Static Inference of
Modes and Data Dependencies in Logic Programs

Saumya K. Debray
University of Arizona

Abstract: Mode and data dependency analyses find many applications in the generation of efficient executable code for logic programs. For example, mode information can be used to generate specialized unification instructions where permissible; to detect determinacy and functionality of programs; to generate index structures more intelligently; to reduce the amount of runtime tests in systems that support goal suspension; and in the integration of logic and functional languages. Data dependency information can be used for various source-level optimizing transformations, to improve backtracking behavior, and to parallelize logic programs. This paper describes and proves correct an algorithm for the static inference of modes and data dependencies in a program. The algorithm is shown to be quite efficient for programs commonly encountered in practice.

Categories and Subject Descriptors: D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors – *compilers, optimization*; I.2 [**Computing Methodologies**]: Artificial Intelligence; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving – *logic programming*.

General Terms: Languages, algorithms

Additional Keywords and Phrases: Dataflow analysis, static inference, mode, data dependency, Prolog

---

Author's address: Department of Computer Science, University of Arizona, Tucson, AZ 85721

1. Introduction

Two kinds of information that are of particular interest in the generation of efficient code for logic programs are mode information for predicates and data dependency information between literals in a clause. This paper is concerned with the static inference of these properties.

In general, logic programs are not directed, in the sense that there is no concept of ‘‘input’’ and ‘‘output’’ arguments to procedures. An argument may be used as either an input or an output argument, 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 arguments bound (the ‘‘input’’ arguments) and another set unbound (the ‘‘output’’ arguments). Traditionally, this information has been supplied by the programmer using what are called ‘‘mode declarations’’ [26].

Mode information finds many different applications in high performance logic programming systems. It can be used to generate specialized code for unification that is more efficient than the general purpose routines because there are fewer cases to take care of [25, 26]. Mode information is important in detecting deterministic and functional computations and reduce the program’s search effort [9, 17]. It can be used to generate index structures for predicates more intelligently. In systems that support goal suspension based on variable instantiations, e.g. MU-Prolog [19] and Sicstus Prolog [23], mode information can be used to reduce the amount of testing required to determine whether a goal should be suspended. Mode information is also important in integrating logic and functional programming languages [21]. Data dependency information is useful in various optimizing transformations of logic programs [7], in improving the backtracking behavior of programs [3] and parallelizing logic programs [2, 27].

Early work on mode inference via static analysis was done by Mellish [16-18], who used dependencies between variables to propagate information regarding their instantiation. This approach, however, had the drawback that builtin predicates such as ‘=’/2 could not be handled very precisely; moreover, since aliasing effects resulting from unification were not taken into account, the procedure sometimes produced erroneous results [18]. A more syntactic approach to mode inference, based on the simple mode set {*in*, *out*}, was proposed by Reddy in connection with work on transforming logic programs into functional languages [21]. This approach, however, applied only to a restricted class of logic programs and often tended to be very conservative. A more accurate treatment based on global flow analysis was described by Debray and Warren [6]. Since then, mode inference procedures related to this have been described by Bruynooghe et. al. [1, 13] and Mannila and Ukkonen [15]. Bruynooghe et al. discuss mode inference as an abstract interpretation problem akin to type inference, and suggest keeping track of the aliases of a variable using two sets of variables, the ‘‘*sure aliases*’’ and the ‘‘*possible aliases*’’; Mannilla and Ukkonen use a simple mode set that is essentially the same as Reddy’s, but focus on the algorithmic aspects of the analysis.

Static inference of data dependencies for Prolog programs has been investigated by Chang et al. [2, 3], and by Warren et al. [27]. They describe how data dependency information can be used both to parallelize Prolog programs, and also to improve its backtracking behavior without incurring significant runtime overhead. Data dependency analysis has also been investigated by Mannilla and Ukkonen [15], who discuss computational aspects of the algorithm; however, because their analysis does not take mode information into account when propagating dependencies, it is quite conservative. Deransart and Małuszynski have used the relationship between logic programs and attribute grammars to reason about properties of logic programs, including the modelling of data dependencies and reduction of occur checks [11]: their treatment, like Reddy's, is also based on a mode set {*input*, *output*} containing only two elements. Debray has considered the incremental synthesis of control strategies for parallel logic programs using mode and data dependency information [5]. Other related work includes flow analysis of logic programs to detect situations where cyclic terms can be created during unification, which involves reasoning about the aliasing behavior of programs [20, 22].

The work described in this paper is based on the ideas introduced in [6], but with several important innovations. The most significant of these is the treatment of aliasing introduced here: in retrospect, the *safety criteria* used to handle aliasing in [6], while sound, were both ad hoc and overly conservative. This paper describes a uniform treatment of aliasing that rectifies these problems. The worst case complexity of the algorithm is analyzed based on this treatment. This suggests a variant of the basic algorithm that has a significantly superior worst case performance, at the cost of a slight potential loss in precision for a small class of programs. We also identify a class of programs, called programs of *bounded variety*, which contains most programs encountered in practice, and for which these algorithms have a significantly better worst case performance. A related treatment of aliasing is given by Bruynooghe et al., who propose that information about the aliases of a variable be maintained using sets of variables representing ''*sure aliases*'' and ''*possible aliases*'' [1, 13]. While such a treatment can lead to more precise analyses in principle, the computational implications of having to maintain this additional information are unclear.

The remainder of the paper is organized as follows: Section 2 discusses some preliminary concepts. Section 3 then discusses the flow analysis algorithm, Section 4 proves its soundness and discusses its complexity. Section 5 sketches a variant of the basic algorithm that has a significantly better worst case performance, though it may be less precise for predicates that are used with a number of different modes and alias patterns. Section 6 discusses some applications of mode and data dependency information in the efficient execution of logic programs. Section 7 concludes with a summary. A formalization of the analysis as an abstract interpretation is given in the appendix.

2. Preliminaries

2.1. The Language

The language considered here is essentially that of first order predicate logic. It has countable sets of variables, function symbols and predicate symbols, these sets being mutually disjoint. Each function and predicate symbol is associated with a unique natural number called its *arity*; a (function or predicate) symbol whose arity is n is said to be an n-ary symbol. A 0-ary function symbol is referred to as a *constant*. A *term* is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms, $1 \le i \le n$. An *atom* is of the form $p\,(t_1, ..., t_n)$, where p is an n-ary predicate symbol and the t_i are terms, $1 \le i \le n$.

A clause in a logic program is a finite set (possibly ordered) of literals, which are either atoms or negations of atoms. A clause is called definite if it has 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*. A predicate definition is assumed to consist of a finite set (possibly ordered) of definite clauses, though the handling of other constructs such as negation is considered later. A logic program consists of a finite set of predicate definitions. The meaning of each clause is the universal closure of the disjunction of its literals; that of the program is the conjunction of its clauses. For the purposes of analysis, it is assumed that we are given a module of the form $\langle P, \text{EXPORTS}(P)\rangle$, where P is a set of predicate definitions, and EXPORTS(P) specifies the predicates in P that are exported, i.e. that may be called from the outside. We adhere to the syntax of Edinburgh Prolog and write clauses in the form

$$\text{p} :- \text{q}_1, \ldots, \text{q}_n.$$

which can be read as ''p *if* q_1 *and* . . . *and* q_n''. The names of variables are written starting with upper case letters, while predicate and function symbols are written starting with lower case letters. In addition, a list with head H and tail Tl is written $[H|Tl]$, while the empty list is written [ ].

While the meaning of logic programs is usually given declaratively in terms of the model theory of first order logic, such programs can also be understood procedurally. In this view, each predicate is a procedure defined by its clauses. Each clause provides an alternate definition of the procedure body. The terms in the head of the clause correspond to the formal parameters, and each literal in the body of the clause corresponds to a procedure call. Parameter passing in such procedure calls is via a generalized pattern matching procedure called *unification*. Briefly, two terms t_1 and t_2 are unifiable if there is some substitution θ of terms for the variables occurring in t_1 and t_2 such that $\theta(t_1) = \theta(t_2)$. Such a substitution is called a *unifier*. For example, the terms $f(X, g(X, Y))$ and $f(a, Z)$ are unifiable with the unifier $\{X \to a, Z \to g(a, Y)\}$. Usually the most general unifier, i.e. one that does not make any unnecessary substitutions, is used. It is a fundamental result of logic programming that if two terms are unifiable, then they have a most general unifier that is unique upto variable renaming. The result of unifying two terms is the term produced by applying their most general unifier to them. If the terms under consideration are not unifiable, then unification is said to *fail*.

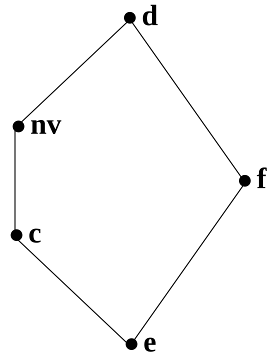
Implementations of logic programming languages typically impose some order of evaluation on the clauses defining a predicate, and the literals within a clause. For example, the execution of a Prolog program follows the textual order of clauses and literals. Execution begins with a goal, or query, from the user, which is a sequence of literals processed from left to right. The processing of a literal proceeds as follows: the clauses for its predicate are tried, in order, until one is found whose head unifies with the literal. If there are any remaining clauses for that predicate whose heads might unify with that literal, a backtrack point is created to remember this. After this, the literals in the body of the clause are executed in their textual left to right order. If unification fails at any point, execution backtracks to the most recent backtrack point: any variables instantiated after the creation of that backtrack point have their instantiations undone, and then the next clause is tried. This process continues recursively until either all the literals have been processed completely, in which case execution is said to succeed, or when no alternatives are left to try, in which case it is said to fail.

The remainder of the paper assumes Prolog's control strategy, with its textual ordering of clauses and literals. However, the techniques described here are not peculiar to Prolog, and can be adapted to other control strategies in a straightforward manner. It is also assumed that the predicates in the program are static, i.e. all executable code is available for analysis at compile time. Thus, the addition or deletion of clauses at runtime through primitives such as *assert* or *retract* is precluded, as is the dynamic construction of goals to be executed via *call*/1 or *not*/1. The analysis of programs that are not static is significantly more complicated because of the need to estimate the possible effects of dynamic code: this issue is beyond the scope of this paper, but a detailed discussion is given in [10].

2.2. Modes

The mode of a predicate in a logic program is an assertion about which of its arguments are input arguments, and which are output arguments, in any call to that predicate arising from that program. Different researchers have considered different sets of modes, e.g. Edinburgh Prolog allows the user to specify the mode of an argument as either bound ('**+**'), unbound ('**–**') or unknown ('**?**') [26]; others consider only the modes {`ground`, `unknown`} [15, 21]. Neither of these is precise enough for our purposes, so we consider the set of modes Δ = {**c**, **d**, **e**, **f**, **nv**}, where **c** (''closed'') denotes the set of ground terms of the first order language under consideration; **d** (''don't-know'') denotes the set of all terms; **e** denotes the empty set; **f** (''free'') denotes the set of uninstantiated variables; and **nv** the set of nonvariable terms. This set of modes is slightly larger than that considered in [6].

The set Δ forms a complete lattice under inclusion:



The ordering on Δ induced by inclusion will be denoted by *$le*, the corresponding join operation being denoted by *$lub*.

Given any set of terms, it is necessary to specify how to find its *instantiation*, i.e. the element of Δ that best ''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. Notice that ι is a closure operator, so that for any set of terms T, $T \subseteq \iota(T)$.

2.3. Comparing Instantiations of Sets of Terms

During the execution of a logic program, terms become progressively more instantiated. 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 are 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 $\trianglelefteq$, as the natural extension of this:

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

The reader may verify that $\trianglelefteq$, 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 $\trianglelefteq$ is also reflexive, and hence a quasi-order, which is easily extended to a partial order in the usual way: given sets of terms T_1 and T_2, let the relation ~ be

$$T_1 \sim T_2 \iff T_1 \trianglelefteq T_2 \; \$and \; T_2 \trianglelefteq T_1.$$

~ is an equivalence relation, and $\trianglelefteq/\sim$, the quotient of $\trianglelefteq$ modulo ~, is a partial order. The discussion that follows concerns itself only with this partial order. With this understanding, we will abuse notation slightly and write the partial order as $\trianglelefteq$. Since each element of the mode set Δ is closed under unification and variable renaming, it follows that $\trianglelefteq$ is a partial – indeed, a total – order over Δ:

$$\mathbf{f} \trianglelefteq \mathbf{d} \trianglelefteq \mathbf{nv} \trianglelefteq \mathbf{c} \trianglelefteq \mathbf{e}$$

The join operation for this order will be written as ∇.

3. The Analysis Framework

The analysis procedure is based on the principles of abstract interpretation [4]. The essential idea behind the analysis is to maintain information about variable bindings at each point in the program. Given some information about the arguments in a call to a predicate, such information can be propagated across each clause for that predicate to obtain a description of the bindings of the call when it returns. The dataflow information is maintained in *instantiation states*, which are abstract representations that describe the terms that each variable in a clause can be bound to at any given point in the clause, as well as possible aliasing and sharing information between variables at that point.

To reason about a call and its return, it is necessary to propagate information about the arguments in the call, from the caller to the callee at the time of the call, and from the callee to the caller at the time of return. While instantiation states describe the bindings of different variables in a clause, it is not natural to use variable names to pass information between clauses. This is because clauses have their variables renamed before they are used in resolution. To handle this, a somewhat different representation, called an *instantiation pattern*, is used to represent calls and returns. An instantiation pattern for a tuple of terms describes the bindings of different elements in the tuple and possible dependencies between them, but without any reference to variable names. Given an instantiation pattern describing a call, it is necessary to specify how to compute the effects of unifying the arguments of the call with the head of a clause. This is done by a procedure that takes an instantiation state A, an instantiation pattern $\bar{I}$ and a tuple of terms $\bar{t}$, and updates A to produce a new instantiation state that incorporates the possible effects of unifying $\bar{t}$ with any tuple of terms represented by $\bar{I}$. This computation proceeds by first considering only the variables appearing in the tuple $\bar{t}$, and then using information about dependencies between variables to propagate the effects of unification to the other variables in the clause.

For the analysis, it is assumed that the user has specified which predicates may be called from the outside; and for each such exported predicate, instantiation patterns that describe how it may be called. The analysis begins with such ‘‘calling patterns’’ for the exported predicates. For each such predicate and calling pattern, the clauses for the predicate are analyzed by propagating instantiation states across the literals of each clause. This yields instantiation patterns describing how the clause may succeed. In order to propagate instantiation states across these clauses, it becomes necessary to analyze predicates called by the exported predicates, and so on. This is repeated until no new calling or success patterns can be found for any predicate in the program, at which point the analysis terminates.

The remainder of this section is organized as follows: Section 3.1 defines the notion of instantiation states. Section 3.2 discusses instantiation patterns. This is followed, in Section 3.3, by a description of how unification is simulated over the abstract domain. Section 3.4 then describes the flow analysis procedure. Section 3.5 considers two examples in detail. Finally, Section 3.6 sketches how other control constructs encountered in logic programming languages may be handled. A formal treatment of the analysis procedure as an abstract interpretation appears in the appendix.

3.1. Instantiation States

The execution of a logic program induces an association, at each point in a clause, between the variables in that clause and the sets of terms they can be instantiated to at that point.† While such sets of terms can be arbitrarily large, it is necessary to have finitely computable approximations to them for static analysis purposes. This can be done by using elements of the mode set to describe these sets of terms. It turns out that information about dependencies between variables also has to be maintained in order to handle unification correctly in the analysis. The behavior of a program is therefore summarized by specifying, at each program point, a description of the set of terms each variable in that clause can be instantiated to at that point, together with the set of variables it can depend on. Such summaries are called ''instantiation states'', or ι-states for short. The finite set of variable names $\mathbf{V}_C$ appearing in a clause C are referred to as the *program variables* of C. The ι-states of a clause C are defined on the program variables of C.

When a clause is selected for resolution against a goal, its variables are renamed so that it is variable-disjoint with the goal. This means that the variables that are used at runtime are, in general, different from the program variables used at compile time. It therefore becomes necessary to ensure that the static analysis takes into account all such possible renamings. This is done via the notion of σ-activations: a use of clause C in a computation, where the variables of C have been renamed via a renaming substitution σ before resolution, is referred to as a σ-*activation* of C. For any term t, let $vars(t)$ be the set of variables occurring in t. Then, we have the following definition:

Definition: An *instantiation state* A_C at a point in a clause C is a mapping

$$A_C : \mathbf{V}_C \rightarrow \Delta \times 2^{\mathbf{V}_C}$$

satisfying the following: if for any variable x in $\mathbf{V}_C$, $A_C(x) = <\delta, V>$, then for any σ-activation of C in the computation,

(i) if $\sigma(x)$ can be instantiated to a term t at that point, then $t \in \delta$; and

(ii) if for any variable y in $\mathbf{V}_C$, $\sigma(y)$ can be instantiated to a term t' at that point such that $vars(t) \cap vars(t') \neq \emptyset$, then $y \in V$. ●

The notion of ι-states extends to arbitrary terms in a straightforward manner: given an ι-state A and a term t, if t is a constant, then $A(t) = \langle \mathbf{c}, \emptyset \rangle$; and if t is a compound term $f(t_1, \ldots, t_n)$, let $A(t_i) = \langle \delta_i, D_i \rangle$, $1 \leq i \leq n$. Then, $A(t) = \langle \delta, D \rangle$, where $\delta = \mathbf{c}$ if $\delta_i = \mathbf{c}$, $1 \leq i \leq n$, and $\mathbf{nv}$ otherwise; and $D = \bigcup_{i=1}^{n} D_i$.

The domain $\mathbf{V}_C$ of the instantiation states of a clause C is fixed once C has been specified. When there is no scope for confusion, therefore, the subscript C will be dropped from the name of the ι-state. If a variable v maps to a pair $<\delta, V>$ in an ι-state A, then δ is said to be the *instantiation* of v in A, written

† This is a straightforward abstraction of denotational semantics of such languages, e.g. see [8, 14].

$inst(A(v))$, while V is its *dependency set*, written $deps(A(v))$. The instantiation component δ acts as a constraint on the terms a variable can be instantiated to at runtime, and the dependency set V acts as a further constraint by allowing us to consider only those terms in δ that depend only on variables in V. Since variables in a clause C are uninstantiated before its head has been unified with the call, the corresponding ''initial ι-state'' for C, where each variable v in $\mathbf{V}_C$ is mapped to the pair <**f**, {v}>, is denoted by A_C^{init}.

Example 1: Consider the program:

p(X) :– q(X, Y), r(Y).
q(Z, Z).
r(a).

Assume that p is called with an uninstantiated argument. If the ι-state at the program point between the literals q and r is A, then A(X) = <**f**, {X, Y}>, indicating that X is still uninstantiated at that point, but may share a variable subterm with Y. Similarly, A(Y) = <**f**, {X, Y}>. ●

3.2. Representing Calls and Returns

When describing how a predicate may be called, or how a call may return, it is necessary to specify not only the instantiation of each argument, but also any sharing of variables between different arguments. Since variables in a clause are renamed before its head is unified with a call, variable names cannot be used in a natural way to propagate sharing information across calls and returns. Instead,a symbolic representation called instantiation patterns, or ι-patterns, are used. These are defined as follows:

Definition: Given an ι-state A and an n-tuple of terms $\bar{t} = <t_1, ..., t_n>$, let $A(t_i) = <\delta_i, V_i>$ for $1 \le i \le n$. Then, the *instantiation pattern* (ι-pattern for short) of $\bar{t}$ induced by A is

$$i\_pat(\bar{t}, A) = < <\delta_1, S_1>, ..., <\delta_n, S_n> >$$

where $S_i = \{ k \mid V_i \cap V_k \ne \emptyset \}$. ●

Example 2: Consider a call $q(X, f(X), h(X, Y), Z)$ in an ι-state A:

$A = \{X \rightarrow \langle \mathbf{f}, \{X\}\rangle, Y \rightarrow \langle \mathbf{f}, \{Y, Z\}\rangle, Z \rightarrow \langle \mathbf{nv}, \{Y, Z\}\rangle\}$.

This call is represented by the ι-pattern

$i\_pat(\langle X, f(X), h(X, Y), Z\rangle, A) = < <\mathbf{f}, \{1,2,3\}>, <\mathbf{nv}, \{1,2,3\}>, <\mathbf{nv}, \{1,2,3,4\}>, <\mathbf{nv}, \{3,4\}> >$.

This indicates that the first argument of the call is a free variable that also shares variables with the second and third arguments; the second argument is a non-variable term that shares variables with the first and third arguments; and so on. ●

For any n, instantiation patterns of length n can be ordered elementwise by inclusion in the obvious way. The least upper bound of two ι-patterns is defined similarly, so that

$<<\delta_{11}, S_{11}>, ..., <\delta_{1n}, S_{1n}>> \mathit{\$lub} <<\delta_{21}, S_{21}>, ..., <\delta_{2n}, S_{2n}>> = <<\delta_{11} \mathit{\$lub}\ \delta_{21}, S_{11} \cup S_{21}>, ..., <\delta_{1n} \mathit{\$lub}\ \delta_{2n}, S_{1n} \cup S_2$

The $k^{th.}$ element $<\delta_k, S_k>$ of an ι-pattern $I = < <\delta_1, S_1>, ..., <\delta_n, S_n> >$, is denoted by $I[k]$. The *share set* of $I[k]$, $1 \le k \le n$, written *share*($I[k]$), is the set S_k. The *instantiation* of the $k^{th.}$ element, written *inst*($I[k]$) where no confusion can arise, is δ_k. Thus, given a tuple of terms $\bar{t} = <t_1, ..., t_n>$ in an ι-state A, if I is the ι-pattern *i_pat*($\bar{t}$, A), then for $1 \le k \le n$, *inst*($I[k]$) describes the instantiation of t_k, while *share*($I[k]$) gives the indices of other elements in $\bar{t}$ that t_k shares variables with. The ι-pattern of a call to a predicate is referred to as a *calling pattern*, while that at the return from a call is referred to as the *success pattern* for that call. Finally, an ι-pattern is said to describe a tuple of terms if the instantiations of, and sharing between, the elements of the tuple of terms are ‘‘consistent’’ with the ι-pattern:

Definition: An ι-pattern $< <\delta_1, S_1>, ..., <\delta_n, S_n> >$ induced by an ι-state A *describes* a tuple of terms $<t_1, ..., t_n>$ if $inst(A(t_i))\ \mathit{\$le}\ \delta_i$, $1 \le i \le n$; and if $deps(A(t_i)) \cap deps(A(t_j)) \ne \emptyset$, $1 \le i, j \le n$, then $j \in S_i$ and $i \in S_j$. ●

3.3. Abstracting Unification

During static analysis, the bindings of variables at different points in a program clause are described by ι-states. It is therefore necessary to specify how unification is to be simulated over ι-states. First, consider a variable x occurring in a term t_1, whose instantiation in the ι-state under consideration is δ_1, and assume that t_1 is being unified with a term t_2 whose instantiation is δ_2. If aliasing effects are temporarily ignored, then the instantiation of the resulting term is no larger than $\delta = \delta_1 \nabla \delta_2$. If x is t_1, then the instantiation of x after unification must also be δ. If x is a proper subterm of t_1, then as long as t_2 is not a variable (i.e. as long as $\delta_2 \ne \mathbf{f}$), x becomes instantiated to some proper subterm of the term resulting from the unification; if t_2 is a variable, however, then the instantiation of x does not change. Given a set of terms T, let *psubs*(T) be the set of all proper subterms of all elements of T, and let *sub_inst*(T) be the instantiation ι(*psubs*(T)). Then, the instantiation ‘‘inherited’’ by a variable x occurring in a term during unification is given by the function *inherited_inst*, defined as follows:

Definition: Let t_1 be a term in an ι-state A, with $inst(A(t_1)) = \delta_1$, and let x be a variable occurring in t_1. The instantiation inherited by x when t_1 is unified with a term whose instantiation is δ_2 is given by

$$inherited\_inst(A, x, t_1, \delta_2) \equiv \textbf{if } x = t_1 \textbf{ then } \delta; \textbf{ else if } \delta_2 = \mathbf{f} \textbf{ then } inst(A(x)); \textbf{ else } sub\_inst(\delta);$$

where $\delta = \delta_1 \nabla \delta_2$. ●

The function *sub_inst* is given by the following table:

δ	*sub_inst*(δ)
d	**d**
nv	**d**
c	**c**
f	**e**
e	**e**

Example 3: Suppose a term $f(X)$ is being unified with a ground term, where the variable X is uninstantiated in the ι-state A under consideration. Then, $\iota(f(X)) =$ **nv**, and the value of *inherited_inst*(A, X, $f(X)$, **c**) is *sub_inst*(**nv** ∇ **c**) = *sub_inst*(**c**) = **c**. This says that any term X could become instantiated to as a result of this unification is in the set denoted by **c**. Notice that this is sound in the sense that if the unification were to fail, the resulting instantiation of X would be **e**, which is contained in **c**. ●

In general, of course, it is not enough to simply consider instantiations when describing the effects of unification: dependencies between variables must be taken into account as well. Consider two ι-patterns I_1 and I_2 representing two n-tuples of terms $\bar{t}_1$ and $\bar{t}_2$ that are being unified. Assume, for the sake of simplicity, that these tuples of terms do not share variables. Consider the k^{th} element t_{1k} of $\bar{t}_1$: before unification, the elements of $\bar{t}_1$ that t_{1k} shares variables with is given by the share set $share(I_1[k])$; the share set for t_{1k} after unification is given by the transitive closure of the ‘‘may share with’’ relation, as expressed by the share sets of the two ι-patterns, starting at the $k^{th.}$ elements. This is described by the notion of ‘‘coupling closure’’, denoted by *c_closure*: the indices of elements of $\bar{t}_1$ that the $k^{th.}$ element t_{1k} may share variables with after unification is given by $c\_closure(k, I_1, I_2)$. More formally, this can be defined as follows:

Definition: Consider two ι-patterns I_1 and I_2, each of length n, and some k, $1 \le k \le n$. The *coupling closure* of k in I_1 induced by I_2, written $c\_closure(k, I_1, I_2)$, is defined to be the least set satisfying

(i) if $m \in share(I_1[\,k\,])$ then m is in $c\_closure(k, I_1, I_2)$; and

(ii) if $n_1 \in c\_closure(k, I_1, I_2)$, $n_2 \in share(I_2[\,n_1\,])$ and $m \in c\_closure(n_2, I_1, I_2)$, then m is in $c\_closure(k, I_1, I_2)$. ●

Example 4: Consider the unification of the tuple of terms $\langle X, f(Y), Y, Z\rangle$ with the tuple $\langle U, f(U), V, W\rangle$, where each of the variables U, V, W, X, Y, Z is uninstantiated and not aliased to any other variable. The ι-patterns for these tuples of terms are

$I_1 = \langle\, \langle \mathbf{f}, \{1\}\rangle, \langle \mathbf{nv}, \{2,3\}\rangle, \langle \mathbf{f}, \{2,3\}\rangle, \langle \mathbf{f}, \{4\}\rangle\, \rangle$, and

I_2 = < <**f**, {1,2}>, <**nv**, {1,2}>, <**f**, {3}>, <**f**, {4}> >

respectively. The value of $c\_closure(1, I_1, I_2)$ is the set {1, 2, 3}, indicating that after unification, the first three arguments can share variables. ●

It is worth pointing out that if it is known exactly what the two tuples τ_1 and τ_2 being unified are, then it is trivial to work out the dependencies resulting from unification: let θ be the most general unifier of τ_1 and τ_2, then this information is easily obtained from the tuple of terms $\theta(\tau_1)$. During analysis, however, one of the tuples consists of the arguments of a call or a return, and is represented by an ι-pattern. Thus, the precise form of one of the tuples of terms will not be known. Indeed, even for the other tuple of terms, only partial information will be available, since substitutions obtained at various program points are approximated by ι-states. It is for this reason that the computation of coupling closures is necessary when reasoning about unification.

We are now in a position to describe ‘‘abstract unification’’. There are two contexts where unification has to be dealt with during analysis: at the time of a call, when unifying the arguments of the call with those in the head of the clause; and at the time of a return from the call, when ‘‘back-unifying’’ to propagate the effects of the return to the caller. Since calls and returns are represented by ι-patterns, in each of these cases the structure of one of the tuples of terms being unified is known, but that of the other is represented by an ι-pattern. The processing of unification in the abstract domain is carried out as follows: first, changes in variable instantiations resulting from the unification are derived without taking dependencies between variables into account, using a function *a_unify_init*; these changes are then propagated to other variables, using two functions: *propagate_deps*, which propagates possible changes to dependency sets, and *propagate_inst*, which propagates possible changes to the instantiations of variables; finally, dependency sets are ‘‘cleaned up’’ using information about variable groundness.

Let $\tau = \langle t_1, ..., t_n \rangle$, and for any variable v, let $occ(v, \tau) = \{ j \mid v \in vars(t_j) \}$ be the indices of the elements of τ in which v occurs. Then, the function *a_unify_init* is defined as follows:

Definition: Consider an ι-state A_0 for a clause C, an n-tuple of terms $\tau = \langle t_1, \ldots, t_n \rangle$ and an ι-pattern $\bar{I} = \langle\langle \delta_1, S_1 \rangle, \ldots, \langle \delta_n, S_n \rangle\rangle$. Then, $a\_unify\_init(A_0, \tau, \bar{I}) = \langle A_1, V_1 \rangle$, where A_1 is an ι-state for C and $V_1 \subseteq \mathbf{V}_C$, is defined as follows:

- for any variable v in $\mathbf{V}_C$, if $occ(v, \tau) \neq \emptyset$ then $A_1(v) = \langle \delta, D \rangle$, where

  $\delta = \nabla \{ inherited\_inst(A_0, v, t_j, \delta_j) \mid j \in occ(v, \tau) \}$, and
  $D = \cup \{ deps(A_0(t_j)) \mid j \in c\_closure(k, i\_pat(\tau, A_0), \bar{I}) \}$, for any $k \in occ(v, \tau)$;

  if $occ(v, \tau) = \emptyset$ then $A_1(v) = A_0(v)$.

- $V_1 = \{ v \in \mathbf{V}_C \mid A_0(v) \neq A_1(v) \}$. ●

The function *a_unify_init* returns a pair consisting of an ι-state and a set of program variables: the ι-state reflects the effects of unification on variables in the clause, without taking into account any dependencies that might exist between them. The set of variables returned consists of those variables whose instantiations changed in this step. These changes can then be propagated to account for dependencies between variables.

Let the ι-state returned by *a_unify_init* be A_1, and for any program variable x in the clause under consideration, let $A_1(x) = \langle \delta, D \rangle$. δ is obtained by considering the instantiations inherited from each position in which x occurs in the tuple of terms $\bar{t}$: from the properties of unification, it follows that the resulting instantiation inherited by x must be the least upper bound of all of these with respect to the instantiation order $\trianglelefteq$. It is possible that x is aliased to some other variable whose instantiation or dependency set changes because of this unification: this is not taken into account at this point, but is handled in the next step by the functions *propagate_deps* and *propagate_inst*. The dependency set D of x is computed as follows: first, the possible elements of $\bar{t}$ that x could share variables with after unification is obtained using *c_closure*. If this indicates that x could share variables with the $j^{\text{th.}}$ element t_j of $\bar{t}$, and t_j could have depended on a variable y before unification, then it is inferred that x can depend on y after unification. The dependency set D of x after unification is thus obtained by taking the union of the dependency sets of all such elements t_j. Notice that when computing the coupling closure in this case, it suffices to consider the index of any one element t_k of $\bar{t}$ in which x occurs: because *c_closure* computes a transitive closure starting with the set of elements with which t_k shares variables, and because the ''shares a variable with'' relation is symmetric, the value of the coupling closure does not depend on which particular k is picked.

The next step is to describe the propagation of changes to dependency sets and instantiations resulting from unification. To simplify the presentation, we decompose this step into two functions: *propagate_deps*, which propagates changes to dependency sets, and *propagate_inst*, which propagates changes to instantiations of variables. Changes to dependency sets are relatively easy to characterize: let $\langle A_1, V_1 \rangle = a\_unify\_init(A_0, \bar{t}, \bar{I})$ for some ι-state A_0, tuple of terms $\bar{t}$ and ι-pattern $\bar{I}$. Consider a variable x whose dependency set changes when *a_unify_init* is computed: there may be a variable y that depends on x, and whose dependency set should also be updated. Any change to the dependency set of x can be propagated to y by simply adding $deps(A_1(x))$ to the dependency set of y. Since changes to dependency sets during the computation of *a_unify_init* are obtained from coupling closures, which compute the transitive closure of the ''may share with'' relation, it suffices to repeat this once for each variable appearing in the dependency set of y, i.e. it is not necessary to compute a transitive closure to compute the updated dependency set for y. The function *propagate_deps* is thus defined as follows:

Definition: Let A_0 be an ι-state defined on a set of program variables $\mathbf{V}_C$, and $V_0 \subseteq \mathbf{V}_C$. Then, $propagate\_deps(\langle A_0, V_0 \rangle) = \langle A_1, V_1 \rangle$, where A_1 is an ι-state defined on $\mathbf{V}_C$ and $V_1 \subseteq \mathbf{V}_C$, is defined as follows:

- for each x in $\mathbf{V}_C$, if $A_0(x) = \langle \delta, D \rangle$, then $A_1(x) = \langle \delta, D' \rangle$, where $D' = \cup\{deps(A_0(v)) \mid v \in D\}$;
- $V_1 = V_0 \cup \{v \in \mathbf{V}_C \mid A_1(v) \neq A_0(v)\}$. ●

The next step is to propagate changes in instantiations of variables by taking into account dependencies between variables. Suppose the set of variables inferred to be affected by unification, in *a_unify_init* and *propagate_deps*, is V. Let the ι-state obtained from *propagate_deps* be A, and consider a variable x with $A(x) = \langle \delta, D \rangle$. Then, x depends on a subset (possibly empty) of D. There are five possibilities:

(1) If $\delta = \mathbf{e}$ then execution cannot reach that point, so δ' must also be $\mathbf{e}$.

(2) If $\delta = \mathbf{f}$, there are three possibilities: (*i*) if x does not depend on any variable in $V \cap D$ at runtime, then its instantiation is unaffected, and $\delta' = \mathbf{f}$; (*ii*) if x actually depends only on some set of variables $D_0 \subseteq D$, but all of the variables in D_0 are still uninstantiated after unification, then the instantiation of x is unaffected by the unification, and $\delta' = \mathbf{f}$; (*iii*) if x depends on some variable $y \in V \cap D$, and $inst(A(y)) \neq \mathbf{f}$, then some variable that x can depend on has become instantantiated to a nonvariable term, so $inst(A(y))$ is either $\mathbf{c}$, $\mathbf{nv}$ or $\mathbf{d}$. Combining these cases, it follows from the structure of the mode set Δ that δ' must be $\mathbf{f}$ *$lub* $inst(A(y))$, which works out to $\mathbf{d}$.

(3) If $\delta = \mathbf{c}$ then x is a ground term and cannot be affected by aliasing effects, so $\delta' = \mathbf{c}$.

(4) If $\delta = \mathbf{nv}$ then there are two possibilities: (*i*) if x does not depend on any variable in $V \cap D$ at run-time, then its instantiation is unaffected, and $\delta' = \mathbf{nv}$; (*ii*) if it does depend on variables in $V \cap D_0$, then its instantiation changes only if all their instantiations change to $\mathbf{c}$ in which case $\delta' = \mathbf{c}$. Combining cases (*i*) and (*ii*), it follows that $\delta' = \mathbf{nv}$ *$lub* $\mathbf{c} = \mathbf{nv}$.

(5) If $\delta = \mathbf{d}$ then, by an argument similar to the $\mathbf{nv}$ case, it follows that $\delta' = \mathbf{d}$.

It does not come as a great surprise that the only case where aliasing and dependency effects make a difference is when the variable under consideration is uninstantiated, and the instantiation of one of its possible aliases changes. Based on the above case analysis, the function *propagate_inst* can be defined as follows:

Definition: If A_0 is an ι-state defined on a set of variables $\mathbf{V}_C$ and $V \subseteq \mathbf{V}_C$, then A_1 = *propagate_inst*($\langle A_0, V \rangle$) is an ι-state defined on $\mathbf{V}_C$, defined as follows: for every x in $\mathbf{V}_C$, if $A_0(x) = \langle \mathbf{f}, D_0 \rangle$ and there is a variable y in $V \cap D_0$ such that $inst(A_0(y)) \neq \mathbf{f}$, then $A_1(x) = \langle \mathbf{d}, D_0 \rangle$; otherwise, $A_1(x) = A_0(x)$. ●

The final step is to ''clean up'' the dependency sets of variables. If a variable x has instantiation $\mathbf{c}$, i.e. is inferred to be ground, then it does not share variables with any other variable; thus, the dependency set of x can be set to $\emptyset$, and x can be deleted from the dependency set of any other variable in the clause. This is described by a function *normalize*:

Definition: Let A_0 be an ι-state defined on a set of program variables $\mathbf{V}_C$, and let $ground(A_0) = \{v \in \mathbf{V}_C \mid inst(A(v)) = \mathbf{c}\}$. Then, $normalize(A)$ is an ι-state A_1, with domain $\mathbf{V}_C$, defined as follows: for each $x \in \mathbf{V}_C$, if $A_0(x) = \langle \delta, D \rangle$, then $A_1(x) =$ **if** $x \in ground(A_0)$ **then** $\langle \delta, \emptyset \rangle$; **else** $\langle \delta, D - ground(A_0) \rangle$. ●

We can now define a function *update_i_state* that simulates unification in the ''abstract domain'': given an ι-state, a tuple of terms $\bar{t}$ and an ι-pattern $\bar{I}$, it describes the ι-state resulting from the unification of $\bar{t}$ with a tuple of terms described by $\bar{I}$:

Definition: If A_0 is an ι-state defined on a set of variables $\mathbf{V}_C$, $\bar{t}$ is an n-tuple of terms all whose variables are in $\mathbf{V}_C$, and $\bar{I}$ is an ι-pattern of length n, then $update\_i\_state(A_0, \bar{t}, \bar{I})$ is an ι-state defined on $\mathbf{V}_C$, defined by

$$update\_i\_state(A_0, \bar{t}, \bar{I}) = normalize(propagate\_inst(propagate\_deps(a\_unify\_init(A_0, \bar{t}, \bar{I})))).$$

●

Note that the propagation of dependencies by *propagate_deps* does not depend on the instantiations of variables, but the propagation of instantiations by *propagate_inst* depends on the dependency sets of variables. It is therefore necessary to ensure that dependency sets of variables have been properly updated before applying *propagate_inst*. For this reason, it is necessary to apply *propagate_deps* before *propagate_inst*.

3.4. Propagating Flow Information

The module being analyzed is assumed to be of the form $\langle P, \text{EXPORTS}(P) \rangle$, where P is a set of predicate definitions, and $\text{EXPORTS}(P)$ is a set of pairs $\langle p, cp \rangle$ specifying the predicates p that are exported by the program, i.e. that may be called from outside the program; and for each such predicate, a calling pattern cp that it may be called with from outside the program. Note that $\text{EXPORTS}(P)$ may contain more than one entry for a predicate if it can be called with different calling patterns.

Given a class of queries that the user may ask of a program, as specified by $\text{EXPORTS}(P)$, 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''. Similarly, given a calling pattern for a predicate, only those success patterns will be considered admissible that might actually correspond to computations for that predicate starting with a call described by that calling pattern. For $n \geq 0$, let Γ_n denote the set of pairs $\Delta \times 2^{\{1, \dots, n\}}$. With each n-ary predicate p in a program we associate a set $\text{CALLPAT}(p) \subseteq (\Gamma_n)^n$, the set of *admissible calling patterns*, and a relation $\text{SUCCPAT}(p) \subseteq (\Gamma_n)^n \times (\Gamma_n)^n$, associating with each calling pattern an *admissible success pattern*. Given a module $\langle P, \text{EXPORTS}(P) \rangle$, these sets are defined to be the smallest sets satisfying the following:

- If $\langle p, I \rangle \in$ EXPORTS(P)⟩, then I is in CALLPAT(p).
- Let q_0 be a predicate in the program, $I_c \in$ 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 ι-state at the point immediately after the literal $q_j(\bar{X}_j)$, $0 \le j \le n$, be A_j, where A^{init} is the initial ι-state of the clause; $A_0 = update\_i\_state(A^{init}, \bar{X}_0, I_c)$; then, for $1 \le i \le n$, $cp_i = i\_pat(\bar{X}_i, A_{i-1})$ is in CALLPAT(q_i); and if $<cp_i, sp_i>$ is in SUCCPAT(q_i), then $A_i = update\_i\_state(A_{i-1}, \bar{X}_i, sp_i)$.

  The success pattern for the clause is given by $I_s = i\_pat(\bar{X}_0, A_n)$, and $<I_c, I_s>$ is in SUCCPAT(q_0). •

Notice that no special provision is required for explicit unification via '='/2: this predicate can be handled simply by considering it to be defined as

'='(X, X).

and proceeding as above.

The global data structures maintained by the algorithm consist of a worklist, NEEDS_PROCESSING, of predicates that have to be processed; and for each predicate p in the program, tables CALLPAT(p) and SUCCPAT(p). Initially, NEEDS_PROCESSING contains the set of predicates appearing in EXPORTS(P). If p is an exported predicate, then CALLPAT(p) contains the calling patterns for it that are specified in EXPORTS(P), otherwise it is empty initially; and for each predicate p in the program, SUCCPAT(p) is initially empty. Before analysis begins, the call graph of the program is constructed, and this is used to compute, for each predicate p, the set CALLERS(p) of the predicates that call p, i.e. those predicates q for which there is a clause in the program of the form

q(...) :– . . ., p(...), . . .

The set CALLERS(p) is used to determine which predicates have to be reanalyzed when a new success pattern is found for p.

The analysis begins with the calling patterns specified in EXPORTS(P), and proceeds as follows: first, the predicates mentioned in EXPORTS(P) are analyzed. This, in turn, causes the predicates called by the exported predicates to be analyzed, and so on. This is repeated until no new calling or success patterns can be obtained for any predicate, at which point the analysis terminates. The algorithm is illustrated in Figure 1.

The function *create_i_state*, given a set of variables V, returns the initial ι-state A over V, i.e. the ι-state whose domain is V, such that for each v in V, $A(v) = \langle \mathbf{f}, \{v\} \rangle$. Given a calling pattern cp for a predicate p, the clauses of p are analyzed by the procedure **analyse_pred**, which returns a set of success patterns for p for that calling pattern. If cp not already present in CALLPAT(p), then it is a new calling pattern: in this case it is added to CALLPAT(p), and each clause of p is analyzed via **analyse_clause**. The set of success patterns so computed is returned by **analyse_pred**. If cp is found to be present in CALLPAT(p), *Input*: A program $\langle P, \text{EXPORTS}(P)\rangle$.

Output: Tables CALLPAT(p) and SUCCPAT(p) giving the admissible calling and success patterns for each predicate p in the program, with respect to the set of exported predicates and external calling patterns specified in EXPORTS(P).

Method: Starting with the exported predicates, iterate over the program until no new calling or success patterns can be inferred for any predicate.

(1) Construct the call graph for P. Hence determine, for each predicate p defined in P, the set CALLERS(p) of predicates that call p.

(2) *Initialization*:

For each n-ary predicate p defined in P, create tables CALLPAT(p) and SUCCPAT(p), initialized to be empty.

For each predicate p mentioned in EXPORTS(P), add p to NEEDS_PROCESSING; for each $\langle p, cp\rangle$ in EXPORTS(P), add cp to CALLPAT(p).

(3) *Analysis*:

```
while NEEDS_PROCESSING ≠ ∅ do
    let p be an element of NEEDS_PROCESSING;
    NEEDS_PROCESSING := NEEDS_PROCESSING − {p};
    for each cp ∈ CALLPAT(p) do
        analyse_pred(p, cp);      /* results are in extension table, return value can be ignored */
    od;
od.   /* while */
```

Figure 1: Algorithm for mode and data dependency analysis

there are two possibilities:

(i) If a nonempty set S of success patterns for p, corresponding to the calling pattern cp, is found in SUCCPAT(p), then they are not recomputed. Instead, the set S is returned directly.

(ii) If no success patterns can be found for p corresponding to the calling pattern cp, this indicates a circularity where no possibility for a successful execution of p can be established. In this case, the set of success patterns returned is $\emptyset$.

This strategy is essentially that of maintaining an *extension table* [12, 24], which is used to ''remember'' the success patterns computed for each calling pattern. It can be shown that the use of extension tables leads to an execution strategy that is complete for finite domains, i.e. that all answers are found for any computation [12]; in our case, this implies that all success patterns corresponding to any given calling pattern for a predicate can be computed in finite time. Once a set of success patterns has been computed for a given calling pattern for a predicate, it can be retrieved for future invocations with the same calling pattern in $O(1)$ expected time, by hashing on the calling pattern.

A clause is analyzed by the procedure **analyse_clause**, which propagates ι-states across the clause as described above, and returns the set of possible success patterns for the given calling pattern. The actual propagation of sets of possible ι-states across the body of the clause is done by the procedure **analyse_body**. This procedure takes a set of ι-states for the clause, and recursively processes the literals in the body, until there are no more literals left to process, at which point it returns the resulting set of ι-states. The processing of each literal consists of (*i*) using an ι-state just before the literal to compute a calling pattern for it; (*ii*) using **analyse_pred** to compute the corresponding success patterns for this calling pattern; and (*iii*) using these success patterns to compute the ι-states just after that literal. The procedures **analyse_pred**, **analyse_clause** and **analyse_body** are given in Figure 2. Some observations on these procedures are worth making:

(1) In the description of the function **analyse_pred**, every clause of a predicate is analyzed for every calling pattern. If so desired, it is straightforward to add a third argument to this function, representing the tuple of arguments to the predicate, and analyse only those clauses whose heads unify with this tuple. For example, consider the program

```
p(X, Y, Z) :– q([X | Y], Z).
q([ ], [ ]).
q([H | L], [H1 | L1]) :– . . .
```

When analyzing the literal $q([X|Y], Z)$ in the body of the clause for the predicate p, such a scheme would supply to **analyse_pred** a third argument ⟨[X | Y], Z⟩, representing the tuple of parameters of this literal. When analyzing the clauses for q, only those clauses whose head arguments unify with this tuple – in this case, only the second clause – would be considered. In principle, this can improve the precision of the analysis somewhat. From our experience with programs usually encountered in practice, however, it is our opinion that the benefits actually accruing from such an extension may not be very significant.

(2) Notice that in the description of **analyse_clause**, there is a call to the function *create_i_state*. This is done primarily to simplify the presentation: in practice, there is no need to create the initial ι-state for a clause repeatedly, each time it is analyzed. Instead, it can be created once, at the beginning of analysis, and associated with the clause. Then, whenever a clause is processed, its initial ι-state can be obtained by a simple lookup.

(3) The procedure **analyse_clause** returns all the success patterns computed for the particular calling pattern. It is possible to consider a variant of the algorithm that returns only the new success patterns found (obtained from the set *NEW_SP* in the pseudo-code for **analyse_clause** in Figure 2). While this does not affect the correctness of the algorithm, a larger number of iterations may be necessary to compute the fixpoint using this variant, leading to decreased efficiency [12].

(4) The correctness of the algorithm is independent of the order in which the elements of the set NEEDS_PROCESSING are processed. However, it is usually more efficient to process them in depth-first order, i.e. maintain NEEDS_PROCESSING as a stack.

```
function analyse_pred(p, cp)        /* p is the predicate to be analyzed; cp is a calling pattern */
begin
    if cp ∈ CALLPAT(p) then return {sp | ⟨cp, sp⟩ ∈ SUCCPAT(p)};    /* use previously computed success patterns */
    else
        add cp to CALLPAT(p);
        for each clause c_i of p do  S_i := analyse_clause(c_i, cp)  od;
        return ∪_i S_i;
    fi;
end.

function analyse_clause(cl, cp)    /* cl is the clause to be analyzed; cp is its calling pattern */
begin
    let cl be of the form ‘‘p(X̄) :– Body’’;
    A^init := create_i_state(V), where V is the set of variables appearing in cl;
    A_0 := {update_i_state(A^init, X̄, cp)};              /* head unification */
    A_n := analyse_body(Body, A_0);
    SP := {i_pat(X̄, A_n) | A_n ∈ A_n};                     /* success patterns for the clause */
    NEW_SP := {⟨cp, sp⟩ | sp ∈ SP $and ⟨cp, sp⟩ $nomem SUCCPAT(p)};        /* new success patterns */
    if NEW_SP ≠ ∅ then
        add NEW_SP to SUCCPAT(p);
        add CALLERS(p) to NEEDS_PROCESSING;
    fi;
    return SP;
end.

function analyse_body(Body, A)        /* Body is the body of a clause C; A is a set of ι-states for C */
begin
    if Body is empty then return A;
    else
        let Body be of the form ‘‘q(X̄) , BodyTail’’;
        A_1 := ∅;
        for each A ∈ A do
            cp := i_pat(X̄, A);                            /* a calling pattern for q(X̄) */
            S := analyse_pred(q, cp);                     /* success patterns for q(X̄) */
            for each sp ∈ S do  add update_i_state(A, X̄, sp) to A_1  od;
        od;
        return analyse_body(BodyTail, A_1);
    fi;
end.
```

Figure 2: the functions **analyse_pred**, **analyse_clause** and **analyse_body**.

(5) A useful heuristic during analysis is to process the non-recursive clauses of a predicate before the recursive ones, and to process facts, i.e. unit clauses, before rules. The recursive clauses can be found by looking for cycles in the call graph of each clause: the time taken for this is proportional to the size of the call graph. Processing clauses in this manner increases the likelihood of finding solutions in the extension table when processing recursive calls, so that they need not be recomputed.

The mode of a predicate p can be computed easily from its set of calling patterns CALLPAT(p) as follows: if *$lub* CALLPAT($p$) is $<<\delta_1, S_1>, \ldots, <\delta_n, S_n>>$ then the mode of p is $<\delta_1, \ldots, \delta_n>$. To compute data dependencies, it is necessary to maintain the ι-state at each program point. Then, two literals $p(\bar{t}_1)$ and $q(\bar{t}_2)$ are independent in an ι-state A if for every variable x in $vars(\bar{t}_1)$ it is the case that $deps(A(x)) \cap vars(\bar{t}_2) = \emptyset$. (The symmetry of the dependence relation between variables implies that this is equivalent to saying that for every x in $vars(\bar{t}_2)$, $deps(A(x)) \cap vars(\bar{t}_1) = \emptyset$.)

3.5. Examples

To illustrate the algorithm described above, we give two examples. The first is the usual *append* program, and illustrates how recursion is handled using the extension table; the second, taken from [6], illustrates the handling of aliasing.

Example 5: Consider the program

```
append([H|L1], L2, [H|L3]) :– append(L1, L2, L3).
append([], L, L).
```

Assume that the user has specified the following calling pattern for this predicate:

$$Cp = <<\mathbf{c}, \emptyset>, <\mathbf{c}, \emptyset>, <\mathbf{f}, \{3\}>>.$$

Initially, The table CALLPAT(*append*) contains only Cp, while SUCCPAT(*append*) is empty. Suppose the clauses are processed according to the heuristic mentioned at the end of Section 3.4, i.e. facts before rules and nonrecursive rules before recursive ones. Processing the unit clause, we have $A^{init} = \{L \rightarrow <\mathbf{f}, \{L\}>\}$. The ι-state A_0 for this clause resulting from the unification of the head with the call is given by

$$update\_i\_state(A^{init}, <[], L, L>, Cp)$$

where Cp is the calling pattern given above. To show how *update_i_state* works, we step through its computation: let $<A_0', V_1> = a\_unify\_init(A^{init}, <[], L, L>, Cp)$. Then, we have $A_0' = \{L \rightarrow <\delta, D>\}$, where $\delta = \nabla \{\mathbf{c}, \mathbf{f}\} = \mathbf{c}$; and $D = \{L\}$. The set V_1 of variables whose instantiations change in this step is $\{L\}$. The functions *propagate_deps* and *propagate_inst* do not have any effect on this ι-state. Since the instantiation of L is **c**, the dependency set of L in the ι-state resulting from *normalize* is $\emptyset$. Thus, we have $A_0 = \{L \rightarrow <\mathbf{c}, \emptyset>\}$. The success pattern of this clause is therefore

$$Sp = i\_pat(<[], L, L>, A_0) = <<\mathbf{c}, \emptyset>, <\mathbf{c}, \emptyset>, <\mathbf{c}, \emptyset>>.$$

The tuple $<Cp, Sp>$ is therefore added to SUCCPAT(*append*), after which the recursive clause is processed. For this, the ι-state resulting from the unification of the head with the call is

$$A_0 = \{\mathrm{H} \rightarrow <\mathbf{c}, \emptyset>;\ \mathrm{L1} \rightarrow <\mathbf{c}, \emptyset>;\ \mathrm{L2} \rightarrow <\mathbf{c}, \emptyset>;\ \mathrm{L3} \rightarrow <\mathbf{f}, \{\mathrm{L3}\}>\}.$$

The calling pattern for the literal in the body is therefore

$$i\_pat(<\mathrm{L1}, \mathrm{L2}, \mathrm{L3}>, A_0) = <<\mathbf{c}, \emptyset>, <\mathbf{c}, \emptyset>, <\mathbf{f}, \{3\}>>.$$

This calling pattern is found to be present in CALLPAT(*append*), and the success pattern for this is found from SUCCPAT(*append*) to be the ι-pattern *Sp* given above. The ι-pattern *Sp* is therefore taken to be the success pattern for the literal in the body, and the ι-state A_1 after the return from it is given by

$$A_1 = update\_i\_state(A_0, <\mathrm{L1}, \mathrm{L2}, \mathrm{L3}>, Sp).$$

The reader may verify that $A_1 = \{\mathrm{H} \rightarrow <\mathbf{c}, \emptyset>;\ \mathrm{L1} \rightarrow <\mathbf{c}, \emptyset>;\ \mathrm{L2} \rightarrow <\mathbf{c}, \emptyset>;\ \mathrm{L3} \rightarrow <\mathbf{c}, \emptyset>\}$. From this, the success pattern of the second clause is obtained as

$$i\_pat(<[\mathrm{H}|\mathrm{L1}], \mathrm{L2}, [\mathrm{H}|\mathrm{L3}]>, A_1) = <<\mathbf{c}, \emptyset>, <\mathbf{c}, \emptyset>, <\mathbf{c}, \emptyset>>.$$

This is seen to be the same success pattern as for the first clause, with the appropriate entry already present in SUCCPAT(*append*). The sets of calling and success patterns do not change further, and the analysis terminates at the next iteration. From this, the mode of *append* is found to be $<\mathbf{c}, \mathbf{c}, \mathbf{f}>$.

It is easy to verify that the same results are obtained even if the clauses are processed in the opposite order, i.e. the recursive clause is processed first. When the recursive call is first encountered, SUCCPAT(*append*) is empty, so the set of success patterns returned is ∅. However, when the unit clause is then processed, a success pattern is found and entered into SUCCPAT(*append*) by **analyse_clause**; this also adds CALLERS(*append*), which in this case is just {*append*}, to the set NEEDS_PROCESSING. Because of this, NEEDS_PROCESSING becomes nonempty, causing the top level loop to go through another iteration. In the second iteration, the success pattern found for the unit clause is propagated through the recursive clause: since no new success patterns are found in this iteration, NEEDS_PROCESSING remains empty at the end of this iteration, and the algorithm terminates. Thus, the same results are obtained, but in this case an additional iteration is necessary to propagate the success pattern computed for the unit clause into the body of the recursive clause. ●

Example 6: Consider the aliasing example from [6]: the program is

```
p(X, Y) :– q(X, Y), r(X), s(Y).
q(Z, Z).
r(a).
s(_).
```

Assume that *p* is the only exported predicate in this program, and that the user-specified calling pattern for it is

$$<<\mathbf{f}, \{1\}>, <\mathbf{f}, \{2\}>>.$$

This ι-pattern is inserted into CALLPAT(*p*). Now consider the clause for *p*: let the ι-state after the $k^{\text{th.}}$ literal (where the head counts as the $0^{\text{th.}}$ literal) be A_k. The ι-state resulting from unification of the head with the call is A_0 = {X $\rightarrow$ <**f**, {X}>, Y $\rightarrow$ <**f**, {Y}>}. The calling pattern for q is therefore <<**f**, {1}>, <**f**, {2}>>, and is added to CALLPAT(q).

The clause for q is then analyzed: the ι-state A_q resulting from the unification of the head of the clause for q with the call is given by A_q = {Z $\rightarrow$ <**f**, {Z}>} from which the success pattern of the call to q is inferred to be <<**f**, {1, 2}>, <**f**, {1, 2}>>. The ι-state A_1 after the literal $q(X, Y)$ in the clause for p is therefore

A_1 = {X $\rightarrow$ <**f**, {X, Y}>, Y $\rightarrow$ <**f**, {X, Y}>}.

The calling pattern for the literal for r is therefore <**f**, {1}>, and the success pattern in this case is <**c**, ∅>. The ι-state A_2 after this literal is given by *update_i_state*(A_1, <X>, <**c**, ∅>). Let A_2(X) = <δ_X, D_X>, and A_2(Y) = <δ_Y, D_Y>. Let <A_1', V_1> = *a_unify_init*(A_1, <X>, <**c**, ∅>), then A_1' = {X $\rightarrow$ <**c**, {X, Y}>, Y $\rightarrow$ <**f**, {X, Y}>}, and V_1 = {X}. The function *propagate_deps* has no effect on this ι-state, so that *propagate_deps*($\langle A_1', V_1\rangle$) = $\langle A_1', V_1\rangle$. Then, in the computation of *propagate_inst*($\langle A_1', V_1\rangle$), since X is in the dependency set of Y as well as in V_1 and because its instantiation is not **f**, the instantiation of Y is set to **d**. The dependency set of X is set to ∅ in the normalization phase. The ι-state A_2 is therefore

A_2 = {X $\rightarrow$ <**c**, ∅>, Y $\rightarrow$ <**d**, {Y}>}.

The calling pattern for s is therefore inferred as <**d**, {1}>, and its success pattern is the same, so that the ι-state A_3 at the end of the clause is the same as A_2. The success pattern of p is therefore <<**c**, ∅>, <**d**, {2}>>. This is conservative, but sound. ●

3.6. Handling Other Control Constructs

The only connective considered so far for literals in the body of a clause has been sequential conjunction, decoted by '**,**'. For analysis purposes, other connectives, such as disjunction (denoted by '**;**'), conditionals (denoted by '→') and negation, can be handled simply by a preprocessing phase that transforms clauses to a form that contains only '**,**'. A clause of the form ''*p* :– *q*, (*r* ; *s*), *t*'' is transformed to

p :– *q*, *r*, *t*.
p :– *q*, *s*, *t*.

A clause of the form ''*p* :– *q*, (*r* → *s* ; *t*), *u*'' is transformed to

p :– *q*, *r*, *s*, *u*.
p :– *q*, *t*, *u*.

In handling negation, a naive transformation might take a clause of the form ''*p* :– *q*, *not*((*r*, *s*)), *t*'' and yield

p :– *q*, *r*, *s*.
p :– *q*, *t*.

This simple transformation for negations is much too conservative, however, 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 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 sharpened by observing that if a clause for a predicate is guaranteed to fail, then its success pattern set can be taken to be $\emptyset$. The clause ''p :− q, not((r, s)), t'' may therefore be transformed to

p :− q, r, s, $fail$.
p :− q, t.

In this case, the presence of *fail* in the first transformed clause ensures that this clause does not affect p's success patterns.

4. Soundness and Complexity

4.1. Soundness

This section discusses the soundness of the flow analysis algorithm discussed above. First, we show that *update_i_state* safely describes the effects of unification for certain kinds of terms. This is followed by a proof that the ι-states computed by the flow analysis procedure described above are sound.

As is evident from the previous section, *update_i_state* is used for only two purposes: for the unification of the head of a clause with a call, and for back-unification of arguments at the return from a call. When the head of a clause is being unified with a call, the two terms being unified are variable disjoint; in the case of back-unification at the return from a call, the returning argument tuple is an instance of the calling argument tuple that does not contain any variable of the calling clause not present in the calling argument tuples. The correctness proof below restricts itself to unification between terms that satisfy these criteria, which are referred to as *call-compatibility* and *return-compatibility*. Two terms t_1 and t_2 are *call compatible* if and only if they are variable disjoint; t_1 is *return compatible* with t_2 with respect to a set of variables V if and only if t_2 is an instance of t_1, and $vars(t_2) \cap V \subseteq vars(t_1)$.

If A is an ι-state defined on a set of variables $\mathbf{V}$ and, for a given substitution σ and set of variables $V \subseteq \mathbf{V}$ it is the case that for each v in V, (i) $\sigma(v) \in inst(A(v))$; and (ii) for any $x \in V$, $vars(\sigma(x)) \cap vars(\sigma(v)) \neq \emptyset$ implies $x \in deps(A(v))$ and $v \in deps(A(x))$, then σ is said to be *V-consistent* with A. If σ is $\mathbf{V}$-consistent with A, i.e. consistent on all the variables A is defined on, then σ is said to be *consistent* with A. From this definition, we have:

Lemma 4.1: Let A be an ι-state defined on a set of variables V, and $\bar{T}$ a tuple of terms all whose variables are in V. If σ is a substitution consistent with A, then $i\_pat(\bar{T}, A)$ describes $\sigma(\bar{T})$. *$box*

The following lemma shows that *update_i_state* correctly simulates unification in the abstract domain for calls and returns:

Lemma 4.2: Let A_0 be an ι-state defined over a set of variables $\mathbf{V}$; $\bar{t}$ an n-tuple of terms all whose variables are in $\mathbf{V}$; σ a substitution; $\bar{I}$ an ι-pattern of length n; and $\bar{t}'$ a tuple of terms described by $\bar{I}$ such that either (i) $\sigma(\tau)$ and τ' are call-compatible, or (ii) $\sigma(\tau)$ is return-compatible with τ' with respect to $\mathbf{V}$. If σ is consistent with A_0 and $\sigma(\tau)$ and τ' are unifiable with most general unifier ψ, then ψ *\$comp* σ is consistent with $update\_i\_state(A_0, \tau, \bar{\Gamma})$.

Proof: The lemma holds vacuously in the case where unification fails. Assume, therefore, that unification succeeds. Since τ' is either call- or return-compatible with τ, it suffices to consider the variables occurring in τ. Let

$\langle A', V'\rangle = a\_unify\_init(A_0, \tau, \bar{\Gamma})$;

$\langle A'', V''\rangle = propagate\_deps(\langle A', V'\rangle)$;

$A''' = propagate\_inst(\langle A'', V''\rangle)$; and

$A_1 = normalize(A''') = update\_i\_state(A_0, \tau, \bar{\Gamma})$.

Let x be any variable in $\mathbf{V}$, with $A_0(x) = \langle\delta_0, D_0\rangle$ and $A_1(x) = \langle\delta_1, D_1\rangle$.

Clearly, if neither x nor any variable in D_0 appears in τ, then x is unaffected by the unification. In this case, we have $A_1(x) = A_0(x)$ and (ψ *\$comp* $\sigma)(x) = \sigma(x)$. Since σ is consistent with A_0, ψ *\$comp* σ is trivially consistent with A_1 in this case, and the lemma holds. Assume that either x or some variable in D_0 occurs in τ. If $inst(A_0(x)) = \delta_0 = \mathbf{c}$ then, since σ is consistent with A_0, $\sigma(x)$ must be a ground term and hence cannot be affected by a successful unification, nor can it share variables with any other variable y in $\mathbf{V}$. In this case, it follows from the definitions of *update_i_state* that $\delta_1 = \mathbf{c}$ and $D_1 = \emptyset$, and again the lemma holds. Therefore, consider the case where $\delta_0 \neq \mathbf{c}$.

First, consider the dependency set of x. Let $\tau = \langle t_1, ..., t_n\rangle$. If x occurs in t_k, then the indices of the elements of $\sigma(\tau)$ that $\sigma(x)$ can share variables with is contained in $c\_closure(k, i\_pat(\tau, A_0), \bar{\Gamma})$. It follows that if, for some variable $y \in \mathbf{V}$,

$$vars((\psi \ \$comp \ \sigma)(x)) \cap vars((\psi \ \$comp \ \sigma)(y)) \neq \emptyset$$

then y occurs in t_j for some $j \in c\_closure(k, i\_pat(\tau, A_0), \bar{\Gamma})$. Therefore, from the definition of *update_i_state*, y must be in $deps(A_1(x))$. If x does not occur in τ and, for some variable $y \in \mathbf{V}$,

$$vars((\psi \ \$comp \ \sigma)(x)) \cap vars((\psi \ \$comp \ \sigma)(y)) \neq \emptyset$$

then either (i) $y \in D_0$; or (ii) y *\$nomem* D_0. In the former case, it is easy to verify from the definition of *update_i_state* that since the result of unification is not a ground term, y is in D_1. In the latter case, since σ is consistent with A_0, we must have $vars(\sigma(x)) \cap vars(\sigma(y)) = \emptyset$. It follows that for some variable z occurring in τ, $z \in deps(A_0(x))$ and z becomes dependent on y as a result of unification. But in this case, as argued above, $y \in deps(A'(z))$ and $z \in V_1$. Then, from the definition of *propagate_deps*, we have $y \in deps(A''(x))$. Since *propagate_inst* does not affect the dependency set of any variable, $y \in deps(A'''(x))$.

Next, consider the instantiation of the variable x. Recall that we are considering the case where $inst(A_0(x)) = \delta_0$ is not $\mathbf{c}$. Since *propagate_deps* does not affect instantiations, $inst(A''(x)) = inst(A'(x))$. From the first part of the proof of this lemma, it can be seen that if there is any other variable y that x can share a variable with, then y is in $deps(A''(x))$. From the case analysis before the definition of *propagate_inst*, it can be seen that the only interesting case is where $inst(A''(x)) = \mathbf{f}$ and there is some v in $V'' \cap deps(A''(x))$ for which $inst(A'(v)) \neq \mathbf{f}$. It follows from the definition of *propagate_inst* that in this case, $inst(A'''(x)) = \mathbf{d}$, so that $(\psi \$comp\ \sigma)(x) \in inst(A'''(x))$.

Finally, it is necessary to take into account the effect of the function *normalize*. From its definition, it can be seen that the only effect of *normalize* is the following: if $inst(A'''(v)) = \mathbf{c}$ for a variable v then the dependency set of v is set to $\emptyset$, and v is deleted from the dependency set of the other variables. Since there is no other effect on variables that are not ground, our earlier conclusions about such variables in A''' apply also to the ι-state $A_1 = normalize(A''') = update\_i\_state(A_0, \bar{\tau}, \bar{I})$. This shows that for every variable x in $\mathbf{V}$,

(i) $(\psi \$comp\ \sigma)(x) \in inst(A_1(x))$; and

(ii) For any $y \in \mathbf{V}$, if $vars((\psi \$comp\ \sigma)(x)) \cap vars((\psi \$comp\ \sigma)(y)) \neq \emptyset$, then $y \in deps(A_1(x))$.

Thus, $\psi \$comp\ \sigma$ is consistent with $A_1 = update\_i\_state(A_0, \bar{\tau}, \bar{I})$, and the lemma follows. *$box*

When considering the unification of a tuple of terms $\bar{\tau}$ with a tuple $\bar{\tau}'$ described by an ι-pattern $\bar{I}$, it has been assumed that the effects of this unification can be described by first considering the effects on the variables occurring in $\bar{\tau}$, and then propagating these effects to the other variables in the ι-state under consideration. This assumption does not hold if the compatibility requirements of Lemma 4.2 are violated. To see this, consider $\bar{\tau} = \langle X, a \rangle$, and $\bar{I} = \langle \langle \mathbf{f}, \{1\} \rangle, \langle \mathbf{f}, \{2\} \rangle \rangle$. Let the ι-state A under consideration be $A = \{X \rightarrow \langle \mathbf{f}, \{X\} \rangle\}$. Then, $update\_i\_state(A, \bar{\tau}, \bar{I}) = A$. It is evident that while $\bar{I}$ describes the tuple of terms $\bar{\tau}' \equiv \langle Y, X \rangle$, the result of unifying $\bar{\tau}$ and $\bar{\tau}'$ is not reflected in $update\_i\_state(A, \bar{\tau}, \bar{I})$. The reason for this is that $\bar{\tau}$ and $\bar{\tau}'$ are not call-compatible. Similarly, consider $\bar{\tau} \equiv \langle X \rangle$, $\bar{\tau}' \equiv \langle Y \rangle$, and let $\bar{I} = \langle \langle \mathbf{f}, \{1\} \rangle \rangle$. Let the ι-state A under consideration be

$$A = \{X \rightarrow \langle \mathbf{f}, \{X\} \rangle;\ Y \rightarrow \langle \mathbf{f}, \{Y\} \rangle\}.$$

Then, $update\_i\_state(A, \bar{\tau}, \bar{I}) = A$. The unification of $\bar{\tau}$ and $\bar{\tau}'$ aliases together the variables X and Y, and even though $\bar{I}$ describes $\bar{\tau}'$, this aliasing is not reflected in $update\_i\_state(A, \bar{\tau}, \bar{I})$. The problem in this case is that $\bar{\tau}$ and $\bar{\tau}'$ are not return-compatible.

To establish that the analysis algorithm is sound, it is necessary to show that for any computation of a ‘‘legitimate’’ call to an exported predicate in a module, any call to a predicate p that can arise is described by a tuple in CALLPAT(p), and any return from such a call is described by a tuple in SUCCPAT(p). This follows directly from the fact, established in Lemma 4.2, that unification in calls and returns are handled correctly:

Theorem 4.3 (Soundness): Let CALLPAT(p) and SUCCPAT(p) represent the admissible calling and success patterns for a predicate p in a module $\langle P, \text{EXPORTS}(P) \rangle$, and let $\langle q, c_q \rangle \in \text{EXPORTS}(P)$. In any computation of q with calling pattern c_q,

(1) for any call C_p to p that can arise at runtime, there is a calling pattern $I_c \in$ CALLPAT(p) such that I_c describes C_p; and

(2) if the call C_p can succeed with its arguments bound to a tuple of terms S_p, i.e. with success pattern $\iota(S_p)$, then there is a pair $<I_c, I_s>$ in SUCCPAT(p) such that I_s describes S_p.

Proof: By induction on the number of resolution steps in the computation. Consider a call $C_p \equiv p(\bar{T}_{in})$ obtained after k resolution steps.

Base case: If $k = 0$, then C_p must be the first literal in the user's query, i.e. the query must be of the form

$$?\text{-}\ p(\bar{T}_{in}),\ .\ .\ .$$

Since this is a user query, it has to conform to a calling pattern for p specified in EXPORTS(P), so there must be a user-specified calling pattern I_c for the exported predicate p, such that I_c describes C_p. It follows from the definition of CALLPAT(p) that I_c is in CALLPAT(p).

For success patterns, the base case must be for one resolution step. In this case, the query must be as above, and there must be a unit clause

$$U\ :\ p(\bar{S})$$

such that for some θ-activation of U, $\bar{T}_{in}$ and $\theta(\bar{S})$ are unifiable with most general unifier σ. The arguments of the call on success are therefore $\sigma(\bar{T}_{in}) \equiv (\sigma\ \$comp\ \theta)(\bar{S})$. The ι-state for U, after head-unification, is given by $A_0 = update\_i\_state(A^{init}, \bar{S}, I_c)$. From Lemma 4.2, the substitution $\sigma\ \$comp\ \theta$ is consistent with A_0. It follows from Lemma 4.1 that the success pattern $S_p \equiv i\_pat(\bar{S}, A_0)$ describes $(\sigma\ \$comp\ \theta)(\bar{S})$. Since $\sigma(\bar{T}_{in}) \equiv (\sigma\ \$comp\ \theta)(\bar{S})$, it follows that S_p describes $\sigma(\bar{T}_{in})$, whence the theorem holds.

Induction case: Assume that the theorem holds for k resolution steps, $k < n$. Consider a call $p(\bar{T}_{in})$ that arises after n steps. For this, there must be a clause for a predicate r,

$$r(\bar{X}_0) :\!-\ q_1(\bar{X}_1),\ .\ .\ .,\ q_m(\bar{X}_m),\ p(\bar{S}),\ .\ .\ .,\ q_n(\bar{X}_n)$$

where r is called with arguments $\bar{T}_r$. Certainly this call to r is obtained in fewer than n resolution steps, whence from the induction hypothesis there is an ι-pattern $I_r \in$ CALLPAT(r) that describes $\bar{T}_r$. Because $I_r \in$ CALLPAT(r), the analysis algorithm will have processed this clause with the calling pattern I_r.

Consider the analysis of a θ-activation of this clause for the calling pattern I_r. Let A_i be the ι-state immediately after the $i^{th.}$ literal in the clause (the head counts as the 0^{th} literal), and ψ_i any substitution for the variables of the clause obtained at that point for this call using SLD-resolution. For simplicity of notation, let $\sigma_i = \psi_i\ \$comp\ \theta$: the call to p being considered is thus $p(\bar{T}_{in}) \equiv p(\sigma_m(\bar{S}))$. We show, by induction on i, that σ_i is consistent with A_i, $0 \le i \le m$. Let ψ_0 be the most general unifier of the arguments $\bar{T}_r$ in the call and $\theta(\bar{X}_0)$ in the head, and $A_0 = update\_i\_state(A^{init}, \bar{X}_0, I_r)$. In the base case, it follows from Lemma 4.2 that σ_0 is consistent with A_0. Suppose that σ_i is consistent with A_i for $1 \le i < j$, and consider σ_j, where $j \le m$. Clearly, the call $q_j(\bar{X}_j)$ is obtained in fewer than n resolution steps, as is the return from this call. From the induction hypothesis for the theorem, therefore, there must be an ι-pattern $cp = i\_pat(\bar{X}_j, A_{j-1})$ in CALLPAT(q_j) such that cp describes $\sigma_{j-1}(\bar{X}_j)$, and $\langle cp, sp \rangle \in$ SUCCPAT(q_j) such that sp describes $\sigma_j(\bar{X}_j)$. It follows, from Lemma 4.2, that σ_j is consistent with $A_j = update\_i\_state(A_{j-1}, \bar{X}_j, sp)$. This implies that σ_i is consistent with A_i for $1 \le i \le m$. Now the calling pattern inferred for p is $I_c \equiv i\_pat(\bar{S}, A_m)$, and since σ_m is consistent with A_m, it follows from Lemma 4.1 that I_c describes $\sigma_m(\bar{S})$, whence part (1) of the theorem holds.

The inductive argument for part (2) of the theorem, involving success patterns, is similar. *$box*

That the algorithm terminates can be seen from the following: from the definition of *update_i_state* it follows that for any ι-state A, n-tuple of terms $\bar{t}$ and ι-pattern $\bar{I}$ of length n, if a variable x is in the domain of A, then

$$inst(A(x)) \trianglelefteq inst(update\_i\_state(A, \bar{t}, \bar{I})(x)).$$

In other words, instantiations of variables are nondecreasing. Since the mode set Δ is finite, this implies that the instantiation of a variable can only increase through a finite number of values. Given this, and the finite domains of ι-states, nontermination is possible only if the dependency set of some variable oscillates, i.e. if a variable enters and leaves a dependency set repeatedly. It can be seen from the definition of *update_i_state* that the only time a variable y leaves the dependency set of a variable x is in the normalization step, if the instantiation of x or y is **c**. In this case, since instantiations of variables are nondecreasing, y can never be reintroduced into the dependency set of x. Thus, oscillations are not possible, and the algorithm terminates.

4.2. Complexity

Consider a program with p predicates of arity at most a, where each predicate has at most c clauses, and each clause has at most l literals. Suppose there are at most V variables in any clause. For a tuple of size a, computing the ι-pattern for that term in an ι-state involves computing the instantiation of each element of that tuple, and determining sharing of variables, which involves determining whether two terms in the tuple have intersecting variable sets. Determining the instantiation of a term may require the examination of the instantiation of each variable occurring in that term, and may therefore take $O(V)$ time in the worst case;† determining the instantiations of the a terms in the tuple can therefore take $O(aV)$ time. For each pair of terms, determining whether their dependency sets have a nonempty intersection can take time $O(V)$. Since there are $O(a^2)$ such pairs, the total time complexity of *i_pat* is $O(aV + a^2V) = O(a^2V)$.

To compute the cost of *update_i_state*, it is necessary to determine the costs of *a_unify_init*, *propagate_deps*, *propagate_inst* and *normalize*. The worst case cost *a_unify_init* can be obtained as

† By representing sets of variables using bit vectors, and using appropriate data structures for ι-states, the instantiation of a term can actually be determined using $O(1)$ bit vector operations.

follows: each computation of *inherited_inst* takes $O(1)$ time, and processing a variable in *a_unify_init* can involve a computations of *inherited_inst* and ∇, and a union operations. Since each of these can be done in time $O(1)$, their cost is $O(a)$. In computing the coupling closure, there is one *i_pat* operation, which has cost $O(a^2V)$; if coupling closures are computed using straightforward transitive closure methods, then for ι-patterns of size n the time taken to compute the coupling closure of an argument position is $O(n^3)$, with $n = a$ in the worst case. The total cost of *a_unify_init*, which may involve the processing of V variables, is therefore $O((a^3 + a^2V)V) = O(a^3V^2)$. In *propagate_deps*, the dependency set of a variable can be of size V in the worst case, necessitating $O(V)$ union operations for each variable, or $O(V^2)$ operations altogether. In *propagate_inst*, the dependency set of a variable may be of size V in the worst case, so looking through this to determine whether any of them has instantiation $\neq$ **f** can cost $O(V)$. The total cost for processing V variables is therefore $O(V^2)$. In *normalize*, the processing of each variable may involve $O(V)$ unions, so the total cost of *normalize* can be $O(V^2)$ in the worst case. Thus, the worst case time complexity of *update_i_state* is $O(a^3V^2)$.

The processing of each clause involves $O(l)$ applications of *i_pat* and $O(l)$ applications of *update_i_state*. It can be seen from the above that the cost of *i_pat* is dominated by that of *update_i_state*. Thus, the complexity of processing each clause is $O(la^3V^2)$. The cost of processing a predicate, for each calling pattern, is therefore $O(cla^3V^2)$.

Since $|\Delta| = 5$, for each argument position of a predicate there are 5 different instantiations possible, so the total number of instantiations possible for a predicate of arity a is 5^a. The number of possible share sets for such a predicate is the number of partitions of the set $\{1, ..., a\}$. This is given by Bell's number B_a, with $2^a < B_a < 2^{a \log a}$. Thus, the total number of calling and success patterns possible is $5^a B_a$. For simplicity of notation, we write this quantity as $\Phi(a)$.

By using extension tables, each predicate is processed exactly once for any particular calling and success pattern for it: once a success pattern for a predicate has been computed for a calling pattern, on future invocations of that predicate with that calling pattern, this success pattern is not recomputed, but rather is looked up in $O(1)$ time. Thus, the total time complexity of the algorithm, given p predicates in the program, is $O(pcla^3V^2\,\Phi(a))$.

Let the size of a term be the number of symbols in the term, i.e. the number of nodes in its tree representation. If the size of the largest term in a clause is s, then the size of the clause is las, and the size of the program is $N = pclas$. Since each variable in a clause must account for at least one node in the tree representation of the clause, we have $laV = O(las)$. The time complexity of the algorithm can therefore be written as $O((pclas)(a^2V)\Phi(a)) = O(Na^2V\,\Phi(a))$.

This worst case complexity can be misleading, however, for two reasons. The first is that predicates in a program very rarely exhibit all possible calling and success patterns. Typically, a predicate in a program is used consistently with some of its arguments as input arguments and others as output arguments: it is this sort of consistent usage that makes mode analysis meaningful at all. The second reason is that while the analysis above indicates that the algorithm can be worse than exponential in the maximum arity a of a predicate in the program, the arities of predicates commonly encountered tend to be small, so that this is often not a big problem in practice. Define a program to be of *bounded variety* if the number of calling patterns for each predicate in the program can be bounded by a constant. Most programs encountered in practice are of bounded variety. For such programs, the number of calling patterns per predicate is $O(1)$ by definition, and the complexity of the algorithm reduces to $O(Na^2V)$.

In practice, the arity a and number of variables V per clause do not increase as the size of the program increases. For practical purposes, therefore, the algorithm takes time linear in the size of the program.

5. Trading Precision for Speed

While the flow analysis algorithm developed in the previous section is reasonably efficient for programs that are likely to be encountered in practice, there may be occasions when it performs badly. This section discusses how the algorithm may be modified to improve its worst case behavior. While this may involve some loss in precision, our experience indicates that for most programs encountered in practice, this loss in precision tends to be insignificant.

The modification proposed affects only how the tables CALLPAT and SUCCPAT are managed. Recall that given two ι-patterns $I_1 = <<\delta_{11}, S_{11}>, ..., <\delta_{1n}, S_{1n}>>$ and $I_2 = <<\delta_{21}, S_{21}>, ..., <\delta_{2n}, S_{2n}>>$, I_1 *$le* I_2 if and only if δ_{1i} *$le* δ_{2i} and $S_{1i} \subseteq S_{2i}$, $1 \le i \le n$. For any value of $n \ge 0$, the set of ι-patterns form a complete lattice under this ordering, so that for any two ι-patterns I_1 and I_2 of the same length, we can find I_1 *$lub* I_2. In the modified algorithm, each of the tables CALLPAT and SUCCPAT contains only one entry per predicate. Let $I_{\$bottom}(n) = \langle\langle \mathbf{e}, \emptyset\rangle, \ldots, \langle \mathbf{e}, \emptyset\rangle\rangle$ be the ''bottom'' ι-pattern of length n. Initially, the entry in CALLPAT(p) for an n-ary predicate p is $I_{\$bottom}(n)$, and that in SUCCPAT($p$) is $<I_{\$bottom}(n), I_{\$bottom}(n)>$. When a call to a predicate p with calling pattern I is encountered, let the entry in SUCCPAT(p) be $<I_c, I_s>$. If I *$le* I_c, then the ι-pattern I_s is returned as the success pattern for that call; otherwise, the entry in CALLPAT(p) is replaced by the ι-pattern I *$lub* I_c; flow analysis is then carried out as described, but for the calling pattern I *$lub* I_c. If the success pattern computed for this calling pattern is $I_s{}'$, then the entry in SUCCPAT(p) is replaced by $<I$ *$lub* I_c, I_s *$lub* $I_s{}'>$.

The soundness of this algorithm follows from that of the algorithm described in the previous section. Its worst case complexity can be computed as follows: the cost of *update_i_state* remains the same as before, so that the cost of processing c clauses of a predicate, each containing l literals, is, as before, $O(cla^3V^2)$). In this case, however, the number of different calling patterns for which the analysis will be carried out will be far fewer: for each predicate with arity a, the number of different instantiations that can be considered is $3a$, since the height of $\langle\Delta$, *$le_triangle*$\rangle$ is 4 but flow analysis is not carried out for the ''bottom'' ι-patterns $I_{\$bottom}(a)$ initially in each CALLPAT and SUCCPAT tables; the number of different share sets possible for such a predicate is a+1. Thus, the total number of calling patterns that may be considered for a predicate of arity a is now $O(a^2)$, down from 5^aB_a in the previous case. If there are p

predicates in the program, then the total cost of the new algorithm, in the worst case, is therefore $O(Na^4V)$, where $N = pclas$ is the size of the program. For programs of bounded variety, the number of calling patterns is $O(1)$, and the complexity reduces, as before, to $O(Na^2V)$.

It is not difficult to set up an analysis system where the CALLPAT and SUCCPAT tables for some predicates are managed according to the old algorithm, while those for others are managed according to the new algorithm above, as directed by the user. This can yield a spectrum of analysis systems that are intermediate in speed and precision between the two described here, and that can be tuned to an application by simple hints from the user. A point to be noted in this case, however, is that the treatment of unification via =/2 as the predicate defined as

$$=(X, X)$$

can lead to substantial loss in precision. Therefore, =/2 should be handled specially, in the same way as head unification.

6. Applications

Information regarding predicate modes and data dependencies between literals finds many uses in the static optimization of logic programs. The classic application of mode information is in the use of specialized unification routines that have fewer cases to test than the general unification algorithm, and hence are more efficient [25, 26]. Mode information is important in detecting deterministic and functional computations, information regarding which can be used to improve the search behavior of logic programs [6, 17]. Mode information is important in the transformation of logic programs into functional languages [21]. Mode information can also be used to improve the efficiency of programs in systems such as MU-Prolog [19] and Sicstus Prolog [23], which permit the suspension of certain goals depending on the instantiation of variables at runtime: if it can be ascertained that certain arguments of a predicate will always be instantiated (or uninstantiated) at a call via mode analysis, the corresponding runtime tests can be eliminated, thereby leading to faster code. Also, mode information can be used to generate index structures for predicates more intelligently: for example, Prolog compilers typically generate an index on the first argument of each predicate. However, if the mode of the predicate indicates that the first argument of calls to a predicate will always be uninstantiated but some other argument will be instantiated, an intelligent compiler will be able to generate an index on the instantiated argument, thereby reducing the amount of shallow backtracking required.

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,

$$\mathrm{p}(\bar{X}) :- Body_1.$$

and

$$\mathrm{p}(\bar{X}) :- Body_2.$$

it is possible to merge them to produce the clause

$$\mathrm{p}(\bar{X}) :- Body_1 \,;\, Body_2.$$

Amongst 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 7: 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 <**c**, **c**, **f**, **f**> 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 [9], 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 alternatives in the second clause. ●

Knowledge of data dependencies can be used to devise semi-intelligent backtracking schemes for programs that do not incur the runtime overhead of intelligent backtracking [3]. They can also be used in the parallelization of programs [2, 27]. A related algorithm can be used to synthesize control strategies for parallel logic programs [5]. Data dependency information is also necessary for various optimizing transformations of logic programs, e.g. recursion removal, loop fusion, code motion out of loops, etc. [7]. The following example illustrates one such application:

Example 8: Consider the following predicate to compute the maximum and minimum values in a binary tree whose leaves are labelled by integers:

```
maxmin(T, Max, Min) :– maxval(T, Max), minval(T, Min).

maxval(leaf(N), N).
maxval(tree(L, R), Max) :– maxval(L, M1), maxval(R, M2), max(M1, M2, Max).

minval(leaf(N), N).
minval(tree(L, R), Min) :– minval(L, M1), minval(R, M2), min(M1, M2, Min).
```

With this definition, the computation of *maxmin*(*T*, *Max*, *Min*) for any tree *T* requires two traversals of the tree. Unfold/fold transformations can be used to ''fuse'' the two loops in this definition. The transformation begins by unfolding the literals for *maxval* and *minval* in the clause for *maxmin*, which yields

```
maxmin(leaf(N), N, N).
maxmin(tree(L, R), Max, Min) :–
    maxval(L, Mx1), maxval(R, Mx2), max(Mx1, Mx2, Max),
    minval(L, Mn1), minval(R, Mn2), min(Mn1, Mn2, Min).
```

Next, literals that can be shown to be independent using data dependency analysis may be rearranged (provided that other relevant considerations, e.g. termination, are also satisfied). This yields

```
maxmin(leaf(N), N, N).
maxmin(tree(L, R), Max, Min) :–
    maxval(L, Mx1), minval(L, Mn1),
    maxval(R, Mx2), minval(R, Mn2),
    max(Mx1, Mx2, Max), min(Mn1, Mn2, Min).
```

Finally, folding is carried out using the original definition of *maxmin*, which yields the definition

```
maxmin(leaf(N), N, N).
maxmin(tree(L, R), Max, Min) :–
    maxmin(L, Mx1, Mn1),
    maxmin(R, Mx2, Mn2),
    max(Mx1, Mx2, Max), min(Mn1, Mn2, Min).
```

With this definition, the computation of *maxmin*(*T*, *Max*, *Min*) for any tree *T* requires only one traversal of *T*. Further, if the predicates *maxval* and *minval* are not used elsewhere in the program, they may now be discarded, leading to a decrease in code size. The crucial step in the transformation is the rearrangement of literals, where data dependency information is necessary to ensure that the transformation does not change the behavior of the program in unacceptable ways by altering producer-consumer relationships between literals. ●

7. Conclusions

Mode and data dependency information play an important role in the compilation of logic programs to efficient executable code. This paper shows how the two analyses can be combined. It describes a mechanism for managing dependencies between variables in a uniform manner, and uses it to describe a flow analysis algorithm that is then proved sound. This yields an algorithm that is precise, yet efficient for most programs encountered in practice. A variation on the algorithm is also described that offers significantly superior worst-case performance while retaining precision and efficiency for most commonly encountered programs. Some applications of mode and data dependency information to logic programs are also described.

Acknowledgements

Comments by the anonymous referees helped improve the contents and presentation of this paper substantially.

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. J. Chang and A. M. Despain, Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, pp. 10-21.
4. 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.
5. S. K. Debray, Synthesizing Control Strategies for AND-Parallel Logic Programs, Tech. Rep. 87-12, Dept. of Computer Science, The University of Arizona, Tucson, AZ, May 1987.
6. S. K. Debray and D. S. Warren, Automatic Mode Inference for Logic Programs, *J. Logic Programming 5,* 3 (Sep. 1988), pp. 207-229.
7. S. K. Debray, Unfold/Fold Transformations and Loop Optimization of Logic Programs, in *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988, pp. 297-307. SIGPLAN Notices vol. 23 no. 7.
8. S. K. Debray and P. Mishra, Denotational and Operational Semantics for Prolog, *J. Logic Programming 5*, 1 (Mar. 1988), pp. 61-91.
9. 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.
10. S. K. Debray, Flow Analysis of Dynamic Logic Programs, *J. Logic Programming 7,* 2 (Sept. 1989), pp. 149-176.
11. P. Deransart and J. Małuszynski, Relating Logic Programs and Attribute Grammars, *J. Logic Programming 2,* 2 (July 1985), pp. 119-156.

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. G. Janssens and M. Bruynooghe, An Instance of Abstract Interpretation Integrating Type and Mode Inferencing, in *Proc. Fifth International Conference on Logic Programming*, Seattle, Aug. 1988, pp. 669-683. MIT Press.

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

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

16. 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.

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

18. C. S. Mellish, Abstract Interpretation of Prolog Programs, in *Proc. Third International Logic Programming Conference*, London, July 1986. Springer-Verlag LNCS vol. 225.

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

20. D. Plaisted, The Occur-check Problem in Prolog, in *Proc. 1984 Int. Symposium on Logic Programming*, IEEE Computer Society, Atlantic City, New Jersey, Feb. 1984, pp. 272-280.

21. 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.

22. H. Søndergaard, An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction, in *Proc. ESOP ’86*, Saarbrucken, Mar. 1986.

23. *Sicstus Prolog User’s Manual*, Swedish Institute of Computer Science, Sweden, Sep. 1987.

24. 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.

25. 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.

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

27. R. Warren, M. Hermenegildo and S. K. Debray, On the Practicality of Global Flow Analysis of Logic Programs, in *Proc. Fifth International Conference on Logic Programming*, MIT Press, Seattle, Aug. 1988.

Appendix: Mode and Data Dependency Analysis as an Abstract Interpretation

This appendix casts the mode and data dependency analysis presented in the paper as an abstract interpretation. In an actual execution of a program, each program point has associated with it a set of substitutions. If **Subst** be the set of all substitutions, the concrete domain is the set $2^{\textbf{Subst}}$, which forms a complete lattice under inclusion. The abstract domain for a program is the disjoint sum of the abstract domains for each of its clauses. The abstract domain for a clause C is its set of ι-states,

$$\iota\textbf{-State}_C = \mathbf{V}_C \to \Delta \times 2^{\mathbf{V}_C}$$

where $\mathbf{V}_C$ denotes the set of program variables of C, i.e. the variables appearing in the program text for C. For simplicity of notation in the discussion that follows, we consider the abstraction and concretization functions for only a single clause. The extension of this to the entire abstract domain is a tedious but conceptually straightforward construction involving the usual injection and projection operations.

The set of ι-states $\iota\textbf{-State}_C$ for a clause C can be ordered as follows: for any A_1 and A_2 in $\iota\textbf{-State}_C$, $A_1 \ \mathit{\$le}\ A_2$ if and only if for every variable x in $\mathbf{V}_C$, $\mathit{inst}(A_1(x)) \ \mathit{\$le}\ \mathit{inst}(A_2(x))$ and $\mathit{deps}(A_1(x)) \subseteq \mathit{deps}(A_2(x))$, and forms a complete lattice under this ordering. Recall that if A is an ι-state defined on a set of variables $\mathbf{V}$, then a substitution σ is said to be consistent with A if and only if for every x in $\mathbf{V}$, (i) $\sigma(x) \in \mathit{inst}(A(x))$; and ($ii$) if for any $y \in \mathbf{V}$, $\mathit{vars}(\sigma(x)) \cap \mathit{vars}(\sigma(y)) \neq \emptyset$, then $y \in \mathit{deps}(A(x))$. The abstraction function $\alpha : 2^{\textbf{Subst}} \to \iota\textbf{-State}_C$ is defined as follows:

Definition: Given a set of substitutions Θ at a point in a clause C, $\alpha(\Theta) = A$ is the least ι-state in $\iota\textbf{-State}_C$ such that every θ in Θ is consistent with A. ●

The concretization function $\gamma : \iota\textbf{-State}_C \to 2^{\textbf{Subst}}$ is defined as follows:

Definition: Given an ι-state A in $\iota\textbf{-State}_C$, $\gamma(A) = \{\theta \in \textbf{Subst} \mid \theta \text{ is consistent with } A\}$. ●

It is straightforward to show that α and γ are monotonic and adjoint. Consider sets of substitutions Θ_1 and Θ_2, with $\Theta_1 \subseteq \Theta_2$. By definition, every member of Θ_2 is consistent with $\alpha(\Theta_2)$, which implies that every member of Θ_1 is consistent with $\alpha(\Theta_2)$. Since $\alpha(\Theta_1)$ is the *least* ι-state that every member of Θ_1 is consistent with, it follows that $\alpha(\Theta_1) \ \mathit{\$le}\ \alpha(\Theta_2)$, i.e. that α is monotonic.

Let A_1 and A_2 be ι-states defined on a set of variables $\mathbf{V}$, with $A_1 \ \mathit{\$le}\ A_2$. Consider any substitution σ consistent with A_1: since $A_1 \ \mathit{\$le}\ A_2$, $\mathit{inst}(A_1(x)) \ \mathit{\$le}\ \mathit{inst}(A_2(x))$ for any x in $\mathbf{V}$, and $\mathit{deps}(A_1(x)) \subseteq \mathit{deps}(A_2(x))$, whence it follows that σ is consistent with A_2. This implies that if σ is in $\gamma(A_1)$ then σ is also in $\gamma(A_2)$, i.e. $\gamma(A_1) \subseteq \gamma(A_2)$. This establishes that γ is monotonic.

By definition, $\alpha(\Theta)$ is the least ι-state that every element of a set of substitutions Θ is consistent with; it is easy to see that $\gamma(A)$ is the largest set of substitutions all whose elements are consistent with the ι-state A. It follows immediately that $\alpha(\gamma(A)) = A$, and $\Theta \subseteq \gamma(\alpha(\Theta))$, i.e. that α and γ are adjoint.

It remains only to show that ''unification'' in the abstract domain, as defined by *update_i_state*, is faithful to unification in the concrete domain. This proof is essentially that of Lemma 4.2, and is not repeated here. This completes the formalization of our mode and data dependency analysis as an abstract interpretation. *$box*