does subsumption checks up to variable renaming) [3, 33]. An *optimal* evaluation is one that computes no other facts. Consider any implementation strategy that evaluates a program optimally and that proceeds in a top-down fashion by invoking subgoals. We make the following important observation: *The set of goals, i.e., calls, arising during the execution of a program P is identical to the set of facts computed for the magic predicates in P^{mg}.* Our approach to abstract interpretation of logic programs rests upon this property of P^{mg}.

Abstract interpretation of a program consists essentially of “simulating” its execution over an abstract domain. This is done by specifying, as part of the abstract interpretation, an “abstract operation” *abs_f* for each primitive operation f of the language. To see how this should be done, it is necessary to make the primitive operations of the language – in our case, unification and the collection of results from different computation paths – explicit.

Consider the computation of a rule r: $p(\bar{T}_0)\ :-\ q_1(\bar{T}_1), \ldots, q_n(\bar{T}_n)$. Initially, each variable in the clause is uninstantiated. First, the arguments in the head of the clause are unified with those in the call to yield a substitution θ_0. The first literal in the body is then evaluated in the context of this substitution; if this succeeds yielding a new substitution θ_1, the next literal in the body is evaluated in the context of θ_1, and so on. Finally, when all the literals in the body have been successfully evaluated, yielding a substitution θ_n, the “return value” is obtained by applying θ_n to the tuple of arguments in the head of the clause.

This operational behavior can be made explicit by rewriting the corresponding modified rule in

P^{mg}. Recall that a k-ary predicate symbol is applied to a single term that is a tuple of k terms. We use this convention here also: each $T_i, i = 0, \ldots, n$, is taken to be a single term. In the following, $X_\downarrow$, $X_\uparrow$, $T_{i\downarrow}$ and $T_{i\uparrow}$, $i = 1, \ldots, n$, are distinct new variables, and **id** is the identity substitution. The *explicated version* of rule r is:

$p(X_\downarrow, X_\uparrow)$:−
  $\mathit{magic\_p}(X_\downarrow)$, $\mathit{unify}(\mathbf{id}, X_\downarrow, \bar{T}_0, \theta_0)$,
  $\mathit{app\_subst}(\theta_0, \bar{T}_1, T_{1\downarrow})$, $q_1(T_{1\downarrow}, T_{1\uparrow})$, $\mathit{unify}(\theta_0, T_{1\downarrow}, T_{1\uparrow}, \theta_1)$,
  $\mathit{app\_subst}(\theta_1, \bar{T}_2, T_{2\downarrow})$, $q_2(T_{2\downarrow}, T_{2\uparrow})$, $\mathit{unify}(\theta_1, T_{2\downarrow}, T_{2\uparrow}, \theta_2)$,
  $\ldots$,
  $\mathit{app\_subst}(\theta_{n-1}, \bar{T}_n, T_{n\downarrow})$, $q_n(T_{n\downarrow}, T_{n\uparrow})$, $\mathit{unify}(\theta_{n-1}, T_{n\downarrow}, T_{n\uparrow}, \theta_n)$,
  $\mathit{app\_subst}(\theta_n, \bar{T}_0, X_\uparrow)$.

Each k-ary (non-magic) predicate — which can be thought of a predicate that takes one argument that is a k-tuple of terms — has been modified to have two arguments: the first, subscripted '$\downarrow$', representing the tuple of arguments at the call to the predicate, and the second, subscripted '$\uparrow$', representing the tuple of arguments at the return from that call.

It is important that we maintain separate "calling" and "return" arguments. One reason for doing this is to make explicit the operational aspects of a logic program computation (since this is what an abstract interpretation tries to mimic). We note that it also has declarative virtues, since the application of a substitution to a term is essentially a destructive operation, and it is better to make such operations explicit. The most important reason behind this anticipates a technical difficulty in abstract interpretation — certain kinds of static analyses require that the connection between "calling" and "return" values be maintained explicitly during analysis in order that the analysis be precise. (This point is illustrated in Example 7.1.)

Given that we maintain separate sets of calling and return arguments, it is possible to eliminate the magic predicates. Consider rule r. Every tuple in the magic predicate *magic_p* corresponds to a call, and an examination of the explicated version of r reveals that this tuple is also contained in the first argument position of every tuple in p that is computed as an answer to this call. This suggests that we can simply use the first argument position of the predicate p to play the role of the magic predicate *magic_p*, by adopting the convention that $\perp$ in the second argument indicates the generation of a goal, for which an answer is not as yet known. The explicated version of the rule is then:

$p(\bar{X}_\downarrow, \bar{X}_\uparrow)$:-
  $p(\bar{X}_\downarrow, \_)$, $\mathit{unify}(\mathbf{id}, \bar{X}_\downarrow, \bar{T}_0, \theta_0)$,
  $\mathit{app\_subst}(\theta_0, \bar{T}_1, \bar{T}_{1\downarrow})$, $q_1(\bar{T}_{1\downarrow}, \bar{T}_{1\uparrow})$, $\mathit{unify}(\theta_0, \bar{T}_{1\downarrow}, \bar{T}_{1\uparrow}, \theta_1)$,
  $\mathit{app\_subst}(\theta_1, \bar{T}_2, \bar{T}_{2\downarrow})$, $q_2(\bar{T}_{2\downarrow}, \bar{T}_{2\uparrow})$, $\mathit{unify}(\theta_1, \bar{T}_{2\downarrow}, \bar{T}_{2\uparrow}, \theta_2)$,
  $\ldots$,
  $\mathit{app\_subst}(\theta_{n-1}, \bar{T}_n, \bar{T}_{n\downarrow})$, $q_n(\bar{T}_{n\downarrow}, \bar{T}_{n\uparrow})$, $\mathit{unify}(\theta_{n-1}, \bar{T}_{n\downarrow}, \bar{T}_{n\uparrow}, \theta_n)$,
  $\mathit{app\_subst}(\theta_n, \bar{T}_0, \bar{X}_\uparrow)$.

The corresponding abstract interpretation computation can now be described simply by replacing each primitive operation by the corresponding operation over the abstract domain. The following rule is called the *abstract explicated version* of rule r:

$\mathit{abs\_p}(\bar{X}_\downarrow, \bar{X}_\uparrow)$:-

P The original source program.

P^{mg} The *magic* program: Each clause in P has an additional "magic literal" added to it. This magic literal acts as a filter during bottom-up evaluation, and prunes away provably useless computation branches.

P^{ex} The *explicated magic* program: The primitive operations of the language, unification and substitution application, are made explicit, and separate argument tuples for "calling" and "return" values are added. Additional clauses and literals are added to act as filters during bottom-up evaluation and prune away provably useless computation branches. The operation *merge* is to be used in evaluating the fixpoint.

P^{abs} The *abstract explicated magic* program: The concrete domain operations *unify* and *app_subst* are replaced by corresponding operations *abs_unify* and *abs_app_subst*. The operation *abs_merge* is to be used in evaluating the fixpoint.

Figure 1: Different kinds of rewritten programs

$$
\begin{aligned}
&abs\_p(X_{\downarrow}, \_),\ abs\_unify(\alpha(\{\mathbf{id}\}), \bar{X}_{\downarrow}, \bar{T}_0, A_0),\\
&abs\_app\_subst(A_0, \bar{T}_1, \bar{T}_{1\downarrow}),\ abs\_q_1(\bar{T}_{1\downarrow}, \bar{T}_{1\uparrow}),\ abs\_unify(A_0, \bar{T}_{1\downarrow}, \bar{T}_{1\uparrow}, A_1),\\
&abs\_app\_subst(A_1, \bar{T}_2, \bar{T}_{2\downarrow}),\ abs\_q_2(\bar{T}_{2\downarrow}, \bar{T}_{2\uparrow}),\ abs\_unify(A_1, \bar{T}_{2\downarrow}, \bar{T}_{2\uparrow}, A_2),\\
&\ldots,\\
&abs\_app\_subst(A_{n-1}, \bar{T}_n, \bar{T}_{n\downarrow}),\ abs\_q_n(\bar{T}_{n\downarrow}, \bar{T}_{n\uparrow}),\ abs\_unify(A_{n-1}, \bar{T}_{n\downarrow}, \bar{T}_{n\uparrow}, A_n),\\
&abs\_app\_subst(A_n, \bar{T}_0, \bar{X}_{\uparrow}).
\end{aligned}
$$

where $\alpha(\{\mathbf{id}\})$ represents the abstract domain element corresponding to (the singleton set containing) the identity substitution. The A_i are descriptions, over the abstract domain, of sets of substitutions.

We now describe how to construct the *explicated magic program* P^{ex}, and the *abstract explicated magic program* P^{abs}. [1]

Definition 5.1 The *explicated magic program* P^{ex} is obtained from the given program P by applying the Magic Templates algorithm with the following modifications: In Step 2, construct the explicated version of the rule. In Step 3, the head of the "magic" rule is $q_i(\bar{t}, \perp)$, and the body is constructed using the explicated version of the rule. In Step 4, the seed is $q(\langle\bar{c}\rangle, \perp)$. ■

Definition 5.2 The *abstract explicated magic program* P^{abs} is obtained from the explicated program P^{ex} by replacing every predicate, say p, by its abstract version, *abs_p*; replacing every operation by the corresponding operation over the abstract domain; and replacing every set of concrete substitutions by its abstract description. ■

At this point, we have considered a number of different rewritings of programs. The different kinds of rewritten programs are summarized in Figure 1.

[1]This algorithm is presented as a simple modification to the Magic Templates algorithm presented earlier for the case of left-to-right sips. For general sips, we must use the Magic algorithm presented in [3] and make essentially the same modification.

Example 5.1 Continuing our running example, the explicated magic program P^{ex} is:

$sg(\langle john, U\rangle, \bot).$ `/* Seed from the query rule */`
$sg(V_{2\downarrow}, \bot)$:−
    $sg(U_{\downarrow}, \_),\ unify(\mathbf{id}, U_{\downarrow}, \langle X, Y\rangle, \theta_0),$
    $app\_subst(\theta_0, \langle X, Z1\rangle, V_{1\downarrow}),\ up(V_{1\downarrow}, V_{1\uparrow}),\ unify(\theta_0, V_{1\downarrow}, V_{1\uparrow}, \theta_1),$
    $app\_subst(\theta_1, \langle Z1, Z2\rangle, V_{2\downarrow}).$ `/* From rule 2, 2nd body literal */`
$sg(U_{\downarrow}, U_{\uparrow})$:−
    $sg(U_{\downarrow}, \_),\ unify(\mathbf{id}, U_{\downarrow}, \langle X, Y\rangle, \theta_0),$
    $app\_subst(\theta_0, \langle X, Y\rangle, V_{1\downarrow}),\ flat(V_{1\downarrow}, V_{1\uparrow}),\ unify(\theta_0, V_{1\downarrow}, V_{1\uparrow}, \theta_1),$
    $app\_subst(\theta_1, \langle X, Y\rangle, U_{\uparrow}).$ `/* Explicated rule 1 */`
$sg(U_{\downarrow}, U_{\uparrow})$:−
    $sg(U_{\downarrow}, \_),\ unify(\mathbf{id}, U_{\downarrow}, \langle X, Y\rangle, \theta_0),$
    $app\_subst(\theta_0, \langle X, Z1\rangle, V_{1\downarrow}),\ up(V_{1\downarrow}, V_{1\uparrow}),\ unify(\theta_0, V_{1\downarrow}, V_{1\uparrow}, \theta_1),$
    $app\_subst(\theta_1, \langle Z1, Z2\rangle, V_{2\downarrow}),\ sg(V_{2\downarrow}, V_{2\uparrow}),\ unify(\theta_1, V_{2\downarrow}, V_{2\uparrow}, \theta_2),$
    $app\_subst(\theta_2, \langle Z2, Y\rangle, V_{2\downarrow}),\ down(V_{3\downarrow}, V_{3\uparrow}),\ unify(\theta_2, V_{3\downarrow}, V_{3\uparrow}, \theta_3),$
    $app\_subst(\theta_3, \langle X, Y\rangle, U_{\uparrow}).$ `/* Explicated rule 2 */`

□

We now formalize the connection between the execution of a program and the explicated magic program. This result is a straightforward extension of results in [3, 33], taking into account the differences between P^{mg} and P^{ex}.

Theorem 5.1 *Consider a program P, an input Q containing a query $q \equiv g^e(\bar{t})\_?$, and a sip for each rule of P, for each head adornment. Let P^{ad} be the corresponding adorned program and P^{ex} the explicated program. Let $\mathcal{S}$ be an evaluation strategy that evaluates $\langle P, Q\rangle$, proceeding left-to-right in each rule. Then:*

1. *The fact $p^a(\bar{c}, \bot)$ is in the least fixpoint of $\langle P^{ex}, Q\rangle$ if and only if $p^a(\bar{c})\_?$ is a goal generated by $\mathcal{S}$.*
2. *Let p^a be a predicate in P^{ad}. The fact $p^a(\bar{c}, \bar{s})$, $s \neq \bot$, is in the least fixpoint of $\langle P^{ex}, Q\rangle$ if and only if $p^a(\bar{c})\_?$ is a goal generated by $\mathcal{S}$ and $p^a(\bar{s})$ is a solution to this goal.*

Proof *[Only if:]* The proof is by induction on the height of derivation trees in $\langle P^{ex}, Q\rangle$. The only tree of height 0 is the tree consisting of the single node $g(\bar{t}, \bot)$, corresponding to the query q, and is the basis of our induction. Let the claim hold for all facts that have a derivation tree of height less than N. Consider a fact $p(\bar{c}, \_)$ that is the root of a tree of height N.

Let the children, from left to right, have labels $l_0, l_1, \ldots, l_k$, ignoring *unify* and *apply_subst* literals. (There is exactly one *unify* and one *apply_subst* literal between the node with label l_i and the node with label l_{i+1}.) By construction of P^{ex}, there is a rule, say r: 'h :− $b_1, \ldots, b_n$', $n = k$, in P and a substitution θ such that the following holds. Let $h = p(\bar{t}_0)$ and $b_i = q_i(\bar{t}_i), i = 1 \ldots k$. Then, $l_0 = \theta(p(\bar{t}_0, \_))$, and $l_i = q_i(\bar{c}_i, \bar{s}_i), i = 1 \ldots k$. By the induction hypothesis, l_0 is a goal generated by $\mathcal{S}$, and each $l_i, i = 1 \ldots k$ is a goal-solution pair. By definition of the predicates *unify* and *apply_subst*, it follows that $\mathcal{S}$ must also generate the goal $p(\bar{c})\_?$ if it proceeds according to the sip.

[If:] If $\mathcal{S}$ generates a goal G, there is a chain of goals and solutions such that the following holds. The first element is the given query q, the last element is goal G, and there is a rule r in P such that:

- This rule is invoked by a goal that is a predecessor of G in the chain, and there is a set of predecessor goal-solution pairs that unify with the first k-1 body literals of r. The resulting mgu, applied to the k'th body literal, generates goal G.

If $\mathcal{S}$ generates an answer A to a goal G, there is a chain of goals and solutions such that the following holds. The first element is the given query q, the last element is A, and there is a rule r in P such that:

- This rule is invoked by goal G, which is a predecessor of A in the chain, and there is a set of predecessor goal-solution pairs that unify with the body literals of r. The resulting mgu, applied to the head literal, generates solution A.

We prove that if a goal $p(\bar{c})\_?$ is generated by $\mathcal{S}$, a fact $p(\bar{c},\_)$ is computed in P^{ex}, and that if a solution $p(\bar{s})$ is generated by $\mathcal{S}$, a fact $p(\bar{c},\bar{s})$ is computed in P^{ex}. The proof is by induction on the length of the chain associated with the goal or solution. The basis is the chain of length 1, which is the query q; the fact $p(\bar{c},\perp)$ is the seed in P^{ex}. Let the claim hold for goals and solutions that are generated by chains of length less than N. Consider a solution $p(\bar{s})$ (to a goal $p(\bar{c})\_?$) generated by a chain of length N, say from a rule r in P. Since, by the induction hypothesis, there are facts in P^{ex} corresponding to the goal that invoked rule r (this is a fact $p(\bar{c},\_)$), the goals obtained by instantiating the body literals, and the solutions to these goals, we can instantiate the explicated version of rule r to obtain the solution $p(\bar{c},\bar{s})$. Consider a goal that is generated by a chain of length N, say from the k'th body literal of a rule r in P. The claim is similarly established by considering the magic rule that is generated from this body literal. $\Box$

The careful reader will have noticed that we assume that a set of all generated facts (modulo variable renamings) is maintained. This corresponds to one particular choice for the *merge* operation. The fixpoint computation can be refined by maintaining instead the *irredundant* version of this set [21]. That is, we may discard a generated fact if it is "subsumed" by an existing fact.[2] This may enable us to avoid some derivations of goals and facts, essentially because we know that more general goals and facts are also derived. However, in the worst case — which is that for every version of a generated fact, all generated versions that are more general are generated later — none of the derivations discussed in the above proof can be avoided. The order of derivations is, of course, dependent on the order in which rules (and facts) are considered, and is non-deterministic, as the following example illustrates.

Example 5.2 Consider the following program:

q1 :− p(a).
q1 :− p(X).
p(X) :− r(X).
?- q1.

[2] In an abstract interpretation, a fact A "subsumes" another fact B if the set of concrete facts represented by A contains every concrete fact represented by B, i.e., if $\gamma(B) \subseteq \gamma(A)$. This need not necessarily coincide with the "usual" notion of subsumption of first order terms.

Assuming that the merge operation performs subsumption checks, is the goal $r(a)$ generated? This depends upon the order in which the rules defining $q1$ are considered, since if $p(a)\_?$ is generated after $p(X)\_?$, it is simply discarded. □

The following example, given by Codish *et al.* [6] (who credit it to J. Gallagher), is a simple variant of the previous program. It brings out a subtle problem associated with the Magic Templates algorithm, vis-a-vis a Prolog-style top-down evaluation method.

Example 5.3 Consider the program:

q1 :− p(a), p(X), r(X).
p(X).
Query: ?- q1.

Again assuming that the merge operation performs subsumption checks, is the goal $r(a)$ generated? Prolog will not generate this goal, but a bottom-up fixpoint evaluation of the program generated by the Magic Templates rewriting will generate it. To understand why, we note that both the goals $p(a)$ and $p(X)$ are generated (in Prolog as well as the bottom-up evaluation). These have, respectively, the answers $p(a)$ and $p(X)$. The control strategy of the Prolog evaluation, which waits for the answer to a goal before proceeding, ensures that only the answer $p(X)$ is used for the goal $p(X)$. The Magic Templates approach, on the other hand, will use any generated fact that unifies with a goal, and thus will use $p(a)$ as an answer to the goal $p(X)$ generated by the second p literal. Therefore, the goal $r(a)$ is generated (in addition, of course, to $r(X)$). We note that the explicated version of the Magic Templates rewritten program does not suffer from this problem; it behaves like Prolog on this example. □

Finally, we observe that if there is *any* evaluation method that generates all necessary goals and solutions and halts (independently of the order of derivations), then so will the bottom-up evaluation of P^{ex}. Indeed, we must eventually generate all the goals and facts, given the completeness of bottom-up evaluation, and we must then stop since the fixpoint has been reached. A similar observation holds for the bottom-up evaluation of P^{abs}, discussed in the next section.

5.2 Evaluation of Abstract Explicated Programs

Our proposal is quite simple. Given a program P, a query, a choice of an abstract domain, abstract operations over it, and sips, to do abstract interpretation:

1. Construct the abstract explicated program P^{abs}.
2. Compute the fixpoint of P^{abs} over the abstracted set of facts bottom-up, using the given set of abstract operations.

Definition 5.3 Canonical Computation We refer to the computation of the fixpoint of P^{abs} as a *canonical computation.* ■

Note that P^{abs} is evaluated over the abstracted set of facts, i.e., with each fact h replaced by $\alpha(h)$. There are at least two options available when a new "abstract tuple" is inferred in the fixpoint computation of P^{abs} — it can either be added to the set of known facts, or the LUB can be taken of this tuple and (some summary of) the set of known facts. The latter alternative, which essentially computes a worst-case summary of the available information, is less precise (because information is lost in taking the LUB) but more efficient (because fewer tuples have to be stored). To articulate this, we introduced an *abstract merge* operation *abs_merge* in an earlier section. This is a parameter of the bottom-up evaluation strategy that will usually be implicit in the discussion that follows. Note that it does not appear in the explicated program; rather, it is part of the fixpoint evaluation phase.

It is important to understand the role of the operator *abs_merge*: The "set of known facts" is always represented in some form, either by explicitly listing all members, or by some summary that is a "safe" approximation in that the set of facts represented by the summary includes the set of facts being summarized. The choice of *abs_merge* reflects this representation, and *abs_merge* can be understood as a LUB operator over the representation domain. When sets are represented by listing all elements, the LUB is a set union. If the set contains non-ground tuples, we may choose an *irrset* representation (for irredundant set, in which no element is subsumed by another; see [21]). If so, the LUB must include subsumption checks. If the set is represented by an element of the abstract domain that is the LUB of the set of known (abstract) facts during an abstract interpretation, then *abs_merge* is simply the LUB operator over the abstract domain.

In order for our analysis to be sound, the following must hold:

1. for every goal $p(\bar{t})$ generated at runtime in the computation of P on input Q, there is a tuple $abs\_p(\bar{a}, \bar{b})$ computed in P^{abs} such that $\bar{t} \in \gamma(\bar{a})$; and
2. for every goal $p(\bar{t})$ so generated that can succeed with its arguments bound to $\bar{t}'$, there is a tuple $abs\_p(\bar{a}, \bar{a}')$ computed in P^{abs} such that $\bar{t}' \in \gamma(\bar{a}')$.

The following lemma establishes an important connection between P^{ex} and P^{abs}, and is used to show the soundness of our analysis.

Lemma 5.2 *Consider a program P, input Q, and a sip for each rule of P for each head adornment. Let P^{abs} be the corresponding abstract explicated program, using abstract domain D_{abs}, abstraction and concretization functions α and γ, and abstract operations abs_unify, abs_app_subst, and abs_merge.*

Let g be a fact in the least fixpoint of $\langle P^{ex}, Q\rangle$. Then, there is a fact h in the least fixpoint of $\langle P^{abs}, \alpha(Q)\rangle$ such that $g \in \gamma(h)$.

Proof (Sketch) There is a straightforward mapping of the derivation tree for g in $\langle P^{ex}, Q\rangle$ into a derivation tree for h in $\langle P^{abs}, \alpha(Q)\rangle$ based on the correspondence between the explicated and the abstract explicated versions of a rule in P. (The proof also utilizes Condition (*) on operators over the abstract domain.) □

The following theorem shows that the results of the abstract interpretation that we propose is sound, in that the goals and facts generated in a computation over the concrete domain are contained in the set of goals and facts represented by the result of the abstract interpretation.

Theorem 5.3 Soundness
A Canonical Computation is a sound abstract interpretation.

Proof Follows immediately from Theorem 6.1 and Lemma 6.2. □

Note that this result does not guarantee termination of the evaluation of the least fixpoint of P^{abs} (i.e. the termination of the abstract interpretation). For this, we must rely upon other properties, such as finiteness of the abstract domain. In general, results on the safety and termination of fixpoint evaluation for logic programs are applicable here, e.g., [1, 19, 20]. As in the bottom-up evaluation of P^{ex}, the existence of a terminating abstract interpretation strategy that generates all the abstract "goals" and "facts" required by the sip definition and terminates assures that bottom-up evaluation of P^{abs} will also terminate. (Due to space constraints, we do not prove this formally, but the development is straightforward.) Thus, while termination issues are not addressed in this paper, the proposed method is at least no worse than any other abstract interpretation technique that uses the same abstract domain and operations and mimics the same choice of sips.

6 On the Precision of Canonical Computations

We have shown that our analysis is sound in that anything that can happen at runtime is inferred during analysis. It is desirable to also be able to go in the other direction, and reason about how tightly the results of the analysis bound the runtime possibilities. In this section, we show that the results of our analysis are at least as precise as those of any abstract interpretation that uses the same abstract domain and operations—in other words, that no imprecision is introduced due to rewriting and subsequent bottom-up fixpoint evaluation.

Theorem 6.1 Relative Precision
Consider a program P and input Q. Let P^{abs} be the corresponding abstract explicated program, using abstract domain D_{abs}, abstraction and concretization functions α and γ, and abstract operations abs_unify, abs_app_subst, and abs_merge. For any abstract interpretation that uses the same abstract domain and operations, and approximates the evaluation of $\langle P, Q\rangle$ proceeding left-to-right within each rule, the following hold for every fact $abs\_p(\bar{a}, \bar{b})$ in the least fixpoint of $\langle P^{abs}, \alpha(Q)\rangle$:

1. $\gamma(\bar{a}) \subseteq calls(\mu_p)$; *and*
2. *let* $S = \{\bar{a}' \mid \langle \bar{a}, \bar{a}'\rangle \in abs\_p\}$, *then* $\gamma(S) \subseteq succs(\mu_p(\bar{a}))$.

Proof By induction on the heights of the derivation trees for the facts.

Consider a tuple $\langle \bar{a}, \bar{b}\rangle$ in the relation *abs_p* that has a derivation tree of height 0. This means that it must be the "seed fact" describing the query $p(\bar{t})$. Then, $\bar{a} = \alpha(\{\bar{t}\})$. Assuming that the abstract interpretation is sound, it follows that $\bar{a}$ must be in $dom(\mu_p)$, whence $\gamma(\bar{a}) \subseteq calls(\mu_p)$.

Consider a fact $abs\_p(\bar{a}, \bar{a}')$ whose derivation tree is of height 1. In this case, the original program P has a fact $p(\bar{u})$, corresponding to which there is a clause in the abstract explicated program P^{abs} of the form

$$abs\_p(\bar{X}_{\downarrow}, \bar{X}_{\uparrow}) \ :- \ abs\_p(\bar{X}_{\downarrow}, \_), abs\_unify(\alpha(\{\mathbf{id}\}), \bar{X}_{\downarrow}, \bar{u}, A_0), abs\_app\_subst(A_0, \bar{u}, \bar{X}_{\uparrow})$$

such that $\bar{a} = A(\bar{X}_{\downarrow})$, and $\bar{a}' = A(\bar{X}_{\uparrow})$, for some abstract substitution A such that $abs\_unify(\alpha(\{\mathbf{id}\}), \bar{X}_{\downarrow}, \bar{u}, A)$. For this to be true, there must be a fact $abs\_p(A(\bar{X}_{\downarrow}), A(\_)) \equiv abs\_p(\bar{a}, \bar{b})$, for some $\bar{b}$, in the least fixpoint of $\langle P^{abs}, \alpha(Q)\rangle$, whose derivation tree is of height 0; this, in turn, implies that the input contains a query $p(\bar{t})$ such that $\bar{a} = \alpha(\bar{t})$, and it follows from the above that $\gamma(\bar{a}) \subseteq calls(\mu_p)$. Since the abstract interpretation under consideration is also using the abstract operations *abs_unify* and *abs_app_subst*, it must also infer that a call described by $\bar{a}$ can have success pattern $\bar{a}'$. It follows that $\bar{a}' \in \mu_p(\bar{a})$, which implies that $\gamma(\bar{a}') \subseteq succs(\mu_p(\bar{a}))$.

Assume that the theorem holds for all facts that have derivation trees of height less than N. Consider a fact $abs\_p(\bar{a}, \bar{a}')$ whose smallest derivation tree is of height N: this must have been derived from a clause in the explicated abstract program of the form

$$
\begin{array}{l}
abs\_q_0(\bar{V}_\downarrow, \bar{V}_\uparrow) \;:- \\
\quad abs\_q_0(\bar{V}_\downarrow, \_),\ abs\_unify(\alpha(\{\mathbf{id}\}), \bar{V}_\downarrow, \bar{T}_0, A_0), \\
\quad abs\_app\_subst(A_0, \bar{T}_1, \bar{T}_{1\downarrow}),\ abs\_q_1^{a_1}(\bar{T}_{1\downarrow}, \bar{T}_{1\uparrow}),\ abs\_unify(A_0, \bar{T}_{1\downarrow}, \bar{T}_{1\uparrow}, A_1), \\
\quad \ldots, \\
\quad abs\_app\_subst(A_{n-1}, \bar{T}_n, \bar{T}_{n\downarrow}),\ abs\_q_n^{a_n}(\bar{T}_{n\downarrow}, \bar{T}_{n\uparrow}),\ abs\_unify(A_{n-1}, \bar{T}_{n\downarrow}, \bar{T}_{n\uparrow}, A_n), \\
\quad abs\_app\_subst(A_n, \bar{T}_p, \bar{U}_\downarrow), \\
\quad abs\_p(\bar{U}_\downarrow, \bar{U}_\uparrow), \\
\quad \ldots
\end{array}
$$

where $A_0, \ldots, A_n$ are abstract substitutions such that $\bar{a} = A_n(\bar{T}_p)$. In turn, this must have been derived from a clause in the original program of the form

$$q_0(\bar{T}_0) \;:-\; q_1(\bar{T}_1), \ldots, q_n(\bar{T}_n), p(\bar{T}_p), \ldots$$

Further, it must be the case that

1. $abs\_q_0(\bar{b}_0)$ is in the least fixpoint of $\langle P^{abs}, \alpha(Q)\rangle$, where $\bar{b}_0 = A(\bar{V}_\downarrow)$ for some abstract substitution A, such that $abs\_unify(\alpha(\{\mathbf{id}\}), \bar{b}_0, \bar{T}_0, A_0)$ holds; and
2. $abs\_q_i(A_{i-1}(\bar{T}_i), A_i(\bar{T}_i))$, $1 \le i \le n$, are in the least fixpoint of $\langle P^{abs}, \alpha(Q)\rangle$

and each of these facts has a derivation tree whose height is less than N. We show, by induction on i, that the abstract substitution A_i safely describes the set of substitutions that may be obtained at the program point immediately after the literal $q_i(\bar{T}_i)$, $0 \le i \le n$, in any computation of this clause starting with a call to q_0 described by $\bar{b}$. Since the fact $abs\_q_0(\bar{b})$ has a derivation tree of height less than N, it follows from the induction hypothesis of the theorem that $\gamma(\bar{b}_0) \subseteq calls(\mu_{q_0})$, whence from the soundness of *abs_unify* it follows that A_0 safely describes the substitutions that may be obtained after head unification for any call to q_0 described by $\bar{b}_0$. Assume that a call described by $\bar{T}_{1\downarrow}$ can return with success pattern $\bar{T}_{1\uparrow}$. From the induction hypothesis of the theorem, $\gamma(\bar{T}_{1\uparrow}) \subseteq succs(\mu_{q_1}(\bar{T}_{1\downarrow}))$. Let $abs\_unify(A_0, \bar{T}_{1\downarrow}, \bar{T}_{1\uparrow}, A_1)$ hold, then it follows, from the soundness of *abs_unify*, that A_1 safely describes the set of substitutions that may be obtained at the point immediately after the literal q_1. Suppose the argument holds for all values of i less than k, and consider the literal $q_k(\bar{T}_k)$: since A_{k-1} safely describes the set of substitutions that may be obtained at the program point immediately before this literal, it follows from the soundness of *abs_app_subst* that $\bar{T}_{k\downarrow}$ safely describes all calls that can arise for this literal in this computation.

From the induction hypothesis of the theorem, $\bar{T}_{k\uparrow}$ safely describes any success pattern that can be obtained for such a call. From the soundness of *abs_unify*, it follows that A_k safely describes the set of substitutions that can be obtained immediately after this literal. This establishes that the abstract substitution A_i safely describes the set of substitutions that may be obtained at the program point immediately after the literal $q_i(\bar{T}_i)$, $0 \le i \le n$. It follows that $A_n(\bar{T}_p)$ safely describes all the calls to the literal $p(\bar{T}_p)$ that can arise here. This implies that $\gamma(A_n(\bar{U})) \subseteq calls(\mu_p)$. But $A_n(\bar{U}) = \bar{a}$, so this implies that $\gamma(\bar{a}) \subseteq calls(\mu_p)$. $\Box$

What Theorem 6.1 shows is that a canonical computation is at least as precise as any other abstract interpretation that uses the same abstract domain and abstract operations. To understand the significance of this, it is useful to compare it to Theorem 5.1. Theorem 5.1 extends the sip-optimality results of [3, 33] to the explicated programs considered in this paper; the extension is based on a straightforward correspondence between the first argument of an explicated program predicate and the argument (vector) of a "magic" predicate, and between the second argument of an explicated program predicate and the argument of a (user) program predicate. The result is with respect to computations over the concrete domain, and essentially the same result holds for the magic program P^{mg}. In contrast, Theorem 6.1 is a result about computations over the abstract domain. Note that the analogous result does *not* hold for the magic program P^{mg}; the explication of the program (in particular, the introduction of "input" and "output" copies of the arguments of a predicate) is crucial. Theorem 6.1 does not address the question of how much precision can be attained for a given abstract domain. Clearly, the converse of Lemma 6.2, viz. "anything that is inferred during analysis will happen at runtime", may not hold. Even a more conservative statement, of the form "for any given input Q to the program, for any calling or success pattern inferred at analysis time, there is some input described by $\alpha(Q)$ that causes that calling or success pattern to be realized at runtime", may not hold, because we assume that the abstract operations *abs_app_subst*, *abs_unify* and *abs_merge* are given to us by the designer of the abstract interpretation, and it may happen that these operations are very imprecise. The point, however, is that the rewriting and subsequent bottom-up computation does not contribute, in any way, to loss of precision: for any given abstract domain, any loss of precision during analysis is due only to various parameters of the abstract interpretation, such as the abstract operations *abs_app_subst*, *abs_unify* and *abs_merge*.

The results of this section show that the canonical computations approach is as precise as any top-down abstract interpreter with respect to sets of $\langle call, success\rangle$ pairs. However, suppose that a predicate appears in two different clauses; then we cannot tell which pairs correspond to which occurrence. Thus, we could potentially lose information at the program point level.[3] Fortunately, it is easy to modify the transformation to retain information at the level of predicate occurrences rather than predicates. We can distinguish between different occurrences of the same predicates by introducing variants of predicates so that every predicate occurs in exactly one clause-body position in the entire program. (Equivalently, we could use an extra argument position to distinguish between these variants.) We omit the details of the modified transformation as they are straightforward; it is sufficient to observe that since there are only a finite number of predicate occurrences in the program, this process is guaranteed to terminate yielding a finite transformed program.

We have not compared the complexity of the bottom-up interpreter presented here with the complexity of top-down interpreters. However, such a comparison should be similar to the comparison in the case of the concrete domain. Evaluation over a finite abstract domain, which is common in abstract interpretation, is very similar to the case of Datalog computation (i.e. all arguments are restricted to be constants or variables). Further, for abstract interpretation, a top-down interpreter must incorporate some form of memoing to insure termination. Ullman [37] has shown that for the case of Datalog, a bottom-up computation using Magic "dominates", asymptotically, under a detailed cost model, any top-down memoing computation. This result is generalized to all Horn clause logic programs in [36].

[3] We thank an anonymous referee for pointing this out.

7 Examples

In this section, we present several detailed examples to illustrate the application of our technique. Each example is chosen to illustrate an important aspect of the approach.

7.1 Precision

Our first example illustrates the need to maintain the connection between calling and return arguments, by introducing separate "input" and "output" arguments in the explicated program, to prevent an undesirable loss of precision.

Example 7.1 Consider an abstract interpretation that performs data dependency analysis of programs, e.g. as in [14, 15, 30]. Such analyses find applications in parallelization of logic programs.

Let the abstract representation of calls and returns to an n-ary predicate be as follows: each argument is represented by a subset of $\{1, \ldots, n\}$ that indicates which argument positions it can possibly share variables with. For example, if two variables X and Y are possible aliases, then, corresponding to the call $q(f(X), g(Y, Z), Z)$, in the explicated abstract program, we have the fact

$$abs\_q(\langle\{1,2\},\{1,2,3\},\{2,3\}\rangle, \perp).$$

Consider the predicate p defined by the single clause

p(X, Y, Y).

First, consider a call where the first and second arguments are aliases: given the call $p(\{1,2\},\{1,2\},\{3\})$, the analysis infers that all arguments can be aliased together on success. Now suppose that the connection between "calling" and "success" patterns is not maintained, but that these are factored separately into relations `call_abs_p` and `succ_abs_p`. This results in the relations

$$call\_abs\_p(\langle\{1,2\},\{1,2\},\{3\}\rangle).$$
$$succ\_abs\_p(\langle\{1,2,3\},\{1,2,3\},\{1,2,3\}\rangle).$$

Then, if there is a subsequent call where only the second and third arguments are aliased, i.e. the call is `abs_p`$(\langle\{1\},\{2,3\},\{2,3\}\rangle)$, the computation generates the tuple

$$succ\_abs\_p(\langle\{1\},\{2,3\},\{2,3\}\rangle)$$

but then discards this tuple because the tuple $succ\_abs\_p(\langle\{1,2,3\},\{1,2,3\},\{1,2,3\}\rangle)$, computed earlier, is "more general" in the sense that

$$\gamma(\langle\{1\},\{2,3\},\{2,3\}\rangle) \subseteq \gamma(\langle\{1,2,3\},\{1,2,3\},\{1,2,3\}\rangle).$$

Even if the tuple $succ\_abs\_p(\langle\{1\},\{2,3\},\{2,3\}\rangle)$ is not discarded, however, the connection between the call $abs\_p(\langle\{1\},\{2,3\},\{2,3\}\rangle)$ and its success pattern $succ\_abs\_p(\langle\{1\},\{2,3\},\{2,3\}\rangle)$ is lost, since if we have only the success patterns $\langle\{1\},\{2,3\},\{2,3\}\rangle$, and $\langle\{1,2,3\},\{1,2,3\},\{1,2,3\}\rangle$, the calling pattern $call\_abs\_p(\langle\{1\},\{2,3\},\{2,3\}\rangle)$ can "unify" with both success patterns, and we are forced to conclude that it can succeed with all its arguments aliased together. This is a factor in the imprecision in the mode analysis algorithm of Mellish [27]. This loss of precision can be avoided, in general, only by maintaining explicit "calling" and "return" arguments to maintain the connection between calling and corresponding success patterns. If this is done, the first call becomes

$$abs\_p(\langle\{1,2\},\{1,2\},\{3\}\rangle, X),$$

and it evaluates to the binding $X = \langle\{1,2,3\},\{1,2,3\},\{1,2,3\}\rangle$, with the resulting relation being

$$abs\_p(\langle\{1,2\},\{1,2\},\{3\}\rangle, \langle\{1,2,3\},\{1,2,3\},\{1,2,3\}\rangle).$$

The second call then computes the tuple

$$abs\_p(\langle\{1\},\{2,3\},\{2,3\}\rangle, \langle\{1\},\{2,3\},\{2,3\}\rangle),$$

and since nothing in the relation abs_p computed so far is more general than this tuple, it is not discarded, whence we can infer that the first argument of the call is independent of the other two when the call returns.

The crux of the matter is also illustrated by Example 5.3. It is interesting to consider the question of when this explicit connection between calling and success patterns need not be maintained without any loss in the precision of the analysis, since in this case the dataflow information inferred can be stored more compactly. The problem is the following. Suppose that the connection between calling and success patterns is not maintained. In other words, we maintain two relations, one consisting of all the calling patterns, and the other of all the success patterns, that have been encountered. In this case, given an arbitrary calling pattern C and an arbitrary success pattern S, if there is an abstract substitution A such that $S = A(C)$ then we have to assume that a call described by C can succeed with its bindings upon success described by S. Now consider a calling pattern C_1 for a predicate p, whose success pattern is S_1, and another calling pattern C_2 for p with success pattern S_2: if the connection between calling and success patterns is maintained explicitly, the tuples corresponding to these are $\langle C_1, S_1\rangle$ and $\langle C_2, S_2\rangle$. Clearly, there must be abstract substitutions A_1 and A_2 such that $S_1 = A_1(C_1)$ and $S_2 = A_2(C_2)$. Assume that C_1 is more precise than C_2, i.e. denotes a smaller set of values, so $\gamma(C_1) \subseteq \gamma(C_2)$. From monotonicity considerations, it follows that $\gamma(S_1) \subseteq \gamma(S_2)$. If the connection between calling and success patterns is not maintained, and there is an abstract substitution A' such that $S_2 = A'(C_1)$, then we must infer that the calling pattern C_1 can give rise to the success pattern S_2. There is a loss of precision in this case if S_2 is less precise than S_1; stated differently, there is no loss of precision if S_2 is at least as precise as S_1, i.e. if $\gamma(S_2) \subseteq \gamma(S_1)$. But from monotonicity, we have $\gamma(S_1) \subseteq \gamma(S_2)$, whence we have $S_1 = S_2$. Thus, in the general case, there is no loss of precision if, whenever there is an abstract substitution A such that $S_2 = A(C_1)$, it is the case that $S_2 = S_1$. That is, whenever there are two different calling patterns C_1 and C_2 for a predicate, with one of them more precise than the other, there is no loss of precision in separating calling and success patterns only if both C_1 and C_2 have the same success pattern. This clearly does not hold in general, so it is necessary to maintain the connection between calling and success patterns explicitly. □

7.2 Depth Abstractions: Replacing Unification by Matching

We now consider an abstract interpretation described by Marriott and Søndergaard [24], based on a scheme proposed by Sato and Tamaki [34]. An application for this analysis is that of replacing unification by matching, which is significantly more efficient (e.g. see [22]): when two terms are being unified, matching can be used if one of the terms is ground. Once the analysis has been carried out for a program, if a relation *abs_p* is such that in each tuple $\langle \bar{a}, \bar{a}' \rangle$ in *abs_p*, the elements in positions Pos in $\bar{a}$ denote ground terms, then head unification for those argument positions in the corresponding predicate p in the original program can be replaced by matching.

Another application, cited by Sato and Tamaki [34], is that of transforming nondeterministic programs to deterministic ones, which are more efficient, based on the success patterns inferred.

The basic idea is to describe a term using a "depth abstraction", i.e. where subterms at depths greater than a specified bound are replaced by variables.[4] For example, the depth-1 abstraction of the term $f(g(a), h(X, f(b, X)), Y)$ is $f(g(U), h(X, V), Y)$ (the principal functor is at depth 0). The analysis of a program is carried out using depth-k abstractions, for some fixed k specified beforehand. An abstract substitution at a point within a clause is maintained as a mapping from the variables occurring in that clause to depth-k abstractions of terms. The application of an abstract substitution, given by *abs_app_subst*, is essentially the same as the application of substitutions over the concrete domain; unification over the abstract domain, given by *abs_unify*, is ordinary unification followed by an abstraction of the resulting terms to depth k. Thus, let $\delta_k(t)$ denote the depth-k abstraction of a term t, and extend this to substitutions as follows: given a (idempotent) substitution θ, $\delta_k(\theta)$ is the depth-k abstraction of the image of each variable in the domain of θ, i.e.

$$\delta_k(\theta) = \{x \mapsto \delta_k(\theta(x)) \mid x \in dom(\theta)\}.$$

Then, the abstract operations can be defined as follows:

$$abs\_app\_subst(A, t_1, t_2) \Leftrightarrow app\_subst(A, t_1, t2), \text{ and}$$
$$abs\_unify(A_1, t_1, t_2, A_2) \Leftrightarrow unify(A_1, t_1, t_2, A_1') \wedge A_2 = \delta_k(A_1').$$

Because subterms are discarded during depth abstraction, analysis using a depth-k abstraction may fail to detect any aliasing that occurs at depths greater than k. Because of this, a variable occurring in a depth-k abstracted term may not necessarily correspond to a free variable at runtime. Soundness therefore requires that such variables be interpreted as denoting all possible terms (i.e. a depth-k abstracted term denotes the set of all its instances).

Example 7.2 Consider the "aliasing" example from [13]:

```
p(X, Y) :- q(X, Y), r(X), s(Y).
q(Z, Z).
r(a).
r(b).
s(b).
s(c).
p(U, V).
```

Assume depth-2 abstraction, and consider a sip that follows Prolog's left-to-right execution strategy. The explicated abstract program is shown in Figure 2. When the rewritten explicated program

[4] If no restrictions are imposed, then a term may have, in general, a number of different "best" depth-k abstractions, and so there may not be adjoint functions α and γ: a simple way around this is to ensure that depth abstractions are *linear*, i.e. do not contain repeated variables [24].

$\texttt{abs\_p}\ (\langle X_\downarrow, Y_\downarrow\rangle,\ \langle X_\uparrow, Y_\uparrow\rangle)\ :-$
$\quad\texttt{abs\_p}(\langle X_\downarrow, Y_\downarrow\rangle, \_),$
$\quad\texttt{abs\_unify}(\mathbf{id},\ \langle X_\downarrow, Y_\downarrow\rangle, \langle X, Y\rangle, A_1),$
$\quad\texttt{abs\_app\_subst}(A_1, \langle X, Y\rangle, \langle X_1, Y_1\rangle),$
$\qquad\quad\texttt{abs\_q}(\langle X_1, Y_1\rangle, \langle X_2, Y_2\rangle), \texttt{abs\_unify}(A_1, \langle X, Y\rangle, \langle X_2, Y_2\rangle, A_2),$
$\quad\texttt{abs\_app\_subst}(A_2, \langle X\rangle, \langle X_3\rangle),\ \texttt{abs\_r}(\langle X_3\rangle, \langle X_4\rangle),\ \texttt{abs\_unify}(A_2, \langle X\rangle, \langle X_4\rangle, A_3),$
$\quad\texttt{abs\_app\_subst}(A_3, \langle Y\rangle, \langle Y_3\rangle),\ \texttt{abs\_s}(\langle Y_3\rangle, \langle Y_4\rangle),\ \texttt{abs\_unify}(A_3, \langle Y\rangle, \langle Y_4\rangle, A_4),$
$\quad\texttt{abs\_app\_subst}(A_4, \langle X, Y\rangle, \langle X_\uparrow, Y_\uparrow\rangle).$
$\texttt{abs\_q}(\langle U_\downarrow, V_\downarrow\rangle, \langle U_\uparrow, V_\uparrow\rangle)\ :-$
$\quad\texttt{abs\_q}(\langle U_\downarrow, V_\downarrow\rangle, \_),$
$\quad\texttt{abs\_unify}(\mathbf{id}, \langle U_\downarrow, V_\downarrow\rangle, \langle Z, Z\rangle, A_1),\ \texttt{abs\_app\_subst}(A_1, \langle Z, Z\rangle, \langle U_\uparrow, V_\uparrow\rangle).$
$\texttt{abs\_r}(\langle X_\downarrow\rangle, \langle X_\uparrow\rangle)\ :-$
$\quad\texttt{abs\_r}(\langle X_\downarrow\rangle, \_), \texttt{abs\_unify}(\mathbf{id}, \langle X_\downarrow\rangle, \langle a\rangle, A_1),\ \texttt{abs\_app\_subst}(A_1, \langle a\rangle, \langle X_\uparrow\rangle).$
$\texttt{abs\_r}(\langle X_\downarrow\rangle, \langle X_\uparrow\rangle)\ :-$
$\quad\texttt{abs\_r}(\langle X_\downarrow\rangle, \_), \texttt{abs\_unify}(\mathbf{id}, \langle X_\downarrow\rangle, \langle b\rangle, A_1),\ \texttt{abs\_app\_subst}(A_1, \langle b\rangle, \langle X_\uparrow\rangle).$
$\texttt{abs\_s}(\langle X_\downarrow\rangle, \langle X_\uparrow\rangle)\ :-$
$\quad\texttt{abs\_s}(\langle X_\downarrow\rangle, \_), \texttt{abs\_unify}(\mathbf{id}, \langle X_\downarrow\rangle, \langle b\rangle, A_1),\ \texttt{abs\_app\_subst}(A_1, \langle b\rangle, \langle X_\uparrow\rangle).$
$\texttt{abs\_s}(\langle X_\downarrow\rangle, \langle X_\uparrow\rangle)\ :-$
$\quad\texttt{abs\_s}(\langle X_\downarrow\rangle, \_), \texttt{abs\_unify}(\mathbf{id}, \langle X_\downarrow\rangle, \langle c\rangle, A_1),\ \texttt{abs\_app\_subst}(A_1, \langle c\rangle, \langle X_\uparrow\rangle).$

$\texttt{abs\_p}(\langle U, V\rangle, \bot).$
$\texttt{abs\_q}(\langle U, V\rangle, \bot)$:-
$\quad\texttt{abs\_p}(\langle X_0, Y_0\rangle, \_),$
$\quad\texttt{abs\_unify}(\mathbf{id},\ \langle X_0, Y_0\rangle, \langle X, Y\rangle, A_1),$
$\quad\texttt{abs\_app\_subst}(A_1, \langle X, Y\rangle, \langle U, V\rangle).$
$\texttt{abs\_r}(\langle U\rangle, \bot)$:-
$\quad\texttt{abs\_p}(\langle X_0, Y_0\rangle, \_),$
$\quad\texttt{abs\_unify}(\mathbf{id},\ \langle X_0, Y_0\rangle, \langle X, Y\rangle, A_1),$
$\quad\texttt{abs\_app\_subst}(A_1, \langle X, Y\rangle, \langle X_1, Y_1\rangle),$
$\qquad\quad\texttt{abs\_q}(\langle X_1, Y_1\rangle, \langle X_2, Y_2\rangle), \texttt{abs\_unify}(A_1, \langle X, Y\rangle, \langle X_2, Y_2\rangle, A_2),$
$\quad\texttt{abs\_app\_subst}(A_2, \langle X\rangle, \langle U\rangle).$
$\texttt{abs\_s}(\langle U\rangle, \bot)$:-
$\quad\texttt{abs\_p}(\langle X_0, Y_0\rangle, \_),$
$\quad\texttt{abs\_unify}(\mathbf{id},\ \langle X_0, Y_0\rangle, \langle X, Y\rangle, A_1),$
$\quad\texttt{abs\_app\_subst}(A_1, \langle X, Y\rangle, \langle X_1, Y_1\rangle),$
$\qquad\quad\texttt{abs\_q}(\langle X_1, Y_1\rangle, \langle X_2, Y_2\rangle), \texttt{abs\_unify}(A_1, \langle X, Y\rangle, \langle X_2, Y_2\rangle, A_2),$
$\quad\texttt{abs\_app\_subst}(A_2, \langle X\rangle, \langle X_3\rangle),\ abs\_r(\langle X_3\rangle, \langle X_4\rangle),\ \texttt{abs\_unify}(A_2, \langle X\rangle, \langle X_4\rangle, A_3),$
$\quad\texttt{abs\_app\_subst}(A_3, \langle Y\rangle, \langle U\rangle).$
$?-\ \texttt{abs\_p}(\langle U, V\rangle, \_).$

Figure 2: The Explicated Abstract Program from Example 8.2

is evaluated bottom-up, the (minimal) relations computed are as follows:

$\texttt{abs\_p}(\langle U, V\rangle, \langle b, b\rangle)$.
$\texttt{abs\_q}(\langle U, V\rangle, \langle U, U\rangle)$.
$\texttt{abs\_r}(\langle X\rangle, \langle a\rangle)$.
$\texttt{abs\_r}(\langle X\rangle, \langle b\rangle)$.
$\texttt{abs\_s}(\langle a\rangle, \bot)$.
$\texttt{abs\_s}(\langle b\rangle, \langle b\rangle)$.

As the reader will notice, the first argument of the relations *abs_p*, *abs_q*, *abs_r* and *abs_s* contain the calling patterns to the respective predicates, while the second arguments contain the corresponding success patterns. In particular, note that the relation *abs_q* captures clearly the aliasing behavior of the predicate $q/2$.

On the other hand, consider a different abstract interpretation, where all constants are mapped to a single abstract domain element **ATOM**, while compound terms are subjected to depth-2 abstraction, and the abstract functions *abs_app_subst* and *abs_unify* modified appropriately. The relations computed in this case are

$\texttt{abs\_p}(\langle U, V\rangle, \langle \texttt{ATOM}, \texttt{ATOM}\rangle)$.
$\texttt{abs\_q}(\langle U, V\rangle, \langle U, U\rangle)$.
$\texttt{abs\_r}(\langle X\rangle, \langle \texttt{ATOM}\rangle)$.
$\texttt{abs\_s}(\langle \texttt{ATOM}\rangle, \langle \texttt{ATOM}\rangle)$.

□

8 Related Work

Early work on abstract interpretation of logic programs was carried out by Mellish, who described a framework for the abstract interpretation of Prolog programs [28]. His approach was to define a set of dataflow equations defining relationships between dataflow information at different program points, and then to solve these equations by computing a least fixpoint in a bottom-up manner. This early approach, which is similar in spirit to ours, did not explicitly maintain the connection between calling and success patterns, resulting in loss of precision (as seen in Example 8.1). There is also a limitation in the class of sips that can be dealt with; a single linear sip is chosen for each rule, and is fixed for all patterns of restricted arguments. (Although this is true of our method as well, our approach rests upon the Magic Templates rewriting, for which the extension to general sips is known.) No characterization was given of the precision of this approach.

At about the same time, Jones and Søndergaard gave a somewhat different framework for the abstract interpretation of Prolog programs, based on a denotational description of Prolog [17]. The meaning of a program is specified by a set of mutually recursive functions, with analogous definitions specifying the abstract meaning. The least solution to the equations defines the abstract semantics that gives the desired flow information. The connection between calling and success patterns is implicit in the "logs" kept by the abstract computations. However, specific algorithms for computing the least solution are not discussed, and no characterization is given of the precision that may be achieved using this approach. Closely related to the Jones-Søndergaard work is that of Winsborough [39], who gives a minimal function graph semantics for logic programs. This also retains the connection between calling and return values. However, no attention is given to algorithmic aspects of analyses.

Bruynooghe gives a framework for abstract interpretation of logic programs [4]. This is based on a top-down execution model, where the computation over the abstract domain is represented by an abstract AND-OR tree. Because of this, termination is somewhat awkward: when a recursive call is encountered that has already been encountered earlier, the bottom element is assumed as the "success value", and the computation continued to a fixpoint; if the recursive call has not been encountered earlier, things become more complicated. Debray discusses a family of abstract interpretations that admit efficient analysis algorithms, using extension tables to guarantee termination [11]. A scheme similar to Bruynooghe's in many ways, involving the top-down construction of abstract AND-OR trees, is described by Corsini and Filè [8]. In none of these cases is it possible to carry out the abstract interpretation of a Prolog program using an ordinary Prolog interpreter and still be able to guarantee termination, because "vanilla" top-down interpreters do not use a complete evaluation strategy and do not explicitly compute a fixpoint (since they do not keep track of all solutions, but compute only one solution at a time). This exemplifies the problems encountered when trying to capture a bottom-up fixpoint computation within a top-down framework.

Marriott and Søndergaard discuss a bottom-up approach to abstract interpretation [23]. However, this work differs from ours in several ways. The most significant difference is that they are concerned with abstract interpretations that approximate the declarative semantics of logic programs, which is given in terms of the model theory of first order logic [38]; this is manifested in their omission of the rewriting step that introduces auxiliary literals and clauses into the program. Because of this, their approach cannot capture abstract interpretations that are based on the operational or denotational semantics of the language. Because of the rewriting to introduce auxiliary literals and clauses that act as filters, our approach is able to capture abstract interpretations based on operational and denotational semantics as well as those based on the declarative semantics.

The connection between top-down abstract interpretation and the Magic Sets transformation is mentioned by Marriott and Søndergaard [25]. Recent work by Mellish discusses the application of Magic Sets evaluation techniques for the computation of fixpoints in mode analysis of logic programs [29]. This work focusses on the application of partial evaluation techniques to derive efficient analysis systems, and does not address issues of precision.

Some time after the writing of this paper, we became aware of independent and essentially simultaneous work by a number of researchers on the application of the Magic Sets transformation to dataflow analysis of logic programs [6, 18, 32]. While these papers are very similar to this paper in spirit, the details of how the transformation is realized differ. For example, Kanamori suggests augmenting the transformed program with indexes to relieve some inefficiencies associated with a naive Magic Sets transformation [18]. Like us, both Nilsson [32] and Codish *et al.* [6] note that a straightforward application of the Magic Sets transformation can result in a loss of precision because the connection between calling and success patterns is not maintained. Codish *et al.* suggest a modified transformation that, however, results in programs whose bottom-up semantics no longer corresponds to the operational behavior of the original program. In contrast, the explicated programs we consider retain the connection between calling and success patterns: as a result, precision is not compromised despite the use of a straightforward Magic Sets rewriting.

9 Conclusions

Dataflow analysis of logic programs requires a synthesis of top-down and bottom-up information flow: a top-down component to mimic the control strategy of the language under consideration, and a bottom-up component to compute fixpoints. In much of the literature on dataflow analysis of logic programs, this synthesis is either not addressed, or is given using techniques such as memoization or ad hoc termination rules that are extraneous to the operational semantics of the language under consideration. This paper discusses the application of the *magic templates* algorithm, originally devised as a technique for efficient bottom-up evaluation of logic programs, to dataflow analysis of logic programs. The principal contributions of this work is to demonstrate how the fixpoint evaluation algorithm can be decoupled from the control strategy of the language under consideration. It turns out that a straightforward application of Magic Templates rewriting can lead to an undesirable loss in the precision of analysis. We show how the original Magic Templates strategy can be modified to avoid this problem, and prove that the resulting analysis algorithm is at least as precise as any other abstract interpretation that uses the same abstract domain and abstract operations.

References

[1] Afrati, F., Papadimitriou, C., Papageorgiou, G., Roussou, A., Sagiv, Y. and Ullman, J.,D. "Convergence of Sideways Query Evaluation", Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.

[2] Bancilhon, F., Maier, D., Sagiv, Y. and Ullman, J.D., " Magic sets and other strange ways to implement logic programs", In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Boston, Massachusetts, March 1986.

[3] Beeri, C. and Ramakrishnan, R. " On the power of magic", In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[4] Bruynooghe, M., "A Framework for the Abstract Interpretation of Logic Programs", Rsearch Report 62, Katholieke Universiteit Leuven, Belgium, Oct. 1987.

[5] Chang, J.-H., Despain, A. M., and DeGroot, D., "AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis", Digest of Papers, Compcon 85, IEEE Computer Society, FEB 1985.

[6] Codish, M., Dams, D., and Yardeni, E., "Bottom-up Abstract Interpretation of Logic Programs", Technical Report CS90-24, Dept. of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, Oct. 1990. (To appear in *Theoretical Computer Science.*)

[7] Codognet, C., Codognet, P., and Corsini, M., "Abstract Interpretation for Concurrent Logic Languages", in *Proc. NACLP-90*, Austin, TX, Oct. 1990 (to appear).

[8] Corsini, M. M., and Filè, G., "The Abstract Interpretation of Logic Programs: A General Algorithm and its Correctness", Research Report, Dept. of Pure and Applied Mathematics, University of Padova, Italy, Dec. 1988.

[9] Cousot, P., and Cousot, R., "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", Proc. Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 238-252.

[10] Cousot, P., and Cousot, R., "Systematic Design of Program Analysis Frameworks", Proc. Sixth ACM Symposium on Principles of Programming Languages, 1979, pp. 269-282.

[11] Debray, S. K., "Efficient Dataflow Analysis of Logic Programs", Proc. Fifteenth ACM Symposium on Principles of Programming Languages, San Diego, CA, Jan. 1988.

[12] Debray, S. K., and Mishra, P., "Denotational and Operational Semantics for Prolog", *J. Logic Programming* vol. 5 no. 1, March 1988, pp. 61-91.

[13] Debray, S. K., and Warren, D. S., "Automatic Mode Inference for Logic Programs", *J. Logic Programming* vol. 5 no. 3, Sept. 1988, pp. 207-229.

[14] Debray, S. K., "Static Inference of Modes and Data Dependencies in Logic Programs", *ACM Transactions on Programming Languages and Systems* vol. 11 no. 3, July 1989, pp. 418-450.

[15] Jacobs, D., and Langen, A., "Accurate and Efficient Approximation of Variable Aliasing in Logic Programs", Proc. NACLP-89, Cleveland, OH, Oct. 1989.

[16] Jones, N. D., and Mycroft, A., "Stepwise Development of Operational and Denotational Semantics for PROLOG", Proc. 1984 Int. Symposium on Logic Programming, Atlantic City, New Jersey, IEEE Computer Society, Feb. 1984. pp. 289-298.

[17] Jones, N. D., and Søndergaard, H., "A Semantics-Based Framework for the Abstract Interpretation of Prolog", in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (eds.), Ellis Horwood, 1987.

[18] Kanamori, T., "Abstract Interpretation based on Alexander Templates", Technical Report TR-549, ICOT, Tokyo, March 1990.

[19] Kifer, M. and Lozinskii, E., "SYGRAF: Implementing Logic Programs in a Database Style", IEEE Trans. on Software Engineering, 1988.

[20] Krishnamurthy, R., Ramakrishnan, R. and Shmueli, O., "A Framework for Testing Safety and Effective Computability of Extended Datalog", Proc. SIGMOD 88, Chicago.

[21] Maher, M., and Ramakrishnan, R., "Déjà Vu in Fixpoints of Logic Programs", Proc. NACLP-89, Cleveland, OH, Oct. 1989.

[22] Maluszynski, J., and Komorowski, H., "Unification-free Execution of Logic Programs", Proc. 1985 Symposium on Logic Programming, Boston, July 1985, pp. 78-86. IEEE Press.

[23] Marriott, K., and Søndergaard, H., "Bottom-up Abstract Interpretation of Logic Programs", Proc. Fifth International Conference on Logic Programming, Seattle, WA, August 1988. MIT Press.

[24] Marriott, K., and Søndergaard, H., "On Describing Success Patterns of Logic Programs", Technical Report 88/12, Department of Computer Science, University of Melbourne, Australia, May 1988.

[25] Marriott, K., and Søndergaard, H., "Semantics-based Dataflow Analysis of Logic Programs", *IFIP-89*, pp. 601-609, 1989.

[26] Marriott, K., Søndergaard, H., and Jones, N. D., "Denotational Abstract Interpretation of Logic Programs", Manuscript, Dept. of Computer Science, University of Melbourne, June 1990.

[27] Mellish, C. S., "The Automatic Generation of Mode Declarations for Prolog Programs", DAI Research Paper 163, Dept. of Artificial Intelligence, University of Edinburgh, Aug 1981.

[28] Mellish, C. S., "Abstract Interpretation of Prolog Programs", Proc. Third International Conference on Logic Programming, London, July 1986 (Springer-Verlag LNCS vol. 225).

[29] Mellish, C. S., "Using Specialisation to Reconstruct Two Mode Inference Systems", Manuscript, Department of Artificial Intelligence, University of Edinburgh, June 1990.

[30] Muthukumar, K., and Hermenegildo, M., "Determination of Variable Dependence Information at Compile Time Through Abstract Interpretation", In Proceedings of the North American Conference on Logic Programming, Cleveland, OH, Oct. 1989.

[31] F. Nielson, "Strictness Analysis and Denotational Abstract Interpretation", *Information and Computation* **76**, 29-92 (1988).

[32] Nilsson, U., "Abstract Interpretation: A Kind of Magic", Technical Report, Dept. of Computer Science, Linköping University, Sweden, 1990. (To appear in *Theoretical Computer Science*.)

[33] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3):189–216, 1991.

[34] Sato, T., and Tamaki, H., "Enumeration of Success Patterns in Logic Programs", *Theoretical Computer Science* **34**, (1984), pp. 227-240.

[35] Søndergaard, H., "Semantics-Based Analysis and Transformation of Logic Programs", Doctoral Dissertation, Dept. of Computer Science, University of Copenhagen, Denmark, Dec. 1989. (Also available as Technical Report 89/21, Dept. of Computer Science, University of Melbourne, Australia.)

[36] S. Sudarshan and Raghu Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In *Proceedings of the International Logic Programming Symposium*, 1993.

[37] Ullman, J.D., "Bottom-Up Beats Top-Down for Datalog", In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 140-150, Philadelphia, PA, 1989.

[38] van Emden, M., and Kowalski, R., "The Semantics of Predicate Logic as a Programming Language", *JACM 28*, no. 4, Oct. 1976, pp. 733-742.

[39] Winsborough, W., "A Minimal Function Graph Semantics for Logic Programs", Technical Report No. 711, Dept. of Computer Science, The University of Wisconsin-Madison, August 1987.