Automatic Mode Inference for Logic Programs†

Saumya K. Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

David S. Warren
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract: In general, logic programs are undirected, i.e. there is no concept of ''input'' and ''output'' arguments to a procedure. An argument may be used either as an input or as an output argument, and programs may be executed either in a ''forward'' direction or in a ''backward'' direction. However, it is often the case that in a given program, a predicate is used with some of its arguments used consistently as input arguments and others as output arguments. Such *mode information* can be used by a compiler to effect various optimizations.

This paper considers the problem of automatically inferring the modes of the predicates in a program. The dataflow analysis we use is more powerful than approaches relying on syntactic characteristics of programs, e.g. [18]. Our work differs from that of Mellish [14, 15] in that (*i*) we give a sound and efficient treatment of variable aliasing in mode inference; (*ii*) by propagating instantiation information using state transformations rather than through dependencies between variables, we achieve greater precision in the treatment of unification, e.g. through =/2; and (*iii*) we describe an efficient implementation based on the dynamic generation of customized mode interpreters. Several optimizations to improve the performance of the mode inference algorithm are described, as are various program optimizations based on mode information.

This work was supported in part by the National Science Foundation under grant number DCR-8407688.

† This is a revised version of a paper presented at the Third Symposium on Logic Programming, Salt Lake City, Utah, Sept. 1986.

1. Introduction

In general, logic programs are not directed, in the sense that there is no concept of ‘‘input’’ and ‘‘output’’ variables. A variable may be used as either an input or an output variable, and programs may be executed in either a ‘‘forward’’ or a ‘‘backward’’ direction. However, it is often the case that in a particular program, a predicate is executed in one direction only, i.e. it is always called with a particular set of its variables bound (the ‘‘input’’ variables) and another set unbound (the ‘‘output’’ variables). If a compiler is aware of this usage, it can make various optimizations based on this fact, e.g. use efficient special-purpose unification routines instead of the general unification algorithm where appropriate, infer determinacy, etc.

Traditionally, this information has been supplied by the programmer using what are called ‘‘mode declarations’’ [21]. This has the problem that errors made by the programmer in declaring modes can lead to some very strange program behaviour, whose cause can be hard to find. We consider the alternative solution of having the compiler infer the modes, using these either to optimize a program without mode declarations, or to verify the declarations made by the programmer, much as type-checkers in other languages verify type declarations.

Some researchers have considered the question of verifying the consistency of mode declarations supplied by the user [3, 17, 19]. The issue of automatic mode inference has been considered by Mellish [14, 15] and Reddy [18], and more recently by Bruynooghe et al. [5] and Mannila and Ukkonen [13]. We focus our attention on the inference of modes rather than on verifying the consistency of user-supplied mode declarations. Unlike Reddy, who uses a syntactic analysis of the program to assign modes to predicates, we use a dataflow analysis that is usually more precise. The work of Bruynooghe et al. is very similar to ours; Mannila and Ukkonen restrict their attention to a much simpler mode set, consisting of two types of modes, *ground* and *nonground*, and hence the precision of their algroithm is not as good as ours. The principal contributions of this work are as follows:

(1) We consider the problem of *aliasing* explicitly, and give a sound and efficient method of dealing with aliasing in the analysis framework (to our knowledge, this has not been done elsewhere in the literature on mode inference). This enables us to prove the soundness of our algorithm.

(2) While Mellish uses dependencies between variables to propagate information regarding their instantiation, we do this by treating literals in a clause as state transformers. This results in greater precision in the treatment of clauses containing predicates like =/2, which occur quite frequently in programs.

(3) Our treatment suggests an efficient implementation where a mode inference program customized to the program being analyzed can be generated dynamically. This can then be

executed to get the modes. This generative approach results in greater efficiency because an extra level of interpretation can be avoided.

(4) Our approximation domain is much simpler than that used by Mellish. This has two positive effects: the treatment of aliasing can be simplified, and the inference procedure can be made more efficient. On the other hand, it may result in the loss of more information than would have been the case if a richer approximation domain had been used.

We assume the reader is acquainted with the basic concepts of logic programming. The rest of this paper is organized as follows: Section 2 develops some of the ideas and definitions that are used later in the paper. Section 3 considers, for expository purposes, a simple mode inference scheme. Section 4 generalizes the results of Section 3 to a more realistic scheme. Section 5 considers implementation issues and possible optimizations. Section 6 describes some applications of mode information, and Section 7 describes the performance of our system.

2. Preliminaries

A Prolog program consists of a number of definite Horn clauses P (i.e. clauses containing exactly one positive literal), together with a negative clause Q called the *query*.

The mode of a predicate in a program indicates how its arguments will be instantiated when that predicate is called. The modes of a program thus represent statements about all computations that are possible from it. For the sake of simplicity, we classify terms occurring in a program into four classes with regard to how they are instantiated: *empty*, *closed*, *free* and *don't-know* which refer, respectively, to the empty set, the set of closed (i.e. ground) terms, the set of free variables and the set of all terms. We thus consider modes over the domain $\Delta = \{c, d, e, f\}$, where e denotes to the empty set, c denotes closed terms, f denotes free variables, and d denotes ''don't-know'' terms (i.e. terms which may be partially instantiated, or which are not known to be closed or free variables).

This serves to approximate the unbounded number of terms that may exist during the execution of a program by a finite, bounded set that may be reasoned about statically. This is done by defining an equivalence relation over the set of terms that partitions it into a finite, bounded number of equivalence classes, and then approximating the execution of the program by computing over these equivalence classes. This is an instance of a more general approach to program analysis called abstract interpretation [7]. In this case there are four such equivalence classes, represented by the elements of Δ. Clearly, modes are meaningful only when the control strategy has been specified. For the purposes of this paper, we assume the control strategy of Prolog, with its left-to-right order of evaluation of the literals in a clause; however, the techniques described here are applicable to other evaluation strategies as well.

Given any set of terms, it is necessary to specify how to find its *instantiation*, i.e. the element of Δ that ‘‘describes’’ it. This is given by the *instantiation function* ι, which is defined as follows:

Definition: The *instantiation* $\iota(T)$ of a set of terms T is given by $\iota(T) = \cap\{\delta \in \Delta \mid T \subseteq \delta\}$. ●

Thus, any set of terms containing only ground terms will have instantiation c, any set of terms containing only uninstantiated variables will have instantiation f, and so on. It is not difficult to verify that if $T = T_1 \cup T_2$, then $\iota(T) = \iota(T_1)$ *$lub* $\iota(T_2)$.

When a clause is selected for resolution against a goal, its variables are renamed so that it is variable-disjoint with the goal. Consider a use of clause C in a computation where the variables of C have been renamed via a renaming substitution σ: we refer to this as a σ-*activation* of C. The variable names appearing in a clause are referred to as its *program variables*; the program variables of any clause form a finite set. For convenience, we use the notation ‘‘*v $curly_arrow t*’’ to indicate that at runtime, the variable v can be instantiated to the term t. The set of terms a variable can be instantiated to at any point in a program is described using instantiation states (ι-states for short):

Definition: Let V be the set of program variables of a clause C in a program. Then, an *instantiation state* ζ at a point in that clause is a mapping

$$\zeta : V \rightarrow \Delta$$

such that for any v in V, if for any σ-activation of C in the computation it is the case that $\sigma(v)$ *$curly_arrow* t at that point, then $t \in \zeta(v)$. ●

The extension of the mapping ζ to terms is straightforward: constants are mapped to c, and a compound term is mapped to c if every proper subterm of it is mapped to c, and to d otherwise. We denote the extended mapping by ζ. We refer to the ι-state of a clause where each variable is mapped to f as the *initial ι-state* ζ_{init}.

Tuples of instantiations will be referred to as *instantiation patterns*, or ι-*patterns*. In particular, the ι-patterns at the entry to a call will be referred to as *calling patterns*, and ι-patterns at the return from a call as *success patterns* for that call. For example, the call

$$\ldots, p(X, f(X), g(a)), \ \ldots$$

with the variable X uninstantiated, has the calling pattern *$langle f,d,c$rangle*. If the call succeeds binding X to the constant ‘*b*’, then its success pattern is *$langle c,c,c$rangle*.

The once-only nature of Prolog's assignment means that terms can only become more instantiated as execution progresses. The notion of ''more instantiated than'' is quite straightforward when dealing with individual terms: a term t_2 is more instantiated than another term t_1 if t_2 is a substitution instance of t_1. However, during static analysis, variables will be associated with sets of terms, which makes it necessary to ''lift'' this order to sets of terms. Define unification over sets of terms, denoted by *s_unify*, as follows:

Definition: Given sets of terms T_1 and T_2, $s\_unify(T_1, T_2)$ is the least set of terms T such that for each pair of unifiable terms $t_1 \in T_1$, $t_2 \in T_2$ with most general unifier θ, $\theta(t_1)$ is in T. ●

It can be seen that given two terms t_1 and t_2, t_2 is more instantiated than t_1 if and only if the result of unifying t_1 and t_2 is the term t_2. We define the instantiation order over sets of terms, denoted $\unlhd$, as the natural extension of this:

Definition: Given sets of terms T_1 and T_2, $T_1 \unlhd T_2$ if and only if $s\_unify(T_1, T_2) = T_2$. ●

The reader may verify that $\unlhd$, as defined above, is transitive. If the set of terms T under consideration is closed under unification (i.e. for any t_1 and t_2 in T, if t_1 and t_2 are unifiable with most general unifier θ, then $\theta(t_1)$ is also in T), then $\unlhd$ is also reflexive, and hence a preorder, which is easily extended to a partial order $\unlhd/\sim$ modulo variable renaming. In what follows, we will concern ourselves only with this partial order, and with this understanding, abuse notation slightly and write the partial order as $\unlhd$. Since each element of Δ is closed under unification and variable renaming, it follows that $\unlhd$ is a partial order over Δ:

$$f \unlhd d \unlhd c \unlhd e$$

The join operation for this order will be written as ∇. It is easy to verify that for any two elements δ_1, δ_2 in Δ, $s\_unify(\delta_1, \delta_2) = \delta_1 \nabla \delta_2$.

The elements of Δ form a complete lattice with respect to inclusion, as shown in Figure 1. The ordering on this lattice will be written *$le*, with the least upper bound operation denoted by *$lub*. The lower an element is in this lattice, the smaller the corresponding set of terms, and intuitively, the greater the amount of information it conveys.

The ordering *$le* extends to tuples (ι-patterns) in the natural way via element-wise comparison. Let $\downarrow$ denote the selection operation on tuples: *$langle* t_1, ..., t_n*$rangle* $\downarrow k = t_k$ if $1 \le k \le n$, and is undefined otherwise. Then, given two tuples $T_1, T_2 \in \Delta^n$, T_1 *$le* T_2 if and only if $T_1 \downarrow i$

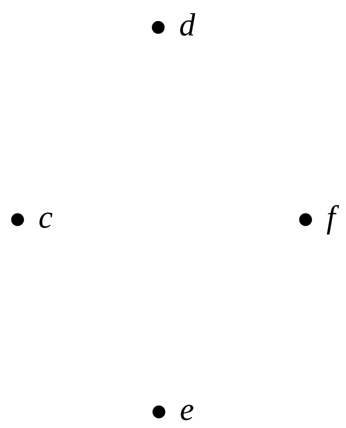

Figure 1: The lattice *\$langle* Δ, *\$le* *\$rangle*

\$le $T_2{\downarrow}i$ for $1 \le i \le n$. Similarly, the greatest lower bound and least upper bound of ι-patterns are defined as the ι-patterns obtained by taking the appropriate bound elementwise. The definitions are similar for $\trianglelefteq$.

The mode of a predicate in a program is a conservative statement of how its arguments will be instantiated in any call to it:

Definition: The mode of a predicate p in a program, given the set of all calling patterns $CALL_p$ for p, is defined to be *\$lub* $CALL_p$. ●

The mode of an n-ary predicate is therefore an element in Δ^n. In general, if the calling patterns are computed over a domain S, the corresponding mode will be referred to as an S-mode. E.g. if all calling patterns are tuples over $\{c,d,e\}$ then the corresponding mode will be referred to as a ‘‘$\{c,d,e\}$-mode’’.

A mode inferred for a predicate in a program is sound if and only if for each argument position of that predicate, the set of terms that can occur in that position in all calls to that predicate at runtime is contained in the set of terms denoted by the corresponding element of the mode. Thus, if an argument position of a predicate is inferred to have mode c, then soundness demands that this argument be instantiated with a ground term in any call to that predicate. More formally,

Definition: An inferred mode I_M of a predicate in a program is *sound* if and only if the calling pattern I_C of any call to that predicate that can arise in the program satisfies I_C *$le* I_M. ●

A mode inference procedure is sound if the mode inferred by it is sound for all predicates in all programs.

Clearly, a procedure that always infers the mode of every predicate to be *$langle* d, d, . . ., d*$rangle* is sound, if somewhat dull. We will be interested in sound inference procedures that strive to be more precise than this, though complete precision will in general be unattainable.

Mode inference requires the inference of the calling patterns of a predicate. The computation of calling patterns, in turn, is dependent on a knowledge of success patterns. We now consider the relationship between success patterns for a clause and success patterns for the literals in the clause. For this, we have to take into account the left-to-right evaluation strategy of Prolog, and the manner in which this changes the instantiations of terms as execution progresses. This requires the ability to go from ι-states to instantiations of argument positions and vice versa:

Definition: Given a tuple of terms T = *$langle*$T_1,...,T_n$*$rangle* appearing in a clause and an ι-state ζ for that clause, the *projection* of ζ on T, written $\pi_T(\zeta)$, is defined to be the tuple *$langle* $\zeta(T_1), . . ., \zeta(T_n)$*$rangle* . ●

Projections of ι-states permit the inference of instantiations of argument positions given the instantiations of variables. This is necessary, for example, in determining the calling patterns for a literal in the body of a clause. It is also necessary to be able to go in the other direction, so as to determine, for example, how success through a literal affects ι-states. Consider the unification of an n-tuple of terms $\bar{T}$ with another n-tuple of terms $\bar{T}'$, whose ι-pattern is I, in an ι-state ζ. This amounts to the unification of terms $\bar{T}\downarrow j$ with terms $\bar{T}'\downarrow j$ with instantiation $I\downarrow j$, $1 \le j \le n$. Now from our choice of Δ, if the instantiation of any term t_0 in an ι-state is I_0, then the instantiation of any subterm of t_0 in that ι-state is no worse than I_0. Thus, since the instantiation of $\bar{T}'\downarrow j$

in ι-state ζ is $I\downarrow j$, the instantiation of any subterm of $\bar{T}'\downarrow j$ is also $I\downarrow j$. If a variable v occurs as a subterm of the k^{th} element of $\bar{T}$, then its instantiation in ι-state ζ is, in general, $\zeta(\bar{T}\downarrow k)$, so that after unification its instantiation is given by $\zeta(\bar{T}\downarrow k)\ \nabla\ I\downarrow k$. Since v may occur in more than one element of $\bar{T}$, we have to consider all such positions k and take the least upper bound ∇ to determine the final instantiation of v.[1] This is done by defining a transformation δ on ι-states:

Definition: Given an n-tuple of terms T in an ι-state ζ and an ι-pattern $I \in \Delta^n$, the ι-state transformation δ is defined as follows: if a variable v occurs in T, then

$$\delta(T,I,\zeta)(v) = \nabla\{(\zeta(T\downarrow k)\ \nabla\ I\downarrow k) : v \text{ is a subterm of } T\downarrow k\},$$

else $\delta(T, I, \zeta)(v) = \zeta(v)$. ●

Thus, given a goal '$p(\bar{T})$' in a clause, let ζ be an ι-state for the clause just before this goal. The calling pattern for the goal is $\pi_{\bar{T}}(\zeta)$. Let I be a success pattern for this goal. Then, the ι-state just after this call is given by $\delta(\bar{T}, \mathrm{I}, \zeta)$. It should be noted, however, that possible aliasing effects were not considered here; indeed, ι-states do not contain enough information to cope with aliasing and, as we will see, this can be a problem when considering the soundness of mode inference.

We conclude this section by mentioning a restriction we place on programs. If sound mode inference is to be possible, the entire search tree that might be traversed during execution should be available statically for analysis. Programs where this is satisfied, i.e. which do not contain any calls to *call*, *assert* etc., will be referred to as *static*. Throughout the rest of the paper, we will assume that we are dealing with static programs. Somewhat more limited analyses may be carried out for certain classes of dynamic programs using techniques discussed in [Debray Analysis Dynamic 1987].

3. Calling and Success Patterns over { c, d, e }

For expository purposes, we restrict our attention in this section to a very simple approximation domain, consisting of sets of terms that are either empty ('*e*'), closed ('*c*') or the universe ('*d*'). In the next section, the ideas of this section are extended to the full domain Δ.

---

[1] For the sake of simplicity we assume the two tuples of terms do not share variables, but this restriction is easily gotten around.

3.1. Admissible Success Patterns

Given a calling pattern for a call, not all success patterns can be considered ''reasonable'' for it. Intuitively, a success pattern for a call is ''reasonable'' only if it agrees with what is already known to be a ''reasonable'' computation of the call. We define an *admissible success pattern* relation *SUCCPAT*(p) over calling and success patterns for a predicate p as follows:

Definition: Given a calling pattern I_C for a predicate p and a success pattern I_S, the tuple $\$langle\, I_C, I_S \$rangle$ is in *SUCCPAT*(p) if and only if there exists a clause

$$p(\bar{T}_0) :- \mathrm{q}_1(\bar{T}_1), \ldots, \mathrm{q}_n(\bar{T}_n), \qquad n \geq 0,$$

in the program such that $I_S = \pi_{\bar{T}_0}(\zeta_n)$, where ζ_{init} is the initial ι-state of the clause; $\zeta_0 = \delta(\bar{T}_0, I_C, \zeta_{init})$; $\zeta_j = \delta(\bar{T}_j, I_j, \zeta_{j-1})$ for $1 \leq j \leq n$; and $\$langle\, \pi_{\bar{T}_j}(\zeta_{j-1}), I_j \$rangle \in$ *SUCCPAT*(q_j). ●

An admissible success pattern is thus obtained by determining the ι-state resulting after all literals in the body of the clause have been evaluated, and then determining the resultant instantiations of the terms in the head of the clause. This is done by first determining the ι-state ζ_0 resulting from the unification of the calling literal with the head. From this, ι-states after successive literals in the body are determined by computing the calling pattern, and an admissible success pattern corresponding to it, for each literal, proceeding from left to right in accordance with Prolog's evaluation strategy.

Example: Consider the program

$p(X, Y) :- q(X, Z), r(Z, Y).$
$q(a, \_).$
$r(b, b).$

Here, $\bar{T}_0 = \$langle\, \mathrm{X,Y} \$rangle$, $\bar{T}_1 = \$langle\, \mathrm{X,Z} \$rangle$, $\bar{T}_2 = \$langle\, \mathrm{Z,Y} \$rangle$. Suppose p is called with the calling pattern $\$langle\, d,d \$rangle$. Then, $\zeta_0 = \{X \rightarrow d, Y \rightarrow d, Z \rightarrow d\}$, where $T \rightarrow s$ indicates that the instantiation of the term T is s.

It follows from this that the calling pattern for q is $\pi_{\bar{T}_1}(\zeta_0) = \$langle\, d,d \$rangle$. From the clauses for q, an admissible success pattern for this calling pattern can be deduced to be $\$langle\, c,d \$rangle$. From this, $\zeta_1 = \delta(\bar{T}_1, \$langle c, d\$rangle, \zeta_0) = \{X \rightarrow c, Y \rightarrow d, Z \rightarrow d\}$.

The calling pattern of r is then the projection of ζ_1 on $\bar{T}_2$, which is $\$langle\, d,d \$rangle$. The only admissible success pattern for r corresponding to this is $\$langle\, c,c \$rangle$. This gives

$\zeta_2 = \delta(\bar{T}_2, \$langlec\ , c\$rangle\ , \zeta_1) = \{X \rightarrow c, Y \rightarrow c, Z \rightarrow c\}$.

Then, an admissible success pattern for p relative to its calling pattern $\$langle\ d,d\$rangle$ is $\pi_{\bar{T}_0}(\zeta_2) = \$langle\ c, c\$rangle$. ●

3.2. Admissible Calling Patterns

Given a class of queries that the user may ask of a program, only some of the possible calling patterns will in fact be encountered during computations. During static analysis, therefore, not all calling patterns for a predicate will be ‘‘admissible’’. The admissible success pattern relation can be used to define admissible calling patterns. The set of *admissible calling patterns* $CALLPAT(p)$ of a predicate p are defined as follows:

Definition: The set of admissible calling patterns $CALLPAT(p)$ for a predicate p in a program for a class of queries Q is the least set such that

(1) If p is an exported predicate and I is a calling pattern for p in the class of queries Q, then I is in $CALLPAT(p)$;

(2) If q_0 is a predicate in the program, $I_c \in CALLPAT(q_0)$, and there is a clause in the program of the form

$$q_0(\bar{T}_0) :- q_1(\bar{T}_1), ..., q_n(\bar{T}_n).$$

Let the ι-state at the point immediately after the literal $q_j(\bar{X}_j)$, $0 \le j \le n$, be ζ_j, where ζ_{init} is the initial ι-state of the clause; $\zeta_0 = \delta(\bar{T}_0, I_c, \zeta_{init})$; then, for $1 \le i \le n$, $cp_i = \pi_{\bar{T}_i}(\zeta_{i-1})$ is in $CALLPAT(q_i)$; and if $\$langle\ cp_i, sp_i\$rangle$ is in $SUCCPAT(q_i)$, then $\zeta_i = \delta(\bar{T}_i, sp_i, \zeta_{i-1})$. ●

The set $CALLPAT(p)$ is in fact a conservative approximation of the set of calling patterns $CALL_p$ of the predicate p in a program. That the sets $CALLPAT(p)$ and $SUCCPAT(p)$ are in fact computable can be seen from the fact that given a set of calling patterns for the predicates exported by a module, we can begin by considering these calling patterns and propagate instantiations, collecting success and calling patterns until no more can be found. The termination of this procedure follows from the finiteness of the program and the approximation domain Δ.

3.3. Aliasing

An issue that we have not considered so far is that of *aliasing*. Consider, for example, the program

$$p(X, Y) :- q(X, Y), r(X), s(Y).$$

q(*Z*, *Z*).
r(*a*).
s(_).

Once execution succeeds past the literal for *q* in the clause for *p*, the variables *X* and *Y* are aliased together, so that when the goal *r*(*X*) binds *X* to *a*, *Y* also gets bound to *a*. Thus, if the calling pattern for *p* is *$langle d, d $rangle* , we infer the success pattern *$langle c, d $rangle* , even though it is really *$langle c, c$rangle* .

Aliasing, as in this case, refers to the situation where two or more variables point to the same object, so that changing the instantiation of one term might result in changes to the instantiation of another. For example, in the goal

$$. . ., X = f(Y), . . ., Y = a, . . .$$

the unification of *Y* with *a* affects the instantiation of *X*.

The reason aliasing is a problem is that since ι-patterns do not contain variables, they are not sufficiently expressive to capture the effects of aliasing. It turns out, as we will show, that aliasing does not affect the soundness of mode inference as long as we restrict our attention to the approximation domain {*c*, *d*, *e*}. However, if we are also interested in knowing when an argument will be a free variable, aliasing will have to be taken into account explicitly.

3.4. Soundness

Soundness requires that the mode inferred for any predicate must be above any calling pattern that can arise at runtime with respect to *$le* . For this, it is sufficient to ensure that any calling pattern inferred during mode analysis is above any calling pattern that can arise at that point in the program at runtime, with respect to *$le* . For this, we define the notion of *safe* instantiations:

Definition: An inferred instantiation I_0 for a term *T* at a point in a clause is defined to be *safe* if and only if for any execution of the clause, the instantiation I_1 of *T* at runtime at that point in the clause satisfies I_1 *$le* I_0.

Lemma 3.1: Admissible calling and success patterns over {*c*, *d*, *e*} are safe.

Proof: If ι-patterns are restricted to {*c*, *d*, *e*}, then the worst that can happen due to aliasing is that the instantiation of a variable can change from *d* to *c*. In this case, the actual instantiation is *c*, the inferred instantiation is *d*, and since *c $le d*, the instantiation is safe. *$always*

This result, in fact, follows from a more general result in [10], which states that admissible calling and success patterns over an arbitrary approximation domain Δ are safe whenever each element of Δ is closed under instantiation, i.e. for each δ in Δ, if x is in δ then every substitution instance of x is also in δ.

Lemma 3.2: If a call to a predicate p with arguments $\bar{T}_{in}$ in a static program succeeds with arguments $\bar{T}_{out}$, then for some I_S, $\$langle\, \iota(\bar{T}_{in}),\ I_S\$rangle \in$ *SUCCPAT*(p) and $\iota(\bar{T}_{out})\ \$le\ \ I_S$, where the ι-patterns are over $\{c, d, e\}$.

Proof: By induction on the number of steps k of deduction.

Consider the case $k = 1$. For the call to succeed in one step of deduction, there must be a unit clause $p(\bar{T})$ which unifies with $p(\bar{T}_{in})$. In this case, the calling pattern is $I_C = \iota(\bar{T}_{in})$, and an admissible success pattern is $I_S = \pi_{\bar{T}}(\delta(\bar{T}, I_C, \zeta_{\mathrm{init}}))$, where ζ_{init} is the initial ι-state of the clause. By definition, $\$langle\, \iota(\bar{T}_{in}),\ I_S\$rangle \in$ *SUCCPAT*(p), so the lemma holds.

Now assume that the lemma holds for values of $k < n$. Since the program is static, each clause and each call can be considered when computing admissible success patterns. Consider a goal $p(\bar{T}_{in})$ that succeeds in n steps. Then, there is a clause for p

$$\mathrm{p}(\bar{T}) :- \mathrm{q}_1(\bar{T}_1), \ldots, \mathrm{q}_n(\bar{T}_n)$$

where each of the subgoals q_j succeeds in fewer than n steps. From the induction hypothesis, if q_j succeeds with arguments $\bar{T}_{j\,(out)}$, then there is an ι-pattern I_j such that $\$langle\, \iota(\bar{T}_{in}),\ I_j\$rangle \in$ *SUCCPAT*(q_j) and $\iota(\bar{T}_{j\,(out)})\ \$le\ \ I_j$. Since success patterns are being computed over $\{c, d, e\}$, from Lemma 3.1, they are safe. Thus, if I_{infer} and I_{actual} represent the inferred and actual instantiations of any term at a point just after q_j, then $I_{actual}\ \$le\ \ I_{infer}$. In particular, this holds for $j = n$, so that for each element of $\bar{T}$, the inferred instantiation is above the actual instantiation with respect to $\$le$. The lemma follows directly from this. $\$always$

Theorem 3.1: $\{c, d, e\}$-Mode inference over admissible calling patterns is sound for static programs.

Proof: By induction on the number of steps k in the deduction.

Let *CALLPAT*(p) be the set of admissible calling patterns for a predicate p, and consider a call $p(\bar{T}_{in})$ after k deductions.

If $k = 0$, then the user's query must be

$$:- \mathrm{p}(\bar{T}_{in}), \ldots, \mathrm{q}_n(\bar{T}_n).$$

Since no variable bindings have been set up at this point, the calling pattern is $\pi_{\bar{T}_{in}}(\zeta_{init})$, and this is in *CALLPAT*(p) by definition.

Assume that the statement is true of calls after k deductions, for $k < n$, and consider a call $p(\bar{T}_{in})$ after n deductions. For this, there must be a clause for a predicate r,

$\mathrm{r}(\bar{T}_0)$:– $\mathrm{q}_1(\bar{T}_1), \ldots, \mathrm{q}_n(\bar{T}_n)$

where $q_j = p$ for some j between 1 and n, and r was called with the arguments $\bar{T}_r$. Then, the calling pattern for r is $I = \iota(\bar{T}_r)$, and by the induction hypothesis, this calling pattern is in *CALLPAT*(r). Let ζ_{init} be the initial ι-state of this clause, $\zeta_0 = \delta(\bar{T}_0, I, \zeta_{init})$ and $\zeta_j = \delta(\bar{T}_j, I_j, \zeta_{j-1})$ for $1 \le j \le n$, where I_j is in *SUCCPAT*(q_j). Let the calling pattern for q_j be I_{C_j}. From Lemma 3.2, if q_j succeeds with success pattern I_j', then there exists an I_j such that *\$langle*I_{C_j}, I_j *\$rangle* $\in$ *SUCCPAT*(q_j), and I_j' *\$le* I_j, i.e. the inferred instantiation of any term is above the actual instantiation of that term at runtime with respect to *\$le*. It follows that if the actual calling pattern for p is I_C while the inferred calling pattern for p is $\pi_{\bar{T}_j}(\zeta_{j-1})$, then $\pi_{\bar{T}_j}(\zeta_{j-1}) \in$ *CALLPAT*(p), and I_C *\$le* $\pi_{\bar{T}_j}(\zeta_{j-1})$. Therefore, I_C *\$le* *\$lub* *CALLPAT*(p). *\$always*

4. Calling and Success Patterns over { c, d, e, f }

This section extends the results of the previous section to consider modes over a larger and more interesting domain, {*c, d, e, f*}. The definitions introduced in the previous section do not change. However, once we begin to consider whether or not a variable is free at any point in a program, aliasing assumes a crucial role in soundness considerations. It will turn out, however, that the ideas developed in the previous section can still be applied, with slight modifications, to the richer approximation domain we now consider.

4.1. Aliasing Revisited

When inferring modes over the full approximation domain Δ, care has to be exercised in order not to infer a variable as having instantiation f when its instantiation is, in fact, d or c due to aliasing. This is illustrated by the following example:

```
p(X, Y) :- q(X, Y), r(X), s(Y).
q(Z, Z).
r(a).
s(W).
```

Once execution succeeds past q, the variables X and Y are aliased together. As a result, even though s appears to be called with an uninstantiated argument, the goal r actually instantiates the argument to the goal *s*.

We can distinguish between two kinds of aliasing: *call-aliasing*, where there are repeated variables in a call, and *return-aliasing*, where distinct variables in a call are aliased (in general, share variables) on return. The example above illustrates an instance of return-aliasing. Problems that can arise due to call-aliasing are illustrated by the following example:

p :– $q(X, X)$.
$q(a, Y)$:– $r(Y)$.

Naive $\{c, d, e, f\}$-mode inference obtains the mode *$langle f, f$rangle* for q, but then erroneously infers a mode *$langle f$rangle* for r when in fact the mode of r is *$langle c$rangle*.

As mentioned earlier, ι-patterns are not expressive enough to capture aliasing effects. A possible solution is global analysis to detect possible occurrences of aliasing; however, this can be expensive. Instead, we use a conservative local analysis. Since the analysis is local, i.e. restricted to the ι-patterns of the literals in a clause, it cannot be as thorough as a global analysis of the program. It does, however, guarantee soundness.

It can be seen that in the first aliasing example above, problems arise because aliasing makes the instantiation of the argument to s unsafe; in the second example, the instantiation of the argument to r is unsafe. While the safety of an instantiation in the full approximation domain Δ is undecidable in general, it is possible to give sufficient conditions for safety. Let $vars(T)$ denote the set of variables occurring in a term T. Then, we have:

Lemma 4.1: Consider a clause $\mathrm{q}_0(\bar{T}_0)$:– $\mathrm{q}_1(\bar{T}_1), \ldots, \mathrm{q}_n(\bar{T}_n)$ with calling pattern I_0, and let ζ_j be the ι-state after the literal q_j. Then, the input instantiation of a term $\bar{T}_j \downarrow k$, given by $I_j \downarrow k = \zeta_{j-1}(\bar{T}_j \downarrow k)$, is safe either of the following are satisfied:

(1) $I_j \downarrow k \neq f$; or

(2) There are no indices $k1$, $k2$, $m1$, $m2$, with $k1 < k2 < j$, such that $vars(\bar{T}_{k1} \downarrow m1) \cap vars(\bar{T}_{k2} \downarrow m2) \neq \emptyset$, and $\zeta_{k1}(\bar{T}_{k1} \downarrow m1) \neq c$.

Proof:

(1) In this case, $I_j \downarrow k$ is either c or d. If it is c then it contains no variables, and therefore cannot be affected by aliasing. If it is d then aliasing can only change its instantiation to c. Since c *$le* d, the instantiation is safe.

(2) In order that an instantiation become unsafe, it is necessary that (*i*) aliasing occurs; and (*ii*) an aliased variable is then instantiated. For condition (*i*) to be satisfied, some predecessor q_{k1} to the goal q_j under consideration (with the head of the clause counting as a predecessor

as well) must have had a non-ground output argument $\bar{T}_{k1}{\downarrow}m1$; for condition (*ii*) to be satisfied, there must have been a goal q_{k2} between q_{k1} and q_j, such that for some variable $Z \in vars(\bar{T}_{k1}{\downarrow}m1)$, Z occurs in an input argument to q_{k2}. Therefore, if these conditions are not satisfied, then the instantiation must be safe. *$always*

Since the second condition of the lemma includes the head of the clause as a predecessor literal, the lemma holds for both call- and return-aliasing. However, while this lemma gives sound criteria for inferring safety, it is very conservative: case (2) of the lemma essentially says ''if a literal takes a non-ground input and returns a non-ground output, assume that it can cause aliasing''. It is possible to improve the analysis by considering more extensive examinations of the program for aliasing. Several global analysis algorithms for the detection of aliasing, in different contexts, have been proposed, e.g. see [4, 6, 8]. However, these are fairly elaborate algorithms that tend to be quite expensive. Our emphasis is on practically useful algorithms that are efficient to use, yet reasonably precise. We outline below two algorithms for the detection of aliasing that represent, we feel, a reasonable compromise between precision and speed. With each predicate *p* is associated a bit $alias_p$, which we call its *alias bit*. The idea is to determine by static analysis whether or not a predicate can cause return-aliasing, i.e. alias together distinct variables in a call to it, and accordingly set the value of its alias bit. Algorithm I sets the alias bit of a predicate to **1** if that predicate either has a clause with repeated variables in the head, or can call a predicate with such a clause. Algorithm II, which is more discriminating, requires additionally that a predicate must be able to succeed with non-ground output arguments in order to set its alias bit. The algorithms are outlined in Figure 2.

That each of the algorithms terminates follows from the fact that since a program can have only finitely many predicates, the number of alias bits must be finite, and each alias bit can only go from **0** to **1**, never vice versa. Soundness can be established by showing that if a predicate can alias together distinct variables in a call to it, then its alias bit is set to **1** by either algorithm. It is evident that whenever the alias bit for a predicate is set to **1** by Algotithm II, it is also set to **1** by Algorithm I, so that the soundness of Algorithm II implies that of Algorithm I. The soundness of Algorithm II is given by the following:

Theorem 4.1: Let $\{X_1, ..., X_n\}$ be all the variables of an *n*-tuple of terms $\bar{T}$, such that each of the variables X_i, $1 \le i \le n$, occurs exactly once in $\bar{T}$, and let '$p(\bar{T})$' be a call to an *n*-ary predicate *p* in a program. If the call can succeed with two distinct variables X_j and X_k, $1 \le j, k \le n$, aliased together, then $alias_p = \mathbf{1}$.

Algorithm I:

begin
for each predicate p in the program **do**
if there is a clause for p with repeated variables in the head
then $alias_p$:= **1**
else $alias_p$:= **0**;
repeat
for each predicate p with $alias_p$ = **0 do**
if a clause for p has a literal '$q(\ldots)$' in its body such that $alias_q$ = **1** **then**
$alias_p$:= **1**;
until there is no change in any alias bit;
end.

Algorithm II:

begin
for each predicate p in the program **do**
if there is a clause for p with repeated variables in the head
and $\$exists\ I_C \in CALLPAT(p)$ such that $\$exists\ \$langle I_C, I_S\$rangle \in SUCCPAT(p)$
and for some j, $1 \leq j \leq arity(p)$, $I_S \downarrow j \neq c$
then $alias_p$:= **1**
else $alias_p$:= **0**;
repeat
for each predicate p with $alias_p$ = **0 do**
if a clause for p has a literal '$q(\ldots)$' in its body such that $alias_q$ = **1**
and $\$exists\ I_C \in CALLPAT(p)$ such that $\$exists\ \$langle I_C, I_S\$rangle \in SUCCPAT(p)$
and for some j, $1 \leq j \leq arity(p)$, $I_S \downarrow j \neq c$ **then**
$alias_p$:= **1**;
until there is no change in any alias bit;
end.

Figure 2 : Two Simple Algorithms for Detecting Potential Sources of Aliasing

Proof: By induction on the number of steps N in the deduction.

In the base case, consider $N = 1$. For the call to succeed in one step of deduction, there must have been a unit clause $p(\bar{T}_0)$ which unifies with $p(\bar{T})$. In this case, distinct variables in the call can become aliased together only if there are repeated variables in $\bar{T}_0$, and if the call can succeed with a non-ground argument. In this case, it follows from the definition that there will be a success pattern I_S for p such that *$langle* $\iota(\bar{T})$, I_S*$rangle* is in *SUCCPAT*($p$), and for some j, $1 \le j \le n$, $I_S \downarrow j \ne c$. It follows, from the description of Algorithm II above, that $alias_p$ will be set to **1**.

Assume that the theorem holds for calls involving fewer than k steps of deduction, and consider a call that requires k steps to succeed. Clearly, in order to return aliased variables, the call must have succeeded with at least one non-ground argument. Let the actual arguments at the return from the call be $\bar{T}_{out}$, and let the inferred success pattern be I_S. Since the call succeeds with some non-ground argument, there is a j, $1 \le j \le n$, such that $\iota(\bar{T}_{out})\downarrow j \ne c$. Aliasing and subsequent instantiation of aliased variables can only cause them to become more instantiated than might be evident from a straightforward propagation of instantiation patterns. This implies that $I_S \trianglelefteq \iota(\bar{T}_{out})$, and therefore that $I_S \downarrow j \ne c$.

There are two possibilities, in the inductive case, regarding where the aliasing occurs: if it occurs in the head, this must be due to the occurrence of repeated variables in the head, and in this case it follows immediately from the description of the algorithm that $alias_p$ is set to **1**; if it occurs in a call to a predicate q arising from a literal in the body of the clause, the call to q must have required fewer than k steps of deduction to succeed, and must have succeeded with a non-ground argument. It follows from the induction hypothesis that $alias_q$ = **1**, so that the algorithm sets $alias_p$ to **1** as well. *$always*

The condition for safety can now be improved to the following:

Lemma 4.2: Consider a clause $q_0(\bar{T}_0)$:– $q_1(\bar{T}_1)$, . . ., $q_j(\bar{T}_j)$, . . ., $q_n(\bar{T}_n)$ with calling pattern I_0, and let ζ_j be the ι-state after the literal q_j. In the absence of call-aliasing, the input instantiation of a term $\bar{T}_j \downarrow k$, given by $I_j \downarrow k = \zeta_{j-1}(\bar{T}_j \downarrow k)$, is safe if there are no indices $k1$, $k2$, $m1$, $m2$, with $k1 < k2 < j$, such that

(i) $alias_{q_{k1}}$ = **1**;

(ii) $vars(\bar{T}_{k1} \downarrow m1) \cap vars(\bar{T}_{k2} \downarrow m2) \ne \emptyset$; and

(iii) $\zeta_{k1}(\bar{T}_{k1} \downarrow m1) \ne c$.

Proof: As before, in order for an instantiation to become unsafe it is necessary both that aliasing occur and that an aliased variable become instantiated. Since we assume that there is no call-aliasing, we need concern ourselves only with return-aliasing. For return-aliasing to have occurred, it is necessary that there be some predecessor q_{k1} to the literal q_j under consideration which could have caused aliasing, i.e. whose alias bit $alias_{q_{k1}}$ has value **1**, and which had a non-ground output argument $\bar{T}_{k1}{\downarrow}m1$; for an aliased variable to have then become instantiated, there must have been a goal q_{k2} between q_{k1} and q_j, such that for some variable $Z \in vars(\bar{T}_{k1}{\downarrow}m1)$, Z occurs in an input argument to q_{k2}. Therefore, if these conditions are not satisfied, then the instantiation must be safe. *$always*

This lemma gives us conditions for the safety of instantiations in the absence of call-aliasing. At this point, it is not difficult to see how call-aliasing can be handled: consider the program

p :– $q(X, Y)$, $r(X, Y)$, $s(f(Z), Z)$.
$q(X, X)$.
$r(a, Y)$:– $r1(Y)$.
$s(f(a), X)$:– $s1(X)$.

In the clause defining p, call-aliasing occurs both for r and s: in the case of r, this is because of return-aliasing caused by the earlier literal q, while for s the cause is the repetition of variables in the call. In the former case, the alias bit for q would have been set to **1** by the algorithms above; the latter case can be detected simply by checking for repeated variables in a literal in the source program. The mechanism we propose for handling call-aliasing is simple: with each literal L in the body of a clause ‘‘p(...) :– *Body*’’ is associated a bit, its call-alias bit. This bit is set to **1** if either (*i*) the alias bit of the predicate for any of the literals to its left is set to **1**; or (*ii*) for some literal L' in the program with predicate symbol p, the call-alias bit of L' is **1**; or (*iii*) if there are repeated occurrences of a variable in the literal L; otherwise, it is set to **0**. When determining the ι-state at the point in a clause between the head and the body, i.e. immediately after unification of the arguments in the call with the arguments in the head of the clause, the value of the call-alias bit of the literal from which the call arose is taken into account to see if call-aliasing might have occurred:

Lemma 4.3: Consider a clause $q_0(\bar{T}_0)$:– $q_1(\bar{T}_1), \ldots, q_n(\bar{T}_n)$ with calling pattern I_0, and let the call-alias bit of the literal from which the call arose be *c-alias*. Let ζ_0 be the ι-state at the point in the clause between the head and the body. The instantiation of a term $\bar{T}_0{\downarrow}k$ at this point, given by $\zeta_0(\overline{T}_0 \downarrow k)$, is safe if *c-alias* = **0**.

Proof: By induction on the number of steps N in the deduction, similar to that of Theorem 4.1. *$always*

Calling patterns can be constrained to be safe by replacing the instantiation of any term whose instantiation cannot be inferred to be safe, e.g. from Lemmas 4.1, 4.2 or 4.3, by *d*, which is the top element in the lattice *$langle* Δ, *$le$rangle* . This can be thought of as asserting that we know nothing about the actual instantiation of any term that is not safe. Clearly, instantiations so obtained are guaranteed to be safe. Such calling patterns will be referred to as *safe calling patterns*.

Returning to the example of aliasing given earlier,

p :– *q*(*X*, *Y*), *r*(*X*), *s*(*Y*).
q(*Z*, *Z*).
r(*a*).
s(*W*).

we find that though the variable Y in the call to *s* appears to be free, the predecessor *q* has a non-closed output argument X which is an input to *r*, and further that $alias_q$ = **1** because of the repeated variables in the head of the clause for *q*. This instantiation of Y is therefore not safe, and a safe calling pattern is obtained by replacing it by *d*. Alternatively, Bruynooghe et. al. have recently proposed a strategy for handling return-aliasing with greater precision, by duplicating literals in the body of the clause [5]. Using this strategy, the program above might be transformed to

p :– *q*(*X*, *Y*), *r*(*X*), *q*(*X*, *Y*), *s*(*Y*).
q(*Z*, *Z*).
r(*a*).
s(*W*).

This would enable us to infer the mode *$langle c$rangle* for the predicate *s*, rather than *$langle d$rangle* , as one might by simply replacing all unsafe instantiations by *d*. This strategy could be used in conjunction with that for handling call-aliasing suggested by Lemma 4.3. One potential problem is that uncontrolled duplication of literals could adversely affect the efficiency of the algorithm: this could be ameliorated by only duplicating those literals capable of causing aliasing, i.e. whose alias bits had been set to **1**. Also, this strategy may not work if the program contains metalanguage constructs like *var*/1 and *nonvar*/1.

4.2. Soundness

It is not difficult to show that a mode inference strategy based on the previous section, but constrained to work with safe calling patterns, is sound:

Theorem 4.2: $\{c,d,e,f\}$-Mode inference over safe calling patterns is sound for static programs.

Proof: Similar to Theorem 1. Since inferred calling patterns are constrained to be safe, if I_i is an inferred calling pattern for a literal and I_a an actual calling pattern for that literal at runtime, then I_a *$le* I_i. Therefore, if $CALLPAT^{(safe)}$ is the set of all safe calling patterns inferred for the predicate, then I_a *$le $lub* $CALLPAT^{(safe)}$. *$always*

5. Computing Modes : Implementation Issues

Conceptually, there are two phases to mode inference: computation of the admissible success pattern relation, and computation of the calling pattern set. From the definition of the success pattern relation, it can be seen that there is a direct correspondence between a program clause C and a Horn clause C' defining the admissible success pattern relation defined by C. Let ‘‘st_trans($\overline{T}$,I,S_0,S_1)’’ be the state transition relation denoting $\delta(\overline{T},I,S_0) = S_1$, and let ‘‘project(S,$\overline{T}$,I)’’ denote $\pi_{\overline{T}}(S) = I$. Then, given a clause

$$p(\overline{T}_0) :- q_1(\overline{T}_1), \ldots, q_n(\overline{T}_n), \qquad n \geq 0,$$

the admissible success pattern relation defined by this clause is specified by the clause in Figure 3. By executing this clause it is possible to obtain the admissible success patterns resulting from the corresponding clause in the original program. Note that if processing begins with the user’s query, then for any predicate *p*, the set of first arguments for the corresponding predicate *succ_p* is precisely the set of admissible calling patterns for *p*, so that these can be computed and recorded at the same time (the extension to safe calling patterns is straightforward). We will refer to such a program which, when executed, will compute the calling and success patterns for a given program, as a mode interpreter for that program.

Obtaining the mode interpreter for a program is not difficult. However, a naive implementation tends to be inefficient for three reasons:

(1) Since ultimately a least upper bound is taken when computing the mode of a predicate from its set of calling patterns, most calling patterns do not contribute useful information.

(2) A large number of success patterns are computed, but only a small subset of them are considered when computing calling patterns.

succ_p(I_C, I_S) :–

  st_trans($\overline{T}_0$,I_C,S_{init},S_0),

  project(S_0,$\overline{T}_1$,I_{C1}), succ_q_1(I_{C1}, I_1), st_trans($\overline{T}_1$,I_1,S_0,S_1),

  . . .,

  project(S_{j-1},$\overline{T}_j$,I_{Cj}), succ_q_j(I_{Cj}, I_j), st_trans($\overline{T}_j$,I_j,S_{j-1},S_j),

  . . .,

  project(S_{n-1},$\overline{T}_n$,I_{Cn}), succ_q_n(I_{Cn}, I_n), st_trans($\overline{T}_n$,I_n,S_{n-1},S_n),

  project(S_n,$\overline{T}_0$,I_S).

Figure 3

(3) Two levels of interpretation are involved, since the mode inference essentially involves an abstract interpretation of the program by the compiler, which, in turn, is interpreted by the underlying system.

We will consider how these problems might be better handled, to make for a more efficient mode inference algorithm.

5.1. Eliminating Redundant Success Patterns

Since the mode of a predicate is ultimately computed as the least upper bound of the set of its calling patterns, it suffices to maintain, for any given calling pattern, only the least upper bound of the corresponding admissible success patterns. This serves to control the combinatorial explosion that could otherwise arise. Also, rather than compute the entire admissible success pattern relation beforehand, it is significantly more efficient to combine the computations of calling and success patterns into one phase, so that only relevant success patterns are computed.

5.2. Efficient Computation of Fixpoints

A mode interpreter generated directly via a naive transformation of the user's program can very well loop forever because of circular dependencies set up by recursion. However, since the fixpoints being computed are finite, there are finite computations that will obtain the sets we seek. This problem has been investigated extensively by database researchers (see, for example, [1]).

We compute these fixpoints iteratively, in a bottom-up manner, using an *extension table* [12] to avoid repetitions of the same computation. The essential idea here is to maintain a table of *$langle Call, Return$rangle* pairs, where *Return* is the set of solutions corresponding to the call *Call*, so that these need not be recomputed, but can be returned by looking up the table. The amount of redundant computation may be reduced still further by maintaining the least upper bound of calling and success patterns, and terminating computations that will clearly not improve this bound.

A point to take into account is the tradeoff between the cost of redundant computation and the cost of additional bookeeping to avoid redundant computation, since it may not always be apparent, in advance, which will be predominant in a specific situation. In addition, we have to take into account the cost of generating the mode interpreter, which tends to increase with the complexity of the program being generated.

5.3. Structure of Mode Interpreters

A mode interpreter for a predicate consists of two components: the the ι-pattern propagation component and the extension table component. When a literal is being processed, its calling pattern is sent to the extension table component of the corresponding predicate. This checks the extension table, and if necessary calls its propagation component to compute success patterns. Initially, each predicate has the empty mode. The user has to specify which predicates in a file are exported and what instantiation patterns these exported predicates may be called with. This information is used to start the inference process.

The propagation component contains a clause for each clause of the original program. Its function is to approximate the execution of the corresponding clause in the user's program by propagating the calling pattern through the body and finally projecting the instantiation state on the terms in the head to produce a success pattern. This is essentially an optimized version of the mode interpreter clause in Figure 3, modified so that entire ι-states are not passed around explicitly. Instead, a series of variables is used to maintain the instantiation of each term at different points in the program, so that only the terms involved in a call to a literal need be passed into the corresponding call in the mode interpreter. Since each clause of the user program has to be processed, the order in which they are processed is not important for mode inference. Therefore, for greater efficiency in the processing of recursive programs, the clauses in the mode interpreter are ordered so that those corresponding to facts in the user program precede those corresponding to rules.

Each literal in the body of a clause in the original program corresponds, in general, to a three-literal sequence in the corresponding clause in the mode interpreter: a projection literal (corresponding to the operator π), a call to the extension table manager corresponding to the called literal, and a state-transition literal (corresponding to the operator δ). The projection and state-transition literals are used only to relate the instantiations of compound terms to the instantiations of their subterms, and can therefore be optimized away in calls not having any non-closed compound term as an argument. The extension table manager for a predicate, when called with a calling pattern, records this pattern in its table, evaluates the propagator clauses if necessary, and returns the least upper bound, with respect to *$le*, of the resulting success patterns. An example of the mode interpreter (excluding the extension table manager, details of which may be found in [12], and the various alias bits) for the quicksort predicate, defined below, is given in Figure 4.

```
qsort([ M | L ], R) :–
        part(M,L,U1,U2),
        qsort(U1,V1),
        qsort(U2,V2),
        append(V1,[ M | V2 ],R).
qsort([ ],[ ]).
```

The discussion so far has not concerned itself with clauses that contain disjunctions (via the connective '**;**') or negation. These constructs can be handled simply by preprocessing the clauses to yield clauses of the form already discussed: a clause of the form

```
p :– q, (r ; s), t.
```

is transformed to the clauses

```
p :– q, r, t.
p :– q, s, t.
```

A clause of the form

```
p :– q, not( (r, s) ), t.
```

may be transformed, for mode inference purposes, to

```
p :– q, r, s.
p :– q, t.
```

Given this straightforward transformation for negations, the inferred success pattern for *p* may be overly conservative, since success patterns for the clause ''p :– q, r, s'' will be considered even though in reality, the calls to *r* and *s*, being within a negation, will not affect *p*'s success

```
mode_qsort(1, _, _, [c, c]).
mode_qsort(2, [X1, Y1], ExtTbl, [X2, Y2]) :–
        project(X1, [M1, L1]),
        safe_instance(M1, M1s),
        safe_instance(L1, L1s),
        safe_instance(Y1, Y1s),
      mode_part([M1s, L1s, f, f], ExtTbl, [M2, L2, U1, U2]),
      mode_qsort([U1, f], ExtTbl, [U1a, V1]),
        safe_instance(U2, U2s),
      mode_qsort([U2s, f], ExtTbl, [U2a, V2]),
        safe_instance(M2, M2s),
        safe_instance(V1, V1s),
        safe_instance(V2, V2s),
        safe_instance(Y1s, Y1t),
        st_trans([M1s, V2s], W),
      mode_append([V1s, W, Y1t], ExtTbl, [V1a, W1, Y2]),
        safe_instance(M2s, M2t),
        safe_instance(L2, L2s),
        st_trans([M2t, L2s], X2).
```

Figure 4: Mode Interpreter for qsort/2.

patterns in any way – the creation of this clause is necessary only to ensure that calling patterns for *r* and *s* are obtained correctly. The analysis may be shrapened by observing that if a clause is guaranteed to fail, then the success pattern resulting from it may be taken to be the most instantiated possible, consisting only of *e*'s and denoting the empty set of terms. Then, the clause

p :– q, not((r, s)), t.

may be transformed to

p :– q, r, s, fail.

p :– q, t.

In this case, the presence of *fail* in the first transformed clause can be used to ensure that this clause does not affect *p*'s success patterns.

6. Applications

Knowledge of modes enables a compiler to make various optimizations to the program. We briefly list some of these in this section.

Mode information can be used in a structure-sharing implementation to decrease the space requirements of a program by allocating more variables on the local stack [2, 21]. Another application of mode information is in the use of special-purpose unification routines where appropriate [20]. These have fewer cases to test than the general-purpose routine, and therefore are faster. In implementations that permit delaying of goals, e.g. MU-Prolog (see [16]), mode information may also be used to reduce the number of tests necessary at runtime, thereby improving efficiency.

Mode information is also useful in further analysis of the program. For example, it may be used to infer determinacy and functionality of predicates, which enables earlier reclamation of space on the local stack, insertion of cuts where appropriate to control backtracking, and program transformations that depend on functionality. The reader is referred to [11] for details.

Another application of mode information is in *clause fusion* to reduce the amount of nondeterminism in a predicate. In general, given two clauses with identical heads,

p($\bar{X}$) :– $Body_1$.

and

p($\bar{X}$) :– $Body_2$.

it is possible to merge them to produce the clause

p($\bar{X}$) :– $Body_1$; $Body_2$.

Among the advantages of doing this are that if $Body_1$ fails, then the arguments in the call will not have to be restored from the choice point and unified again with the head of the second clause; if an index is present on the clauses of the predicate, it will be slightly smaller; and finally, if $Body_1$ and $Body_2$ contain literals in common, they may be factored to reduce the amount of redundant computation. In practice, however, it is rarely the case that two clauses for a predicate have identical heads. Mode information can sometimes be used in such cases to transform their heads in a manner that allows fusion to be carried out. The basic idea is to take ''output'' arguments, i.e. those with mode *f*, and move their unification from the head into the body of the clause. This is illustrated by the following example:

Example: Consider the following predicate:

```
part([],_,[],[]).
part([E|L], M, [E|U1], U2) :– E =< M, part(L, M, U1, U2).
part([E|L], M, U1, [E|U2]) :– E  > M, part(L, M, U1, U2).
```

The second and third clauses for the predicate cannot be merged, since the arguments in their heads differ. However, if we know that *part* has the mode *$langle c, c, f, f$rangle* then the clauses can be transformed to produce

```
part([E|L], M, U1a, U2) :– E =< M, U1a = [E|U1], part(L, M, U1, U2).
part([E|L], M, U1, U2a) :– E  > M, U2a = [E|U2], part(L, M, U1, U2).
```

At this point, it is possible to merge the two clauses. Moreover, noticing that the complementary literals ‘E =< M’ and ‘E > M’ imply that the two bodies are mutually exclusive [11], we can generate the transformed predicate defined by

```
part([],_,[],[]).
part([E|L], M, U1a, U2a) :– E =< M →
        (U1a = [E|U1], part(L, M, U1, U2)) ;
        (U2a = [E|U2], part(L, M, U1, U2)).
```

The transformed predicate does not create a choice point for the predicate, since a type test on the first argument suffices to discriminate between the two clauses, and an arithmetic comparison can be used to discriminate between the two disjuncts in the second clause. ●

A transformation system based on the principles illustrated in the example above has been implemented in a prototype compiler for SB-Prolog [9]. The code produced by the compiler for the transformed program of the example above executes more than 30% faster than the original program.

7. Performance

The mode inference procedure turns out to be about an order of magnitude faster with the program generation approach than with interpretation on top of the compiler. Typically, the time taken for mode inference, which includes the generation, execution and clean-up of the mode interpreter, is roughly equal to the time taken to then compile the program. This makes it a practical and useful option during compilation.

The precision of analysis will inevitably depend on the richness of the approximation domain. Even with the very simple approximation domain Δ, the precision of mode inference was quite acceptable. For example, even {*c*, *d*, *e*}-mode inference, over large fragments of our Prolog compiler, typically inferred 60% to 70% of the ‘‘interesting’’ modes (in this case, the *c* modes) correctly.

8. Conclusions

We present a procedure for the automatic inference of modes for Prolog programs and proved its soundness. Our approach differs from previous ones in that (*i*) it gives a sound and efficient treatment of aliasing; (*ii*) uses a notion of state transformations by literals to obtain greater precision in the treatment of unification via predicates such as =/2; and (*iii*) rather than have the compiler interpret the user program, it indicates how to dynamically generate another program which, when executed, yields the modes for the original program. This program uses extension tables to efficiently compute its results bottom-up and guarantee termination. This approach enables us to eliminate an extra level of interpretation between the underlying system and the mode interpreter, yielding a significant performance improvement which makes the inference procedure a practical option in a Prolog compiler.

9. Acknowledgements

Thanks are due to Chris Mellish and Harald Sϕndergaard for many helpful comments on an earlier draft of this paper.

References

1. F. Bancilhon and R. Ramakrishnan, An Amateur’s Introduction to Recursive Query Processing Strategies, MCC Tech. Report No. DB-091-86, Microelectronics and Computer Tehnology Corp., Austin, TX, Mar. 1986.
2. M. Bruynooghe, The Memory Management of PROLOG Implementations, in *Logic Programming*, K. L. Clark and S. Tarnlund (ed.), Academic Press, London, 1982. A.P.I.C. Studies in Data Processing No. 16.
3. M. Bruynooghe, Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs, in *Proc. 1st. Int. Logic Programming Conference*, Marseille, France, 1982.
4. M. Bruynooghe, Compile time Garbage Collection, in *Proc. IFIP Working Conference on Program Transformation and Verification*, Elsevier-North Holland, 1986.

5. M. Bruynooghe, B. Demoen, A. Callebaut and G. Janssens, Abstract Interpretation: Towards the Global Optimization of Prolog Programs, in *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, CA, Sep. 1987.

6. J. Chang, A. M. Despain and D. DeGroot, AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis, in *Digest of Papers, Compcon 85*, IEEE Computer Society, Feb. 1985.

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

8. S. K. Debray, Global Optimization of Logic Programs, Ph.D. Thesis, SUNY at Stony Brook, NY 11794, Aug. 1986.

9. S. K. Debray, The SB-Prolog System, Version 2.3.2: A User Manual, Tech. Rep. 87-15, Department of Computer Science, University of Arizona, Tucson, AZ, Dec. 1987. (Revised March 1988).

10. S. K. Debray, Efficient Dataflow Analysis of Logic Programs, in *Proc. Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, Jan. 1988.

11. S. K. Debray and D. S. Warren, Functional Computations in Logic Programs, *ACM Transactions on Programming Languages and Systems 11,* 3 (July 1989), pp. 451-481.

12. S. W. Dietrich, Extension Tables: Memo Relations in Logic Programming, in *Proc. Fourth IEEE Symposium on Logic Programming,* San Francisco, CA, Sep. 1987, pp. 264-272.

13. H. Mannila and E. Ukkonen, Flow Analysis of Prolog Programs, in *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, CA, Sep. 1987.

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

15. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *J. Logic Programming 2*, 1 (Apr. 1985), 43-66.

16. L. Naish, *Negation and Control in Prolog*, Springer-Verlag, 1986. LNCS vol. 238.

17. R. O'Keefe, Mode Error Diagnosis in Interpreted Code − a Prolog Debugging Aid, Internal Note, Dept. of Artificial Intelligence, University of Edinburgh, 1981.

18. U. S. Reddy, Transformation of Logic Programs into Functional Programs, in *Proc. 1984 Int. Symposium on Logic Programming*, IEEE Computer Society, Atlantic City, New Jersey, Feb. 1984, pp. 187-196.

19. G. Smolka, Making Control and Data Flow in Logic Programs Explicit, in *Proc. 1984 Symposium on LISP and Functional Programming*, Austin, TX, Aug. 1984.

20. P. Van Roy, B. Demoen and Y. D. Willems, Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection and Determinism, in *Proc. TAPSOFT 1987*, Pisa, Italy, Mar. 1987.

21. D. H. D. Warren, Implementing Prolog – Compiling Predicate Logic Programs, Research Reports 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.