Flow Analysis of Dynamic Logic Programs†

Saumya K. Debray

Abstract: Research on flow analysis and optimization of logic programs typically assumes that the programs being analyzed are *static*, i.e. any code that can be executed at runtime is available for analysis at compile time. This assumption may not hold for ‘‘real’’ programs, which can contain dynamic goals of the form *call*(X), where X is a variable at compile time, or where predicates may be modified via features like *assert* and *retract*. In such contexts, a compiler must be able to take the effects of such dynamic constructs into account in order to perform nontrivial flow analyses that can be guaranteed to be sound. This paper outlines how this may be done for certain kinds of dynamic programs. Our techniques allow analysis and optimization techniques that have been developed for static programs to be extended to a large class of ‘‘well-behaved’’ dynamic programs.

Address for correspondence and proofs:

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

Running Title: Flow Analysis of Dynamic Logic Programs

---

† A preliminary version of parts of this paper appeared in *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Fransisco, Sept. 1987.

1. Introduction

There has been a considerable amount of research on static analysis of logic programs (e.g. see [1, 2, 5-7, 9-11]). All of this research, however, has concerned itself with programs that are *static*, i.e. where the entire program is available for inspection at compile time. It has been assumed that programs fragments are not created ''on the fly'' and executed. While such an assumption is adequate for a large class of logic programs, there are many cases where program fragments are created and executed dynamically. There are two ways in which this can happen: a goal may be constructed dynamically and executed, e.g. via literals of the form $call(X)$ or $not(X)$, where X is a variable in the program; or, the program itself may be modified dynamically, e.g. through language features like Prolog's *assert*. Researchers investigating flow analyses of logic programs have typically assumed that programs do not display this sort of dynamic behavior. As a result, analyses that have been proposed to date either fail to be sound for a large class of programs, or fail to give any meaningful information about their runtime behavior.

This seems an undesirable state of affairs, because very often, the effects of dynamic program modifications, e.g. via *assert*, tend to be quite localized. It should be possible, at the very least, to isolate portions of the program that may be affected by dynamic constructs. The remainder of the program can then be analyzed and optimized as before. This paper takes a first step in this direction by outlining how some dynamic logic programs can be analyzed statically to isolate ''well-behaved'' portions, for which static analyses using conventional flow analysis techniques can be guaranteed to be sound even in the presence of dynamic program constructs.

It is assumed that the reader is acquainted with the basic concepts of logic programming. The remainder of this paper is organized as follows: Section 2 discusses preliminary concepts, and introduces some of the terminology used in the paper. Section 3 discusses conditions under which predicates can be guaranteed to be unaffected by dynamic program updates. Section 4 considers a class of programs, called *simple programs,* where the effects of runtime program modifications can be estimated in a relatively straightforward way. Section 5 discusses another class of programs, called *stable programs*, where independence of runtime modifications can be used to characterize the changes possible at runtime. Section 6 outlines how some of the restrictions on programs assumed earlier in the paper may be relaxed. Section 7 concludes with a summary.

2. Preliminaries

2.1. The Language

The language we consider is essentially that of Horn clauses, augmented with three primitives, *assert*/1, *retract*/1, and *call*/1. A predicate definition in such a language consists of a finite multiset (possibly ordered) of clauses. Each clause is a finite multiset (possibly ordered) of literals, which are either atomic goals or negations of atomic goals. Clauses are generally constrained to be definite Horn, i.e. have exactly one positive literal. The positive literal is called the *head* of the clause, and the remaining literals, if any, constitute the *body* of the clause; a clause with only negative literals is referred to as a *goal*. The meaning of each clause is the disjunction of its literals, that of the program being the conjunction of the clauses. We adhere to the syntax of Edinburgh Prolog [3] and write clauses in the form

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

which can be read as ''p *if* q_1 *and* . . . *and* q_n'', where the conjunction of literals in the body of the clause is represented using the connective '**,**'. The names of variables begin with upper case letters, while the names of function and predicate symbols begin with lower case letters. A program consists of a finite set of predicate definitions. We assume throughout this paper that with each program is associated a class of queries that may be asked of it; this essentially specifies what the exported predicates are, and how they may be called. It is assumed that the only calls arising in a program are those resulting from the execution of queries from this class.

It is assumed that the only primitives that can modify a program at runtime are *assert* and *retract*. When a goal

assert(*C*)

is executed, the term that *C* is instantiated to, interpreted as a clause, is added to the clauses of the program. The position of the added clause relative to the other clauses is not specified. When a goal

retract(*C*)

is executed, one of the clauses in the program that matches *C* is deleted from the program. In each case, the argument *C* must be instantiated to a nonvariable term that can be interpreted as a clause, i.e. the predicate symbol of each literal must be defined. Finally, when a goal

call(*G*)

is executed, the term that *G* is instantiated to, interpreted as a goal, is executed.

It is easy to apply our ideas to programs that also contain features such as *not*, *cut*, and metalanguage features such as *var*/1 and *nonvar*/1. However, we specifically exclude programs containing features such as *univ* (Prolog's '=..'/2) and *name*/2. The reason for this restriction is that without it, the static determination of what sorts of clauses might be asserted or retracted becomes difficult. The presence of *name* makes it possible to construct an unbounded number of new constants at runtime. This, together with *univ*, makes it possible to create an unbounded number of different function symbols at runtime which were not present in the program at compile time. As a result, the task of reliable flow analysis becomes difficult. Despite these restrictions, however, the procedure described enables logic program analyzers to handle a larger class of programs, under more realistic assumptions, than most others that have been proposed in the literature. Section 6 of this paper discusses situations under which these restrictions can be relaxed.

For the sake of simplicity, we assume the control regime of Prolog, with its textual ordering on clauses and literals, in the rest of this paper. However, this control strategy is not in any way intrinsic to the analysis procedure outlined here, and the analysis may be carried out for other control strategies as well, as long as that control strategy is specified to the compiler beforehand.

2.2. ''Green'' Asserts

A common use of *assert* is in the recording of lemmas that have been proved. Intuitively, if we can prove G in a program, then adding G to the program database does not really change anything, except to make the proving of G more efficient in the future. We refer to such uses of *assert*, where the meaning of the program is not affected, as ''green'' *assert*s.

The act of adding a clause C to the program via *assert* causes a change in the quantification of variables in C, from existential (before the *assert*) to universal (in the *assert*ed version of C). It is not difficult to show that this does not pose a problem in green asserts. Consider a clause of the form

$$p :- \ldots, call(\ G\ ), \ldots, assert(\ G\ ), \ldots$$

where G is an atom whose principal functor q is a predicate symbol defined in the program (so that q cannot be, for example, ':–'/2). At runtime, suppose that the instance of G that is called is G_1, and that it succeeds with substitution θ, such that $\theta(G_1) \equiv G_2$. If G_2 is ground then it is clear that asserting G" does not affect the success or finite failure sets of the program. Assume G_2 is not ground, and consider any SLD-derivation for it. For any variable v in G_2, and any ground term t, the substitution $\sigma : \{v \rightarrow t\}$ can be systematically applied to each step of the SLD-derivation, yielding an SLD-derivation for $\sigma(G_2)$. Since this argument applies to each variable appearing in G_2, the universal closure of G_2 is true in the program, so asserting it (or any instance of it) does not change the success or finite failure set of the program.

Whether or not a particular literal ''*assert*(G)'' in a program is green or not is clearly undecidable. However, sufficient conditions can be given for greenness: it is easy to see that if a clause is of the form

$$p(\bar{T}\,) :- Lits_1, call(G), Lits_2, assert(G), Lits_3.$$

then the literal ''*assert*(G)'' is green, and the clause is equivalent to

$$p(\bar{T}\,) :- Lits_1, call(G), Lits_2, Lits_3.$$

Green *assert*s can therefore be ignored during static analyses of programs without affecting soundness. To simplify the discussion that follows, the remainder of this paper assumes that programs being analysed are preprocessed, and any *assert*s that are identifiable as green are deleted for the purposes of analysis.

2.3. Dynamic Logic Programs

Most researchers investigating the static analysis of logic programs have assumed that the programs being considered are ''pure''; in some cases, a limited degree of impurity, in the form of operational features like Prolog's *cut* or metalanguage features like *var*/1 and *nonvar*/1, is tolerated. However, as far as we know, all proposals for the analysis of logic programs to date have assumed that the programs are *static*, i.e. that anything that can be executed at runtime is available for analysis at compile time. This assumption is not always satisfied for ''real'' programs, which sometimes construct and execute goals dynamically through *call*/1, or undergo limited amounts of change at runtime through *assert*/1 or *retract*/1. As things stand, since such programs are not static, analysis algorithms proposed in the literature are not applicable, and the prospects for the global optimization of such programs seem fairly limited. In most cases, however, such programs, even though they may be dynamic, tend to be largely unaffected by the dynamic constructs, which tend to be ''well-behaved'' and localized. As an example, consider the following predicate defining a compiler:

compile(*InFile*, *OutFile*) :–
 getclauses(*InFile*, *Clauses0*),
 preprocess(*Clauses0*, *Clauses1*),
 code_gen(*Clauses1*, *AsmList*),
 assemble(*AsmList*, *OutFile*).

where the predicates have their intuitive meanings. The predicate *code_gen*/2 might need to generate new integers, e.g. for numbering variables as it is processing them, or for creating new labels. One possibility is to obtain these integers through a *gensym* utility:

gensym(*N*) :–
 $gensymcount(*M*),
 N is *M* + 1,
 retract(*$gensymcount*(*M*)),
 assert(*$gensymcount*(*N*)).

The only predicate being asserted into is *$gensymcount*/1. Suppose that neither *preprocess*/2 nor any of the predicates it calls ever refers to *$gensymcount*/1. Then, it is clear that whether or not any clauses are asserted for *$gensymcount*/1 cannot affect the success or failure of any call to any of the predicates called by *preprocess*/2.

It is desirable, in such cases, to isolate those portions of the program that are unaffected by dynamic constructs, and proceed with the analysis and optimization of these portions as before. This paper considers some simple classes of ''well-behaved'' dynamic logic programs, and describe procedures for identifying portions of such programs that are not affected by runtime changes. The following terminology is convenient in the discussion that follows:

Definition [occurrence]: A predicate p *occurs positively* in a goal G if either G is of the form ''$q_1, ..., q_k, ..., q_n$'' where the predicate symbol of the literal q_k is p, or if G is of the form ''q_1, ..., not(G'), ..., q_n'' and p occurs negatively in G'.

A predicate p *occurs negatively* in a goal G if G is of the form ''q_1, ..., not(G'), ..., q_n'' and p occurs positively in G'. ●

Note that a predicate may occur both positively and negatively in the same goal.

Definition [dependence]: A predicate p is *s-dependent* on a predicate q in a program if q occurs positively in the body of a clause for p; p is *f-dependent* on q if q occurs negatively in the body of a clause for p. If a predicate p is s-dependent or f-dependent on a predicate q, then p is *dependent* on q. ●

If a predicate p is s-dependent on a predicate q, then the success of p depends on the success of q; if p is f-dependent on q, then the success of p depends on the failure of q. In general, it is possible for a predicate to be both s-dependent and f-dependent on another predicate, as is evident from the following:

p :– . . ., q, . . .
p :– . . ., $not(q)$, . . .

Definition [reachability]: A predicate q is *s-reachable* from a predicate p if either (i) p is s-dependent on q; or (ii) there is a predicate r such that p is s-dependent on r and q is s-reachable from r; or (iii) there is a predicate r such that p is f-dependent on r and q is f-reachable from r.

A predicate q is *f-reachable* from a predicate p in a program if either (i) p is f-dependent on q; or (ii) there is a predicate r such that p is f-dependent on r and q is s-reachable from r; or (iii) there is a predicate r such that p is s-dependent on r and q is f-reachable from r.

If a predicate p is s-reachable or f-reachable from a predicate q, then p is *reachable* from q. ●

Thus, ‘‘reachability’’ can be thought of as a sort of transitive closure of ‘‘dependence’’. If a predicate q is s-reachable from a predicate p, then the success of p depends on the success of q; if q is f-reachable from p, then the success of p depends on the failure of q. As in the case of dependence, a predicate may be both s-reachable and f-reachable from another predicate.

Definition: A predicate is said to be *assertive* if *assert* is reachable from it, and *retractive* if *retract* is reachable from it. If an assertive predicate only asserts facts, it is said to be *unit-assertive*.

A predicate that is assertive or retractive is also said to be *modifying*. ●

If the predicate symbol of a clause being asserted (retracted) is p, then p is said to be *asserted into* (*retracted from*).

Definition: A predicate is *assertable* if it can be asserted into, and *retractable* if it can be retracted from.

A predicate that is assertable or retractable is also said to be *modifiable*. ●

3. Static Predicates

Conceptually, there are two components to the analysis of a dynamic program. In the presence of dynamic constructs, it is necessary to determine, first, what goals can be executed at runtime; and second, what the effects of such executions can be on the program. Accordingly, the analysis consists of two relatively independent phases: the program is first examined to determine which predicates may be called from dynamic goals, asserted into, or retracted from. After this, the program is examined to determine how far the effects of such dynamic constructs may extend. Especially important, in this context, is the estimation of the effects of dynamic program updates via *assert*/1 and *retract*/1. The kind of analysis necessary for the first phase depends on what kinds of clauses may be asserted, and is discussed in much of the remainder of this paper. This section assumes that the sets of assertable and retractable predicates are known, and considers the second phase. First, to specify which predicates might ''see'' the dynamic updates effected by a given predicate, we define the notion of one predicate being ''downstream'' from another. Let $\succ$ denote the reflexive closure of the reachability relation between predicates, i.e. $p \succ q$ if and only if either $q = p$ or q is reachable from p. Then, we have the following:

Definition [downstream]: A predicate q is *downstream* from a predicate p in a program if

(1) there is a clause in the program of the form

$$Head \text{ :– } \ldots, p_1(\ldots), \ldots, q_1(\ldots), \ldots$$

such that $p_1 \succ p$ and $q_1 \succ q$; or

(2) for some predicate r in the program, there are two clauses

$r(\ldots) \text{ :– } \ldots, p_1(\ldots), \ldots$

$\ldots$

$r(\ldots) \text{ :– } \ldots, q_1(\ldots), \ldots$

where the first clause precedes the second in the clause evaluation order, such that $p_1 \succ p$ and $q_1 \succ q$.

•

Intuitively, q is downstream from p if a call to q can arise after a call to p has been executed. A predicate is said to be *static* in a program if it can be guaranteed not to be affected by runtime changes to the program. We have the following result:

Theorem 3.1: A predicate p in a program is static if

(i) p is not downstream from any modifiable predicate; and

(ii) neither p nor any predicate reachable from p is modifiable in the program.

Proof: There are two ways in which runtime modifications can affect a predicate p: (i) the ways in which p can be called may change; and (ii) the ways in which a call to p can succeed may change. In the first case, p must be downstream from a modifiable predicate; in the second case, either p or some predicate reachable from p must be modifiable. Thus, if p is neither downstream from a modifiable predicate, and neither p nor any predicate reachable from p is modifiable, then no runtime modification to the program will ever be ''visible'' to p, i.e. p is static. $\square$

A better estimate of the static predicates in a program may be obtained if more is known about the kinds of program properties we are interested in. In general, *assert*s affect universally quantified statements about properties of execution paths, e.g. predicate modes [5, 10], types [1, 11], etc, while *retract*s affect existentially quantified statements about properties of execution paths, e.g. the success or possible termination of a goal. Statements about both calling properties (properties at the point of calls to predicates, e.g. modes) and success properties (properties that hold at returns from calls, e.g. success types) may be affected for predicates that are downstream from a modifiable predicate. However, if a predicate is not downstream from a modifiable predicate but is modifiable or can reach a modifiable predicate, then only its success properties may be affected by runtime changes to the program.

Thus, in the general case it is necessary to identify only (*i*) predicates that are downstream from a modifiable predicate; and (*ii*) predicates that are modifiable or can reach modifiable predicates. When more is known about the program properties of interest, the analysis may be sharpened further. For example, if only calling properties are sought, then it is necessary only to exclude those predicates that are downstream from modifiable predicates.

The remainder of the paper discusses how the sets of assertable and retractable predicates may be identified. First we consider a class of programs, called *simple programs,* where the arguments to *assert*, *retract*, *call*, etc., are fully determined. Later we relax this requirement and consider a class of programs called *stable programs*, where dynamic updates are required to satisfy an independence criterion.

4. Simple Programs

As mentioned earlier, the essence of our approach is to determine which predicates may be modified in a program, and how far the effects of such modifications might extend. In some systems, e.g. Quintus Prolog [12], compiled predicates that are modifiable have to be declared by the user as ‘‘dynamic’’. This can provide a crude approximation to the sets of assertable and retractable predicates, and has the merit that it involves practically no analysis of the program. However, it can be overly conservative, because not every dynamic predicate need necessarily be both assertable and retractable. As noted in the previous section, it may be possible to ignore the modifiability of some predicates depending on the kinds of analysis that are of interest. However, it is not possible to distinguish between assertable and retractable predicates using only *dynamic* declarations. Moreover, some systems, e.g. SB-Prolog [4], do not require modifiable compiled predicates to be so declared beforehand by the user, and for such systems it is not possible to rely on *dynamic* declarations to estimate the set of modifiable predicates. It is possible that a more fine-grained system of declarations could aid, and possibly replace, the type of analysis described here; however, it seems neither reasonable nor desirable to require programmers to produce and maintain declarations that are both sound and precise, especially for nontrivial programs.

In the simplest case, for every literal ‘‘*assert*(T)’’ and ‘‘*retract*(T)’’ occurring in the program, the predicate symbol of each literal in T (interpreted as a clause) can be determined simply by inspection, without further analysis. Such programs are referred to as *simple*:

Definition [fully determined]: A term t is *fully determined* if

(1) it is of the form $q(\bar{T})$, and q is not :–/2; or

(2) it is of the form '$q_0(\bar{T}_0)$:– $q_1(\bar{T}_1), \ldots, q_n(\bar{T}_n)$', i.e. each of the symbols q_i, $0 \le i \le n$, is determined; and if any of the q_j, $1 \le j \le n$, is *assert*, *retract*, *not* or *call*, then the corresponding argument $\bar{T}_j$ is fully determined.

Definition [simple]: A program is simple if, for every literal '*assert*(T)', '*retract*(T)', '*not*(T)' and '*call*(T)' occurring in the program, T is fully determined. ●

Example: The program

> *p*(*X*, *Y*) :– *assert*((*q*(*Z*) :– *r*(*Z*, *X*))).
> *r*(*V*, *g*(*V*)) :– *retract*(*q*(*V*)).

is simple. However, the program

> *p*(*X*, *Y*) :– *assert*((*q*(*Z*) :– *r*(*Z*, *X*))).
> *r*(*V*, *g*(*V*)) :– *X* = *q*(*V*), *retract*(*X*).

is not, because the argument to *retract* in the clause for *r*/2 is not fully determined. ●

For simple programs, the determination of static predicates is straightforward. Consider a program P. First, the modifiable predicates in P are obtained, as follows: If there is a clause

> p(...) :– . . ., *assert*(q_0(...)), . . .

or

> p(...) :– . . ., *assert*((q_0(...) :– *Body*)), . . .

in P, then q_0 is assertable; and similarly for *retract*. If, for any literal for *assert*, *call* or *not* in P, any literal in the body of its argument has the predicate symbol *assert*, *retract*, *not* or *call*, then its arguments are processed recursively as described above.

Next, the program P is used to compute an augmented program P^*, whose purpose is to allow the estimation of the reachability and ''downstream from'' relations between predicates that can exist at runtime, in the presence of dynamic program changes. P^* is obtained as follows: initially, P^* is the same as P. The following rules are then applied until there is no change to P^*:

(1) If there is a clause C in P^* of the form

> $p(\bar{T})$:– $Lits_1$, *assert*(($q(\bar{U})$:– *Body*)), $Lits_2$

then delete C from P^* and add the clauses

> $p(\bar{T})$:– $Lits_1$, $Lits_2$.
> $q(\bar{U})$:– *Body*.

Let the new clause for q be C_q. Copies of C_q are made as necessary, so that for each clause C'_q for q

(including $C_q' = C_q$), C_q precedes C_q', and C_q' precedes C_q, in the clause evaluation order. This is necessary because we do not know the relative position, within the clauses for q, where the asserted clause may be added. Notice that only two copies of C_q need be added: one that precedes all the other clauses for q, and one that is preceded by all the others. The case $C_q = C_q'$ is included because the literal for *assert* in the clause C may be called more than once at runtime, resulting in multiple instances of C_q being asserted.

(2) If there is a clause C in P^* of the form

$$p(\overline{T}) :- Lits_1, call(CallLits), Lits_2$$

then delete C from P^* and add the clause

$$p(\overline{T}) :- Lits_1, CallLits, Lits_2.$$

(3) If there is a clause C in P^* of the form

$$p(\overline{T}) :- Lits_1, not(NegLits), Lits_2$$

then delete C from P^* and add the clause

$$p(\overline{T}) :- Lits_1, NegLits, Lits_2.$$

That the augmented program can always be computed follows from the fact that the arguments to *assert*, *call* and *not* are fully determined in a simple program. Since the total number of occurrences of *assert*, *call* and *not* decreases by at least one at each application of these rules, the rewriting of P^* is guaranteed to terminate. The reachability relations that can exist between predicates at runtime, in the presence of dynamic updates, is captured by reachability relations computed from P^*:

Proposition 4.1: Let P be a simple program. If p is reachable from q during the execution of P, then p is reachable from q in P^*. If p is downstream from q during the execution of P, then p is downstream from q in P^*. $\square$

It should be emphasized that the augmented program P^* is used only to estimate the reachability and downstream relations that might exist at runtime because of dynamic updates: no dataflow analysis is performed on P^* itself. This accounts for the treatment of *not* in rule (3) above. It also explains why *assert*s are taken into account but *retract*s are not.

Example: Consider the program P, consisting of the clauses

$$p(X, Y) :- s(X, Z), not(assert((q(Z) :- call(not(assert(r(Z, X) :- q(X))))))).$$
$$q(a).$$

The augmented program P^* is computed as follows: let the clauses in P^* at iteration i be written as P_i^*.

Then, we have the following:

P_0*: $p(X, Y)$:– $s(X, Z)$, $assert((q(Z)$:– $call(not(assert(r(Z, X)$:– $q(X)))))).

$q(a)$.

P_1*: $p(X, Y)$:– $s(X, Z)$.

$q(Z)$:– $call(not(assert(r(Z, X)$:– $q(X))))$.

$q(a)$.

$q(Z)$:– $call(not(assert(r(Z, X)$:– $q(X))))$.

P_2*: $p(X, Y)$:– $s(X, Z)$.

$q(Z)$:– $not(assert(r(Z, X)$:– $q(X)))$.

$q(a)$.

$q(Z)$:– $not(assert(r(Z, X)$:– $q(X)))$.

P_3*: $p(X, Y)$:– $s(X, Z)$.

$q(Z)$:– $assert(r(Z, X)$:– $q(X))$.

$q(a)$.

$q(Z)$:– $assert(r(Z, X)$:– $q(X))$.

P_4*: $p(X, Y)$:– $s(X, Z)$.

$q(Z)$.

$q(a)$.

$q(Z)$.

$r(Z, X)$:– $q(X)$.

$r(Z, X)$:– $q(X)$.

P_4* is the final augmented program. •

Once the program has been processed as above, and the sets of assertable and retractable predicates, together with the reachability and downstream relations in the augmented program P* have been determined, the static predicates in the program can be identified in a straightforward way using Theorem 3.1.

Returning to the example from Section 2, consider the predicate

compile(*InFile*, *OutFile*) :–
    getclauses(*InFile*, *Clauses0*),
    preprocess(*Clauses0*, *Clauses1*),
    code_gen(*Clauses1*, *AsmList*),
    assemble(*AsmList*, *OutFile*).

where the only modifiable predicate is *$gensymcount*/1, which is reachable only from *code_gen*/2. The only calls to *assert*/1 and *retract*/1 are from the *gensym* predicate, defined as

gensym(*N*) :–

$\$gensymcount(M)$,
$\quad N$ is $M + 1$,
$\quad retract(\$gensymcount(M))$,
$\quad assert(\$gensymcount(N))$.

It can be seen that the arguments to *assert*/1 and *retract*/1 are fully defined, so that the program is simple. Assume that none of the program predicates reachable from *preprocess*/2 are also reachable from *code_gen*/2 or *assemble*/2. Then, the predicate *preprocess*/2, and all the predicates reachable from it, are static.

5. Stable Programs

In general, programs may not always be simple, and the straightforward treatment described in the previous section may not apply. This section considers the problem of flow analysis for programs where dynamic updates satisfy an independence criterion. Such programs are referred to as ‘‘stable’’. First, the notion of stability is defined and a simple syntactic sufficient condition given for it. This is followed by a discussion of how such programs may be analyzed at compile time. Initially we consider stable programs that are unit-assertive, i.e. assert only facts; this restriction is later relaxed. When considering this class of programs, it is assumed that if a program contains the *read* predicate, then no function symbol in any term read in matches any of the predicate symbols in the program. If the implementation does not provide some sort of module facility, this restriction must be enforced by the user.

5.1. Stability

The task of predicting, at compile time, the behavior of programs that can assert or retract arbitrary clauses can be extremely difficult. It is therefore necessary to make some assumptions regarding the ‘‘well-behavedness’’ of dynamic programs. The issue is whether or not a call to *assert* or *retract* at runtime can modify a program in a way that creates opportunities for further calls to *assert* or *retract*, and thereby further changes to the program, that had been absent earlier: in other words, whether or not a change to a program is dependent on another change. Programs where changes are independent are said to be stable.

Consider a call ‘$p(\bar{X})$’ in a program P_0: if this call returns (with either success or failure), then the set of clauses P_1 comprising the program at the return from the call may be different from the original program P_0, i.e. the call may have modified the program, by adding clauses through *assert* or deleting them through *retract*. Such a change to a program can be described as a pair $\langle \mathbf{add}(A), \mathbf{delete}(D) \rangle$ where A and D are sets of clauses, with $A \cap D = \emptyset$. Such a pair is referred to as a *modification*.

Definition [modifiability]: Let C be a call, and M a modification $\langle \mathbf{add}(A), \mathbf{delete}(D) \rangle$. A program P_0 is $\langle C, M \rangle$*-modifiable* to a program P_1 (written "$P_0 \rightarrow_{C,M} P_1$") if, when the call C is executed in the program P_0, the program that results at the return from the call is P_1, and $P_1 = P_0 \cup A - D$. ●

Independence of program modifications is captured by the notion of stability:

Definition [stability]: A program P_0 is *stable* if, whenever there are calls C_0, C_1, modifications M_0, M_1 and programs P_1, P_2 such that $P_0 \rightarrow_{C_0,M_0} P_1$ and $P_1 \rightarrow_{C_1,M_1} P_2$, there exists a program P_1' such that $P_0 \rightarrow_{C_1,M_1} P_1'$ and $P_1' \rightarrow_{C_0,M_0} P_2$. •

The requirement for program stability may be represented pictorially as in Figure 1. Modifications are independent in a stable program, in the following sense: if a call C_0 in a program P_0 can result in modification M_0 and yield a program P_1, such that another call C_1 in P_1 can result in modification M_1 and yield the program P_2, then the call C_1 in P_0 would also result in modification M_1; and if the resulting program were P_1', then the call C_0 in P_1' would still result in modification M_0 and yield the program P_2. In this case, it is clear that the modification M_1 does not depend on the modification M_0.

Example: Consider the program P_0, defined by the clauses

p(*X*, *Y*) :– *r*(*Y*), *assert*((*q*(*X*) :– *retract*(*Y*))).
r(*r*(_)).

The call ‘*p*(*U*, *V*)’, with *U* and *V* uninstantiated, succeeds in P_0 and yields the program P_1:

$$\begin{array}{ccccc} & & P_0 & & \\ & \swarrow^{C_0,\,M_0} & & \searrow^{C_1,\,M_1} & \\ P_1 & & & & P_1' \\ & \searrow_{C_1,\,M_1} & & \swarrow_{C_0,\,M_0} & \\ & & P_2 & & \end{array}$$

Figure 1 : Stability of Programs

$p(X, Y)$:– $r(Y)$, *assert*(($q(X)$:– *retract*(Y))).
$q(V)$:– *retract*($r(W)$).
$r(r(\_))$.

Now consider the call '$q(0)$': it succeeds in P_1 and results in the deletion of the clause '$r(r(\_))$'. However, it fails immediately in P_0. The program P_0 is therefore not stable. •

If an assertive predicate in a program can assert a clause whose body contains a literal p(...), and p is a modifying predicate, then the program is said to be *fluid*. An obvious situation where a program is unstable is when it is fluid, as in the above example. Our experience indicates, however, that fluid programs are rare in practice.

There is another situation where a modification to the program at runtime can open up avenues for further changes to the program. Consider, for example, the following program:

$p(X)$:– *assert*($r(0, X)$).
$q(X)$:– $r(X, Y)$, *assert*($q(X)$), *assert*(Y).

This program is certainly not fluid, and none of the calls to *assert* are especially intimidating. Initially, calls to $q/1$ fail because there are no clauses for $r/1$. However, as soon as $p/1$ is called, a clause is asserted for $r/1$. Subsequent calls to $q/1$ may now succeed, resulting in further changes to the program. In this case, the problem arises because there is an assertable predicate r that is s-reachable from a modifying predicate q. An analogous situation may arise involving *retract* and negation, as the following program illustrates:

$p(X)$:– *retract*($r(0)$).
$q(X)$:– *not*($r(0)$), *assert*($q(X)$), *assert*($s(f(X))$).
$r(0)$.

Initially, calls to $q/1$ fail because of the negated goal in the body of its clause. However, as soon as $p/1$ is called, the clause for $r/1$ is retracted. Subsequent calls to $q/1$ may now succeed, resulting in further changes to the program. In this case, the problem arises because there is a retractable predicate r that is f-reachable from a modifying predicate q.

If the criterion for the well-behavedness of programs is that runtime program modifications be independent, i.e. that no runtime modification should open up new avenues for further changes to the program via *assert* or *retract*, then it suffices to exclude the above cases:

Theorem 5.1: A program is stable if

(i) it is not fluid;

(ii) there is no assertable predicate in the program that is s-reachable from a modifying predicate; and

(iii) there is no retractable predicate in the program that is f-reachable from a modifying predicate.

Proof: We sketch the outline for the proof. There are basically two ways in which program modifications can fail to be independent: (i) an asserted clause can contain, in its body, a literal that, when called, eventually calls *assert* or *retract*; and (*ii*) the asserted or retracted clause permits another call '$p(\overline{T})$' to succeed that would have failed otherwise, and that then results in a call to *assert* or *retract*. In the first case, the program is fluid. In the second case, let the predicate asserted into or retracted from be q. Then, there are two possibilities: the call '$p(\overline{T})$' can succeed after the modification to the program either because a call to q now succeeds that had failed earlier, or because a call to q now fails that had succeeded earlier. In the first case, it must be that q is s-reachable from p and was asserted into, while in the second case it must be that q is f-reachable from p and was retracted from (assuming the usual finite failure semantics for negation). Thus, if these cases are excluded then the program must be stable. $\square$

This theorem can be strengthened somewhat, since from the argument above, if an assertable predicate p in a program is s-reachable from a modifying predicate q, then in order that the program be unstable, it is necessary that *assert* or *retract* be downstream from p, and similarly for the retractable case. For example, the program

$p(X)$:– *assert*($r(0)$).
$q(X)$:– $r(X)$, *assert*($q(X)$), *assert*($r(f(X))$).

is unstable, but the program

$p(X)$:– *assert*($r(0)$).
$q(X)$:– *assert*($q(X)$), *assert*($r(f(X))$), $r(X)$.

is not.

It should be pointed out that programs can be stable but not simple (if the argument of an *assert* or *retract* is not fully determined). Similarly, programs can be simple but not stable (e.g. if it is fluid). Thus, the two classes of programs are not directly comparable.

The reason for considering stable programs is that, when analyzing the program to estimate the effects of dynamic program modifications through *assert* and *retract*, it is necessary to guarantee that every execution path that can exist at runtime has been taken into account during analysis. This can be difficult if a runtime modification can open up execution paths that can then cause further modifications that would not have been possible earlier, i.e. if the program is not stable. For example, if a predicate is undefined in a program at the time of analysis, flow information cannot be usefully propagated across literals for it. In order that the analysis be sound, it is necessary to guarantee, therefore, that execution cannot succeed past such literals at runtime either, e.g. by having the predicate become defined via *assert*. It is for this reason that we impose additional restrictions on the reachability relation between predicates and require programs to be stable.

5.2. Analysis of Unit-Assertive Stable Programs

We first restrict our attention to unit-assertive stable programs, i.e. stable programs that assert only facts (this restriction is relaxed later). A unit-assertive program cannot be fluid, so it suffices to enforce the second and third constraints from Theorem 5.1. To simplify the discussion that follows, we also assume that if the program contains literals for *call*/1 and *not*/1, then the arguments to such literals are fully determined. This restriction is relaxed in Section 6.

5.2.1. The Analysis Procedure

Given the restriction that arguments of *call*/1 and *not* are fully determined, it is easy to determine which predicates in the program are modifying. The analysis proceeds as follows: first, the sets of terms each *assert* and *retract* in the program can be called with are determined. From this, the sets of modifiable predicates are determined, and also whether or not the program is unit-assertive. The stability of the program is ascertained by determining the reachability of assertable and retractable predicates from modifying predicates. Finally, reachability and downstream relations are used to identify the static predicates in the program.

The key to the analysis lies in being able to obtain a sound approximation to the set of terms that may be asserted or retracted when the program under analysis is executed. In other words, it is necessary to obtain the *calling types* of the primitives *assert* and *retract* in that program, where the calling type of a predicate describes the terms it can be called with. For simplicity of exposition, a very simple (and crude) algorithm for the inference of calling types is outlined below; more sophisticated and precise type inference algorithms can be used to improve the precision of the analysis.

The analysis tries to determine the principal functors of terms that can be asserted or retracted. Since only unit-assertive programs are being considered, this gives the sets of assertable and retractable predicates. To this end, we define the notion of functor sets:

Definition [functor set]: Let Φ be the set of function symbols appearing in a program, then the *functor set* $FS_\Phi(t)$ of a term t is defined as follows:

(1) if t is a variable, then $FS_\Phi(t) = \Phi$;

(2) if t is a term of the form (t_1, t_2), then $FS_\Phi(t) = FS_\Phi(t_1) \cup FS_\Phi(t_2)$;

(3) if t is a term of the form $not(t_1)$ or $call(t_1)$, then $FS_\Phi(t) = FS_\Phi(t_1)$;

(4) otherwise, t must be a nonvariable term whose principal functor is not '**,**'/2 *not*/1, or *call*/1, and $FS_\Phi(t) = \{functor(t)\}$.

The function $functor(t)$ yields the principal functor of t if t is a nonvariable term, and is undefined if t is a variable. The intent, in clause (2) of the definition, is to represent the body of a clause by the set of principal functors of the literals in it, where for simplicity we only consider the connective '**,**' (the idea extends in a straightforward way to other connectives). Clauses (2) and (3) of this definition are not really necessary for our purposes at this point, but will be useful when we extend the analysis scheme to consider non-unit-assertive programs later. For any given program, the set of function symbols Φ is finite. The set of all functor sets $\mathbf{FS}_\Phi$ for a program with function symbols Φ, which is just the powerset 2^Φ, is therefore also finite. Define the ordering $\sqsubseteq$ on functor sets as follows: given S_1 and S_2 in $\mathbf{FS}_\Phi$, $S_1 \sqsubseteq S_2$ if and only if $S_1 \subseteq S_2$. $\mathbf{FS}_\Phi$ forms a complete lattice under $\sqsubseteq$ with $\emptyset$ and Φ as the bottom and top elements respectively. The meet operation on functor sets, denoted by $\sqcap$, corresponds to set intersection. The ordering $\sqsubseteq$ on $\mathbf{FS}_\Phi$ extends elementwise to tuples of functor sets.

For the sake of simplicity in the description of the algorithm, we assume that each clause in the program is transformed to a normal form where each argument of each literal (except for those whose predicate symbol is =/2) is a distinct variable, and explicit unifications have been introduced via =/2. For example, the clause

$p(X, f(X, g(Y))) :- q(h(Z), X), r(f(X, Z), Y, Y).$

would be transformed to the normal form representation

$p(X, V1) :-$
$\quad V1 = f(X, g(Y)), V2 = h(Z), q(V2, X), V3 = f(X, Z), V4 = Y, r(V3, V4, Y).$

With every point in a clause (i.e. point between two literals) is associated an *abstract state* A, which maps the variables of the clause to elements of $\mathbf{FS}_\Phi$. The functor set of a variable V at any point in a clause is given by $A(V)$, where A is the abstract state at that point. The mapping A extends to arbitrary terms and tuples of terms, as follows: if t is a nonvariable term, then $A(t) = FS_\Phi(t)$; if t is a tuple $\langle t_1, \ldots, t_n \rangle$ then $A(t) = \langle A(t_1), \ldots, A(t_n) \rangle$.

The tuple of functor sets for the arguments of a literal at the point of a call is referred to as the *calling pattern* for that literal, while the tuple of functor sets at the return from that call are referred to as a *success pattern* for that literal. The set of calling patterns of a predicate p in a program is the set of calling patterns for all literals with predicate symbol p over all possible executions of that program; the set of success patterns for a predicate is defined similarly. Let $\downarrow$ be the selection operator on tuples: $\langle x_1, \ldots, x_n \rangle \downarrow k = x_k$ if $1 \le k \le n$, and is undefined otherwise. Then, given an n-tuple of distinct variables $\bar{V}$, an n-tuple of functor sets $\bar{\tau}$ and an abstract state A, the *updated abstract state* $A[\bar{V} \leftarrow \bar{\tau}]$ is defined to be the following:

$$A[\bar{V} \leftarrow \bar{\tau}](v) = \begin{cases} \bar{\tau} \downarrow k & \text{if } v = \bar{V} \downarrow k,\ 1 \le k \le n \\ A(v) & \textit{otherwise} \end{cases}$$

We first define the treatment of unification, via the predicate '='/2, in the analysis. Consider a literal

$$T_1 = T_2$$

in a clause, and let A be the abstract state just before it. Suppose a variable X occurs in either term (possibly both). If X is in fact the term T_1 (the case where X is T_2 is symmetric), then after unification its functor set is given by

$$A(T_1) \sqcap A(T_2).$$

On the other hand, if X occurs as a proper subterm of T_1, then since functor sets contain information only about the possible principal functors for a term, nothing can be said about what X might become instantiated to due to this unification; it is obvious, however, that it is still safe to give the functor set of X after unification as $A(X)$ in this case. This can be summarized by defining a function *a_unify* that, given a abstract state and two terms to be unified via '='/2, returns the abstract state describing the functor sets of variables resulting from the unification:

Definition: Given terms T_1 and T_2 to be unified in an abstract state A, the resulting abstract state $A' = a\_unify(A, T_1, T_2)$ is defined as follows: for each variable v that A is defined on,

$$A'(v) = \textbf{if } (v \equiv T_1 \textbf{ or } v \equiv T_2) \textbf{ then } A(T_1) \sqcap A(T_2) \textbf{ else } A(v).$$

●

Given a class of queries that the user may ask of a program, not all the different calling patterns that are possible for a predicate are in fact encountered during computations. During static analysis, therefore, not all calling patterns for a predicate are ''admissible''. Similarly, given a calling pattern for a predicate, only certain success patterns actually correspond to computations for that predicate starting with a call described by that calling pattern. With each n-ary predicate p in a program, therefore, is associated a set $\text{CALLPAT}(p) \subseteq \mathbf{FS}_\Phi^n$, the set of *admissible calling patterns*, and a relation $\text{SUCCPAT}(p) \subseteq \mathbf{FS}_\Phi^n \times \mathbf{FS}_\Phi^n$, associating calling patterns with *admissible success patterns*. Define the *initial abstract state* of a clause to be the abstract state that maps each variable in the clause to the top element Φ of the lattice of functor sets. The admissible calling and success pattern sets are defined to be the least sets satisfying the following:

- If p is an exported predicate and I is a calling pattern for p in the class of queries specified by the user, then I is in $\text{CALLPAT}(p)$.
- Let q_0 be a predicate in the program, $I_c \in \text{CALLPAT}(q_0)$, and let there be a clause in the program of the form

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

  Let the abstract state at the point immediately after the literal $q_k(\bar{X}_k)$, $0 \le k \le n$, be A_k, and let A_{init} be the initial abstract state of the clause. Then, $A_0 = A_{init}[\bar{X}_0 \leftarrow I_c]$. For $1 \le k \le n$, if the predicate symbol q_k is not '='/2, then the calling pattern for that literal is $cp_k = A_{k-1}(\bar{X}_k)$, and cp_k is in $\text{CALLPAT}(q_k)$; and if $\langle cp_k, sp_k \rangle$ is in $\text{SUCCPAT}(q_k)$, then the abstract state just after that literal is

  $$A_k = A_{k-1}[\bar{X}_k \leftarrow sp_k].$$

  If $q_k(\bar{T}_k)$ is '$T_1 = T_2$' then the abstract state after the unification is given by

  $$A_k = a\_unify(A_{k-1}, T_1, T_2).$$

  The success pattern for the clause is given by $I_s = A_n(\bar{X}_0)$, and $\langle I_c, I_s \rangle$ is in $\text{SUCCPAT}(q_0)$. ●

The analysis begins with the calling patterns specified by the user for the exported predicates. Given an admissible calling pattern for a predicate, abstract states are propagated across each clause for that predicate as described above: first, the abstract state resulting from unification of the arguments in the call with those in the head of the clause is determined. This is used to determine the calling pattern for the first literal in the body. This predicate is processed similarly, and a success pattern corresponding to its calling pattern is determined. This is used to update the previous abstract state and obtain the abstract state immediately after that literal, whence the calling pattern for the second literal is determined, and so on. Since arguments to *call*/1 and *not*/1 are assumed to be fully determined, they can be processed in the obvious way without difficulty. When all the literals in the body have been processed the success pattern for that clause is obtained by determining the instantiation of the arguments in the head in the abstract state after the last literal in the body. This is repeated until no new calling or success patterns can be obtained for any predicate, at which point the analysis terminates.

In order to avoid repeatedly computing the success patterns of a predicate for a given calling pattern, an *extension table* can be used [8, 13]. This is a memo structure that maintains, for each predicate, a set of pairs $\langle Call, RetVals \rangle$ where *Call* is a tuple of arguments in a call and *RetVals* is a list of solutions that have been found for that (or a more general) call to that predicate. At the time of a call, the extension table is first consulted to see if any solutions have already been computed for it: if any such solutions are found, these are returned directly instead of repeating the computation. If the extension table indicates that the call has been made earlier but no solutions have been returned, then the second call is suspended until solutions are returned for the first one. This can in many cases yield solutions where a more naive evaluation strategy such as Prolog's would have looped. The extension table idea can be modified in a straightforward way to deal with calling and success patterns rather than actual calls and returns. In this way, once a success pattern has been computed for a given calling pattern for a predicate, success patterns for future invocations of that predicate with the same calling pattern can be obtained in $O(1)$ time on the average by hashing on the calling pattern. That the sets of calling and success patterns for the predicates in the program can be computed in finite time follows from the fact that both the program and the set of functor sets $\mathbf{FS}_{\Phi}$ is finite, which implies that the space of possible calling and success patterns for any predicate is also finite, and the monotonicity of the functions used in the analysis. Since each element of $\mathbf{FS}_{\Phi}$ is closed under instantiation, aliasing does not pose a problem, and for most programs the time complexity of the algorithm is $O(N)$ for a program of size N [6]. The reader is also referred to [5], which considers the propagation of abstract states in more detail, and in addition discusses several issues relating to efficiency of inference for a related algorithm for mode analysis.

The following establishes the soundness of the analysis described above:

Theorem 5.2: Consider a program with function symbols Φ, no occurrences of *name* or *univ*, and where the arguments to *call*/1 and *not*/1 are fully determined. For any predicate p in the program,

(1) if p can be called with arguments $\langle t_1, ..., t_n \rangle$, then there is a tuple $\langle S_1, ..., S_n \rangle$ in CALLPAT(p) such that

$$FS_{\Phi}(t_i) \sqsubseteq S_i, \quad 1 \le i \le n.$$

(2) if this call can succeed with its arguments instantiated to $\langle t_1', ..., t_n' \rangle$, then there is a pair $\langle I_c, I_s \rangle$ in SUCCPAT(p) such that

$I_c = \langle S_1, ..., S_n \rangle$ and $FS_\Phi(t_i) \sqsubseteq S_i$, $1 \le i \le n$; and
$I_s = \langle T_1, ..., T_n \rangle$ and $FS_\Phi(t_i') \sqsubseteq T_i$, $1 \le i \le n$.

Proof: By induction on the number of steps in the computation. $\square$

Corollary [soundness]: Consider a program with function symbols Φ, with no occurrences of *name* or *univ*, and where the arguments to *call*/1 and *not* are fully determined. If C is a clause that can be asserted in the program, then there is a calling pattern $\langle I_c \rangle$ in CALLPAT(*assert*) such that $FS_\Phi(C) \sqsubseteq I_c$, and similarly for *retract*. $\square$

Example: Consider the following normalized program:

$p(X) :- q(X), r(X).$
$q(Y) :- Y = a.$
$q(Y) :- Y = f(b).$
$q(Y) :- Y = g(a, f(a)).$
$r(Z) :- Z = a.$
$r(Z) :- Z = b.$
$r(Z) :- Z = f(\mathrm{c}).$
?– p(W).

The set Φ is $\{a/0, b/0, c/0, f/1, g/2\}$. Starting from the query, the initial calling pattern for $p/1$ is $\langle \Phi \rangle$. The abstract state in the clause for $p/1$ just after the head is therefore $\{X \rightarrow \Phi\}$, and the calling pattern induced for $q/1$ is also $\langle \Phi \rangle$. The reader may verify that the success pattern for $q/1$ is $\langle \{a/0, f/1, g/2\} \rangle$, so that the abstract state in the clause for $p/1$, between the literals for $q/1$ and $r/1$, is $\{X \rightarrow \{a/0, f/1, g/2\}$, and the calling pattern for $r/1$ is $\langle \{a/0, f/1, g/2\} \rangle$. The success pattern for the first clause for $r/1$ for this calling pattern is $\langle \{a/0\} \rangle$, that for the second clause is $\langle \emptyset \rangle$, and for the third clause is $\langle \{f/1\} \rangle$. The success pattern for $r/1$ (and hence for $p/1$) is therefore $\langle \{a/0, f/1\} \rangle$. •

5.2.2. Verifying Unit-Assertivity and Stability

Given a procedure to determine the calling types of predicates, it is a simple matter to determine whether a program is unit-assertive: if the calling type of the predicate *assert* in a program does not contain ':–'/2, the program is unit-assertive. From the definition of calling types, the predicates that are assertable are given by the calling type of *assert*, while those that are retractable are given by the calling type of *retract*.

Once the sets of assertable and retractable predicates in the program have been determined, and the program has been confirmed to be unit-assertive, it is necessary to verify that it is stable. From Section 3, this requires that (*a*) no assertable predicate in the program be s-reachable from any modifying predicate;

and (b) no retractable predicate be f-reachable from any modifying predicate. Since the arguments of *call*/1 and *not*/1 are fully determined, the determination of the modifying predicates in the program is straightforward. It is therefore a simple matter to analyze reachability relationships between predicates and ascertain that the conditions for stability are satisfied.

Example: Consider the (unnormalized) program

$p(X)$:– $q(X, Z)$, $assert(Z)$.
$q(a, g(b))$.
$q(X, h(X, Y))$:– $r(f(X), g(Y))$.
$r(X, Y)$:– $assert(X)$, $retract(Y)$.
$f(a)$.
$f(U)$:– $g(U)$, $h(U, V)$, $f(V)$.
?– $p(a)$.

The set Φ is $\{a/0, b/0, f/1, g/1, h/2\}$. The calling type of $p/1$ is $\langle\{a/1\}\rangle$. The success pattern for the first clause for $q/2$ is $\langle\{a/0\}, \{g/1\}\rangle$, while that for the second clause for $q/2$ is $\langle\Phi, \{h/2\}\rangle$, so that the success type for $q/2$ in the clause for $p/1$ is $\langle\Phi, \{g/1, h/2\}\rangle$, and the calling pattern for *assert* resulting from this is $\langle\{g/1, h/2\}\rangle$. Also, the call to $r/2$ in the second clause for $q/2$ gives a calling pattern of $\langle\{f/1\}\rangle$ for *assert* and $\langle\{g/1\}\rangle$ for *retract*. Thus, the calling type of *assert* is $\{f/1, g/1, h/2\}$, while that of *retract* is $\{g/1\}$. It is evident that the program is unit-assertive, and that it satisfies the conditions for stability. ●

The discussion above assumes, however, that the set of function symbols that appear in the program at compile time are the only ones that need to be considered during the analysis. In the absence of *univ* and *name*, the program will not be able to create new function symbols dynamically. However, it is still possible for new function symbols to be introduced at runtime, that did not appear in the program at compile time, if the program contains the predicate *read*. In this case, it is necessary to assume that no function symbol in any term that is read in will match any predicate symbol in the program. It can be seen that if this condition holds, reachability relations from the predicates exported by the program will not be disturbed. For example, even if a goal of the form

$$\ldots, read(X), assert(X), \ldots$$

is encountered, the asserted predicate will not be any of the predicates in the program, and moreover will not be reachable from any predicate in the program. Any code so asserted will thus never be executed by any of the predicates being analyzed. Ignoring this and performing the analysis as if the *read* were absent will therefore not affect the soundness of the algorithm.

5.3. Analysis of Non-Unit-Assertive Stable Programs

If the calling type of *assert* or *retract* is found to contain the function symbol ‘:–’/2 in the analysis above, the program may assert rules, and hence may not be unit-assertive. In this case, further analysis is necessary to determine the static predicates in the program. For this, a different abstraction for terms, called *extended functor sets*, is considered:

Definition [extended functor sets]: Let Φ be the set of function symbols appearing in a program. Then, the *extended functor set* $EFS_\Phi(t)$ of a term in the program is defined as follows:

- if t is a variable, then $EFS_\Phi(t) = \langle \Phi, \Phi \rangle$;
- if t is a term '*Head* :– *Body*', then $EFS_\Phi(t) = \langle FS_\Phi(Head), FS_\Phi(Body) \rangle$;
- if t is a nonvariable term whose principal functor is not ':–'/2, then $EFS_\Phi(t) = \langle FS_\Phi(t), \emptyset \rangle$. ●

Example: Let t_0 be the term '$p(X, Y)$'. Then $EFS_\Phi(t_0) = \langle \{p/2\}, \emptyset \rangle$.

Let t_1 be the term '$p(X, Z)$:– $q(X, Y)$, $not(r(Y, Z))$'. Then $EFS_\Phi(t_1) = \langle \{p/2\}, \{q/2, r/2\} \rangle$.

Let t_2 be the term '$p(X, Z)$:– $q(X, Y)$, Z'. Then, $EFS_\Phi(t_2) = \langle \{p/2\}, \Phi \rangle$.

Let t_3 be the term 'W :– $q(X, Y)$, $r(Y, Z)$'. Then $EFS_\Phi(t_3) = \langle \Phi, \{q/2, r/2\} \rangle$. ●

Analogously as with functor sets, an ordering $\sqsubseteq$ can be defined on extended functor sets, as follows: given two extended functor sets $S_1 = \langle S_{11}, S_{12} \rangle$ and $S_2 = \langle S_{21}, S_{22} \rangle$, $S_1 \sqsubseteq S_2$ if and only if $S_{11} \subseteq S_{21}$ and $S_{12} \subseteq S_{22}$. The set of extended functor sets for a program, $\mathbf{EFS}_\Phi$, forms a complete lattice under this ordering, with top element $\langle \Phi, \Phi \rangle$ and bottom element $\langle \emptyset, \emptyset \rangle$. The meet operation on extended functor sets, denoted by $\sqcap$, is defined in terms of elementwise set intersection in the obvious way.

The extended functor set of a term t is an abstraction of the interpretation of t as a clause, representing the predicate symbols that can occur in the head and in the body:

Proposition 5.3: For any term t, let θ be a substitution such that $\theta(t)$ is a fully determined term which, when interpreted as a clause, contains no literal for *assert* or *retract* in the body. If $EFS_\Phi(t) = \langle S_0, S_1 \rangle$, then,

(1) if $\theta(t)$ represents a unit clause with predicate symbol p, then $p \in S_0$.

(2) if $\theta(t)$ represents a non-unit clause '$p(\bar{T})$:– $q_1(\bar{T}_1), \ldots, q_n(\bar{T}_n)$', then $p \in S_0$ and $\{q_1, \ldots, q_n\} \subseteq S_1$.

Proof: The proposition follows directly from the definitions above if t is fully determined. Suppose t is not fully determined. We have the following cases:

(1) If t is a variable, then $EFS_\Phi(t) = \langle \Phi, \Phi \rangle$, and the proposition follows trivially.

(2) If t is not a variable, then its principal functor p has been determined. If p is not ':–'/2, then t represents a unit clause with predicate symbol p. In this case, it follows from the definition of EFS_Φ that $S_0 = \{p\}$, so the proposition holds. If p is ':–'/2, then, since t is not fully determined, t is of the form

t_0 :− t_1, . . ., t_n

where one or more of the t_i, $0 \le i \le n$, is a variable. From the definition of FS_Φ, if t_0 is a variable then $S_0 = \Phi$; and if any of the t_j, $1 \le j \le n$, is a variable, then $S_1 = \Phi$. Since Φ is the set of all function symbols appearing in the program, it is easy to see that the proposition holds in each case. ●

5.3.1. The Analysis Procedure

The analysis procedure in this case is very similar to that described earlier for the unit-assertive case, with minor modifications to deal with extended functor sets. An abstract state A at a point in a clause now maps each program variable in that clause to an extended functor set, and extends to arbitrary terms and tuples of terms as follows: if t is a nonvariable term, then $A(t) = EFS_\Phi(t)$; if t is a tuple $\langle t_1, \ldots, t_n \rangle$ then $A(t) = \langle A(t_1), \ldots, A(t_n) \rangle$. Given an abstract state A, an n-tuple of distinct variables $\bar{V}$ and an n-tuple of extended functor sets $\bar{\tau}$, the updated abstract state $A[\,\bar{V} \leftarrow \bar{\tau}\,]$ is defined exactly as before:

$$A[\bar{V} \leftarrow \bar{\tau}](v) = \begin{cases} \bar{\tau} \downarrow k & \text{if } v = \bar{V} \downarrow k,\ 1 \le k \le n \\ \\ A(v) & \textit{otherwise} \end{cases}$$

Reasoning as before, ‘‘abstract unification’’ for extended functor sets, given by the function *ea_unify* (for ‘‘extended *a_unify*’’) is defined as follows:

Definition: Given terms T_1 and T_2 to be unified in an abstract state A, the resulting abstract state $A' =$ $ea\_unify(A, T_1, T_2)$ is defined as follows: for each variable v that A is defined on,

$$A'(v) = \textbf{if } (v \equiv T_1 \textbf{ or } v \equiv T_2) \textbf{ then } A(T_1) \sqcap A(T_2) \textbf{ else } A(v).$$

●

Abstract states are propagated across clauses, and sets of admissible calling and success patterns CALLPAT and SUCCPAT computed, as before: consider a clause

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

At the entry to a clause, the initial abstract state A_{init} maps every variable in that clause to the top element $\langle \Phi, \Phi \rangle$ in the lattice of extended functor sets. Given a calling pattern $\bar{\tau}_0$, the abstract state A_0 at the point in the clause just after unification has succeeded through the head is given by $A_0 = A_{init}[\,\bar{X}_0 \leftarrow \bar{\tau}_0\,]$. Let the abstract state at the program point just before the literal $q_k(\bar{X}_k)$ be A_{k-1}. If the predicate symbol q_k is not ‘=’/2, then the calling pattern for that literal is $cp_k = A_{k-1}(\bar{X}_k)$, and cp_k is in CALLPAT(q_k); and if $\langle cp_k, sp_k \rangle$ is in SUCCPAT(q_k), then the abstract state just after that literal is

$$A_k = A_{k-1}[\,\bar{X}_k \leftarrow sp_k\,].$$

If $q_k(\bar{T}_k)$ is ‘$T_1 = T_2$’ then the abstract state after the unification is given by

$$A_k = ea\_unify(A_{k-1}, T_1, T_2).$$

Let A_n be the abstract state just after the last literal in the body of the clause, q_n: then, the success pattern for the clause, is $A_n(\bar{X}_0)$. The algorithm begins by considering user-specified calling patterns for the predicates exported by the program, and iteratively propagates abstract states across clauses in the program until no new calling and success pattern can be found. As before, since arguments to *call*/1 and *not*/1 are assumed to be fully determined, they can be processed in the obvious way without difficulty.

A soundness result exactly analogous to that for the analysis for the unit-assertive case, given in Theorem 5.2, can be proved by induction on the number of steps in the computation:

Theorem 5.4: Consider a program with function symbols Φ, with no occurrences of *name* or *univ*, and where the arguments to *call*/1 and *not*/1 are fully determined. For any predicate *p* in the program,

(1) if *p* can be called with arguments $\langle t_1, ..., t_n \rangle$, then there is a tuple $\langle S_1, ..., S_n \rangle$ in CALLPAT(p) such that

$EFS_\Phi(t_i) \sqsubseteq S_i, \quad 1 \le i \le n.$

(2) if this call can succeed with its arguments instantiated to $\langle t_1', ..., t_n' \rangle$, then there is a pair $\langle I_c, I_s \rangle$ in SUCCPAT(p) such that

$I_c = \langle S_1, ..., S_n \rangle$ and $EFS_\Phi(t_i) \sqsubseteq S_i$, $1 \le i \le n$; and
$I_s = \langle T_1, ..., T_n \rangle$ and $EFS_\Phi(t_i') \sqsubseteq T_i$, $1 \le i \le n$.

$\square$

Corollary [soundness]: Consider a program with function symbols Φ, with no occurrences of *name* or *univ*, and where the arguments to *call*/1 and *not* are fully determined. If *C* is a clause that can be asserted in the program, then there is a calling pattern $\langle I_c \rangle$ in CALLPAT(*assert*) such that $EFS_\Phi(C) \sqsubseteq I_c$, and similarly for *retract*. $\square$

It is not difficult to see that extended functor sets are closed under substitution, i.e. for any term *t* and substitution θ, if $EFS_\Phi(t) = \langle S_0, S_1 \rangle$ and $EFS_\Phi(\theta(t)) = \langle S_0', S_1' \rangle$, then $S_0' \subseteq S_0$ and $S_1' \subseteq S_1$. It follows, from the results of [6], that for most programs commonly encountered in practice, the analysis can be carried out in time proportional to the size of the program.

5.3.2. Verifying Program Stability

Once the sets of calling patterns for *assert* and *retract* have been determined, the sets of assertable and retractable predicates are first determined as follows: if an extended functor set $\langle S_0, S_1 \rangle$ is in CALLPAT(*assert*), then every predicate symbol *p* in S_0 is an assertable predicate. S_1 is checked to ensure that it does not contain *call*/1, *name*, *univ* or *not*/1, nor *assert*, *retract*, or any modifying predicate: this ensures that the program is not fluid. The treatment of *retract* is analogous.

After this, an augmented program P^* is constructed from the given program P. As in the case for simple programs, the idea is that the reachability and downstream relations in the augmented program cover any such relationships that can occur in the program at runtime because of dynamic updates. Initially, P^* is the same as P. P^* is then augmented as follows: for each extended functor set $\langle S_0, S_1 \rangle$ in CALLPAT(*assert*), let $\hat{S}_1$ be any enumeration of S_1; then, for every p in S_0, and every permutation *Body* of the elements of S_1, the clause

$$p(\overline{X}) :- Body, \hat{S}_1$$

is added to P^*, with copies made as necessary so that this clause precedes all other clauses for p, and is also preceded by all clauses for p (including itself). The reason $\hat{S}_1$ is appended to *Body* is that there may be more than one literal with the same predicate symbol in the body of the asserted clause: this would not be reflected in the extended functor set, but must be taken into account when estimating the reachability and downstream relations that can exist at runtime. It should be emphasized that the augmented program P^* is constructed solely to estimate reachability and downstream relations that can exist at runtime: no dataflow analysis is performed on the augmented program, so the values of the arguments to literals in the clauses added during this augmentation are really immaterial. As before, notice that only two copies of each clause need be added: one preceding every other clause for p, and one preceded by every other clause. However, since all permutations of literals in the body have to be taken into account, the number of clauses that have to be added grows quickly with the size of the extended functor sets.

It is certainly very conservative to consider all permutations of the literals in the body when computing the reachability and downstream relations in the augmented program. The precision of the analysis can be improved significantly by modifying the definition of extended functor sets to maintain sequences of function symbols in the body (notice that in this case, it is necessary to bound the number of times any symbol can appear in these sequences, in order to guarantee termination), rather than unordered sets as we have considered, so that more information regarding the relative order of literals in the bodies of asserted clauses is available. The tradeoff in this case is that the analysis of the program to compute the sets CALLPAT and SUCCPAT now becomes more expensive (though, given the fact that each such sequence is closed under substitution, the asymptotic complexity of the analysis is still linear in the size of the program [6]).

Once the augmented program P^* has been computed, the reachability and downstream relations between predicates can be computed in a straightforward way, and static predicates identified using Theorem 3.1.

The augmented program construction described above is given primarily for expository reasons. Since its only purpose is to extend the reachability and downstream relations between predicates, in practice it would suffice – and be significantly more efficient – to augment the reachability and downstream relations directly, instead of going through the intermediate step of constructing an augmented program.

Example: Consider the program

p(*X*) :– *q*(*X*), *r*(*Y*), *s*(*X*, *Y*, *Z*), *assert*(*Z*).
q(*r*(*X*)).
r(*a*).
r(*b*).
s(*U*, *V*, *W*) :– *W* = ‘:–’(*U*, *V*).
?- *p*(*Z*).

Given that the only exported predicate is *p*, and that the calling pattern for it is $\langle\!\langle \Phi, \Phi \rangle\!\rangle$, the calling pattern for *q* is obtained as $\langle\!\langle \Phi, \Phi \rangle\!\rangle$, and the corresponding success pattern is $\langle\!\langle \{r/1\}, \emptyset \rangle\!\rangle$. The calling pattern for *r* is also $\langle\!\langle \Phi, \Phi \rangle\!\rangle$, and the success pattern is $\langle\!\langle \{a/0, b/0\}, \emptyset \rangle\!\rangle$. The calling pattern for *s* is therefore obtained as

$$\langle\!\langle \{r/1\}, \emptyset \rangle, \langle \{a/0, b/0\}, \emptyset \rangle, \langle \Phi, \Phi \rangle\!\rangle.$$

From the definition of extended functor sets, it follows that the success pattern of *s* is

$$\langle\!\langle \{r/1\}, \emptyset \rangle, \langle \{a/0, b/0\}, \emptyset \rangle, \langle \{r/1\}, \{a/0, b/0\} \rangle\!\rangle.$$

so that the calling pattern of *assert* is obtained as $\langle\!\langle \{r/1\}, \{a/0, b/0\} \rangle\!\rangle$. As the reader may easily verify, this is the only tuple in CALLPAT(*assert*).

The augmented program *P** is now computed as follows: the clauses added are

r(*X*) :– *a*, *b*, *a*, *b*.

and

r(*X*) :– *b*, *a*, *a*, *b*.

It suffices to add two copies of each of these clauses, so that the augmented program is obtained as

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

r(*X*) :– *b*, *a*, *a*, *b*.
r(*X*) :– *a*, *b*, *a*, *b*.
r(*a*).
r(*b*).
r(*X*) :– *a*, *b*, *a*, *b*.
r(*X*) :– *b*, *a*, *a*, *b*.

s(*U*, *V*, *W*) :– *W* = ‘:–’(*U*, *V*).

When the reachability and downstream relations are computed from this augmented program, it can be seen that the only static predicate is *q*/1. ●

6. Relaxing Some Restrictions

This section outlines how several of the restrictions on programs, assumed in previous sections, may be relaxed.

First, during the analysis of stable programs it was assumed that literals for *call*/1 and *not*/1 were fully determined. This restriction is not really essential: if the program contains literals for *call*/1 or *not*/1 whose arguments are not fully determined, it is possible to proceed with the analysis, as before, and obtain the calling patterns for the *call* and *not* literals. The functor sets so obtained gives the sets of predicates that can be called from such literals. A conservative analysis for the program can then be carried out, as follows: first, not knowing anything about the instantiations of the arguments to the predicates called via *call* and *not*, we have to be pessimistic and assume that they can be called with every possible argument. Then, when determining reachability and downstream relations, it is necessary to consider every permutation of the predicates accessed via *call* and *not*, and take into account the possibility of repeated literals, as discussed in Section 5.3.2. This is in fact pertinent even for static programs: researchers investigating static analyses of logic programs have typically assumed either that literals for *call* and *not* do not appear in programs, or that their arguments are fully determined, i.e. that the programs are simple. Clearly, this may not always be the case even if the program does not contain *assert*s or *retract*s. If the program contains literals of the form '$call(X)$', then an analysis such as the one described in this paper is necessary to guarantee that the static analysis is sound. As far as we know, this issue has not been addressed elsewhere in the literature.

Another restriction that can be relaxed is that on the presence of literals for *name* and *univ* in the program. The reason for this restriction was that calls to these predicates could make it possible to construct, dynamically, function symbols that had not been present in the program at compile time, thereby making it very difficult to certify that the functor sets computed statically were sound. However, it is easy to see that this problem cannot arise if *name* and *univ* are guaranteed to be called with the proper modes (i.e., given the usual usage for these predicates, with the first argument always ground). Thus, the presence of literals for *name* and *univ* can be tolerated provided that (i) a mode analysis of the program, ignoring *assert* and *retract*, indicates that *name* and *univ* have modes that guarantee that no new function symbols are constructed dynamically; and (ii) the extended functor set analysis guarantees that *name* and *univ* are not reachable via *call* or *not*, or from any predicate that is not static in the program.

In some cases, a limited amount of static analysis may be carried out even for predicates that are not static by the criteria discussed in this paper. As noted, for example, if we want to know only about certain calling properties of predicates, then it is sufficient to exclude those predicates that are downstream from modifiable predicates. However, consider a predicate p in a program such that p is downstream from a modifiable predicate, but neither p nor any predicate reachable from p is modifiable. Further, assume that none of the predicates that p depends on are depended on by any other predicate. Then, we can analyze the predicates reachable from p under the most conservative assumptions regarding the terms that p may be called with. For example, if we are doing mode inference, then we can assume that nothing is known about the instantiation of any of the arguments for p. This is illustrated by the following example:

Example: Consider mode analysis of the program

$p(X)$:− $assert(q(X))$, $q(Y)$, $r(X, Y)$.
$r(U, V)$:− $s(U, N, W)$, $t(W, V)$.
$s([], 0, [])$.
$s([X \mid L1], f(N), [f(N) \mid L2])$:− $s(L1, N, L2)$.
$t([], [])$.
$t([H \mid L1], [g(H) \mid L2])$:− $t(L1, L2)$.

Assume that the only predicate that depends on *s*/3 or *t*/2 is *r*/2. The predicate *r*/2 is downstream from the modifiable predicate *q*/1, and hence is not static. However, we can assume that nothing is known about the instantiation of the arguments to *r*/2, and still infer that *t*/2 is always called with its first argument ground, and further that *r*/2 succeeds binding its second argument to a ground term. Information about the success pattern of *r*/2 can now be used to improve mode analysis in other parts of the program. ●

7. Conclusions

The focus of research on static analysis and optimization of logic programs has been primarily on static programs, where code is never created and executed ‘‘on the fly’’. It has been felt that if a program uses dynamic constructs, then the program being executed may not be the same as the program analyzed at compile time, and hence that results of static analysis may not be valid at runtime.

While this is true in general, it can be overly conservative. It is often the case that runtime changes are localized to one part of the program, and do not interact with other parts. It is desirable, in such cases, to be able to identify those portions of the program that are unaffected by such changes, so that these portions can be analyzed and optimized using static analysis techniques already developed for static programs. This paper takes a first step in this direction by considering how certain kinds of dynamic programs can be analyzed for static program fragments. The restrictions are intended principally to ensure that runtime modifications to the program are reasonably well-behaved, so that a sound approximation to the kinds of changes that can occur at runtime can be obtained via static analysis. Our approach enables compilers for logic programming languages to apply static optimization techniques to some dynamic programs as well.

The reader should not infer from this that the author endorses or encourages in any way the use of *asserts* and *retracts* in logic programs.

Acknowledgements

The author is grateful to David S. Warren for many very helpful comments on an earlier draft of this paper, and to John C. Peterson for pointing out a technical problem in an earlier version of the paper. Detailed comments by the referees were also very helpful in improving the content and presentation of the paper.

References

1. 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.
2. 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.
3. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
4. 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).
5. S. K. Debray and D. S. Warren, Automatic Mode Inference for Logic Programs, *J. Logic Programming 5*, 3 (Sep. 1988), pp. 207-229.
6. 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.
7. 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.
8. 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.
9. H. Mannila and E. Ukkonen, Flow Analysis of Prolog Programs, in *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, CA, Sep. 1987.
10. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *J. Logic Programming 2*, 1 (Apr. 1985), 43-66.
11. P. Mishra, Towards a theory of types in Prolog, in *Proc. 1984 Int. Symposium on Logic Programming*, IEEE Computer Society, Atlantic City, New Jersey, Feb. 1984, pp. 289-298.
12. *Quintus Prolog Reference Manual*, Quintus Computer Systems, Inc., Mountain View, CA, Apr. 1986.
13. H. Tamaki and T. Sato, OLD-Resolution with Tabulation, in *Proc. 3rd. International Conference on Logic Programming*, London, July 1986, 84-98. Springer-Verlag LNCS vol. 225.